ВОЛОДИМИР ГАВРИЛКІВ

ПРАКТИЧНІ МЕТОДИ РОЗРОБКИ КОМПІЛЯТОРІВ

НАВЧАЛЬНИЙ ПОСІБНИК

Розробимо лексичний та синтаксичний аналізатори мови програмування, в якій представлено всього два типи змінних: цілий та булевий. Значення змінних цілих типів завжди є обмеженими, і цей факт відображається в їх описі:

VarName: integer[min..max] := IntValue;

Булеві змінні можуть набувати двох значень: true та false. Опис булевої змінної має простіший вигляд:

VarName: boolean := BoolValue;

ПРИКАРПАТСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
Імені Василя Стефаника
1940
UNIVERSITAS

Міністерство освіти і науки України
Прикарпатський національний університет імені Василя Стефаника

Володимир Гаврилків

Практичні методи розробки компіляторів

Навчальний посібник

Івано-Франківськ

2023

УДК 510.5:004.423.24
ББК 22.123
Г 12

Гаврилків В.М. Практичні методи розробки компіляторів: навчальний посібник / В.М. Гаврилків. – Івано-Франківськ: Голіней, 2023. – 72 с.

У посібнику викладено основні практичні методи розробки лексичних та синтаксичних аналізаторів при проектуванні компіляторів мов програмування. Кожен параграф супроводжується завданнями для самостійного розв'язування.

Рекомендовано Вченою радою факультету математики та інформатики як навчальний посібник для студентів освітньої програми "Математика комп'ютерних технологій" (протокол № 5 від 7 червня 2023 р.).

Рецензенти: доктор фізико-математичних наук, професор
Бандура Андрій Іванович, професор кафедри вищої математики Івано-Франківського національного технічного університету нафти і газу.

доктор фізико-математичних наук, доцент
Никифорчин Олег Ростиславович, завідувач кафедри алгебри та геометрії Прикарпатського національного університету імені Василя Стефаника;

Зміст

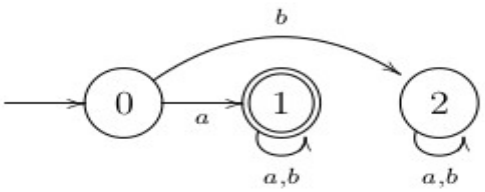
§ 1. Приклади програмної реалізації детермінованих скінченних автоматів

Розробимо програму на мові програмування C++ для реалізації скінченного автомата без виходу, який розпізнає всі слова в алфавіті {a, b}, які починаються з букви 'a'.

Задамо автомат як п'ятірку $A = (Q, X, \delta, q_0, F)$, де $X = \{a, b\}$ — вхідний алфавіт, $Q = \{q_0, q_1, q_2\}$ — множина станів керуючого пристрою, q_0 — початковий стан, $F = \{q_1\}$ — множина кінцевих станів, а функція переходів δ задана таблицею:

δ	a	b
q_0	q_1	q_2
q_1	q_1	q_1
q_2	q_2	q_2

Помічений орієнтований граф цього автомата має наступний вигляд:



Для зручності записів у графі автомата та програмі для позначення станів використовуємо їх індекси. У наступній програмі змінна state позначає активний стан автомата, яким на початку роботи є початковий стан q_0.

```
#include<iostream>
#include<string>
using namespace std;

int main()
{
  //SetConsoleCP(1251); SetConsoleOutputCP(1251); //кодування кирилиці у Windows

      string s;
      cout << "\nВведіть слово в алфавіті {a, b}:\n";
      cin >> s;

      int state = 0;

      int l = s.length();

      for (int i = 0; i < l; i++)
      {
            switch (state)
            {
                  case 0: if (s[i] == 'b') state = 2; else state = 1; break;
                  case 1: if (s[i] == 'b') state = 1; else state = 1; break;
                  case 2: if (s[i] == 'b') state = 2; else state = 2; break;
```

```
            }
      }

      if (state == 1)
      cout << "Слово розпізнається скінченним автоматом!";
            else cout << "Слово не розпізнається скінченним автоматом!";

      return 0;
}
```

Розглянемо приклади роботи програми, яка реалізує скінченний автомат:

Введіть слово в алфавіті {a, b}:
abb (дані користувача)
Слово розпізнається скінченним автоматом!

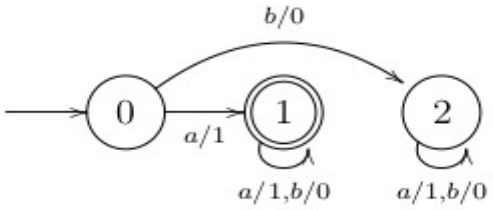
Введіть слово в алфавіті {a, b}:
bba (дані користувача)
Слово не розпізнається скінченним автоматом!

Наступна програма реалізує скінченний автомат з виходом, який розпізнає всі слова в алфавіті {a, b}, які починаються з букви 'a', і замінює всі букви 'a' на '1', а букви 'b' на '0'. Вхідні слова обирає випадково.

Для того, щоб автомат обирав вхідні слова випадковим чином використано наступні функції:

- time(0) – видає в секундах системний час, починаючи з півночі 01.01.1970;
- srand(time(0)) – ініціалізує генератор псевдовипадкових чисел.

Помічений орієнтований граф цього автомата має наступний вигляд:




```
#include<iostream>
#include<cstdlib>
#include<ctime>
using namespace std;

int main()
{
      char s[30];
      srand(time(0));
      int l = 1 + rand() % 30; //генерує випадкову довжину l<=30 вхідного слова

      cout << "Результатом роботи автомата на вхідному слові ";

      for (int i = 0; i < l; i++) //генерує випадкове слово довжини l
      {
```

```
		s[i] = 'a' + rand() % 2;
		cout << s[i];
	}

	int state = 0;

	for (int i = 0; i < l; i++)
	{
		switch (state)
		{
			case 0:	if (s[i] == 'b') {state = 2; s[i] = '0';}
					else  {state = 1; s[i] = '1';} break;
			case 1:	if (s[i] == 'b') {state = 1; s[i] = '0';}
					else  {state = 1; s[i] = '1';} break;
			case 2:	if (s[i] == 'b') {state = 2; s[i] = '0';}
					else  {state = 2; s[i] = '1';} break;
		}
	}

	cout <<  "\nє вихідне слово ";

	for (int i = 0; i < l; i++)
	{
		cout << s[i];
	}

	if (state == 1)
	 cout << "\nВхідне слово розпізнається скінченним автоматом!\n";
	   else cout << "\nВхідне слово не розпізнається скінченним автоматом!\n";

	return 0;
}
```

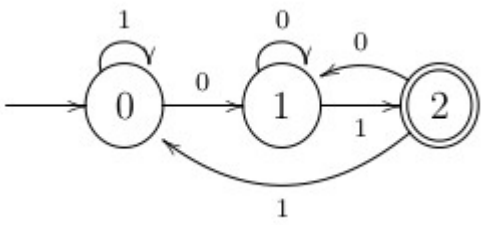
Розглянемо приклади роботи програми, яка реалізує скінченний автомат:

Результатом роботи автомата на вхідному слові aaabbbbaaabb
є вихідне слово 111000011100
Вхідне слово розпізнається автоматом!

Результатом роботи автомата на вхідному слові bbaabbbbaabbabbbabbaabaaaba
є вихідне слово 00110000110010001011011101
Вхідне слово не розпізнається автоматом!

Розробимо програму для ДСА без виходу, який розпізнає всі слова в алфавіті {0,1}, які закінчуються на '01'. Функцію переходів задамо двовимірним масивом.

Помічений орієнтований граф цього автомата має наступний вигляд:



```
#include<iostream>
#include<string>
using namespace std;

int main()
{
    //SetConsoleCP(1251);SetConsoleOutputCP(1251);//кодування кирилиці у Windows

      int delta[3][2] =
      {
            {1, 0},
            {1, 2},
            {1, 0}
      };

      string s;
      cout << "ДСА, який розпізнає слова, що закінчується на 01.\n";
      cout << "Введіть слово в алфавіті {0, 1}: ";
      cin >> s;

      int l = s.length();
      int state = 0;

      cout << "\nТакти роботи автомата: \n";

            for (int i = 0; i < l; i++)
            {
                  cout << "delta(q" << state << "," << s[i] << ")=q";

                  state = delta[state][s[i]-'0'];
                  cout << state << endl;
            }
      if (state == 2)
        cout << "\nСлово розпізнається скінченним автоматом!";
            else cout << "\nСлово не розпізнається скінченним автоматом!";

      //system("pause");
      return 0;
 }
```

Приклади роботи програми (жирним шрифтом виділено дані користувача):

ДСА, який розпізнає слова, що закінчується на 01.
Введіть слово в алфавіті {0, 1}: ***1001***

Такти роботи автомата:
delta(q0,1)=q0
delta(q0,0)=q1
delta(q1,0)=q1
delta(q1,1)=q2

Слово розпізнається скінченним автоматом!

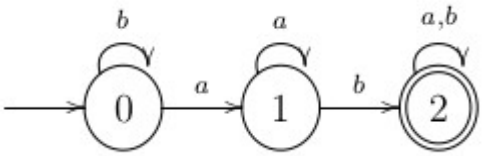
ДСА, який розпізнає слова, що закінчується на 01.
Введіть слово в алфавіті {0, 1}: ***1010***

Такти роботи автомата:
delta(q0,1)=q0
delta(q0,0)=q1
delta(q1,1)=q2
delta(q2,0)=q1

Слово не розпізнається скінченним автоматом!

Наступна програма реалізує ДСА без виходу, який розпізнає всі слова в алфавіті {a,b}, які містять підслово 'ab'. Функція переходів задана двовимірним масивом.

Помічений орієнтований граф цього автомата має наступний вигляд:



Оскільки індекси масивів нумеруються цілими невід’ємними числами, то у наступній програмі на C++ для задання функції переходів двовимірним масивом нумеруємо вхідні букви (змінна num_letter): букву 'a' — номером 0, а букву 'b' — номером 1.

```
#include<iostream>
#include<string>
using namespace std;

int main()
{
	int delta[3][2] =
	{
		{1, 0},
		{1, 2},
		{2, 2}
	};

	string s;
	cout << "ДСА, який розпізнає слова, що місять підслово 'ab'.\n";
	cout << "Введіть слово в алфавіті {a, b}: ";
	cin >> s;

	int l = s.length();
	int state = 0, num_letter;

	cout << "\nТакти роботи автомата: \n";

	for (int i = 0; i < l; i++)
	{
		cout << "delta(q" << state << "," << s[i] << ")=q";
		if ( s[i] == 'a' ) num_letter = 0; else num_letter = 1;
		state = delta[state][num_letter];
		cout << state << endl;
	}
```

```
      if (state == 2)
        cout << "\nСлово розпізнається скінченним автоматом!";
            else cout << "\nСлово не розпізнається скінченним автоматом!";

      return 0;
}
```

Приклади роботи програми (жирним шрифтом виділено дані користувача):

ДСА, який розпізнає слова, що місять підслово 'ab':
Введіть слово в алфавіті {a, b}: ***baaba***

Такти роботи автомата:
delta(q0,b)=q0
delta(q0,a)=q1
delta(q1,a)=q1
delta(q1,b)=q2
delta(q2,a)=q2

Слово розпізнається скінченним автоматом!

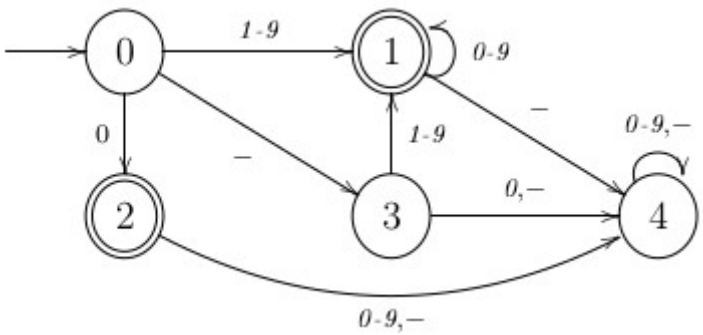
ДСА, який розпізнає слова, що місять підслово 'ab':
Введіть слово в алфавіті {a, b}: ***bbaa***

Такти роботи автомата:
delta(q0,b)=q0
delta(q0,b)=q0
delta(q0,a)=q1
delta(q1,a)=q1

Слово не розпізнається скінченним автоматом!

Розробимо програму для скінченного автомата без виходу, який розпізнає довільне ціле число, яке не містить незначущих нулів. За вхідний алфавіт візьмемо множину X = {0,1,...,9,-}.

Помічений орієнтований граф цього автомата має наступний вигляд:



Розіб’ємо вхідний алфавіт X на такі попарно неперетинні класи, що на кожній букві з одного класу автомат в усіх станах працює ідентично. Класи занумеруємо невід’ємними цілими числами для задання функції переходів у вигляді двовимірного масиву.

Номер класу	Множина символів
0	цифра 0
1	цифри 1-9
2	знак -

Для визначення класу, до якого належить вхідна буква, у наступній програмі використовуємо допоміжну функцію class_symbol(char w). Аргументами функції переходів δ є пари, першою компонентою яких є номер j стану q_j скінченного автомата, а другою — номер class_symbol(w) класу вхідної букви w.

```
#include<iostream>
#include<string>
using namespace std;

int class_symbol(char w)
{
	switch (w)
	{
		case '0' : return 0;
		case '-' : return 2;
		default  : if (isdigit(w)) return 1; else return -1;
	}
}

int main()
{
	int delta[5][3] =
	{
		{2, 1, 3},
		{1, 1, 4},
		{4, 4, 4},
		{4, 1, 4},
		{4, 4, 4}
	};

	string s;
	cout << "ДСА, який розпізнає довільне ціле число.\n";
	cout << "Введіть слово в алфавіті {0,1,...,9,-}: ";
	cin >> s;

	int l = s.length();
	int state = 0;

	cout << "\nТакти роботи автомата: \n";
```

```
        for (int i = 0; i < l; i++)
        {
         if (class_symbol(s[i]) == -1)
              { cout << "\nВведено помилковий символ '" << s[i] << "'.\n";
                    cout << "Слово не є цілим числом!"; return 0; }

        cout << "delta(q" << state << "," << s[i] << ")=q";
        state = delta[state][class_symbol(s[i])];
        cout << state << endl;
        }
    if (state == 1 || state == 2)
      cout << "\nВведено слово є цілим числом без незначущих нулів!";
        else cout << "\nСлово не є цілим числом без незначущих нулів!";

    return 0;
}
```

Розглянемо приклади роботи програми, яка реалізує скінченний автомат (жирним шрифтом виділено вхідні дані користувача):

ДСА, який розпізнає довільне ціле число.
Введіть слово в алфавіті {0,1,...,9,-}: ***-2020***

Такти роботи автомата:
delta(q0,-)=q3
delta(q3,2)=q1
delta(q1,0)=q1
delta(q1,2)=q1
delta(q1,0)=q1

Введено слово є цілим числом без незначущих нулів!

ДСА, який розпізнає довільне ціле число.
Введіть слово в алфавіті {0,1,...,9,-}: ***0020***

Такти роботи автомата:
delta(q0,0)=q2
delta(q2,0)=q4
delta(q4,2)=q4
delta(q4,0)=q4

Слово не є цілим числом без незначущих нулів!

ДСА, який розпізнає довільне ціле число.
Введіть слово в алфавіті {0,1,...,9,-}: ***19x***

Такти роботи автомата:
delta(q0,1)=q1
delta(q1,9)=q1

Введено помилковий символ 'x'.
Слово не є цілим числом!

Завдання для самостійної роботи до параграфа 1

1. Напишіть програму для скінченного автомата без виходу, який розпізнає всі слова в алфавіті {a,b}, які починаються і закінчуються на букву 'a'.

2. Напишіть програму для скінченного автомата без виходу, який розпізнає всі слова в бінарному алфавіті {0,1}, що закінчуються на 101.

3. Напишіть програму для скінченного автомата без виходу, який розпізнає всі слова в алфавіті {0,1,2}, які починаються і закінчуються на однакову букву. Функцію переходів задайте двовимірним масивом.

4. Напишіть програму для скінченного автомата без виходу $A_{m,d} = (Q,X,\delta,q_0,F)$, де $m,d \in N$, $Q = \{q_0,q_1,...,q_{m-1}\}$, $X = \{0,1,...,d-1\}$, $F = \{q_1\}$ і $\delta(q_i,k) = q_{(di+k)\,res\,m}$ ($n\ res\ m$ - остача від ділення n на m). Функцію переходів задайте двовимірним масивом.

5. Напишіть програму для скінченного автомата з виходом, який розпізнає всі слова в алфавіті {a,b,c,d}, і шифрує букви вхідного слова, які знаходяться на парних позиціях, шифром зсуву на дві букви праворуч, а на непарних – на три букви ліворуч. Слова обирає випадковим чином.

