ЛАБОРАТОРНА РОБОТА №4.
РОЗРОБКА ЛЕКСИЧНОГО АНАЛІЗАТОРА (2 год)

Мета: оволодіти навиками побудови лексичних аналізаторів. Навчитись моделювати роботу скінченного автомату за допомогою програмних засобів.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Лексичний аналізатор (або сканер) – це частина транслятора, яка читає ланцюжки символів вхідної програми і виділяє з них окремі неподільні одиниці, які називають **лексемами**. На вхід лексичного аналізатора надходить текст вихідної програми, а вихідна інформація передається для подальшої обробки транслятором на етапі синтаксичного аналізу і розбору.

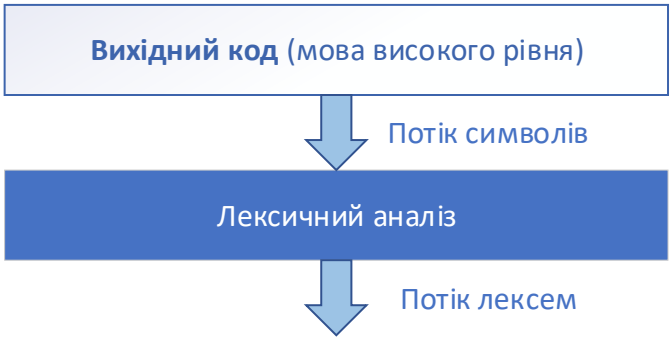

Рис. 4.1 Лексичний аналіз

З теоретичної точки зору лексичний аналізатор не є обов'язковою, необхідною частиною транслятора. Його функції можуть виконуватися на етапі синтаксичного розбору. Однак існує кілька причин, виходячи з яких до складу практично всіх трансляторів включають лексичний аналіз. Ці причини полягають в наступному:

- спрощується робота з текстом вхідної програми на етапі синтаксичного розбору, адже скорочується обсяг оброблюваної інформації, оскільки лексичний аналізатор структурує вхідний текст програми, перетворює його в послідовність лексем, і відкидає незначущу інформацію;
- для виділення і розбору лексем можливо застосовувати просту, ефективну і теоретично добре опрацьовану техніку аналізу, в той час як на етапі синтаксичного аналізу конструкцій вхідної мови використовуються досить складні алгоритми розбору;
- сканер відокремлює складний по конструкції синтаксичний аналізатор від роботи безпосередньо з текстом вхідної програми. При такій структурі спрощується створення транслятора для нових версії мови, інколи достатньо лише модифікувати відносно простий сканер.

- блок лексичного аналізу природним чином вписується в ієрархічну структуру транслятора, що теж важливо при системному підході до розробки трансляторів.

Основні функції лексичного аналізу:

- формування лексем;
- видалення коментарів та пробілів;
- формування інформаційних таблиць (таблиць ідентифікаторів, числових констант, рядкових констант, символьних констант);
- формування додаткової інформації для діагностування помилок.

Функції, які виконує лексичний аналізатор, і склад лексем, які він виділяє в тексті вхідної програми, можуть змінюватися в залежності від версії транслятора.

Поняття лексичних класів

Вхідне текстове представлення програми не є зручним для роботи транслятора, тому під час лексичного аналізу вхідна програма розбивається спершу на рядки, а в подальшому на лексичні класи. Лексеми потрапляють в один лексичний клас, якщо їх не потрібно розпізнавати з погляду синтаксичного аналізатора. Наприклад, під час синтаксичного аналізу всі ідентифікатори можна вважати однаковими.

Розміри лексичних класів різні. Наприклад, лексичний клас ідентифікаторів – нескінченний. З іншого боку, є лексичні класи, що складаються тільки з однієї лексеми, наприклад, лексема `if`. У більшості мов програмування використовуються такі лексичні класи:

- ключові слова (по одному на кожне ключове слово);
- знаки операцій (по одному на кожну операцію);
- ідентифікатори;
- рядкові літерали;
- числові константи;
- роздільники.

Кожному лексичному класу ставиться у відповідність деяке число – ідентифікатор лексичного класу (token).

Приклад. Розглянемо оператор `const pi = 3.1416;`

Цей оператор складається з таких лексем:

- `const` – лексичний клас “ключові слова”;
- `pi` – лексичний клас “ідентифікатори”;
- `=` – лексичний клас “знаки операцій”;
- `3.1416` – лексичний клас “числова константа”;
- `;` – лексичний клас “роздільники”.

Алгоритм лексичного аналізу

В загальному випадку задача сканера при лексичному аналізі дещо ширша, ніж просто перевірка ланцюжка символів лексеми на відповідність її правилам граматики вхідної мови. Сканер повинен виконати як розпізнавання так і запам'ятовування лексеми (занесення її в таблицю лексем). Набір дій визначається реалізацією транслятора.