6. Напишіть програму для скінченного автомата з виходом, який розпізнає всі слова в алфавіті {0,1,2}, і шифрує вхідне слово, замінюючи цифру d на позиції 3n+k≥1, де n≥0, k∈{0,1,2}, остачею від ділення d+k на 3. Слова обирає випадковим чином.

7. Напишіть програму для скінченного автомата з виходом, у якого вхідний алфавіт X={0,1,a} і вихідний Y={0,1,p,n}. Вихідна послідовність з '0' та '1' збігається з вхідною, а на кожний символ запиту 'a' друкується 'p', якщо кількість '0' від початку роботи парна, і 'n' – якщо непарна.

8. Напишіть програму для скінченного автомата без виходу, який розпізнає довільне додатне десяткове число з плаваючою крапкою в звичайній і експоненціальній формах.

9. Напишіть програму для скінченного автомата без виходу, який розпізнає ідентифікатори: слова в алфавіті {a,b,c,0,1,...,9}, які починаються буквою і закінчуються цифрою.

10. Напишіть програму для скінченного автомата без виходу, який розпізнає наступні ідентифікатори: слова в алфавіті {a,b}, які починаються з 'ab' і не містять підслова 'aa'.

11. Напишіть програму для скінченного автомата без виходу, який розпізнає наступні ідентифікатори: слова в алфавіті {a,b}, які закінчуються на 'ba' і не містять підслова 'bb'.

12. Напишіть програму для скінченного автомата без виходу, який розпізнає слова, які належать регулярній мові $\{0, 10\}^*$.

§ 2. Розробка та реалізація лексичного аналізатора як детермінованого скінченного автомата на мові С++

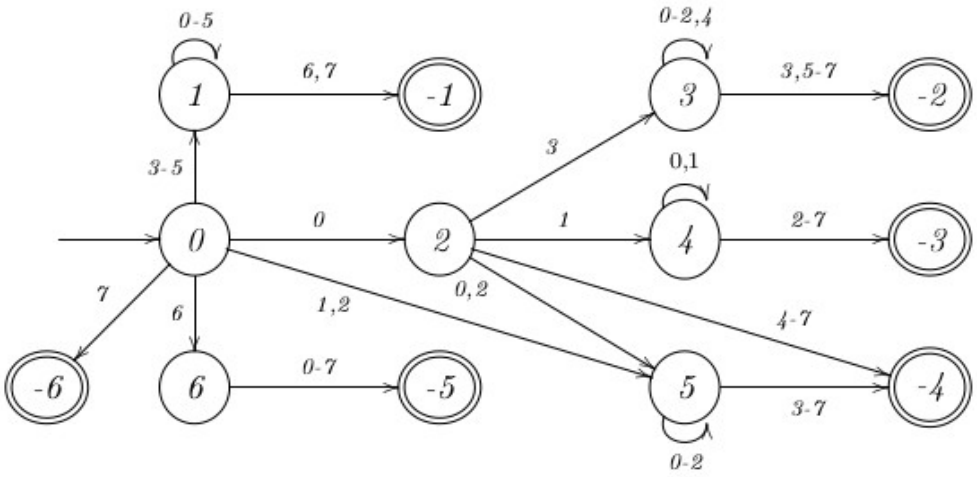
Побудуємо лексичний аналізатор на мові програмування С++ для розпізнавання ідентифікаторів, невід’ємних цілих вісімкових, десяткових і шістнадцяткових констант та операцій '+', '-', '*', '/', '>', '<', '=' мови програмування С. Вхідну послідовність символів збережено у файлі. Задача полягає у знаходженні зазначених лексем у вхідній послідовності символів. Лексичний аналізатор має ігнорувати пропуски та виводити повідомлення про наявність у вхідному тексті помилок, які можуть виникати на етапі лексичного аналізу. Нагадаємо, що у мові С вісімкова константа містить рівно один незначущий нуль і складається з цифр від '0' до '7', шістнадцяткова – починається з символів '0x' і складається цифр від '0' до '9' та букв від 'a' до 'f' і від 'A' до 'F'.

Множину вхідних символів розбиваємо на попарно неперетинні класи. Класи символів обираємо таким чином, щоб забезпечити однозначність переходів автомата, якщо аналізуються не самі символи, а їхні класи.

Номер класу	Множина символів
0	цифра 0
1	цифри 1-7
2	цифри 8-9
3	буква x
4	букви A-F,a-f
5	інші букви, крім букв x, A-F,a-f
6	операції '+', '-', '*', '/', '>', '<', '='
7	інші символи

Таким чином, символи, на які детермінований автомат має забезпечувати різну реакцію, включено у різні класи. Оскільки лексичний аналізатор має розпізнавати різні типи лексем, слід визначити різні типи заключних станів. Тому незаключні стани занумеруємо невід’ємними числами, а заключні – від’ємними. Заключний стан -1 буде означати, що знайдено ідентифікатор; -2 – знайдено шістнадцяткову константу; -3 – вісімкову константу; -4 – десяткову; -5 — арифметичні бінарні операції '+', '-', '*', '/', операції порівняння '>', '<' і операцію присвоєння '='; -6 – інший символ. У графі лексичного аналізатора стрілки помічаємо цифрами, що відповідають класу прочитаного символу. Будемо вважати, що після розпізнавання будь-якої лексеми автомат переходить у початковий незаключний стан 0.

У графі, зображеному на наступному рисунку, в стані 1 відбувається накопичення ідентифікатора, у стані 3 – накопичення шістнадцяткової константи, у стані 4 – вісімкової, а у стані 5 – десяткової:



У наступній програмі на мові C++ визначено змінні state – поточний стан детермінованого скінченого автомата, та begin – початок наступної лексеми:

```
#include<iostream>
#include<fstream>
#include<string>
using namespace std;

int class_symbol(char w)
{
	switch (w)
	{	case '0': return 0;
		case '8': return 2;
		case '9': return 2;
		case 'x': return 3;
		case '+': return 6;
		case '-': return 6;
		case '*': return 6;
		case '/': return 6;
		case '>': return 6;
		case '<': return 6;
		case '=': return 6;
		default:
			if (isdigit(w)) return 1;
			if (isalpha(w))
			  { if ((toupper(w) >= 'A') && (toupper(w) <= 'F')) return 4;
				return 5;
			  }
			return 7;
	}
}
```

```
int main()
{
      //тип лексеми type_lexem[-state-1], яка розпізнається в стані
                              //state = -1,-2,-3,-4,-5
      const char* type_lexem[] = { "Ідентифікатор", "Шіснадцяткова константа",
                     "Вісімкова константа", "Десяткова константа", "Операція" };

      int delta[7][8]=
      {
            {2, 5, 5, 1, 1, 1, 6,-6},
            {1, 1, 1, 1, 1, 1,-1,-1},
            {5, 4, 5, 3,-4,-4,-4,-4},
            {3, 3, 3,-2, 3,-2,-2,-2},
            {4, 4,-3,-3,-3,-3,-3,-3},
            {5, 5, 5,-4,-4,-4,-4,-4},
            {-5,-5,-5,-5,-5,-5,-5,-5}
      };

      char s[255];
      int counter = 0;

      ifstream file("input.txt");
      if (!file.is_open())  cout << "Помилка: файл не знайдено";
      else
       {
                  while (!file.eof())
                  {
                        char c = file.get();
                        s[counter++] = c;
                  }
        cout << "Кількість символів  у файлі дорівнює " << counter - 2 << ".\n";
        file.close();
       }

      cout << "Вміст файлу:\n";

      for (int i = 0; i < counter - 2; i++) { cout << s[i]; }

      cout << "\n\n";

      int state = 0, begin = 0, i = 0;

      while (i < counter - 1)
      {
            if (state == 0) begin = i;

            state = delta[state][class_symbol(s[i])];
            i++;

            if (state < 0)
            {
                  if (state != -6)
                  {
                        i--;
                        cout << type_lexem[-state-1] << ": ";
                        for (int j = begin; j < i; j++)  cout << s[j];
```

```
                        cout << "\n";
                }


                if ((state == -6) && (s[begin] != ' ') && (s[begin] != '\n'))

                        cout << "Помилка: символ '" << s[begin]
                                        << "' не є лексемою вхідної мови\n";
                state = 0;
        }
    }

    return 0;
}
```

Розглянемо приклад роботи програми, що реалізує лексичний аналізатор, над вмістом файлу input.txt:

```
Кількість символів  у файлі дорівнює 80.
Вміст файлу:
let suma = 312 + 2020;
prod = 0123375 * 065343;
id2020 = 0x678fa3b - 0x5afb4db.

Ідентифікатор: let
Ідентифікатор: suma
Операція: =
Десяткова константа: 312
Операція: +
Десяткова константа: 2020
Помилка: символ ';' не є лексемою вхідної мови
Ідентифікатор: prod
Операція: =
Вісімкова константа: 0123375
Операція: *
Вісімкова константа: 065343
Помилка: символ ';' не є лексемою вхідної мови
Ідентифікатор: id2020
Операція: =
Шіснадцяткова константа: 0x678fa3b
Операція: -
Шіснадцяткова константа: 0x5afb4db
Помилка: символ '.' не є лексемою вхідної мови
```

Завдання для самостійної роботи до параграфа 2

Написати програму на мові C++, яка виконує лексичний аналіз вхідного тексту. Програма має ігнорувати пропуски та виводити повідомлення про наявність у вхідному тексті помилок, які можуть виникати на етапі лексичного аналізу.

Варіанти завдань

1. Вхідна мова містить оператори циклу "cycle(...; ...; ...) do ...", розділені символом ';' (крапка з комою). Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', знак присвоєння ':='.

2. Вхідна мова містить оператори циклу "while(...)...done", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', десяткові числа з плаваючою крапкою, знак присвоєння ':='.

3. Вхідна мова містить логічні вирази, розділені символом ';'. Логічні вирази складені з ідентифікаторів, констант '0' і '1', знака '=', операцій 'or', 'xor', 'and', 'not' і круглих дужок.

4. Вхідна мова містить арифметичні вирази, розділені символом ';'. Арифметичні вирази складені з ідентифікаторів, десяткових чисел з плаваючою крапкою (в звичайній і експоненціальній формі), знака '=', знаків операцій '+', '–', '*', '/' і круглих дужок.

5. Вхідна мова містить оператори умови "if...then...else" та "if...then", розділені символом ';'. Оператори умови містять ідентифікатори, знаки порівняння '<', '>', '=', вісімкові числа, знак присвоєння ':='.

6. Вхідна мова містить оператори циклу "do...while(...)", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', римські числа, знак присвоєння ':='. Римськими вважати числа, записані великими літерами 'X', 'V' та 'I'.

7. Вхідна мова містить арифметичні вирази, розділені символом ';'. Арифметичні вирази складені з ідентифікаторів, римських чисел, знака присвоєння ':=', знаків операцій '+', '–', '*', '/' і круглих дужок. Римськими вважати числа, записані великими літерами 'X', 'V' та 'I'.

8. Вхідна мова містить логічні вирази, розділені символом ';'. Логічні вирази складені з ідентифікаторів, констант 'true' і 'false', знака присвоєння ':=', операцій 'or', 'xor', 'and', 'not' і круглих дужок.

9. Вхідна мова містить оператори умови "if...then...else", розділені символом ';'. Оператори умови містять ідентифікатори, знаки порівняння '<', '>', '=', десяткові числа з плаваючою крапкою в експоненціальній формі, знак присвоєння ':='.

10. Вхідна мова містить оператори циклу "cycle(...; ...; ...) do...", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', двійкові числа, знак присвоєння ':='.

11. Вхідна мова містить оператори циклу "while(...)...done", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', рядкові константи (послідовність символів в подвійних лапках).

12. Вхідна мова містить логічні вирази, розділені символом ';'. Логічні вирази складені з ідентифікаторів, констант 'T' і 'F', знака присвоєння ':=', операцій 'or', 'xor', 'and', 'not' і круглих дужок.

13. Вхідна мова містить оператори циклу "do...while(...);", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<=', '>=', '=', десяткові числа, знак присвоєння ':='.

§ 3. Розробка лексичних аналізаторів за допомогою генератора LEX

LEX — (LEXical analyzer) програма для генерування лексичних аналізаторів (сканерів). Перша версія LEX була написана Еріком Шмідтом (Eric Schmidt) і Майком Леском (Mike Lesk). LEX є стандартним генератором лексичних аналізаторів в ОС Unix. Останню версію GNU FLEX (fast LEX) для MS Windows можна звантажити за посиланням http://gnuwin32.sourceforge.net/packages/flex.htm.

LEX зчитує вхідний текстовий файл (lex-специфікацію, опис лексичного аналізатора, у назву якого домовимося включати розширення l, наприклад, scanner.l), і створює на виході код програми на мові програмування С. Lex-специфікація містить код на мові програмування С та список правил, які містять регулярні вирази, що описують послідовності символів (лексеми), які потрібно шукати у вхідному тексті, і дії, які потрібно виконувати, якщо лексеми знайдено. За замовчуванням, скомпонований командою lex <назва lex-специфікації> файл отримує ім’я lex.yy.c. Вихідний файл lex.yy.c потрібно скомпонувати і скомпілювати компілятором мови С. У результаті отримаємо лексичний аналізатор (виконуваний файл), який зчитує стандартний вхідний потік і виокремлює з нього лексеми, які описані регулярними виразами. За замовчуванням дані обираються з stdin (стандартний файл введення, який відкривається операційною системою) і копіюються в stdout (стандартний файл виведення), які за погодженням пов'язуються з терміналом. Змінна yyin вказує на стандартний вхід, а змінна yyout — на стандартний вихід в LEX.

Для зручності наведемо список усіх метасимволів регулярних виразів і вкажемо їх дію:

\n – позначає символ нового рядка;
\t – символ табуляції;
. – позначає довільний символ, крім символу нового рядка \n;
| – використовують для позначення декількох регулярних виразів, розділених |;
() – використовується для групування декількох регулярних виразів в один, наприклад, a(bc|de) відповідає вхідним послідовностям abc і ade;
[] – позначає клас символів, який відповідає одному з символів, записаних в дужках. Знак '-' вказує на діапазон символів. Наприклад, [0-9] позначає те ж саме, що і [0123456789], [a-z] – будь-яка мала літера англійського алфавіту;
^ – позначає початок рядка; всередині квадратних дужок (відразу після першої дужки) використовується як заперечення, наприклад, регулярний вираз [^\t\ n] відповідає кожному символу, крім табуляції і символу нового рядка;
$ – позначає кінець рядка;
* – метасимвол ітерації, який вказує на те, що попередній символ або група символів має повторитися нуль або більше разів підряд;
+ – вказує на те, що попередній символ або група символів має повторитися один або більше разів;
? – вказує на те, що попередній символ або група символів міститься рівно один раз або відсутня;
\ – ставлять перед символом, який зазвичай інтерпретується як метасимвол, щоб використовувати його як звичайний символ (за винятком \n, \t); наприклад, регулярний вираз "сон\." відповіє "сон.", але не "соня";
{k} – вказує на те, що попередній символ (група символів) має повторитися рівно k разів;
{k,m} – вказує на те, що попередній символ (група символів) має повторитися не менше ніж k, але не більше, ніж m разів;
{k,} – вказує на те, що попередній символ (група символів) має повторитися не

менше, ніж k разів;
/ – так званий приєднаний контекст; наприклад, регулярний вираз 0/1 відповідає 0 у вхідному рядку 01, але нічому не відповідає в рядках 0 або 02;
" " – будь-які символи в лапках розглядаються як рядок символів; наприклад, метасимвол * втрачає своє значення і інтерпретуються як два символи: \ і *.

Файл lex-специфікації поділений на три блоки, які відділені один від одного спеціальними рядками, в яких записано "%%", починаючи з першої позиції рядка:

```
Блок визначень

%%

Блок правил трансляції

%%

Блок коду на мові С
```

У блоці визначень задають оголошення потрібних у lex-специфікації змінних, констант, функцій і бібліотек мови С, а також макроси для визначення регулярних виразів. Якщо рядок у блоці визначень починається з пропуску або включений у дужки %{...}%, то він просто копіюється на стандартний вихід. У цьому блоці вказуються також опції з допомогою команди %option, які мають бути враховані при розробці лексичного аналізатора. Макрос вигляду <назва> <регулярний вираз> працює так: якщо після оголошення макросу в наступних макросах або правилах блоку трансляцій потрібно використати "регулярний вираз", то замість нього в фігурних дужках записують його "назва". При генеруванні лексичного аналізатора кожна "назва" автоматично замінюється на "регулярний вираз". Наприклад, макрос виду digit16 [0-9a-fA-F] описує шіснадцяткову цифру. Якщо далі у блоці визначень чи блоці правил трансляцій при побудові наступних регулярних виразів буде використано назву цього макросу, то її потрібно брати у фігурні дужки {digit16}, щоб не плутати з послідовністю символів d i g i t 1 6. У цьому випадку регулярний вираз 0x{digit16}{digit16}* описує множину шіснадцяткових чисел в мові С.