Друга проблема, яка потребує вирішення, це виділення меж лексем. Адже у вхідному тексті програми лексеми не обмежені спеціальними символами. Якщо говорити в термінах програми-сканера, то визначення меж лексем - це виділення тих рядків в загальному потоці вхідних символів, для яких треба виконувати розпізнавання. У загальному випадку ця задача може бути складною, але для простих вхідних мов межі лексем розпізнаються по заданим термінальним символам. Це символи – пробіли, знаки операцій, символи коментарів, а також роздільники (коми, крапки з комою та ін.). Набір таких термінальних символів може варіюватися в залежності від вхідного мови. Важливо відзначити, що знаки операцій самі також є лексемами, і необхідно не пропустити їх при розпізнаванні тексту.

Алгоритм роботи найпростішого сканера можна описати так:

1) перегляд вхідного потоку символів здійснюється до виявлення чергового символу, який є початком нової лексеми;
2) для відібраної частини вхідного потоку виконується функція розпізнавання лексеми;
3) при успішному розпізнаванні інформація про виділену лексему заноситься в таблицю лексем, і алгоритм повертається в стан для розпізнавання чергової лексеми;
4) при неуспішному розпізнаванні видається повідомлення про помилку, а подальші дії залежать від реалізації сканера – або його робота припиняється, або робиться спроба розпізнати наступну лексему (йде повернення до першого етапу алгоритму);
5) робота програми-сканера триває до тих пір, поки не будуть переглянуті всі символи вхідної програми.

КОНТРОЛЬНІ ПИТАННЯ

1) Що таке лексичний аналіз?
2) Що таке лексема?
3) Які основні функції лексичного аналізу?
4) Що таке лексичні класи?
5) Що таке скінченний автомат?

ЛІТЕРАТУРА

1) Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 – «Комп'ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. – Київ: КПІ ім. Ігоря Сікорського, 2021.
2) Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 с.
3) Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. – 1038 с.

ЗАВДАННЯ

1. Визначити термінальні символи і ключові слова.
2. Розділити лексеми на лексичні класи.
3. Розробити алгоритм роботи лексичного аналізатора на основі скінченного автомату. Визначити стани скінченного автомату і спроектувати діаграму переходів.
4. Написати і налагодити програму, яка моделює роботу розробленого лексичного аналізатора.
5. Скласти звіт про виконану роботу. У звіті мають бути:
   - Список термінальних символів і ключових слів;
   - Список лексичних класів;
   - Діаграма переходів скінченного автомату (алгоритм роботи лексичного аналізатора);
   - Вхідні дані для програми і результати роботи програми;
   - Коди програми, яка моделює роботу лексичного аналізатора.
6. Дати відповідь на контрольні запитання викладача.

Варіанти індивідуальних завдань студент отримує у викладача!

ПРИКЛАД ВИКОНАННЯ

Завдання: реалізувати лексичний аналізатор для вхідної мови програмування, програма на якій міститься у текстовому файлі (у форматі ASCII).

Програма на вхідній мові програмування має починатись з ключового слова **`start`**, далі має йти розділ опису змінних **`var`**. Між розділом **`var`** і ключовим словом **`stop`** розміщуються оператори програми. Операторів є 4: оператор вводу даних **`input`**, оператор виводу даних **`output`**, оператор присвоєння **`:=`** і умовний оператор **`if – then [- else ]`**. Кожен оператор має завершуватись символом крапка з комою **`;`**. Оператор присвоєння дозволяє присвоїти деякій змінній значення арифметичного виразу. Допустимі арифметичні операції - **`+ , -, *, /`**. Операндами можуть бути змінні, цілі додатні константи і інші вирази, взяті в дужки. В умовному операторі використовуються логічні вирази, допустимі такі операції порівняння **`> , < , = , <>`** і такі логічні операції **`! , & , |`**. Ідентифікатори (імена змінних) можуть бути довжиною до 4-х символів і складаються лише з маленьких латинських літер або цифр. Перший символ ідентифікатора завжди літера. Тип даних лише один – **`integer`**, при оголошення декількох змінних вони записуються через кому, вкінці опису змінних ставиться символ крапка з комою **`;`**. Коментарі починаються з двох косих ліній **`// … `** .