Блок правил трансляції описує регулярні вирази для лексем і асоціює їх з діями, описаними кодом на мові програмування С:

```
r_1 { дія_1; }
r_2 { дія_2; }
.............
r_n { дія_n; }
```

де "r_i" – регулярний вираз, а "дія_i" – фрагмент програми, що описує, яку дію на мові програмування С має виконати лексичний аналізатор, якщо аналізує лексему, описану регулярним виразом "r_i", де i = 1,...,n. Регулярні вирази записують, починаючи з першої позиції рядка. Після них має слідувати принаймні один пропуск, а далі відповідна дія. Якщо у деякій дії записано лише одну команду, то фігурні дужки можна опустити.

"Дія", яка вказана на початку блоку в правилі, яке не містить регулярного виразу, виконується до початку розбору. LEX робить спробу виділити найдовшу послідовність символів з вхідного потоку. Якщо декілька правил трансляції

задають послідовність символів однакової довжини, то виконується правило, яке записане швидше. Дія {ECHO;} копіює відповідну їй лексему в stdout. Якщо кожне правило трансляції є незастосовним до деякого вхідного символу (послідовності символів), то вхідний символ (послідовності символів) копіюється у вихідний потік. Якщо такі символи не потрібно копіювати у вихідний потік, то кожен з них слід замінити на порожній символ. Для цього в кінці правил трансляції слід додати правило вигляду:

```
. ;
```

Третій блок містить оператори і функції на мові C. LEX копіює цю частину коду в кінець генерованого файлу на мові C. Вважається, що ці оператори містять код, який викликається правилами з попереднього блоку. У цьому блоці міститься головна функція main, яка викликає функцію yylex(), а також функція yywrap(), яка визначає, що робити, якщо лексичний аналізатор дійшов до кінця вхідного файлу. Ненульове повернуте значення цієї функції зупиняє розбір вхідного файлу, нульове — вказує, що розбір слід продовжити (для продовження потрібно відкрити наступний вхідний файл). Типовий вміст третього блоку має такий вигляд:

```
int main()
{
    yylex();
}

int yywrap()
{
    return 1;
}
```

LEX дозволяє автоматично включити функцію main у генерований файл і третій розділ залишити порожнім. Для цього у блоці визначень потрібно задати команду:

```
%option main
```

Наприклад, найпростіша lex-специфікація міститиме всього два рядки:

```
%option main

%%
```

Створений лексичний аналізатор копіюватиме всі символи у вихідний потік.

Лексичний аналізатор, створений LEX, реалізується у вигляді детермінованого скінченного автомата і має ім’я yylex(). Він зупиняє роботу (виконується повернення значення з функції yylex()), якщо в одній із дій виконано оператор return (результат yylex() збігається з вказаним у операторі) або досягнуто кінець файлу і значення yywrap() відмінне від 0 (результат yylex() дорівнюватиме 0). Слід відключати використання yywrap з допомогою опції noyywrap, якщо в програмі на мові LEX є своя функція main, у якій і визначено, який файл і коли слід сканувати.

Наприклад, lex-специфікація, збережена у файлі blank.l, для підрахунку кількості пропусків у вхідному тексті матиме наступний вигляд:

```
%option noyywrap
```

```
%{
      int number_blanks = 0;
%}

%%

[" "]  ++number_blanks;

%%

int main()
{
      yylex();
      printf("Кількість пропусків = %d\n", number_blanks);
}
```

Для створення програми в LINUX, що реалізує лексичний аналізатор, слід виконати наступні дії:

1. З командного рядка перейти в директорію, в якій міститься файл lex-специфікації. Команда pwd в LINUX відображає інформацію про поточне знаходження у файловій системі (шлях до директорії, у якій ви працюєте), а команда cd <назва директорії> (change directory - змінити поточну директорію) - дозволяє перейти у директорію з назвою <назва директорії>.

2. Транслювати файл lex-специфікації в код на мові С, користуючись командою:

      lex blank.l або flex blank.l

3. За допомогою команди ls переконатися, що був створений файл lex.yy.c (вихідний файл транслятора LEX на мові С, створений командою lex)

4. Скомпілювати вихідний файл lex.yy.c компілятором мови С:

      gcc lex.yy.c або cc lex.yy.c

5. За допомогою команди ls переконатися, що був створений бінарний (виконуваний) файл a.out

6. Щоб запустити програму a.out, введіть:

      ./a.out

Для того, щоб змінити при компіляції назву бінарного файлу a.out на blank.out, слід скористатися опцією -o, після якої вказати нову назву файлу, тобто виконати команду:

      cc -oblank.out lex.yy.c

Інформацію про інші опції LEX можна отримати, виконавши команду lex –help.

Для того, щоб змінити стандартний вхідний або вихідний потік, можна скористатися командою:

      ./a.out < input.txt > output.txt

Тоді ввід буде виконуватися з файлу input.txt, а вивід – у файл output.txt

У Linux для <<EOF>> слід скористатися комбінацією клавіш Ctrl + D, а у Windows – Ctrl + Z.

Лексичний аналізатор за один такт роботи зчитує одну лексему з вхідного потоку і заносить її у символьний масив **yytext**, а у змінну цілого типу **yyleng** – кількість символів розпізнаної лексеми. Далі переходить до зчитування наступної лексеми і т.д. Тому у кожній дії, яка слідує за регулярним виразом, в якому описана певна лексема, можна виконувати обробку цієї лексеми, використовуючи змінні yytext і yyleng.

LEX, за замовчуванням, присвоює змінній yyin вказівник на стандартний потік вводу. Якщо потрібно сканувати текст з файлу, то слід присвоїти змінній yyin результат виклику функції fopen до виклику yylex:

```
yyin = fopen(argv[1], "r");
```

Наступна LEX-програма, яка міститься у файлі putnumtolines.l, ставить перед кожним рядком його номер, дані бере з файлу, який треба попередньо створити і викликати при запуску бінарного файлу a.out. Наприклад, якщо файл з вхідними даними (не плутайте з файлом putnumtolines.l lex-специфікації!) має назву example.txt, то команда матиме вигляд:

```
./a.out example.txt
```

```
%{

    int lineno = 1;

%}

%%

^(.*)\n     printf("%4d\t%s", lineno++, yytext);

%%

int yywrap(void)
{
    return 1;
}

int main(int argc, char *argv[])
{
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

Нехай файл example.txt містить наступний текст:

LEX is a tool for generating scanners:
programs which recognized lexical patterns in text.
LEX reads the given input files,

```
        or its standard input if no file names are given,
        for a description of  a  scanner to  generate.
```

Виконавши команди:

```
lex putnumtolines.l
cc lex.yy.c
./a.out example.txt
```

на терміналі отримаємо текст:

```
        1     LEX is a tool for generating scanners:
        2     programs which recognized lexical  patterns  in text.
        3     LEX reads the given input files,
        4     or its standard input if no file names are given,
        5     for a description of  a  scanner to  generate.
```

Для полегшення роботи з вхідним потоком, наступний оператор дозволяє лексичному аналізатору обирати спосіб введення з консолі чи файлу:

```
    if (argc < 2) yyin = stdin; else yyin = fopen(argv[1], "r");
```

Згенерувавши файл count.l наступної lex-специфікації, отримаємо програму для підрахунку кількості символів, слів і рядків у вхідному потоці.

```
%{
    int nlines, nwords, nchar;
%}

%option noyywrap

NODELIM [^" "\t\n]

%%

{NODELIM}+   { nwords++; nchar += yyleng; }

\n           nlines++;

.            nchar++;

%%

int main()
{
   nlines = nwords = nchar = 0;
   yylex();
   printf("\nКількість рядків - %d \nКількість слів - %d
                        \nКількість символів -  %d\n", nlines, nwords, nchar);
}
```

Виконавши команди:

```
lex  count.l
cc lex.yy.c
./a.out < example.txt
```

на терміналі отримаємо текст:

Кількість рядків - 5
Кількість слів - 38
Кількість символів - 224

Зауважимо, що у вищенаведеному прикладі для зміни вхідного потоку з терміналу на файл example.txt ми використали символ "<" у команді
./a.out < example.txt.

Лексичний аналізатор, описаний наступною lex-специфікацією, виводить найдовше слово у вхідному потоці.

```
%{
    #include <strings.h>

    int longest = 0;
    char longword[60];
%}

%option noyywrap

%%

[a-zA-Z]+ {if (yyleng > longest) {longest = yyleng; strcpy(longword, yytext);}}
.|\n      ;

%%

int main (void)
{
   yylex();
   printf("Найдовшим словом є \"%s\", яке має %d символів.\n", longword,
            longest);
   return 0;
}
```

Скомпілювавши файли та виконавши команду:

```
./a.out < example.txt > long.txt
```

у поточному каталозі буде створено файл long.txt із вмістом:

Найдовшим словом є "description", яке має 11 символів.

Наступна LEX-програма перевіряє на коректність введені дані. Якщо першими двома введеними лексемами є парне та непарне цілі числа, то вхідні дані є коректними, в іншому випадку — некоректними.

```
%{
      #include <stdlib.h>
      #include <stdio.h>

      int number1;
      int number2;
%}

num [0-9]*

%%

{num}[02468]  {printf("Парне %d-цифрове число.", yyleng); return atoi(yytext);}
{num}[13579]  {printf("Непарне %d-цифрове число.", yyleng);return atoi(yytext);}

%%

int main()
{
  printf("\nВведіть парне число і непарне число:\n");
  number1 = yylex();
  number2 = yylex();
  int difference = number1 - number2;
  if (difference % 2 != 0)
    printf("\nВхідні дані були перевірені на коректність.
                              \nРезультат : дані коректні.\n");
  else
    printf("\nВхідні дані були перевірені на коректність.
                              \nРезультат : дані некоректні.\n");

}

int yywrap()
{
  return 1;
}
```

Розглянемо приклад роботи лексичного аналізатора:

Введіть парне число і непарне число:
2020 11 (введено користувачем)
Парне 4-цифрове число. Непарне 2-цифрове число.
Вхідні дані були перевірені на коректність.
Результат: дані коректні.

Згенерувавши файл наступної lex-специфікації, отримаємо програму, яка будує таблицю з двох стовпців: у першому стовпці вказано довжину слова складеного з малих англійських літер, у другому — кількість слів відповідної довжини.

```
%option noyywrap

    int lengs[30];

%%
```

```
[a-z]+   lengs[yyleng]++;

.|\n     ;

%%

int main()
{
    yylex();
    printf("Довжина  Кількість слів\n");
    for (int i = 0; i < 30; i++)
    if (lengs[i] > 0)  printf("%5d%12d\n", i, lengs[i]);
}
```

Виконавши команду ./a.out < example.txt, на екрані отримаємо:

Довжина	*Кількість слів*
1	*3*
2	*7*
3	*5*
4	*3*
5	*8*
7	*2*
8	*5*
10	*2*
11	*1*

Лексичний аналізатор, описаний наступною lex-специфікацією зчитує два числа, записані в римській системі числення, і видає їх суму в десятковій системі числення.

```
%{
   int decimal = 0;
%}

%option noyywrap

WS      [ \t]+

%%

I           decimal += 1;
IV          decimal += 4;
V           decimal += 5;
IX          decimal += 9;
X           decimal += 10;
XL          decimal += 40;
L           decimal += 50;
XC          decimal += 90;
C           decimal += 100;
CD          decimal += 400;
D           decimal += 500;
CM          decimal += 900;
M           decimal += 1000;
{WS}|\n     return decimal;
```

```
%%

int main ()
{
   int first, second;
   first = yylex();
   decimal = 0;
   second = yylex();

   printf("%d + %d = %d\n", first, second, first+second);
   return 0;
}
```

Розглянемо приклад роботи лексичного аналізатора:

XII CIV (введено користувачем)
12 + 104 = 116

Лексичний аналізатор, заданий наступною lex-специфікацією, переводить арифметичні вирази з четвіркової системи числення у двійкову.

```
%{
    char text[50];
    int i, j;
%}

%%

[0-3]+      { j = 0;
              for (i = 0; i < yyleng; i++)
               {
                 switch (yytext[i])
                   {
                      case '0': text[j] = '0'; text[j+1] = '0'; j += 2;  break;
                      case '1': text[j] = '0'; text[j+1] = '1'; j += 2;  break;
                      case '2': text[j] = '1'; text[j+1] = '0'; j += 2;  break;
                       case '3': text[j] = '1'; text[j+1] = '1'; j += 2;  break;
                   }
               }
              for (j = 0; j < 2 * yyleng; j++) { printf("%c", text[j]); }
              printf(" ");
            }

[-+/*=()]     { ECHO; printf(" "); }

[ \t\n]+    ;

.       { printf("\nПомилка: введено некоректний символ '%c'.\n", yytext[0]);
                                  return 0; };

%%

int main()
```

```
{
  printf("\nВведіть арифметичний вираз у четвірковій системі числення:\n");
  yylex();
}

int yywrap()
{
  return 1;
}
```

Приклад роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

Введіть арифметичний вираз в четвірковій системі числення:
3201 + 2 * (322220 - 231) =
*11100001 + 10 * (111010101000 - 101101) =*

Наступна lex-специфікація описує лексичний аналізатор, який розпізнає цілі і дійсні числа, ідентифікатори, службові слова, арифметичні операції, видаляє однорядкові коментарі, включені в фігурні дужки, і пропуски. Паралельно підраховує кількість ідентифікаторів, цілих та дійсних чисел. Функція atoi(const char*) — перетворює рядок s у ціле число типу int. Аналогічно, atof(const char*) — перетворює рядок s у дійсне число типу float.

```
%{
    #include <math.h> /* в цій бібліотеці міститься функція atof() */
    int num_id = 0, num_int = 0, num_float = 0;
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+         { printf("Ціле число: %d\n", atoi(yytext)); ++num_int; }

{DIGIT}+"."{DIGIT}* { printf("Дійсне число: %g\n", atof(yytext)); ++num_float; }

if|then|else|begin|end|function  { printf("Ключове слово: %s\n", yytext); }

{ID}               { printf("Ідентифікатор: %s\n", yytext); ++num_id; }

"+"|"-"|"*"|"/"     { printf("Бінарна операція: %s\n", yytext); }

"{"[^}\n]*"}"       /* вилучає однорядкові коментарі */

[ \t\n]+            /* вилучає пропуски і символи нового рядка */

.                  printf("Нерозпізнаний символ: %s\n", yytext);

%%

int main()
{
    yylex();
```

```
    printf("\nКількість ідентифікаторів: %d", num_id);
    printf("\nКількість цілих чисел: %d", num_int);
    printf("\nКількість дійсних чисел: %d\n", num_float);
}

int yywrap()
{
      return 1;
}
```

Нехай файл cod.txt містить наступний текст:

```
function value2021

begin
      { comment }
      if a > 2.54 then value = 2021 + 11 * 5.5
                  else value = 0.345 / 100 + 45.20244
end
```

Приклад роботи лексичного аналізатора над вмістом файлу cod.txt:

```
Ключове слово: function
Ідентифікатор: value2021
Ключове слово: begin
Ключове слово: if
Ідентифікатор: a
Нерозпізнаний символ: >
Дійсне число: 2.54
Ключове слово: then
Ідентифікатор: value
Нерозпізнаний символ: =
Ціле число: 2021
Бінарна операція: +
Ціле число: 11
Бінарна операція: *
Дійсне число: 5.5
Ключове слово: else
Ідентифікатор: value
Нерозпізнаний символ: =
Дійсне число: 0.345
Бінарна операція: /
Ціле число: 100
Бінарна операція: +
Дійсне число: 45.2024
Ключове слово: end

Кількість ідентифікаторів: 4
Кількість цілих чисел: 3
Кількість дійсних чисел: 4
```

Розглянемо приклад ще однієї вхідної мови, яка містить оператори циклу наступного вигляду: for (...; ...; ...) do ...

Ці оператори розділені між собою символом ";". Оператори циклу містять ідентифікатори, знаки порівняння "<", ">", "=", арифметичні операції "+", "-", "*", "/", десяткові числа з плаваючою крапкою (записані в звичайній та експоненціальній формі), знак присвоєння ":=". Межами лексем є пропуски, знаки табуляції, знаки наступного рядка і повернення каретки, круглі дужки, крапка з комою і знак двокрапки. При цьому круглі дужки і крапка з комою також є лексемами, а знак двокрапки є одночасно межею лексеми і початком наступної лексеми – операції присвоєння.

Використання опції yylineno дозволяє виконувати нумерацію рядків вхідного файлу і у випадку знайденої помилки виводити користувачу номер рядка, в якому цю помилку виявлено. FLEX визначає змінну yylineno і автоматично збільшує її значення на 1, якщо аналізує символ нового рядка "\n". При цьому FLEX не ініціалізує цю змінну. Тому в функції main перед викликом функції лексичного аналізу yylex змінній yylineno слід присвоїти значення 1.

У функції main в наступному лістингу відкривається файл, ім'я якого вказано користувачем при виклику лексичного аналізатора. Нехай, наприклад, вхідний текстовий файл prog.txt містить наступний програмний код:

```
for(m2:=0; m2 < 2019?; m2 = m2 + 1) do x := y + 5;
for(abc1:=.;abc1<11.0E+1;abc1:=.1)do abc1:= 34E5;
```

LEX-програма має наступний вигляд:

```
%option noyywrap yylineno
%{
      #include <stdio.h>
      int pos;
%}

digit   [0-9]
letter  [a-zA-Z]
delim   [();]
oper    [<>=+"-"*/]
ws      [ \t]

%%

for      { printf("Ключове слово (рядок %d, позиція %d): %s\n", yylineno, pos,
                  yytext); pos += yyleng; }

do       { printf("Ключове слово (рядок %d, позиція %d): %s\n", yylineno, pos,
                  yytext); pos += yyleng; }

("_"|{letter})("_"|{letter}|{digit})*   { printf("Ідентифікатор (рядок %d,
                  позиція %d): %s\n", yylineno, pos, yytext); pos += yyleng; }

[-+]?({digit}*\.{digit}+|{digit}+\.|{digit}+)([eE][-+]?{digit}+)?[flFL]?  {
             printf("Число (рядок %d, позиція %d): %s\n", yylineno, pos,
                           yytext); pos += yyleng; }

{oper}       { printf("Операція (рядок %d, позиція %d): %s\n", yylineno, pos,
                     yytext); pos += yyleng; }

":="         { printf("Операція (рядок %d, позиція %d): %s\n", yylineno, pos,
```

```
                        yytext); pos += yyleng; }

{delim}      { printf("Обмежувач (рядок %d, позиція %d): %s\n", yylineno, pos,
                        yytext); pos += yyleng; }

{ws}+        { pos += yyleng; }

\n           { pos = 1; }

.            { printf("Помилка: невідомий символ (рядок %d, позиція %d): %s\n",
                        yylineno, pos, yytext); pos += yyleng; }

%%

int main(int argc, char **argv)
{

    if (argc < 2)
    {
      printf("\nНедостатньо аргументів. Додайте назву файлу з даними.\n");
      return -1;
    }

    if ((yyin = fopen(argv[1], "r")) == NULL)
    {
      printf("\nНе можливо відкрити файл %s.\n", argv[1]);
      return -1;
    }

    /* наступний оператор дозволяє обирати спосіб введення з консолі чи файлу */
    /*  if (argc < 2) yyin = stdin; else yyin = fopen(argv[1], "r");  */

    pos = 1;
    yylineno = 1;
    yylex();
    fclose(yyin);
    return 0;
}
```

Результатом роботи лексичного аналізатора на вмісті файлу prog.txt є:

Ключове слово (рядок 1, позиція 1): for
Обмежувач (рядок 1, позиція 4): (
Ідентифікатор (рядок 1, позиція 5): m2
Операція (рядок 1, позиція 7): :=
Число (рядок 1, позиція 9): 0
Обмежувач (рядок 1, позиція 10): ;
Ідентифікатор (рядок 1, позиція 12): m2
Операція (рядок 1, позиція 15): <
Число (рядок 1, позиція 17): 2019
Помилка: невідомий символ (рядок 1, позиція 21): ?
Обмежувач (рядок 1, позиція 22): ;
Ідентифікатор (рядок 1, позиція 24): m2
Операція (рядок 1, позиція 27): =
Ідентифікатор (рядок 1, позиція 29): m2

```
Операція (рядок 1, позиція 32): +
Число (рядок 1, позиція 34): 1
Обмежувач (рядок 1, позиція 35): )
Ключове слово (рядок 1, позиція 37): do
Ідентифікатор (рядок 1, позиція 40): x
Операція (рядок 1, позиція 42): :=
Ідентифікатор (рядок 1, позиція 45): y
Операція (рядок 1, позиція 47): +
Число (рядок 1, позиція 49): 5
Обмежувач (рядок 1, позиція 50): ;
Ключове слово (рядок 2, позиція 1): for
Обмежувач (рядок 2, позиція 4): (
Ідентифікатор (рядок 2, позиція 5): abc1
Операція (рядок 2, позиція 9): :=
Помилка: невідомий символ (рядок 2, позиція 11): .
Обмежувач (рядок 2, позиція 12): ;
Ідентифікатор (рядок 2, позиція 13): abc1
Операція (рядок 2, позиція 17): <
Число (рядок 2, позиція 18): 11.0E+1
Обмежувач (рядок 2, позиція 25): ;
Ідентифікатор (рядок 2, позиція 26): abc1
Операція (рядок 2, позиція 30): :=
Число (рядок 2, позиція 32): .1
Обмежувач (рядок 2, позиція 34): )
Ключове слово (рядок 2, позиція 35): do
Ідентифікатор (рядок 2, позиція 38): abc1
Операція (рядок 2, позиція 42): :=
Число (рядок 2, позиція 45): 34E5
Обмежувач (рядок 2, позиція 49): ;
```

У деяких випадках необхідні дії зручно описувати у термінах декількох різних станів (чи різних скінченних автоматів) з явним переходом з одного стану в інший. У цьому випадку набір назв станів потрібно записати в спеціальному рядку блоку визначень, який починається з %start (або %s, %S), а перед відповідними певному стану регулярними виразами блоку правил трансляції записувати назву цього стану в кутових дужках. Перехід у новий стан виконується з допомогою оператора BEGIN <назва>. Лексичний аналізатор починає роботу в стані <INITIAL>, до якого належать всі правила трансляції, в яких перед регулярними виразами не записано жодної назви стану. LEX також підтримує переходи в виключні стани, набір назв яких вказується в рядку блоку визначень, який починається з %x (або %X), а далі записують назви виключних станів. Виключні стани відрізняються від звичайних станів (заданих після %start) тим, що правила блоку трансляцій, перед регулярними виразами яких не записано жодну назву стану, не є активними, якщо лексичний аналізатор перебуває у виключному стані. Наприклад, у наступному прикладі у стані one обидві лексеми abc та def можуть бути розпізнані, а у стані two розпізнаною може бути тільки лексема ghi.

```
%s  one
%x  two

%%

abc         { printf("Лексему розпізнано: "); ECHO; BEGIN one; }
<one>def    { printf("Лексему розпізнано: "); ECHO; BEGIN two; }
```

```
<two>ghi    { printf("Лексему розпізнано: "); ECHO; BEGIN INITIAL; }
```

Наступна LEX-програма видаляє коментарі з програми на мові C. У ній out - стан перебування поза межами коментаря, in — всередині.

```
%option main

%start out in

%%

            BEGIN out;
<out>"/*"   BEGIN in;
<out>.|\n   printf("%s", yytext);
<in>"*/"    BEGIN out;
<in>.|\n
```

Згенерувавши наступну lex-специфікацію, отримаємо лексичний аналізатор, який видаляє коментарі і пропуски, а з кожним класом лексем (токеном) асоціює цілочислове значення у блоці визначень. Зауважимо, що для видалення коментарів використано виключні стани, оскільки дія з регулярним виразом "/*".*"*/" буде також видаляти і вміст text2 між двома коментарями /* text1 */ text2 /* text3 */, намагаючись знайти послідовність символів максимальної довжини.

```
%{
#include <stdio.h>

#define IF 258
#define THEN 259
#define ELSE 260
#define ID 261
#define NUMBER 262
#define COMPOP 263
#define LT 264
#define LE 265
#define EQ 266
#define NE 267
#define GT 268
#define GE 269

int yylval;
%}

%x comment0 comment1

start_comment   "/""*"
delim           [ \t\n]
ws              {delim}+
letter          [A-Za-z]
digit           [0-9]
id              {letter}({letter}|{digit})*
number          {digit}+(.{digit}+)?(E[+-]?{digit}+)?
```

```
%%

<INITIAL>{start_comment}    BEGIN comment0;
<comment0>"*"               BEGIN comment1;
<comment0>[^*]              ;
<comment1>"*"               ;
<comment1>"/"               BEGIN INITIAL;
<comment1>[^*/]             BEGIN comment0;

{ws}                 { /* нічого не повертає */ }
[iI][fF]             { return IF; }
[tT][hH][eE][nN]     { return THEN; }
[eE][lL][sS][eE]     { return ELSE; }
{id}                 { yylval = ID; return ID; }
{number}             { yylval = atoi(yytext); return NUMBER; }
"<"                  { yylval = LT; return COMPOP; }
"<="                 { yylval = LE; return COMPOP; }
"="                  { yylval = EQ; return COMPOP; }
"<>"                 { yylval = NE; return COMPOP; }
">"                  { yylval = GT; return COMPOP; }
">="                 { yylval = GE; return COMPOP; }

%%

int main(argc, argv)
int argc;
char *argv[];
{
  while (yylval = yylex()) printf("Значення '%s' дорівнює  %d\n",yytext,yylval);
}

int yywrap()
{
    return 1;
}
```

Приклад роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

```
/* comment1 */  IF a >= 2.54 then Value = 2021 /*comment2*/ else Value = 0.34
Значення 'IF' дорівнює  258
Значення 'a' дорівнює  261
Значення '>=' дорівнює  263
Значення '2.54' дорівнює  262
Значення 'then' дорівнює  259
Значення 'Value' дорівнює  261
Значення '=' дорівнює  263
Значення '2021' дорівнює  262
Значення 'else' дорівнює  260
Значення 'Value' дорівнює  261
Значення '=' дорівнює  263
Значення '0.34' дорівнює  262
```

Лексичний аналізатор можна будувати також і використовуючи мову програмування C++ в lex-специфікації. Для цього потрібно скомпілювати файл lex.yy.c компілятором C++, виконавши команду **c++ lex.yy.c**

Наступна lex-специфікація count.l++ для підрахунку кількості символів, слів та рядків використовує мову C++:

```
%{
    #include <iostream>
    using namespace std;
    int charCount = 0, wordCount = 0, lineCount = 0;
%}

%option noyywrap

word [^ \t\n]+

%%

{word}          { wordCount++; charCount += yyleng; }
[\n]            { charCount++; lineCount++; }
.               { charCount++; }

%%

main()
{
  yylex();
  cout << "Кількість символів: " << charCount << endl;
  cout << "Кількість слів: " << wordCount << endl;
  cout << "Кількість рядків: " << lineCount << endl;
}
```

Виконавши команди:

```
lex count.l++
c++ lex.yy.c
./a.out < example.txt
```

на терміналі отримаємо текст:

Кількість символів: 229
Кількість слів: 38
Кількість рядків: 5

Завдання для самостійної роботи до параграфа 3

За допомогою генератора лексичних аналізаторів LEX написати програму, яка виконує лексичний аналіз вхідного тексту. Програма має видаляти пропуски і коментарі та виводити повідомлення про наявність у вхідному тексті помилок, які можуть виникати на етапі лексичного аналізу (у повідомленні мають бути вказані номери рядків та позиції помилкових символів); а також підраховувати кількість лексем кожного типу і загальну кількість лексем у вхідному тексті. Коментарі можуть бути двох видів: багаторядкові, включені між символами "/*" і "*/"; та однорядкові, які починаються з символів "//" і закінчуються символом нового рядка "\n".

Варіанти завдань

1. Вхідна мова містить оператори циклу "cycle(...; ...; ...) do ...", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', знак присвоєння ':='. Ідентифікатором є слово з малих англійських літер, яке містить не більше 4 літер.

2. Вхідна мова містить оператори циклу "while(...)...done", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', десяткові числа з плаваючою крапкою, знак присвоєння ':='. Ідентифікатором є слово з англійських літер, яке починається з літер 'a', 'b' або 'c' і остання буква якого збігається з першою.

3. Вхідна мова містить логічні вирази, розділені символом ';'. Логічні вирази складені з ідентифікаторів, констант '0' і '1', знака '=', операцій 'or', 'xor', 'and', 'not' і круглих дужок. Ідентифікатором є слово з великих англійських літер та цифр 2, 3, 4, яке починається літерою і закінчується цифрою.

4. Вхідна мова містить арифметичні вирази, розділені символом ';'. Арифметичні вирази складені з ідентифікаторів, десяткових чисел з плаваючою крапкою (в звичайній і експоненціальній формі), знака '=', знаків операцій '+', '–', '*', '/' і круглих дужок. Ідентифікатором є слово з малих англійських літер, яке містить непарну кількість літер.

5. Вхідна мова містить оператори умови "if...then...else" та "if...then", розділені символом ';'. Оператори умови містять ідентифікатори, знаки порівняння '<', '>', '=', вісімкові числа, знак присвоєння ':='. Ідентифікатором є слово, яке починається з великої англійської літери, а далі містить непарну кількість десяткових цифр.

6. Вхідна мова містить оператори циклу "do...while(...);", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', римські числа, знак присвоєння ':='. Римськими вважати числа, записані великими літерами 'X', 'V' та 'I'. Ідентифікатором є слово з малих англійських літер і цифр 0, 1, яке починається буквою і містить непарну кількість символів.

7. Вхідна мова містить арифметичні вирази, розділені символом ';'. Арифметичні вирази складені з ідентифікаторів, римських чисел, знака присвоєння ':=', знаків операцій '+', '–', '*', '/' і круглих дужок. Римськими вважати числа, записані великими літерами 'X', 'V' та 'I'. Ідентифікатором є слово з малих англійських літер і цифр 5, 6, 8, яке починається з букв 'p', 'q' або 'r' і містить парну кількість символів.

8. Вхідна мова містить логічні вирази, розділені символом ';'. Логічні вирази складені з ідентифікаторів, констант 'true' і 'false', знака присвоєння ':=', операцій 'or', 'xor', 'and', 'not' і круглих дужок. Ідентифікатором є слово з великих англійських літер, яке починається і закінчується літерами 'S', 'T' або 'R'.

9. Вхідна мова містить оператори умови "if...then...else", розділені символом ';'. Оператори умови містять ідентифікатори, знаки порівняння '<', '>', '=', десяткові числа з плаваючою крапкою в експоненціальній формі, знак присвоєння ':='. Ідентифікатором є слово з великих англійських літер і двійкових цифр, яке починається буквою і містить непарну кількість символів.

10. Вхідна мова містить оператори циклу "cycle(...; ...; ...) do...", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', двійкові числа, знак присвоєння ':='. Ідентифікатором є слово з великих англійських літер, яке містить не більше 4 літер.

11. Вхідна мова містить оператори циклу "while(...)...done;", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<', '>', '=', рядкові константи (послідовність символів в подвійних лапках). Ідентифікатором є слово з великих англійських літер і цифр 3, 5, 7, яке починається буквою і містить парну кількість символів.

12. Вхідна мова містить логічні вирази, розділені символом ';'. Логічні вирази складені з ідентифікаторів, констант 'T' і 'F', знака присвоєння ':=', операцій 'or', 'xor', 'and', 'not' і круглих дужок. Ідентифікатор складається з 7 англійських літер.

13. Вхідна мова містить оператори циклу "do...while(...);", розділені символом ';'. Оператори циклу містять ідентифікатори, знаки порівняння '<=', '>=', '=', десяткові числа, знак присвоєння ':='. Ідентифікатором є слово довжини 4 з великих англійських літер, яке починається з літер 'B' або 'C' і остання буква якого не збігається з першою.

14. Використовуючи LEX, розробити програму для підрахунку кількості пропусків у кожному рядку деякого тексту, і загальної кількості пропусків у тексті.

15. Побудувати лексичний аналізатор, який переводить вісімкові натуральні числа у двійкові, та з’ясовує і виводить повідомлення, чи є трійкові натуральні числа парними. Вважаємо, що трійкове число містить рівно один незначущий нуль, а вісімкове число не містить незначущих нулів.

§ 4. Розробка синтаксичних аналізаторів за допомогою генератора YACC (BISON)

YACC — програма для генерування синтаксичних аналізаторів (парсерів). YACC є стандартним генератором синтаксичних аналізаторів в ОС Unix. Останню версію GNU BISON (Yacc-сумісний генератор парсерів) для MS Windows можна звантажити за посиланням http://gnuwin32.sourceforge.net/packages/bison.htm.

Синтаксичні аналізатори в YACC задають за допомогою контекстно вільної граматики, записаної у формі Бекуса-Наура. Наприклад, контекстно вільну граматику з продукціями

```
E -> E + E  (r1)
E -> E * E  (r2)
E -> id     (r3)
```

записують у формі

```
E -> E + E | E * E | id .
```

У цій граматиці символ E є нетерміналом, а id — терміналом (лексемою, яку повертає LEX).

Використаємо цю граматику, щоб породити арифметичний вираз x + y * z:

```
E -> E * E          (r2)
  -> E * z          (r3)
  -> E + E * z      (r1)
  -> E + y * z      (r3)
  -> x + y * z      (r3)
```

Для синтаксичного розбору цього виразу потрібно виконати операції в протилежному порядку. Замість того, щоб, починаючи з початкового нетерміналу, породити арифметичний вираз, потрібно згортати цей вираз до початкового нетерміналу. Відомо, що мова, породжена контекстно вільною граматикою, розпізнається також деяким скінченним автоматом з магазинною пам’яттю, який використовує стек для зберігання інформації. Розглянемо виведення для виразу x + y * z в оберненому порядку:

```
1     . x + y * z       зсув
2     x . + y * z       згортка (r3)
3     E . + y * z       зсув
4     E + . y * z       зсув
5     E + y . * z       згортка (r3)
6     E + E . * z       зсув
7     E + E * . z       зсув
8     E + E * z .       згортка (r3)
9     E + E * E .       згортка (r2)          обчислює добуток
10    E + E .           згортка (r1)          обчислює суму
11    E .               вираз розпізнано
```

Символи, які розміщені ліворуч від символу '.' (крапка), містяться у стеку, а нерозпізнану автоматом частину виразу розміщено праворуч від крапки. Аналіз виразу починається із занесення символів у стек (операція "зсув"). У випадку, коли верхня частина стеку збігається з правою частиною деякої продукції, автомат замінює її на ліву частину цієї продукції (операція

"згортка"). Іншими словами, символи, які збіглися з правою частиною продукції вискакують зі стеку, а відповідні символи лівої частини продукції заносяться у стек. Цей процес триває доти, доки весь вираз не буде занесено у стек, і лише початковий нетермінал залишиться у стеку. На кроці 1 автомат з магазинною пам'яттю заносить x у стек. Крок 2 застосовує правило r3 до верхнього символу стеку, щоб замінити x на E. Автомат продовжує виконувати операції занесення у стек та заміни верхніх символів стеку на відповідні ліві частини продукцій, доки у стеку не залишиться лише початковий нетермінал. На кроці 9, коли виконано згортку за правилом r2, автомат видає добуток. Аналогічно, автомат видає суму на кроці 10. Таким чином, операція добутку має вищий пріоритет, ніж операція суми. Розглянемо детальніше зсув на кроці 6. Замість зсуву автомат міг би виконати згортку за правилом r1. У цьому випадку додавання мало б вищий пріоритет, ніж множення. Ситуація такого типу називається *конфліктом зсуву/згортки*. Наша граматика є неоднозначною, бо є декілька виведень для арифметичного виразу. У такому випадку в YACC слід використовувати оператор для задання пріоритету виконання операцій.

Наступна граматика має *конфлікт згортки/згортки*. Якщо термінал id міститься на вершині стеку, то можна його згорнути як до T, так і до E.

```
E -> T
E -> id
T -> id
```

Якщо в процесі розбору виникає конфлікт і явно не вказано пріоритет виконання, то YACC обирає дію за замовчуванням. Для конфлікту зсуву/згортки обирає зсув, а для конфлікту згортки/згортки обирає продукцію, яка записана швидше.

YACC зчитує вхідний текстовий файл (yacc-специфікацію, опис синтаксичного аналізатора, в назву якого домовимося включати розширення yacc, наприклад, parser.yacc, і створює на виході код програми на мові програмування C.

Файл yacc-специфікації поділений на три блоки, які відділені один від одного спеціальними рядками, в яких записано "%%", починаючи з першої позиції рядка:

```
Блок визначень

%%

Блок продукцій контекстно вільної граматики

%%

Блок коду на мові C
```

У блоці визначень задають оголошення потрібних у yacc-специфікації змінних, констант, функцій і бібліотек мови C, а також оголошення терміналів контекстно вільної граматики командою %token NAME1 NAME2 ... та початкового нетермінала командою %start name. Термінали описують очікувані синтаксичним аналізатором лексеми, які отримано на етапі лексичного аналізу. У цьому випадку NAME1, NAME2,... автоматично визначаються при генерації синтаксичного аналізатора як цілочислові константи, і їх значення очікуються аналізатором серед інших кодів лексем, які повернуться функцією yylex() лексичного

аналізатора. Якщо початковий нетермінал прямо не визначений, то за замовчування ним є нетермінал у лівій частині першої продукції у блоці продукцій контекстно вільної граматики. Імена, які не є задекларованими командою token, вважаються нетерміналами. Якщо рядок у блоці визначень починається з пропуску або включений в дужки %{...}%, то він просто копіюється на стандартний вихід.

У блоці продукцій контекстно вільної граматики об’єднана продукція у формі Бекуса-Наура (тобто в яку включені всі продукції з однаковими лівими частинами) виду $A \rightarrow \omega_1 \mid \omega_2 \mid ... \mid \omega_n$ записується як $A : \omega_1 \mid \omega_2 \mid ... \mid \omega_n$;. Кожне слово ω_i складається з розділених пропусками нетерміналів, терміналів та семантичних дій на мові програмування С, взятих у фігурні дужки.

Третій блок містить оператори і функції мови С. YACC копіює цю частину коду в кінець генерованого файлу на мові С. У цьому блоці міститься головна функція main, яка викликає функцію yyparse(), а також функція yyerror (char *s), яка видає повідомлення про помилки. Типовий вміст третього блоку має такий вигляд:

```
int main (void)
{
    yyparse();
    return 0;
}

void yyerror (char *s)
{
    fprintf(stderr, "YACC: %s\n", s);
}
```

Генератор YACC створює програму — синтаксичний аналізатор, яка може обробляти конструкції з декількох слів відповідно до заданої контекстно вільної граматики. Такий синтаксичний аналізатор може працювати з лексичним аналізатором, створеним програмою LEX. У цьому випадку при поверненні типу лексеми (наприклад, NAME) кожне правило lex-специфікації має завершуватися наступним оператором:

```
return NAME
```

На основі файлу yacc-специфікації програма YACC створює файл y.tab.c на мові С. Після компіляції файлу командою cc формується код функції yyparse, яка повертає ціле число. При роботі функція yyparse викликає для читання лексем функцію yylex. Функція yylex повертає лексеми доти, доки синтаксичний аналізатор не виявить помилку або yylex не поверне маркер кінця, що означає завершення вхідного потоку. При виникненні непереборної помилки функція yyparse повертає в функцію main значення 1. При виявленні маркера кінця функція yyparse повертає в функцію main значення 0.

Генератор YACC присвоює ціле значення кожній лексемі, визначеній у yacc-специфікації за допомогою команди #define препроцесора С. У лексичного аналізатора має бути доступ до цих макроозначень, щоб він міг повертати лексеми синтаксичному аналізатору. Опція **-d** команди yacc (bison) дозволяє створити файл y.tab.h (name.tab.h, де name.y — назва файлу yacc/bison-специфікації). Цей файл далі має бути підключений до файлу lex-специфікації шляхом додавання наступного рядка у блок визначень:

```
%{
      #include "y.tab.h"
%}
```

Якщо трансляцію файла name.y в код мови програмування C виконано з допомогою команди bison -d name.y, то директиву #include "y.tab.h" слід замінити на директиву #include "name.tab.h".

Можна також включити файл lex.yy.c в програму, створену YACC, додавши наступний рядок після другого роздільника %% у файлі yacc-специфікації:

```
#include "lex.yy.c"
```

Для створення програми в LINUX, що реалізує лексичний аналізатор, слід виконати наступні дії:

1. З командного рядка перейти в директорію, в якій містяться файли lex-специфікації та yacc-специфікації. Команда pwd в LINUX відображає інформацію про поточне знаходження у файловій системі (шлях до директорії, у якій ви працюєте), а команда cd <назва директорії> (change directory - змінити поточну директорію) - дозволяє перейти у директорію з назвою <назва директорії>.

2. Транслювати текстовий файл name.y yacc-специфікації в код на мові C, користуючись командою:

      yacc -d name.y або bison -d name.y

3. За допомогою команди ls переконатися, що були створені файли y.tab.h та y.tab.c (name.tab.h та name.tab.c для команди bison).

4. Транслювати текстовий файл name.l lex-специфікації, у блоці визначень якого має бути команда #include "y.tab.h" (#include "name.tab.h" для bison), в код на мові C, користуючись командою:

      lex name.l або flex name.l

5. За допомогою команди ls переконатися, що був створений файл lex.yy.c

6. Скомпонувати та скомпілювати файли lex.yy.c та y.tab.c (name.tab.c для bison) компілятором мови C:

      cc lex.yy.c y.tab.c або gcc lex.yy.c y.tab.c
      (cc lex.yy.c name.tab.c або gcc lex.yy.c name.tab.c для bison)

7. За допомогою команди ls переконатися, що був створений бінарний (виконуваний) файл a.out

8. Для того щоб запустити програму a.out, слід виконати команду: ./a.out

Для того, щоб змінити назву бінарного файла a.out на name.out, можна скористатися командою:

      mv -f a.out name.out

Інформацію про інші опції YACC (BISON) можна отримати, виконавши команду:

yacc --help (bison --help)

Для того, що змінити стандартний вхідний або вихідний потік, можна скористатися командою:

./a.out < input.txt > output.txt

Тоді ввід буде виконуватися з файлу input.txt, а вивід - у файл output.txt

У Linux для <<EOF>> слід скористатися комбінацією клавіш Ctrl + D, а у Windows - Ctrl + Z.

Наступний синтаксичний аналізатор реалізує контекстно вільну граматику, що породжує формальну мову $\{a^n b^n : n \in \mathbb{N}\}$.

Вміст файлу anbn.l lex-специфікації:

```
%{
    #include "y.tab.h"
%}

%%

a   return A;
b   return B;
.   return yytext[0];
\n  return '\n';

%%

int yywrap()
{
    return 1;
}
```

Вміст файлу anbn.y yacc-специфікації:

```
%{
    #include <stdio.h>
    int yylex();
    void yyerror(char*);
%}

%token A B

%%

start :   anbn '\n' { printf("Вхідне слово породжується граматикою.\n");
                      return 0; }
      ;

anbn  :   A B
```

```
      |   A anbn B
      ;

%%
int main()
{
    printf("Введіть, будь ласка, слово в алфавіті {a, b}:\n");
    yyparse();
    return 0;
}

void yyerror(char *s)
{
     printf("Вхідне слово не породжується граматикою.\n");
}
```

Приклад роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

Введіть, будь ласка, слово в алфавіті {a, b}:
aaabbb
Вхідне слово породжується граматикою.

Введіть, будь ласка, слово в алфавіті {a, b}:
abba
Вхідне слово не породжується граматикою.

Якщо разом з типом неодносимвольної лексеми потрібно передати і її значення (для односимвольних лексем цілочисельний ASCII-код символу і його значення збігаються), то для цього використовують глобальну змінну **yylval**, яка за замовчуванням має цілий тип int. Значення від 0 до 257 є зарезервованими для символів і керуючих функцій відповідно до кодування ASCII. Тому цілі значення для неодносимвольних лексем починаються з 258.

Кожне слово ω_i правої частини продукції A : ω_1 | ω_2 |...| ω_n ; може мати, наприклад, наступний вигляд:

A_{i1} {семантична дія_1;} A_{i2} {семантична дія_2;} ... A_{in} {семантична дія_n;} ;,

де A_{ij} є терміналом або нетерміналом.

У процесі роботи синтаксичного аналізатора з кожним терміналом, нетерміналом чи семантичною дією може бути пов’язане певне значення. Складові слова ω_i нумеруються, починаючи зліва. Значення складової з номером і поміщено у змінну $i. У нашому випадку, $1 – значення A_{i1}, $2 – значення { семантична дія_1; }, $3 – значення A_{i2}, $4 – значення { семантична дія_2; } і т.д. Значення лівої частини продукції A поміщено у змінну $$. За замовчування значення $$ збігається з $1.

Наступні lex- та yacc-специфікації описують калькулятор, який обчислює значення виразів, складених з натуральних чисел та операцій додавання і віднімання.

Вміст файлу lex-специфікації:

```
%{
    #include "y.tab.h"
    int pos = 0;
%}


%%

[0-9]+      { yylval = atoi(yytext); printf("Просканована число %d.\n",
                yylval); pos += yyleng; return NUMBER; }
[ \t]       { printf("Проігноровано пропуск.\n"); pos += yyleng; }
\n          { printf("Досягнуто кінець рядка.\n");    pos += yyleng; return 0; }
"+"         { printf("Знайдено знак \"%s\".\n", yytext); pos += yyleng;
                    return yytext[0]; }
"-"         { printf("Знайдено знак \"%s\".\n", yytext);  pos += yyleng;
                return yytext[0]; }
"="         { printf("Знайдено знак \"%s\".\n", yytext);  pos += yyleng;
                return yytext[0]; }
.           { printf("Знайдено невідомі дані \"%s\".\n", yytext);  pos +=
                yyleng; return yytext[0]; }

%%

int yywrap()
{
    return 1;
}
```

Вміст файлу yacc-специфікації:

```
%{
    #include <stdio.h>
    void yyerror(char*);
    int yylex(void);
    extern int pos;
%}

%token NUMBER

%%

statement  :   expression   { printf("Значення виразу дорівнює %d.\n", $1); }
           ;

expression :   expression '+' NUMBER      { $$ = $1 + $3; }
           |   expression '-' NUMBER      { $$ = $1 - $3; }
           |   NUMBER                     { $$ = $1; }
           ;

%%

int main(void)
{
      printf("Введіть, будь ласка, арифметичний вираз з операціями \"+\"
                  та \"-\":\n");
```

```
      yyparse();
      return 0;
}

void yyerror (char *s)
{
      printf("Знайдено помилку на позиції %d!\n", pos);
}
```

Приклади роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

Введіть, будь ласка, арифметичний вираз з операціями "+" та "-":
3+5
Просканована число 3.
Знайдено знак "+".
Просканована число 5.
Досягнуто кінець рядка.
Значення виразу дорівнює 8.

Введіть, будь ласка, арифметичний вираз з операціями "+" та "-":
3 +
Просканована число 3.
Проігноровано пропуск.
Знайдено знак "+".
Досягнуто кінець рядка.
Знайдено помилку на позиції 4!

Розробимо калькулятор, який містить арифметичні вирази з операціями '*', '/', '+', '-'. Контекстно-вільна граматика, яку ми використаємо, є неоднозначною. Для усунення конфліктів зсуву/згортки та згортки/згортки введемо пріоритет операцій і домовимся про порядок виконання зліва направо (лівоасоціативність) для рівнопріоритетних операцій. У YACC використовують команду %left для визначення лівоасоціативності, %right — для правоасоціативності, %nonassoc — щоб вказати відсутність асоціативності. Ці команди разом з операціями слід вказати у блоці визначень yacc-специфікації. Оператори в одній групі мають однаковий пріоритет. Нижче записані оператори мають вищий пріоритет. Для того, щоб додати в мову арифметичних формул унарний мінус, то його пріоритет має бути вище пріоритету бінарних операцій. Домогтися цього можна, запровадивши фіктивну змінну UMINUS з високим пріоритетом (%nonassoc UMINUS в блоці визначень) і використавши оператор %prec, який змінює пріоритет виконання у продукціях, прирівнявши його до пріоритету наступної за ним змінної: '-' expression %prec UMINUS.

Вміст файлу lex-специфікації:

```
%{
     #include "y.tab.h"
%}

%%

[0-9]+      { yylval = atoi(yytext); printf("Просканована число %d.\n",
                                        yylval); return NUMBER; }
[ \t]       { printf("Проігноровано пропуск.\n"); }
```

```
\n          { printf("Досягнуто кінець рядка.\n"); return 0; }
.           { printf("Знайдено невідомі дані \"%s\".\n", yytext); return
                yytext[0]; }

%%

int yywrap()
{
    return 1;
}
```

Вміст файлу yacc-специфікації:

```
%{
    #include <stdio.h>
    void yyerror(char *);
    int yylex(void);
%}

%token NUMBER

%left '-' '+'

%left '*' '/'

%nonassoc UMINUS

%%

statement  :    expression  { printf("Значення виразу дорівнює %d\n", $1); }
           ;

expression :    expression '+' expression { $$ = $1 + $3;   printf("Розпізнано
                                            знак '+'.\n"); }
           |    expression '-' expression { $$ = $1 - $3; printf("Розпізнано
                                         знак '-'.\n"); }
            |    expression '*' expression { $$ = $1 * $3; printf( "Розпізнано
                                            знак '*'.\n"); }
           |   expression '/' expression     { if ($3 == 0) { yyerror("Помилка:
                              ділення на нуль!"); return 0;} else $$ = $1 / $3;
                                   printf("Розпізнано знак '/'.\n"); }
           |    '-' expression %prec UMINUS    { $$ = - $2; printf("Розпізнано
                                                   унарний мінус.\n"); }
           |    '(' expression ')' { $$ = $2;  printf("Розпізнано парні
                                         дужки.\n"); }
           |    NUMBER       { $$ = $1;  printf("Розпізнано число.\n"); }
           ;
%%

int main(void)
{
      printf("Введіть, будь ласка, арифметичний вираз з
                        операціями  \"*\", \"/\", \"+\" та \"-\":\n");
      yyparse();
      return 0;
```