Спочатку **визначаємо термінальні символи і ключові слова**:

- **start** – початок програми
- **var** – оголошення змінних
- **stop** – кінець програми
- **integer** – тип даних
- **input** – оператор вводу
- **output** – оператор виводу
- **if, then, else** – умовний оператор
- **:=** –оператор присвоєння
- **+** – додавання
- **-** – віднімання
- ***** – множення
- **/** – ділення
- **>** – більше
- **<** – менше
- **=** – рівність
- **<>** – нерівність
- **!** – заперечення
- **&** – логічне І
- **|** – логічне АБО
- **;** – кінець оператора
- **,** – розділювач змінних
- **(**– відкрита дужка
- **)** – закрита дужка
- **//** – початок коментаря
- **a … z** – маленькі латинські букви
- **0 … 9** – цифри
- символи табуляції, переходу на новий рядок, пробіл

Розділимо лексеми на типи або лексичні класи:

- Ключові слова (0 – **start**, 1 – **var**, 2 – **stop**, 3 – **input**, 4 – **output**, 5 – **integer**, 6 – **if**, 7 – **then**, 8 – **else**)
- Ідентифікатори (9 – починається з літери, далі або маленька літера або цифра, максимум 4 символи)
- Числові константи (10 - ціле число без знаку)
- Оператор присвоєння (11 – **:=**)
- Знаки операції (12 – **+**, 13 – **-**, 14 – *****, 15 – **/**, 16 – **>**, 17 – **<**, 18 – **=**, 19 – **<>**, 20 – **!**, 21 – **&**, 22 – **|**)
- Розділювачі (25 –**;**, 26 –**,**)
- Дужки (23 – **(**, 24 – **)**)
- Невідома лексема (27 – символи і ланцюжки символів, які не підпадають під вищеописані правила).

Розробимо алгоритм роботи лексичного аналізатора на основі скінченного автомату. Лексичний аналізатор працює за принципом скінченного автомату з такими станами:

- **Start** – початок виділення чергової лексеми;
- **Finish** – кінець виділення чергової лексеми;
- **EndOfFile** – кінець файлу, завершення розпізнавання лексем;
- **Letter** – перший символ буква, розпізнавання слів (ключові слова і ідентифікатори);
- **Digit** – перший символ цифра, розпізнавання числових констант;
- **Separators** – видалення пробілів, символів табуляції і переходу на новий рядок;
- **Scomment** – перший символ “**/**”, можливо далі йде коментар;
- **Comment** – видалення тексту коментаря;
- **Another** – опрацювання інших символів.

Алгоритм роботи скінченного автомату можна зобразити у вигляді діаграми переходів.