```
}
void yyerror(char *s)
{
      fprintf(stderr, "%s\n", s);
}
```

Приклад роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

Введіть, будь ласка, арифметичний вираз з операціями "", "/", "+" та "-":*
3*5
Просканована число 3.
Розпізнано число.
Знайдено невідомі дані "".*
Просканована число 5.
Розпізнано число.
Розпізнано знак ''.*
Досягнуто кінець рядка.
Значення виразу дорівнює 15

У наступному прикладі ми розширимо калькулятор з попереднього розділу, дозволивши використовувати односимвольні змінні в операторах присвоєння. Лексичний аналізатор повертає лексеми типів VARIABLE (односимвольні змінні) та INTEGER (натуральні числа). Для односимвольних змінних змінна yylval вказує індекс у масиві sym. Елемент sym[i] масиву sym просто містить ціле ASCII значення малої англійської літери, яка розміщена під номером і у алфавіті (починаючи відлік з літери ‘a’). Якщо повертаються лексеми типу INTEGER, то змінна yylval містить значення відповідного числа.

Вміст файлу lex-специфікації:

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include "y.tab.h"
    void yyerror(char *);
%}

%%

[a-z]           { yylval = *yytext - 'a'; return VARIABLE; }

[0-9]+          { yylval = atoi(yytext); return INTEGER; }

[-+()=/*\n]     { return *yytext; }

[ \t]           ;

.               yyerror("Помилковий символ");

%%

int yywrap()
{
    return 1;
}
```

Вміст файлу yacc-специфікації:

```
%{
    #include <stdio.h>

    void yyerror(char*);
    int yylex(void);
    int sym[26];
%}

%token INTEGER VARIABLE

%left '+' '-'

%left '*' '/'

%%

program   : program statement '\n'
          |
          ;

statement : expr              { printf("Значення виразу дорівнює %d\n", $1); }
          | VARIABLE '=' expr      { sym[$1] = $3; }
          ;

expr : INTEGER
     | VARIABLE                  { $$ = sym[$1]; }
     | expr '+' expr             { $$ = $1 + $3; }
     | expr '-' expr             { $$ = $1 - $3; }
     | expr '*' expr             { $$ = $1 * $3; }
     | expr '/' expr             { $$ = $1 / $3; }
     | '(' expr ')'              { $$ = $2; }
     ;

%%

int main(void)
{
    printf("Введіть, будь ласка, вирази з присвоєннями значень змінним і
                операціями \"*\", \"/\", \"+\" та \"-\":\n");
    yyparse();
    return 0;
}
```

Приклад роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

Введіть, будь ласка, вирази з присвоєннями значень змінним і операціями "", "/", "+" та "-":*
x=3*5
y=2
x+y
Значення виразу дорівнює 17

За замовчуванням глобальна змінна yylval, яка містить значення лексеми, має цілий тип int. Щоб змінити тип даних використовують макрос YYSTYPE у блоці визначень yacc-специфікації. Наприклад, щоб змінити тип даних на double, слід задати команду:

```
%{
      #define YYSTYPE double
%}
```

Якщо типів даних є більше одного, то є можливість асоціювати різні типи даних з різними лексемами. Для їх задання можна використовувати директиву %union. Директива діє аналогічно оператору union мови C. За ключовим словом у фігурних дужках вказують типи даних і змінні, які зберігають можливі значення, асоційовані з лексемами, наприклад:

```
%union
{
      char* string;
      long number;
}
```

У цьому випадку різні лексеми мають бути оголошені з використанням посилань на різні поля. Директива оголошення лексем, що дозволяє виконати таке зіставлення, буде розглянута нижче. Самі дані асоціюються з лексемами в процесі сканування тексту лексичним аналізатором. Для цього використовують програмний код дії, що задається при оголошенні лексеми в файлі, що описує лексичний аналізатор. Код синтаксичного аналізатора містить глобальну змінну yylval, оголошену зовнішньою у включеному файлі зі списком кодів лексем. Використання подібного файлу в коді сканера текстів дозволяє, наприклад, переписати оголошення дії для лексеми TOKEN наступним чином:

```
TOKEN  {
            yylval.string = strdup(yytext);
            return TOKEN_KW;
       }
```

У цьому прикладі лексичний аналізатор буде передавати в синтаксичний аналізатор, крім коду лексеми TOKEN_KW, ще й безпосереднє значення цієї лексеми через глобальну змінну yylval.string. Отримане значення може бути в подальшому використано в коді дії, що пов'язується з даною лексемою синтаксичним аналізатором.

Всі ідентифікатори, оголошені в директивах token, автоматично визначаються утилітою yacc через директиву #define в результуючому коді аналізатора. Додаткова інформація про тип даних, що асоціюються з лексемами, може бути задана в директиві наступним чином:

```
%token <string> TOKEN_KW
```

У наведеному прикладі з лексемою TOKEN_KW зв'язується рядкове значення (string), для зберігання якого попередньо оголошено поле в директиві %union. При використанні директиви %union в директиві %token обов'язково має бути зазначений тип.

Існує також можливість зв'язування специфічного значення з нетермінальним

символом, який використовується всередині синтаксичного аналізатора. Для цього використовується директива %type, яка аналогічна директиві %token для лексем.

Розглянемо приклад аналізатора для присвоєння змінній дійсного значення з допомогою оператора "<-".

Вміст файлу lex-специфікації:

```
%{
    #include "y.tab.h"
%}

alphabetic      [A-Za-z]
digit           [0-9]
alphanumeric    ({alphabetic}|{digit})

%%

[+-]?{digit}*(\.)?{digit}+  { sscanf(yytext, "%lf", &yylval.real); return
                                    REAL; }

{alphabetic}{alphanumeric}* { strcpy(yylval.str, yytext); return ID; }

\<\-                        return ASS;

\n                          return NL;

%%

int yywrap()
{
    return 1;
}
```

Вміст файлу yacc-специфікації:

```
%{
    #include <stdio.h>
    int yylex();
    void yyerror(char*);
%}

%union
{
    double  real;
    int     integer;
    char    str[30];
}

%token <real> REAL
%token <str> ID
%token ASS NL
%type <real> ass_st

%%
```

```
ass_st:   ID ASS REAL NL  { $$ = $3; printf("%s присвоєно значення %g\n", $1,
                              $$); return 0; }

%%

int main()
{
    yyparse();
    return 0;
}

void yyerror(char *s)
{
     printf("%s, це не є присвоєнням!\n", s);
}
```

Якщо на вхід подати:

x <- 3.21

То на виході отримаємо:

x присвоєно значення 3.21

Обробка помилок при синтаксичному розборі

Якщо вхідний потік не задовольняє заданій граматиці, то синтаксичний аналізатор в момент аналізу лексеми, яка робить неможливим продовження розбору, фіксує помилку. Цю лексему ми на далі будемо називати помилковою лексемою. Насправді помилка може бути викликана не тільки невірними вхідними даними, а й некоректністю самого синтаксичного аналізатора, що є наслідком некоректності граматики.

Стандартною реакцією синтаксичного аналізатора на помилку є видача повідомлення ("синтаксична помилка") і припинення розбору. Цю реакцію можна дещо змінити, наприклад, зробивши повідомлення про помилку дещо більш інформативним, задавши власну функцію yyerror(char*). Однак найбільш важливе завдання полягає в тому, щоб у цьому випадку змусити аналізатор продовжувати аналіз вхідного потоку, зокрема для виявлення інших помилок. У YACC використовується механізм відновлення, який базується на читанні і відкиданні деякої кількості вхідних лексем. Для цього користувачу слід задати додаткові продукції граматики, які вказуватимуть, в яких конструкціях синтаксичні помилки є допустимими (щодо можливості відновлення). Одночасно ці правила визначають шлях подальшого розбору для помилкових ситуацій. Для вказівки точок допустимих помилок використовується зарезервоване з цією метою ім'я лексеми **error**.

Наприклад:

```
list: a b c ;    /* (1) */

list: a b error; /* (2) */

c: d1 d2 d3;     /* (3) */
```

Друге правило вказує шлях розбору у випадку, якщо при розпізнаванні нетермінала list зустрінеться помилка після проаналізованих символів a і b. YACC обробляє правила, що містять лексему error, так само, як і всі інші правила. У результаті для деяких станів побудованого аналізатора передбачено дію для лексеми error (відмінну від дії error). Кажемо, що у цих станах лексема error є допустимою. Розглянемо порядок роботи аналізатора при появі у вхідному потоці помилкової лексеми (тобто лексеми, аналіз якої в даному стані викликає дію error). У першу чергу, фіксується стан помилки та викликається функція yyerror для видачі повідомлення. Шляхом зворотного перегляду пройдених станів, починаючи з даного, робиться спроба визначити стан, в якому є допустимою лексема error. Відсутність такого стану означає неможливість відновлення, і розбір припиняється. Аналізатор повертається у знайдений стан (крім випадку, коли ним є безпосередньо той стан, в якому знайдено помилку). Виконується дія, задана в цьому стані для лексеми error. Наступною вхідною лексемою стає лексема, яка викликала помилку. Розбір триває, але аналізатор залишається в стані помилки доти, доки не будуть успішно прочитані і оброблені три наступні послідовні лексеми. На відміну від звичайного стану, якщо аналізатор перебуває у стані помилки, то повідомлення про помилку не видається, а сама проаналізована лексема ігнорується.

Після обробки трьох допустимих лексем вважається, що відновлення відбулося, і аналізатор виходить зі стану помилки. Отже, синтаксичний аналізатор, проаналізувавши помилку, намагається знайти найближчу точку у вхідному потоці, де дозволеною є лексема error. При цьому спочатку робиться спроба повернення в рамках правила, по якому йшов розбір у момент появи помилкової лексеми, а потім пошук поширюється на правила все більш високого рівня. У прикладі, наведеному на початку розділу, аналіз недопустимої лексеми після того, як було, наприклад, проаналізовано рядок a b d1 d2 викличе повернення до стану, що характеризується конфігураціями:

```
list: a b c;

list: a b error;
```

і продовження розору за правилом (2).

Часто правила, що враховують можливість помилки, задають на рівні основних структурних одиниць вхідного тексту. Наприклад, для пропуску в тексті помилкових операторів може бути використано правило

```
оператор: error;
```

При цьому відновлення зі стану помилки відбудеться після знаходження трьох лексем, які можуть слідувати після оператора, наприклад, для розпізнавання нового оператора. Якщо точно розпізнати початок оператора неможливо, то вихід з помилкового стану може відбутися передчасно, а обробка нового оператора почнеться з середини помилкового, що, ймовірно, призведе до повторного повідомленням про помилку (насправді неіснуючу). З огляду на це, більш надійного результату слід очікувати від правил виду:

```
оператор: error ';'
```

Тут відновлення відбудеться тільки після знаходження символу ";" і двох початкових лексем наступного оператора. Усі лексеми після знайденої помилкової і до ";" будуть відкинуті. З правилами, що включають лексему error, можуть бути пов'язані дії. З їх допомогою користувач може самостійно обробити помилкову ситуацію. Крім звичайних операторів, тут можна використовувати спеціальні оператори yyerrok і yyclearin, які YACC на макрорівні розширює до потрібних послідовностей. Оператор yyerrok анулює стан помилки. Таким чином, можна скасувати дію принципу "трьох лексем". Це допомагає запобігти маскуванню нових помилок у випадку, коли кінець помилкової конструкції розпізнано самим користувачем або однозначно визначається у правилі за меншою кількістю лексем.

```
list  : list ',' ID { yyerrok; }
      | ID
      | error
```

Під наступною лексемою розуміємо лексему, яка буде проаналізована програмою на наступному кроці. При виникненні помилки наступна лексема - це лексема, в якій аналізатор виявив помилку. Якщо дія для відновлення після помилки визначає точку, з якої обробку буде відновлено, то її код має видаляти наступну лексему. Для видалення наступної лексеми використовують оператор yyclearin, якщо пошук потрібної точки для відновлення вводу вказано у заданій користувачем дії. Наведемо загальну форму правила з відновлюваною дією:

```
оператор : error { resynch(); yyclearin; yyerrok; }
```

Передбачається, що процедура resynch() переглядає вхідний потік до початку чергового оператора. Помилкова лексема, що зберігається аналізатором як вхідна лексема, видаляється, після чого відбувається вихід зі стану помилки.

При побудові аналізаторів, що працюють в інтерактивному режимі, для обробки помилок рекомендують використовувати правила виду:

```
input : error '\n'
        {
            printf("Введіть останній рядок ще раз:\n");
        }
            input
            {
               $$ = $4;
            }
            ;
```

Значенням нетермінала після згортки тут стає значення повторно введеного рядка. Однак в цьому варіанті аналізатор залишається у стані помилки впродовж ще трьох лексем. Якщо в одній з трьох перших лексем виправленого рядка міститься помилка, то аналізатор пропустить ці лексеми, не видавши повідомлення про помилку. Для того, щоб врахувати подібну ситуацію, використовують оператор yyerrok, який переводить аналізатор у звичайний стан. Процедура виправлення помилок з цим оператором виглядає так:

```
input : error '\n'
        {
          yyerrok;
          printf("Введіть останній рядок ще раз:\n");
        }
            input
            {
                  $$ = $4;
            }
            ;
```

Для більш інформативних повідомлень можна використовувати номер рядка та номер лексеми, яка викликала помилку, наприклад:

```
extern int lineno, pos;

void yyerror (char *s)
{
    fprintf(stderr, "%s: лексема %d в рядку %d\n", s, pos, lineno);
}
```

Можна також скористатись інструкціями наступного типу:

```
input : '{' list '}'

      | '{' list error { errno = -1; }

      | '{' error '}'

      ...

yyerror(s)

{ if (errno == -1) printf("Опущено праву фігурну дужку\n"); else ... }
```

Синтаксичний аналізатор, описаний наступними специфікаціями, розпізнає послідовність цілих чисел.

Вміст файлу lex-специфікації:

```
%{
    #include "y.tab.h"
%}

%option noyywrap

%%

[0-9]+  { sscanf(yytext, "%d", &yylval); return INTEGER; }

\n      return NEWLINE;

.       return yytext[0];
```

Вміст файлу yacc-специфікації:

```
%{
    #include <stdio.h>
    void yyerror(char*);
    int yylex(void);
%}

%token INTEGER NEWLINE

%%

lines   :
        | lines NEWLINE
        | lines line NEWLINE    { printf("Введено ціле число %d\n", $2); }
        | error NEWLINE         { yyerror("Введіть число ще раз"); yyerrok; }
        ;
line    : INTEGER               { $$ = $1; }
        ;

%%

void yyerror(char *s)
{
    printf("%s: це не є цілим числом\n", s);
}

int main()
{
    printf("Введіть послідовність цілих чисел, будь ласка:\n");
    return yyparse();
}
```

Приклад роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

```
Введіть послідовність цілих чисел, будь ласка:
43124
Введено ціле число 43124
00123
Введено ціле число 123
mam
syntax error: це не є цілим числом
Введіть число ще раз: це не є цілим числом
32
Введено ціле число 32
...
```

Наступний синтаксичний аналізатор обчислює максимальну глибину вкладеності збалансованих круглих дужок. Наприклад, максимальна глибина вкладеності послідовності (((()))()) дорівнює 3. Пропуски та символи табуляції аналізатор ігнорує, а при аналізі інших символів та незбалансованих дужок видає повідомлення про помилку. Для більш інформативних повідомлень використано номер позиції лексеми, яка викликала помилку.