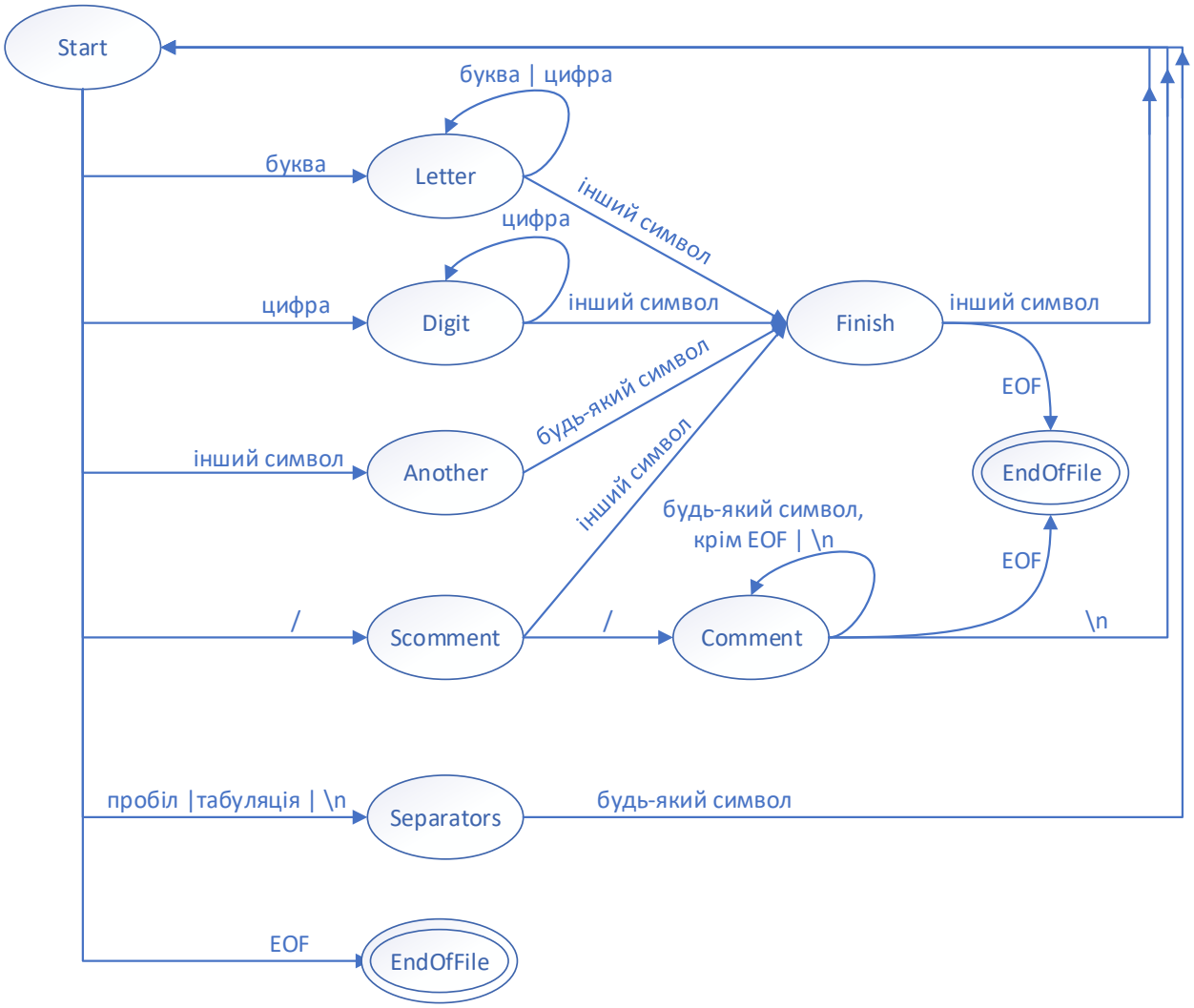

Рис. 4.2 Діаграма переходів скінченного автомату (алгоритм роботи лексичного аналізатора).

Лексичний аналізатор працює наступним чином: читаємо перший символ з файлу, скінченний автомат перебуває у стані **Start**. Далі автомат переходить у один з наступних станів:

- **Letter** – якщо символ це латинська маленька буква;
- **Digit** – якщо символ це цифра;
- **Another** – якщо символ є якимось іншим символом;
- **Scomment** – якщо символ це знак /;
- **Separators** – якщо символ це пробіл, символ табуляції або переходу на новий рядок;
- **EndOfFile** – якщо символ це однака кінця файлу.

У стані **Letter** читаємо по одному символи з файлу і виділяємо ланцюжок символів, який починається з букви, а далі можуть слідувати букви або цифри. Кінець ланцюжка – якщо прочитаний символ відмінний від букви чи цифри. Виділений ланцюжок порівнюємо з ключовими словами, якщо співпадінь немає, вважаємо його ідентифікатором при умові, що довжина ланцюжка не більше 4-х символів, інакше це невизначена лексема. Переходимо до стану **Finish**.

У стані **Digit** читаємо по одному символи з файлу і виділяємо ланцюжок символів, який складається лише з цифр, вважаємо цей ланцюжок числовою константою. Кінець ланцюжка – якщо прочитаний символ відмінний від цифри. Переходимо до стану **Finish**.

У стані **Scomment** читаємо наступний символ, якщо це знак “**/**”, то далі до кінця рядка йде коментар, який можна проігнорувати, переходимо до стану **Comment**. Якщо ж наступний символ не знак “**/**”, то вважаємо що поточна лексема це знак операції ділення, читаємо наступний символ і переходимо до стану **Finish**.

У стані **Comment** читаємо символи, поки не зустрінеться символ переходу на новий рядок, після цього переходимо до виділення нової лексеми – до стану **Start**.

У стані **Separators** читаємо наступний символ і переходимо до виділення нової лексеми – до стану **Start**. Тобто пропускаємо усі пробіли, символи табуляції і переходу на новий рядок.

У стані **Another** порівнюємо поточний прочитаний символ з символами, що позначають знаки операцій, розділювачі і круглі дужки і визначаємо одну з лексем. Є дві лексеми, які вимагають ще читання наступного символу з файлу – це оператор присвоєння “:=” і операція перевірки на нерівність “<>”. Якщо співпадіння не виявлено, то поточний символ – невідома лексема, читаємо наступний символ і переходимо до стану **Finish**.

У стані **Finish** записуємо поточну лексему у таблицю лексем і переходимо до виділення нової лексеми, до стану **Start**.

У стані **EndOfFile** завершуємо обробку вхідного файлу, усі символи з файлу прочитані, усі лексеми записані у таблицю лексем.

Напишемо програму, яка змоделює роботу розробленого алгоритму роботи лексичного аналізатора на основі скінченного автомату. Коди програми розміщено в ДОДАТКУ А.

Для перевірки роботи програми створимо текстовий файл з наступним вмістом:

```
start stop
// comment
start var stop integer input output
if then else
x y result res
:=+-*/><<>!&|;,)(
456 -987 %^:_
// else comment
q:=8;
// hello
integer;
```

Після виконання програми отримаємо таблицю лексем:

```
-------------------------------------------------------------------------
|             TOKEN TABLE                                               |
-------------------------------------------------------------------------
| line number |      token      |    value   | token code | type of token |
-------------------------------------------------------------------------
|           1 |           start |          0 |          0 | StartProgram  |
-------------------------------------------------------------------------
|           1 |            stop |          0 |          3 | EndProgram    |
-------------------------------------------------------------------------
|           3 |           start |          0 |          0 | StartProgram  |
-------------------------------------------------------------------------
|           3 |             var |          0 |          1 | Variable      |
-------------------------------------------------------------------------
|           3 |            stop |          0 |          3 | EndProgram    |
-------------------------------------------------------------------------
|           3 |         integer |          0 |          2 | Integer       |
```

3	input	0	4	Input
3	output	0	5	Output
4	if	0	6	If
4	then	0	7	Then
4	else	0	8	Else
5	x	0	9	Identifier
5	y	0	9	Identifier
5	result	0	27	Unknown
5	res	0	9	Identifier
6	:=	0	11	Assign
6	+	0	12	Add
6	-	0	13	Sub
6	*	0	14	Mul
6	/	0	15	Div
6	>	0	18	Greate
6	<	0	19	Less
6	<>	0	17	NotEquality
6	!	0	20	Not
6	&	0	21	And
6	|	0	22	Or
6	;	0	25	Semicolon
6	,	0	26	Comma
6	)	0	24	RBraket
6	(	0	23	LBraket
7	456	456	10	Number
7	-	0	13	Sub
7	987	987	10	Number
7	%	0	27	Unknown
7	^	0	27	Unknown

7	:	0	27	Unknown
7	_	0	27	Unknown
9	q	0	9	Identifier
9	:=	0	11	Assign
9	8	8	10	Number
9	;	0	25	Semicolon
11	integer	0	2	Integer
11	;	0	25	Semicolon

ДОДАТОК А

У файлі Header.h помістимо необхідні структури даних:

```
#pragma once
#define MAX_TOKENS 1000
#define MAX_IDENTIFIER 10

// перерахування, яке описує всі можливі типи лексем
enum TypeOfTokens
{
    StartProgram,   // start
    Variable,          // var
    Type,              // integer
    EndProgram,        // stop
    Input,                  // input
    Output,                 // output
    If,                // if
    Then,                   // then
    Else,           // else

    Identifier,    // Identifier

    Number,                // number

    Assign,                // :=
    Add,                    // +
    Sub,                    // -
    Mul,                    // *
    Div,                    // /

    Equality,          // =
    NotEquality,       // <>
    Greate,                // >
    Less,                   // <
    Not,                    // !
    And,                    // &
    Or,                 // |

    LBraket,           // (
    RBraket,           // )
    Semicolon,         // ;
    Comma,                  // ,
    Unknown
};

// структура для зберігання інформації про лексему
struct Token
{
    char name[16];      // ім'я лексеми
    int value;          // значення лексеми (для цілих констант)
    int line;           // номер рядка
    TypeOfTokens type;  // тип лексеми
};

typedef  char Id[5];

// перерахування, яке описує стани лексичного аналізатора
enum States
{
    Start,      // початок виділення чергової лексеми
    Finish,     // кінець виділення чергової лексеми
    Letter,     // опрацювання слів (ключові слова і ідентифікатори)
    Digit,      // опрацювання цифри
    Separators, // видалення пробілів, символів табуляції і переходу на новий рядок
    Another,    // опрацювання інших символів
    EndOfFile,  // кінець файлу
    SComment,   // початок коментаря
    Comment     // видалення коментаря
};
```

У файлі Source.cpp помістимо реалізацію необхідних функцій і основну програму.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "Header.h"
```

```
// функція отримує лексеми з вхідного файлу F і записує їх у таблицю лексем TokenTable
// результат функції – кількість лексем
unsigned int GetTokens(FILE* F, Token TokenTable[])
{
	States state = Start;
	Token TempToken;
	// кількість лексем
	unsigned int NumberOfTokens = 0;
	char ch, buf[16];
	int line = 1;

	// читання першого символу з файлу
	ch = getc(F);

	// пошук лексем
	while (1)
	{
		switch (state)
		{
			// стан Start – початок виділення чергової лексеми
			// якщо поточний символ маленька літера, то переходимо до стану Letter
			// якщо поточний символ цифра, то переходимо до стану Digit
			// якщо поточний символ пробіл, символ табуляції або переходу на новий рядок, то
переходимо до стану Separators
			// якщо поточний символ / то є ймовірність, що це коментар, переходимо до стану
SComment
			// якщо поточний символ EOF (ознака кінця файлу), то переходимо до стану
EndOfFile
			// якщо поточний символ відмінний від попередніх, то переходимо до стану Another
		case Start:
		{
			if (ch == EOF)
				state = EndOfFile;
			else
				if (ch <= 'z' && ch >= 'a')
					state = Letter;
				else
					if (ch <= '9' && ch >= '0')
						state = Digit;
					else
						if (ch == ' ' || ch == '\t' || ch == '\n')
							state = Separators;
						else
							if (ch == '/')
								state = SComment;
							else
								state = Another;
			break;
		}

		// стан Finish – кінець виділення чергової лексеми і запис лексеми у таблицю лексем
		case Finish:
		{
			if (NumberOfTokens < MAX_TOKENS)
			{
				TokenTable[NumberOfTokens++] = TempToken;
				if (ch != EOF)
					state = Start;
				else
					state = EndOfFile;
			}
			else
			{
				printf("\n\t\t\ttoo many tokens !!!\n");
				return NumberOfTokens - 1;
			}
			break;
		}