Вміст файлу lex-специфікації:

```
%{
   #include "y.tab.h"
   extern int pos;
%}

%option noyywrap

%%

[ \t]     { /* ігнорування пропусків та символів табуляції */ pos++;}

\n        { return yytext[0]; }

\(        { pos++; return LEFT; }

\)        { pos++; return RIGHT; }

.         { pos++; return yytext[0]; }
```

Вміст файлу yacc-специфікації:

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
    int pos = 0;

%}

%token LEFT RIGHT

%%

P  : S  '\n'  { $$ = $1; printf("Дужки збалансовані.\nГлибина вкладеності дужок:
                              %d\n", $$); return 0; }
   ;
S  :                  { $$ = 0; }
   | S LEFT S RIGHT    { $$ = $3 + 1; if ($$ < $1) $$ = $1; }
   ;

%%

int main()
{
    printf("Введіть послідовність із вкладених одних в інші збалансованих дужок,
                  будь ласка:\n");
    yyparse();
}

void yyerror(char *s)
{
  printf("Синтаксична помилка: на позиції %d дані введено не коректно.\n", pos);
}
```

Приклад роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

Введіть послідовність із вкладених одних в інші збалансованих дужок, будь ласка:
***((()) ()) (((()))) ***
Дужки збалансовані.
Глибина вкладеності дужок: 4

Введіть послідовність із вкладених одних в інші збалансованих дужок, будь ласка:
((()) () (
Синтаксична помилка: на позиції 9 дані введено не коректно.

Синтаксичний аналізатор, заданий наступними специфікаціями, обчислює максимальну ширину вкладеності послідовності збалансованих круглих, фігурних та квадратних дужок, а також кількість пар дужок кожного типу. Наприклад, максимальна ширина вкладеності послідовності ((()()))() дорівнює 2. Всі символи, крім дужок, аналізатор ігнорує, а при аналізі незбалансованих дужок видає повідомлення про помилку.

Вміст файлу lex-специфікації:

```
%{
    #include "y.tab.h"
     extern int pos;
%}

%option noyywrap

%%

[ \t]                    { /* ігнорування пропусків та символів табуляції */ }

\n                       { return yytext[0]; }

"("|")"|"{"|"}"|"["|"]"    { pos++; return yytext[0]; }

.                        { pos++;}
```

Вміст файлу yacc-специфікації:

```
%{
    #include <stdio.h>
    void yyerror(char*);
    int yylex(void);
    int width = 0, n1 = 0, n2 = 0, n3 = 0, pos = 0;
%}

%%

P   :  S '\n'   { printf("Круглі, фігурні та квадратні дужки збалансовані.\n"
                      "Ширина вкладеності всіх дужок: %d\n", width); return 0; }
    ;
S   : '(' S ')' S      { $$ = $4 + 1; if (width < $$) width = $$; n1++; }
    | '{' S '}' S      { $$ = $4 + 1; if (width < $$) width = $$; n2++; }
    | '[' S ']' S      { $$ = $4 + 1; if (width < $$) width = $$; n3++; }
```

```
    |                   { $$ = 0; }
    ;

%%

int main()
{
    printf("Введіть послідовність із вкладених одних в інші збалансованих дужок,
                  будь ласка:\n");
    yyparse();
    printf("Кількість пар круглих дужок: %d\n", n1);
    printf("Кількість пар фігурних дужок: %d\n", n2);
    printf("Кількість пар квадратних дужок: %d\n", n3);
}

void yyerror(char *s)
{
   printf("Синтаксична помилка: на позиції %d дані введено не коректно\n", pos);
};
```

Приклад роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

```
Введіть послідовність із вкладених одних в інші збалансованих дужок, будь ласка:
(() ((())) (()) ()
Круглі, фігурні та квадратні дужки збалансовані.
Ширина вкладеності всіх дужок: 3
Кількість пар круглих дужок: 7
Кількість пар фігурних дужок: 0
Кількість пар квадратних дужок: 0

Введіть послідовність із вкладених одних в інші збалансованих дужок, будь ласка:
if ((x < 2020) and (x > 100)) { a[i] = 5; } else { a[i] = a[0]; }
Круглі, фігурні та квадратні дужки збалансовані.
Ширина вкладеності всіх дужок: 3
Кількість пар круглих дужок: 3
Кількість пар фігурних дужок: 2
Кількість пар квадратних дужок: 3
```

Синтаксичний аналізатор можна будувати також і використовуючи мову програмування C++ у lex- та yacc-специфікаціях. Для цього потрібно скомпілювати файли lex.yy.c та y.tab.c компілятором мови C++, виконавши команду c++ lex.yy.c y.tab.c

Наступні специфікації, які описують калькулятор, який обчислює арифметичні вирази складені з цілих чисел та бінарних операцій '+' і '-' у m-ковій системі числення (m ∈ {2,...,10}), використовують мову C++:

Вміст файлу m-calc.l++ (або m-calc.lpp) lex-специфікації:

```
%{
    #include <iostream>
    using namespace std;
    #include "y.tab.h"
    //#include "m-calc.tab.h++"
```

```
    int m;

%}

%option noyywrap

/* %option c++     //ця опція для генерування C++-файла lex.yy.cc */

%%

[0-9]+      { int suma = 0, prod = 1, digit;

                  for (int i = yyleng - 1; i > -1; i--)
                  {
                        digit = yytext[i] - '0';
                        if (digit > m-1)
                            { cout << "Введено помилкову цифру.\n"; return 0; }
                        suma += prod * digit;
                        prod *= m;
                  }

                  yylval = suma;

                  return NUMBER;
            }

[ \t]       ;

\n          return 0;

.           return yytext[0];
```

Вміст файлу m-calc.y++ (або m-calc.ypp) yacc-специфікації:

```
#include <iostream>
using namespace std;

int yyerror(string);
int yylex(void);
extern int m;

void mNumberSystem(int num)
{
    if (num < 0) { num = -num; cout << "-"; }

    int r[20], i = 0;

    while (num != 0)
    {
        r[i] = num % m;
        num = (num - r[i]) / m;
        i++;
    }
```

```
    for (int j = i - 1; j > -1; j--) cout << r[j];
}


%}

%token  NUMBER

%%

statement    :  expression      { cout << "Значення виразу дорівнює ";
                                  mNumberSystem($$); cout << "\n"; }
             ;

expression   :    expression '+' NUMBER   { $$ = $1 + $3; }
             |    expression '-' NUMBER   { $$ = $1 - $3; }
             |    NUMBER                  { $$ = $1; }
             ;

%%

int main(void)
{
      cout << "Введіть основу m ∈ {2,...,10} системи числення ";
      cout << "і арифметичний вираз,\nскладений з цілих чисел ";
      cout << "у m-ковій системі числення і операцій '+' та '-':\nm=";

      cin >> m;

      return yyparse();
}

int yyerror(string s)
{
    cout << "Синтаксична помилка.\n";
}
```

Приклади роботи аналізатора (жирним шрифтом виділено вхідні дані користувача):

Введіть основу m ∈ {2,...,10} системи числення і арифметичний вираз,
складений з цілих чисел у m-ковій системі числення і операцій '+' та '-':
*m=**3   102111110+102222-101001***
Значення виразу дорівнює 102120101

Введіть основу m ∈ {2,...,10} системи числення і арифметичний вираз,
складений з цілих чисел у m-ковій системі числення і операцій '+' та '-':
*m=**7   16666666+10234322-101+45302***
Значення виразу дорівнює 30312522

Введіть основу m ∈ {2,...,10} системи числення і арифметичний вираз,
складений з цілих чисел у m-ковій системі числення і операцій '+' та '-':
*m=**4 1230+1042+112***
Введено помилкову цифру.
Синтаксична помилка.

Завдання для самостійної роботи до параграфа 4

1. Розробити синтаксичний аналізатор, який реалізує контекстно вільну граматику, що породжує формальну мову:

a) $\{a^n b^{3n} : n \in \mathbb{N}\}$;
b) $\{a^n b^m : n,m \in \mathbb{N}_0,\ n+m \in \mathbb{N}\}$;
c) $\{a^n b^m c^n : n,m \in \mathbb{N}_0,\ n+m \in \mathbb{N}\}$;
d) $\{a^n b^m c^k : n,m,k \in \mathbb{N}_0,\ n+m+k \in \mathbb{N}\}$;
e) $\{a^n b^m c^k : m,k \in \mathbb{N}_0,\ n=m+k \in \mathbb{N}\}$;
f) $\{a^n b^m c^k : m,k \in \mathbb{N}_0,\ 2n=2m+k \in \mathbb{N}\}$;
g) $\{a^n b^m c^k : n,m \in \mathbb{N}_0,\ 6k=2n+3m \in \mathbb{N}\}$;
h) $\{a^n \omega : \omega \in \{b,c\}^n,\ n \in \mathbb{N}\}$.

2. Розробити калькулятор, який над цілими числами виконує арифметичні операції "*", "+", "-", знаходить цілу частину "/" та остачу від ділення "%" у m-ковій системі числення з цифрами 0, 1,..., m-1, де $m \in \{2,...,10\}$.

3. Розробити калькулятор, який над цілими числами виконує арифметичні операції "*", "+", "-", знаходить цілу частину "/" та остачу від ділення "%" у шіснадцятковій системі числення з цифрами 0, 1,..., 9, a, b, c, d, e, f.

4. Розробити калькулятор, який над цілими числами виконує арифметичні операції "*", "+", "-", знаходить цілу частину "/" та остачу від ділення "%" у m-ковій системі числення, $m \in \{11,...,36\}$. Цифрами цієї системи є десяткові цифри та m-10 перших букв англійського алфавіту. Для 16-кової системи числення див. попередній приклад.

5. Розробити калькулятор, який над цілими числами виконує арифметичні операції "*", "+", "-", знаходить цілу частину "/" та остачу від ділення "%" у симетричній m-ковій системі числення з цифрами 0, 1, -1,..., (m-1)/2, -(m-1)/2, де $m \in \{3,5,7,...,19\}$.

6. Розробити калькулятор, який виконує над булевими константами 0 (false), 1 (true) наступні логічні операції (у порядку спадання пріоритету): заперечення (not), кон’юнкція (and), диз’юнкція (or), альтернативне ‘або’ (xor), імплікація (imp), еквіваленція (equ).

7. Розробити калькулятор, який виконує бінарні операції додавання, віднімання, множення та ділення над дійсними двійковими числами.

§ 5. Синтаксичний аналізатор нескладної мови програмування

Розробимо лексичний та синтаксичний аналізатори простої мови програмування, в якій представлено всього два типи змінних: цілий та булевий. Значення змінних цілих типів завжди є обмеженими, і цей факт відображається в їх описі:

VarName: integer[min..max] := IntValue;

Наприклад, опис змінної suma, значення якої належить відрізку [-10,2020] ∩ ℤ і дорівнює 5, має вигляд:

suma: integer[-10..2020] := 5;

Всі елементи цього запису є обов’язковими. Якщо в процесі роботи програми значення змінної виходить за межі вказаного проміжку, то має видаватися повідомлення про помилку.

Булеві змінні можуть набувати двох значень: true та false. Опис булевої змінної має простіший вигляд:

VarName: boolean := BoolValue;

Структуру програми оформимо у стилі мови програмування Паскаль:

program <назва програми>.

declare

// опис змінних

body

// тіло програми

end.

Однорядкові коментарі мають починатися з послідовності '//' і закінчуватися символом нового рядка '\n', а багаторядкові коментарі міститися між символами '/*' та '*/'. Пропуски та символи табуляції мають бути проігноровані. Синтаксичний аналізатор не має бути чутливим до регістру букв у назвах ключових слів, операцій та операторів оголошень типів змінних. У розробленій мові слід визначити всього лише три керуючі конструкції: присвоєння, розгалуження та безумовний перехід.

Наступні три різні форми присвоєння можуть бути використані:

1) просте присвоєння: put a := b; (для будь-яких змінних);

2) присвоєння з операцією not: put a := not b; (тільки для булевих змінних);

3) присвоєння з бінарною операцією: put a := b operation c;

В останньому пункті operation позначає операції '+', '-' та '*' для змінних цілого типу, та 'and' (кон’юнкція), 'or' (диз’юнкція), 'xor' (виключне або), 'imp' (імплікація) та 'equ' (еквіваленція) - для булевого.

Розгалуження задається звичною конструкцією if...goto... з необов’язковою частиною else goto... :

if a comporer b goto mitka1; [else goto mitka2;]

Замість comporer можна використовувати операції порівняння '=', '<>', '<', '>', '<=', '>='. Для булевих значень допустимими є лише операції '=' та '<>'.

Безумовний перехід має простий синтаксис: goto mitka; Мітку визначаємо в if- та goto-конструкціях, а її назву можна ставити перед будь-якою конструкцією, крім ключового слова end.

Якщо лексичний аналізатор просканує недопустимий символ, то має видаватися повідомлення про помилку із вказанням номера рядка та позиції першого недопустимого символу.

Вміст файлу mylanguage.l lex-специфікації:

```
%{
     #include "y.tab.h"
     extern int lineno;
     int pos = 0;
%}

letter  [A-Za-z]
digit   [0-9]
start_comment  "/*"

%option noyywrap
%x comment0 comment1

%%

-?{digit}{digit}*  { printf("Просканована число %s.\n", yytext); pos += yyleng;
                                  return NUMBER; }
=|"<>"|<|>|"<="|">="    { printf("Просканована операцію порівняння '%s'.\n",
                               yytext); pos += yyleng; return COMPOPER; }
":="             { printf("Просканована операцію присвоєння '%s'.\n", yytext);
                               pos += yyleng; return ASS; }
[-+*.:;]        { printf("Просканована знак '%s'.\n", yytext); pos += yyleng;
                                  return yytext[0]; }
\[|\]            { printf("Просканована дужку '%s'.\n", yytext); pos += yyleng;
                               return yytext[0]; }
".."            { printf("Просканована '..'.\n"); pos += yyleng; return DDOT; }

[Ii][Nn][Tt][Ee][Gg][Ee][Rr]    { printf("Просканований оператор оголошення
                               цілого типу.\n"); pos += yyleng; return INTEGER; }
[Bb][Oo][Oo][Ll][Ee][Aa][Nn]  { printf("Просканований оператор оголошення
                          булевого типу.\n"); pos += yyleng; return BOOLEAN; }
[Pp][Rr][Oo][Gg][Rr][Aa][Mm]   { printf("\nПросканована ключове слово '%s'.\n",
                                  yytext); pos += yyleng; return PROGRAM; }
```

```
[Dd][Ee][Cc][Ll][Aa][Rr][Ee]   { printf("Просканована ключове слово '%s'.\n",
                                 yytext); pos += yyleng; return DECLARE; }
[Bb][Oo][Dd][Yy]      { printf("Просканована ключове слово '%s'.\n",
                        yytext); pos += yyleng; return BODY; }
[Ee][Nn][Dd]           { printf("Просканована ключове слово '%s'.\n", yytext);
                                 pos += yyleng; return END; }
[Pp][Uu][Tt]           { printf("Просканована ключове слово '%s'.\n", yytext);
                                 pos += yyleng; return PUT; }
[Ii][Ff]               { printf("Просканована ключове слово '%s'.\n", yytext);
                                 pos += yyleng; return IF; }
[Ee][Ll][Ss][Ee]       { printf("Просканована ключове слово '%s'.\n", yytext);
                                 pos += yyleng; return ELSE; }
[Gg][Oo][Tt][Oo]       { printf("Просканована ключове слово '%s'.\n", yytext);
                                 pos += yyleng; return GOTO; }
[Aa][Nn][Dd]           { printf("Просканована булеву операцію '%s'.\n", yytext);
                                 pos += yyleng; return AND; }
[Oo][Rr]               { printf("Просканована булеву операцію '%s'.\n", yytext);
                                 pos += yyleng; return OR; }
[Xx][Oo][Rr]           { printf("Просканована булеву операцію '%s'.\n", yytext);
                                 pos += yyleng; return XOR; }
[Ii][Mm][Pp]           { printf("Просканована булеву операцію '%s'.\n", yytext);
                                 pos += yyleng; return IMP; }
[Ee][Qq][Uu]           { printf("Просканована булеву операцію '%s'.\n", yytext);
                                 pos += yyleng; return EQU; }
[Nn][Oo][Tt]           { printf("Просканована булеву операцію '%s'.\n", yytext);
                                 pos += yyleng; return NOT; }
[Tt][Rr][Uu][Ee]       { printf("Просканована булеву константу '%s'.\n", yytext);
                                 pos += yyleng; return T; }
[Ff][Aa][Ll][Ss][Ee] { printf("Просканована булеву константу '%s'.\n", yytext);
                                 pos += yyleng; return F; }
{letter}({letter}|{digit})*      { printf("Просканована ідентифікатор '%s'.\n",
                                   yytext); pos += yyleng; return ID; }

<INITIAL>{start_comment}   { pos += 2; BEGIN comment0; }
<comment0>"*"              { pos++; BEGIN comment1; }
<comment0>[^*]      { pos++; if (yytext[0] == '\n') {pos = 0; lineno += 1;} }
<comment1>"*"       { pos++; }
<comment1>"/"       { pos++; printf("Знайдено багаторядковий коментар.\n");
                                                    BEGIN INITIAL; }
<comment1>[^*/]     { pos++; BEGIN comment0; }
"//".*\n { pos = 0; lineno += 1; printf ("Знайдено однорядковий коментар.\n"); }
[ \t\r]             { pos += yyleng;}//{ printf("Проігноровано пропуск.\n"); }
\n             { lineno += 1; pos = 0; printf("Досягнуто кінець рядка.\n");  }
.          { printf("\nЗнайдено невідомий символ. '%s'.\n", yytext);
           printf("Лексична помилка: на позиції %d рядка %d дані введено не
                           коректно\n", pos+1, lineno);  return 0; }
```

Вміст файлу mylanguage.y yacc-специфікації:

```
%{
    #include <stdio.h>
    void yyerror(char *);
    int yylex(void);
    int lineno;
%}
```

```
%token PROGRAM DECLARE BODY END PUT IF ELSE GOTO DDOT ID NUMBER
%token INTEGER T F BOOLEAN AND OR XOR IMP EQU NOT COMPOPER ASS

%%

mylanguage    :   myprogram { printf("\nВведений текст задовольняє синтаксичній
                                        структурі мови.\n\n"); return 0; }
              ;

myprogram       :   PROGRAM ID '.' programbody
                ;

programbody     :   DECLARE declarevar BODY statement END '.'
                |   BODY statement END '.'
                ;

declarevar      :   declarevar ID ':' declarevarbody
                |   ID ':' declarevarbody
                ;

declarevarbody  :   INTEGER '[' NUMBER DDOT NUMBER ']' ASS NUMBER ';'
                |   BOOLEAN ASS T ';'
                |   BOOLEAN ASS F ';'
                ;

statement       :   statement ID ':' statementbody
                |   ID ':' statementbody
                |   statement statementbody
                |   statementbody
                ;

statementbody   :   assignment
                |   branching
                |   GOTO ID ';'
                ;

assignment      :   PUT ID ASS value operation value ';'
                |   PUT ID ASS value ';'
                |   PUT ID ASS NOT value ';'
                ;

operation       :   '+'
                |   '-'
                |   '*'
                |   AND
                |   OR
                |   XOR
                |   IMP
                |   EQU
                ;

value           :   ID
                |   NUMBER
                |   T
                |   F
                ;
```

```
branching       :   IF compareexpr GOTO ID ';'
                |   IF compareexpr GOTO ID ';' ELSE GOTO ID ';'
                ;

compareexpr     :   value COMPOPER value
                ;

%%

int main (void)
{
    lineno = 1;
    yyparse();
    return 0;
}

void yyerror (char *s)
{
      printf ("\nЗнайдено синтаксичну помилку в рядку %d!
                                    \nСкорегуйте вхідний текст!\n", lineno);
}
```

Нехай текстовий файл cod.txt містить наступний текст:

```
PROGRAM CountSum.