		// стан EndOfFile – кінець файлу, можна завершувати пошук лексем
		case EndOfFile:
		{
			return NumberOfTokens;
		}

		// стан Letter – поточний символ – маленька літера, поточна лексема – ключове слово або
ідентифікатор
		case Letter:
		{
			buf[0] = ch;
			int j = 1;
```

```
                        ch = getc(F);

                        while (((ch <= 'z' && ch >= 'a') || (ch <= '9' && ch >= '0')) && j < 15)
                        {
                                buf[j++] = ch;
                                ch = getc(F);
                        }
                        buf[j] = '\0';

                        TypeOfTokens temp_type = Unknown;

                        if (!strcmp(buf, "start"))
                                temp_type = StartProgram;
                        else
                                if (!strcmp(buf, "var"))
                                        temp_type = Variable;
                                else
                                        if (!strcmp(buf, "integer"))
                                                temp_type = Type;
                                        else
                                                if (!strcmp(buf, "stop"))
                                                        temp_type = EndProgram;
                                                else
                                                        if (!strcmp(buf, "input"))
                                                                temp_type = Input;
                                                        else
                                                                if (!strcmp(buf, "output"))
                                                                        temp_type = Output;
                                                                else
                                                                        if (!strcmp(buf, "if"))
                                                                                temp_type = If;
                                                                        else
                                                                                if (!strcmp(buf, "then"))
                                                                                        temp_type = Then;
                                                                                else
                                                                                        if (!strcmp(buf,
"else"))

                                                                                                temp_type =
Else;

                                                                                        else

                                                                                                if
(strlen(buf) < 5)

        temp_type = Identifier;

                        strcpy_s(TempToken.name, buf);
                        TempToken.type = temp_type;
                        TempToken.value = 0;
                        TempToken.line = line;
                        state = Finish;
                        break;
                }

                // стан Digit - поточний символ - цифра, поточна лексема - число
                case Digit:
                {
                        buf[0] = ch;
                        int j = 1;

                        ch = getc(F);

                        while ((ch <= '9' && ch >= '0') && j < 15)
                        {
                                buf[j++] = ch;
                                ch = getc(F);
                        }
                        buf[j] = '\0';

                        strcpy_s(TempToken.name, buf);
                        TempToken.type = Number;
                        TempToken.value = atoi(buf);
                        TempToken.line = line;
                        state = Finish;
                        break;
                }

                // стан Separators - поточний символ пробіл, символ табуляції або переходу на новий
рядок - видаляємо їх
                case Separators:
                {
```

```
                if (ch == '\n')
                        line++;

                ch = getc(F);

                state = Start;
                break;
        }

        // стан SComment – поточний символ /,  можливо це коментар
        case SComment:
        {
                ch = getc(F);
                if (ch == '/')
                        state = Comment;
                else
                {
                        strcpy_s(TempToken.name, "/");
                        TempToken.type = Div;
                        TempToken.value = 0;
                        TempToken.line = line;
                        state = Finish;
                }
                break;
        }

        // стан Comment – поточний символ /,  отже це коментар, видаляємо усі символи до кінця
рядка
        case Comment:
        {
                while (ch != '\n' && ch != EOF)
                {
                        ch = getc(F);
                }
                if (ch == EOF)
                {
                        state = EndOfFile;
                        break;
                }
                //line++;
                //ch = getc(F);
                state = Start;
                break;
        }

        // стан Another – поточний символ – символ, відмінний від попередніх
        case Another:
        {
                switch (ch)
                {

                case '(':
                {
                        strcpy_s(TempToken.name, "(");
                        TempToken.type = LBraket;
                        TempToken.value = 0;
                        TempToken.line = line;
                        ch = getc(F);
                        state = Finish;
                        break;
                }

                case  ')':
                {
                        strcpy_s(TempToken.name, ")");
                        TempToken.type = RBraket;
                        TempToken.value = 0;
                        TempToken.line = line;
                        ch = getc(F);
                        state = Finish;
                        break;
                }

                case ';':
                {
                        strcpy_s(TempToken.name, ";");
                        TempToken.type = Semicolon;
                        TempToken.value = 0;
                        TempToken.line = line;
                        ch = getc(F);
                        state = Finish;
```

```
        break;
}
case  ',':
{
        strcpy_s(TempToken.name, ",");
        TempToken.type = Comma;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
}

case '*':
{
        strcpy_s(TempToken.name, "*");
        TempToken.type = Mul;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
}

case '+':
{

        strcpy_s(TempToken.name, "+");
        TempToken.type = Add;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;

        break;
}

case '-':
{
        strcpy_s(TempToken.name, "-");
        TempToken.type = Sub;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
}

case '&':
{
        strcpy_s(TempToken.name, "&");
        TempToken.type = And;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
}

case '|':
{
        strcpy_s(TempToken.name, "|");
        TempToken.type = Or;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
}
case '!':
{
        strcpy_s(TempToken.name, "!");
        TempToken.type = Not;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
}

case '<':
```

```
		{
			ch = getc(F);
			if (ch == '>')
			{
				strcpy_s(TempToken.name, "<>");
				TempToken.type = NotEquality;
				TempToken.value = 0;
				TempToken.line = line;
				ch = getc(F);
				state = Finish;
			}
			else
			{
				strcpy_s(TempToken.name, "<");
				TempToken.type = Less;
				TempToken.value = 0;
				TempToken.line = line;
				state = Finish;
			}
			break;
		}

		case '>':
		{
			strcpy_s(TempToken.name, ">");
			TempToken.type = Greate;
			TempToken.value = 0;
			TempToken.line = line;
			ch = getc(F);
			state = Finish;
			break;
		}

		case '=':
		{
			strcpy_s(TempToken.name, "=");
			TempToken.type = Equality;
			TempToken.value = 0;
			TempToken.line = line;
			ch = getc(F);
			state = Finish;
			break;
		}

		case ':':
		{
			ch = getc(F);
			if (ch == '=')
			{
				strcpy_s(TempToken.name, ":=");
				TempToken.type = Assign;
				TempToken.value = 0;
				TempToken.line = line;
				ch = getc(F);
				state = Finish;
			}
			else
			{
				strcpy_s(TempToken.name, ":");
				TempToken.type = Unknown;
				TempToken.value = 0;
				TempToken.line = line;
				state = Finish;
			}
			break;
		}

		default:
		{
			TempToken.name[0] = ch;
			TempToken.name[1] = '\0';
			TempToken.type = Unknown;
			TempToken.value = 0;
			TempToken.line = line;
			ch = getc(F);
			state = Finish;
			break;
		}
		}
	}
	}
```

```
	}

}

// функція друкує таблицю лексем на екран
void PrintTokens(Token TokenTable[], unsigned int TokensNum)
{
	char type_tokens[16];
	printf("\n\n-------------------------------------------------------------------------------------\n");
	printf("|             TOKEN TABLE                                                           |\n");
	printf("-------------------------------------------------------------------------------------\n");
	printf("| line number |      token      |    value   | token code | type of token |\n");
	printf("-------------------------------------------------------------------------------");
	for (unsigned int i = 0; i < TokensNum; i++)
	{
		switch (TokenTable[i].type)
		{
		case StartProgram:
			strcpy_s(type_tokens, "StartProgram");
			break;
		case Variable:
			strcpy_s(type_tokens, "Variable");
			break;
		case Type:
			strcpy_s(type_tokens, "Integer");
			break;
		case Identifier:
			strcpy_s(type_tokens, "Identifier");
			break;
		case EndProgram:
			strcpy_s(type_tokens, "EndProgram");
			break;
		case Input:
			strcpy_s(type_tokens, "Input");
			break;
		case Output:
			strcpy_s(type_tokens, "Output");
			break;
		case If:
			strcpy_s(type_tokens, "If");
			break;
		case Then:
			strcpy_s(type_tokens, "Then");
			break;
		case Else:
			strcpy_s(type_tokens, "Else");
			break;
		case Assign:
			strcpy_s(type_tokens, "Assign");
			break;
		case Add:
			strcpy_s(type_tokens, "Add");
			break;
		case Sub:
			strcpy_s(type_tokens, "Sub");
			break;
		case Mul:
			strcpy_s(type_tokens, "Mul");
			break;
		case Div:
			strcpy_s(type_tokens, "Div");
			break;
		case Equality:
			strcpy_s(type_tokens, "Equality");
			break;
		case NotEquality:
			strcpy_s(type_tokens, "NotEquality");
			break;
		case Greate:
			strcpy_s(type_tokens, "Greate");
			break;
		case Less:
			strcpy_s(type_tokens, "Less");
			break;
		case Not:
			strcpy_s(type_tokens, "Not");
			break;
		case And:
			strcpy_s(type_tokens, "And");
			break;
		case Or:
```

```
                        strcpy_s(type_tokens, "Or");
                        break;
                case LBraket:
                        strcpy_s(type_tokens, "LBraket");
                        break;
                case RBraket:
                        strcpy_s(type_tokens, "RBraket");
                        break;
                case Number:
                        strcpy_s(type_tokens, "Number");
                        break;
                case Semicolon:
                        strcpy_s(type_tokens, "Semicolon");
                        break;
                case Comma:
                        strcpy_s(type_tokens, "Comma");
                        break;
                case Unknown:
                        strcpy_s(type_tokens, "Unknown");
                        break;
                }

                printf("\n|%12d |%16s |%11d |%11d | %-13s |\n",
                        TokenTable[i].line,
                        TokenTable[i].name,
                        TokenTable[i].value,
                        TokenTable[i].type,
                        type_tokens);
                printf("-------------------------------------------------------------------------------");
        }
}