Declare suma:  integer[-6..225] := 0; // сума
counter: integer[-3..21] := -3; // лічильник

/* багаторядковий
            коментар */

Body

loop:   PUT suma := suma + counter;
        Put counter := counter + 1;
        If counter <= 25 goto loop;
END.
```

Виконавши команду ./a.out < cod.txt, отримаємо:

```
Просканово ключове слово 'PROGRAM'.
Просканово ідентифікатор 'CountSum'.
Просканово знак '.'.
Досягнуто кінець рядка.
Досягнуто кінець рядка.
Просканово ключове слово 'Declare'.
Просканово ідентифікатор 'suma'.
Просканово знак ':'.
Просканово оператор оголошення цілого типу.
Просканово дужку '['.
Просканово число 0.
Просканово '..'.
```

```
Просканировано число 200.
Просканировано дужку ']'.
Просканировано операцію присвоєння ':='.
Просканировано число 0.
Просканировано знак ';'.
Знайдено однорядковий коментар.
Просканировано ідентифікатор 'counter'.
Просканировано знак ':'.
Просканировано оператор оголошення цілого типу.
Просканировано дужку '['.
Просканировано число -10.
Просканировано '..'.
Просканировано число 25.
Просканировано дужку ']'.
Просканировано операцію присвоєння ':='.
Просканировано число -10.
Просканировано знак ';'.
Знайдено однорядковий коментар.
Досягнуто кінець рядка.
Знайдено багаторядковий коментар.
Досягнуто кінець рядка.
Досягнуто кінець рядка.
Просканировано ключове слово 'Body'.
Досягнуто кінець рядка.
Досягнуто кінець рядка.
Просканировано ідентифікатор 'loop'.
Просканировано знак ':'.
Просканировано ключове слово 'PUT'.
Просканировано ідентифікатор 'suma'.
Просканировано операцію присвоєння ':='.
Просканировано ідентифікатор 'suma'.
Просканировано знак '+'.
Просканировано ідентифікатор 'counter'.
Просканировано знак ';'.
Досягнуто кінець рядка.
Просканировано ключове слово 'Put'.
Просканировано ідентифікатор 'counter'.
Просканировано операцію присвоєння ':='.
Просканировано ідентифікатор 'counter'.
Просканировано знак '+'.
Просканировано число 1.
Просканировано знак ';'.
Досягнуто кінець рядка.
Просканировано ключове слово 'If'.
Просканировано ідентифікатор 'counter'.
Просканировано операцію порівняння '<='.
Просканировано число 25.
Просканировано ключове слово 'goto'.
Просканировано ідентифікатор 'loop'.
Просканировано знак ';'.
Досягнуто кінець рядка.
Просканировано ключове слово 'END'.
Просканировано знак '.'.

Введений текст задовольняє синтаксичній структурі мови.
```

Завдання для самостійної роботи до параграфа 5

Розробити та реалізувати лексичний та синтаксичний аналізатори для нескладної мови програмування, використовуючи генератори LEX (FLEX) та YACC (BISON). Лексичний аналізатор має розпізнавати ключові слова, унарні і бінарні операції та оператори оголошення типів змінних, видаляти пропуски і коментарі та виводити повідомлення про наявність у вхідному тексті помилок, які можуть виникати на етапі лексичного та синтаксичного аналізів (у повідомленні мають бути вказані номери рядків та позиції помилкових символів); а також підраховувати кількість лексем кожного типу і загальну кількість лексем у вхідному тексті. Коментарі можуть бути двох видів: багаторядкові, включені між символами "/*" і "*/"; та однорядкові, які починаються з символів "//" і закінчуються символом нового рядка “\n”.

Структуру програми розробленої мови програмування слід оформити у стилі мови програмування Паскаль:

program <назва програми>.

description

// оголошення та опис змінних

body

// тіло програми

end.

Варіанти завдань

1. Тіло програми містить один або декілька операторів циклу "cycle(id := number; id comproper number; id := id oper number) do <тіло циклу>". Тіло циклу містить оператори присвоєння значень змінним типу "put id := b [oper c];" та розгалуження "if id comproper number then put id := b [oper c];", де b та c — цілі числа або ідентифікатори. У мові слід оголосити змінні цілого типу та операції oper ∈ {'+', '-', '*'}, comproper ∈ {'<', '>', '='}. Ідентифікатором є слово з малих англійських літер, яке містить не більше 4 літер.

2. Тіло програми містить один або декілька операторів циклу "let id = number: while(id comproper number) <тіло циклу> done;’. Тіло циклу містить оператори присвоєння значень змінним типу "let id = b [oper c];" та розгалуження "if id comproper number then let id = b; else let id = b [oper c];", де b та c — додатні дійсні числа або ідентифікатори. У мові слід оголосити змінні додатнього дійсного типу та операції oper ∈ {'+', '-', '*'}, comproper ∈ {'<', '>', '==', '<=', '>='}. Ідентифікатором є слово з англійських літер, яке починається з літер a, b або c і остання буква якого збігається з першою.

3. Тіло програми містить один або декілька операторів присвоєння значень булевим змінним типу "put id := [not] b [booloper c];" та розгалуження "if (id comporper boolconst) then { put id := [not] b [booloper c]; }", де b та c — булеві константи "0", "1", "true", "false" або ідентифікатори. У мові слід оголосити змінні булевого типу та операції booloper ∈ {'or', 'xor', 'and'}, comporper ∈ {'=', '!='}. Ідентифікатором є слово з великих англійських літер та цифр 2, 3, 4, яке починається літерою і закінчується цифрою.

4. Тіло програми містить один або декілька операторів присвоєння значень змінним типу "let id = b [oper c [oper d]];" та розгалуження "if id comporper number { let id = b [oper c [oper d]];} [else { let id = b [oper c];}]", де b, c та d — дійсні числа з плаваючою крапкою (у звичайній і експоненціальній формах) або ідентифікатори. У мові слід оголосити змінні дійсного типу та операції oper ∈ {'+', '-', '*', '/'}, comporper ∈ {'<', '>', '==', '<=', '>='}. Ідентифікатором є слово з малих англійських літер, яке містить непарну кількість літер.

5. Тіло програми містить один або декілька операторів присвоєння значень змінним типу "put id := b [oper c [oper d]];" та розгалуження "if (id comporper number) put id := b [oper c [oper d]]; [else put id := b [oper c];]", де b, c та d — вісімкові натуральні числа або ідентифікатори. У мові слід оголосити змінні вісімкового натурального типу та операції oper ∈ {'+', '*'}, comporper ∈ {'<', '>', '=', '<=', '>='}. Ідентифікатором є слово, яке починається з великої англійської літери, а далі містить непарну кількість десяткових цифр.

6. Тіло програми містить один або декілька операторів циклу "let id := number, run <тіло циклу> while(id comporper number);". Тіло циклу містить оператори присвоєння значень змінним типу "let id := b [oper c];" та розгалуження "if (id comporper number) let id := b [oper c]; else let id := b [oper c];", де b та c — римські числа або ідентифікатори. Римськими вважати числа, записані великими літерами 'X', 'V', 'I'. У мові слід оголосити змінні римського типу (з обмеженим діапазоном значень) та операції oper ∈ {'+', '-'}, comporper ∈ {'<', '>', '==', '<>'}. Ідентифікатором є слово з малих англійських літер і цифр 0, 1, яке починається буквою і містить непарну кількість символів.

7. Тіло програми містить один або декілька операторів присвоєння значень змінним типу "put id = b [oper c [oper d]];" та розгалуження "if (id comporper number) { put id := b [oper c]; } else { put id := b [oper c [oper d]]; }", де b, c та d — римські числа або ідентифікатори. Римськими вважати числа, записані великими літерами 'X', 'V', 'I', 'L', 'C', 'D', 'M'. У мові слід оголосити змінні римського типу (з обмеженим діапазоном значень) та операції oper ∈ {'+', '-'}, comporper ∈ {'<', '>', '==', '<>'}. Ідентифікатором є слово з малих англійських літер і цифр 5, 6, 8, яке починається з букв p, q або r і містить парну кількість символів.

8. Тіло програми містить один або декілька операторів присвоєння значень булевим змінним типу "let id = [not] b [booloper c [booloper d]];" та розгалуження "if (id comporper boolconst) then { let id := [not] b [booloper c]; }", де b, c та d — булеві константи "0", "1", "true", "false" або ідентифікатори. У мові слід оголосити змінні булевого типу та операції booloper ∈ {'or', 'xor', 'and', 'imp', 'equ'}, comporper ∈ {'=', '<>'}. Ідентифікатором є слово з великих англійських літер, яке починається і закінчується літерами S, T або R.

9. Тіло програми містить один або декілька операторів присвоєння значень змінним типу "put id := b [oper c [oper d]];" та розгалуження "if (id comporper number) { put id := b [oper c [oper d]]; }; [else { put id := b [oper c]]; };", де b, c та d – десяткові числа з плаваючою крапкою в експоненціальній формі або ідентифікатори. У мові слід оголосити дійсні змінні та операції oper ∈ {'+', '-' '*', '/'}, comporper ∈ {'<', '>', '=', '!='}. Ідентифікатором є слово з великих англійських літер і двійкових цифр, яке починається буквою і містить непарну кількість символів.

10. Тіло програми містить один або декілька операторів циклу "cycle (id := number; id comporper number; id := id oper number) { <тіло циклу> };". Тіло циклу містить оператори присвоєння значень змінним типу "put id := b [oper c];" та розгалуження "if id comporper number then put id := b [oper c]; else put b [oper c [oper d]];", де b, c та d – двійкові цілі числа або ідентифікатори. У мові слід оголосити змінні двійкового цілого типу та операції oper ∈ {'+', '-', '*'}, comporper ∈ {'<', '>', '='}. Ідентифікатором є слово з великих англійських літер, яке містить не більше 4 літер.

11. Тіло програми містить один або декілька операторів циклу "let id = str1: while(id comporper str2) perform <тіло циклу>". Тіло циклу містить оператори присвоєння значень змінним типу "let id = b [+ c [+ d]];" та розгалуження "if id comporper str3 then { let id = b; } else { let id = b [+ c[+ d]]; }", де b, c та d – ідентифікатори або рядкові змінні та константи. У мові слід оголосити рядкові змінні і константи (послідовності символів у подвійних лапках), односимвольні змінні і константи (послідовності символів в одинарних лапках), операцію конкатенації (+) та операції порівняння '=', '<>'. Ідентифікатором є слово з великих англійських літер і цифр 3, 5, 7, яке починається буквою і містить парну кількість символів.

12. Тіло програми містить один або декілька операторів присвоєння значень булевим змінним типу "put id = [not] b [booloper c [booloper d]];" та розгалуження "if (id comporper b) { let id := b [booloper c [booloper d]]; } else { let id := b [booloper c]; }", де b, c та d – булеві константи "T", "F" або ідентифікатори. У мові слід оголосити змінні булевого типу та операції booloper ∈ {'xor', 'and'}, comporper ∈ {'=', '<>'}. Ідентифікатор складається з 7 англійських літер.

13. Тіло програми містить один або декілька операторів циклу "let id := number: operate <тіло циклу> while (id comporper number);". Тіло циклу містить оператори присвоєння значень змінним типу "let id := b [oper c];" та розгалуження "if (id comporper number) let id := b [oper c]; else let id := b [oper c];", де b та c – десяткові числа або ідентифікатори. У мові слід оголосити змінні дійсного типу та операції oper ∈ {'+', '-', '*', '/'}, comporper ∈ {'<=', '>=', '=', '<>'}. Ідентифікатором є слово довжини 4 з великих англійських літер, яке починається з літер B або C і остання буква якого не збігається з першою.

Список літератури

1. Бєлов Ю.А. Інструментальні засоби програмування: навчальний посібник / Ю.А. Бєлов, В.С. Проценко, П.Й. Чаленко. – К.: Либідь, 1993. – 248 с.

2. Гаврилків В.М. Регулярні вирази у програмних продуктах: навчальний посібник / В.М. Гаврилків. – Івано-Франківськ: Голіней, 2012. – 72 с.

3. Гаврилків В.М. Формальні мови та алгоритмічні моделі: навчальний посібник (вид. друге, доповн.) / В.М. Гаврилків. – Івано-Франківськ: Голіней, 2023. – 180 с.

4. Нікольський Ю.В. Дискретна математика / Ю.В. Нікольський, В.В. Пасічник, Ю.М. Щербина. – К.: Видавнича група BHV, 2007. – 368 с.

5. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: навчальний посібник у двох частинах / Т.М. Сопронюк. – Чернівці: ЧНУ, 2008. – 84 с.

6. Aho A.V. and Ullman J.D. The Theory of Parsing, Translation, and Compiling. Vol. 1. / A.V. Aho and J. D. Ullman. – Englewood Cliffs, N.J.: Prentice Hall, 1972. – 460 p.

7. Brown D. Lex and Yacc, 2nd Edition / Doug Brown, John Levine, Tony Mason. – O'Reilly Media, 2012. – 388 p.

8. Forta B. Regular Expressions in 10 minutes / B. Forta. – Sams Publishing, 2004. – 160 p.

9. Friedl J. Regular Expressions, 3nd Edition / Jeffrey E.F. Friedl. – O'Reilly Media, 2006. – 542 p.

10. Goyvaerts J. Regular Expressions Cookbook / Jan Goyvaerts and Steven Levithan. – O'Reilly Media, 2009. – 494 p.

11. Hunter R. The Design and Construction of Compilers / R. Hunter. – Chichester, New York: John Wiley, 1981. – 272 p.

12. Levine J.R. Flex and Bison / J.R. Levine. — O'Reilly Media, Inc., 2009. – 274 p.

13. Linz P. An Introduction to Formal Languages and Automata / P. Linz. — Peter Linz Univ. of California at Davis. Jones & Bartlett Learning, 2016. – 709 p.

14. Pettorossi A. Automata Theory and Formal Languages: fundamental notions, theorems, and Techniques / A. Pettorossi. — Springer, 2022. – 288 p.

15. Smyth B. Computing Patterns in Strings / Bill Smyth. – Addison-Wesley, 2003. – 440 p.

ПРАКТИЧНІ МЕТОДИ

РОЗРОБКИ КОМПІЛЯТОРІВ

Структуру
програми розробленої
мови програмування слід
оформити у стилі мови
програмування Паскаль:
program <назва програми>.
description
// оголошення та опис змінних
body
// тіло програми
end.