// функція друкує таблицю лексем у файл
void PrintTokensToFile(char* FileName, Token TokenTable[], unsigned int TokensNum)
{
        FILE* F;
        if ((fopen_s(&F, FileName, "wt")) != 0)
        {
                printf("Error: Can not create file: %s\n", FileName);
                return;
        }
        char type_tokens[16];
        fprintf(F, "-------------------------------------------------------------------------------\n");
        fprintf(F, "|             TOKEN TABLE                                                     |\n");
        fprintf(F, "-------------------------------------------------------------------------------\n");
        fprintf(F, "| line number |      token      |    value   | token code | type of token |\n");
        fprintf(F, "-------------------------------------------------------------------------------");
        for (unsigned int i = 0; i < TokensNum; i++)
        {
                switch (TokenTable[i].type)
                {
                case StartProgram:
                        strcpy_s(type_tokens, "StartProgram");
                        break;
                case Variable:
                        strcpy_s(type_tokens, "Variable");
                        break;
                case Identifier:
                        strcpy_s(type_tokens, "Identifier");
                        break;
                case Type:
                        strcpy_s(type_tokens, "Integer");
                        break;
                case EndProgram:
                        strcpy_s(type_tokens, "EndProgram");
                        break;
                case Input:
                        strcpy_s(type_tokens, "Input");
                        break;
                case Output:
                        strcpy_s(type_tokens, "Output");
                        break;
                case If:
                        strcpy_s(type_tokens, "If");
                        break;
                case Then:
                        strcpy_s(type_tokens, "Then");
                        break;
                case Else:
                        strcpy_s(type_tokens, "Else");
                        break;
```

```
                case Assign:
                        strcpy_s(type_tokens, "Assign");
                        break;
                case Add:
                        strcpy_s(type_tokens, "Add");
                        break;
                case Sub:
                        strcpy_s(type_tokens, "Sub");
                        break;
                case Mul:
                        strcpy_s(type_tokens, "Mul");
                        break;
                case Div:
                        strcpy_s(type_tokens, "Div");
                        break;
                case Equality:
                        strcpy_s(type_tokens, "Equality");
                        break;
                case NotEquality:
                        strcpy_s(type_tokens, "NotEquality");
                        break;
                case Greate:
                        strcpy_s(type_tokens, "Greate");
                        break;
                case Less:
                        strcpy_s(type_tokens, "Less");
                        break;
                case Not:
                        strcpy_s(type_tokens, "Not");
                        break;
                case And:
                        strcpy_s(type_tokens, "And");
                        break;
                case Or:
                        strcpy_s(type_tokens, "Or");
                        break;
                case LBraket:
                        strcpy_s(type_tokens, "LBraket");
                        break;
                case RBraket:
                        strcpy_s(type_tokens, "RBraket");
                        break;
                case Number:
                        strcpy_s(type_tokens, "Number");
                        break;
                case Semicolon:
                        strcpy_s(type_tokens, "Semicolon");
                        break;
                case Comma:
                        strcpy_s(type_tokens, "Comma");
                        break;
                case Unknown:
                        strcpy_s(type_tokens, "Unknown");
                        break;
                }

                fprintf(F, "\n|%12d |%16s |%11d |%11d | %-13s |\n",
                        TokenTable[i].line,
                        TokenTable[i].name,
                        TokenTable[i].value,
                        TokenTable[i].type,
                        type_tokens);
                fprintf(F, "-------------------------------------------------------------------------------
");
        }
        fclose(F);

}

int main(int argc, char* argv[])
{
        // таблиця лесем
        //Token TokenTable[MAX_TOKENS];

        Token* TokenTable = new Token[MAX_TOKENS];

        // кількість лексем
        unsigned int TokensNum;
        // таблиця ідентифікаторів

        char InputFile[32] = "";
```

```
	FILE* InFile;

	if (argc != 2)
	{
		printf("Input file name: ");
		gets_s(InputFile);
	}
	else
	{
		strcpy_s(InputFile, argv[1]);
	}

	if ((fopen_s(&InFile, InputFile, "rt")) != 0)
	{
		printf("Error: Can not open file: %s\n", InputFile);
		return 1;
	}

	TokensNum = GetTokens(InFile, TokenTable);

	char TokenFile[32];


	int i = 0;
	while (InputFile[i] != '.')
	{
		TokenFile[i] = InputFile[i];
		i++;
	}
	TokenFile[i] = '\0';
	strcat_s(TokenFile, ".token");

	PrintTokensToFile(TokenFile, TokenTable, TokensNum);
	fclose(InFile);

	printf("\nLexical analysis completed. List of tokens in the file %s\n", TokenFile);
	PrintTokens(TokenTable, TokensNum);

	delete[]TokenTable;
	return 0;

}
```