

Міністерство освіти і науки України
Національний університет «Львівська політехніка»

Кафедра ЕОМ



**Пояснювальна записка
до курсового проєкту**

З дисципліни «Системне програмування, частина 2»

На тему: "Розробка системних програмних модулів
та компонент систем програмування.

Розробка транслятора з вхідної мови програмування"

Варіант №23

Виконав:
ст. групи КІ-309
Чорноморд Я. С.
Прийняв:
Ст.в. каф ЕОМ
Козак Н. Б.

Анотація

Цей курсовий проєкт приводить до розробки транслятора, який здатен конвертувати вхідну мову, визначену відповідно до варіанту, у мову С. Процес трансляції включає в себе лексичний аналіз, синтаксичний аналіз та генерацію коду.

Лексичний аналіз розбиває вхідну послідовність символів на лексеми, які записуються у відповідну таблицю лексем. Кожній лексемі присвоюється числове значення для полегшення порівнянь, а також зберігається додаткова інформація, така як номер рядка, значення (якщо тип лексеми є числом) та інші деталі.

Синтаксичний аналіз: використовується висхідний метод аналізу без повернення. Призначений для побудови дерева розбору, послідовно рухаючись від листків вгору до кореня дерева розбору.

Генерація коду включає повторне прочитання таблиці лексем та створення відповідного коду на мові С для кожного блоку лексем. Отриманий код записується у результуючий файл, готовий для виконання.

Зміст

Анотація	2
Завдання до курсового проекту.....	4
Вступ	5
1.Огляд методів та способів проектування трансляторів	6
2.Формальний опис вхідної мови програмування	8
2.1.Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура.....	8
2.2.Опис термінальних символів та ключових слів.....	10
3.Розробка транслятора вхідної мови програмування.....	12
3.1.Вибір технології програмування	12
3.2.Проектування таблиць транслятора	13
3.3.Розробка лексичного аналізатора.....	16
3.3.1.Розробка блок-схеми алгоритму.....	17
3.3.2.Опис програми реалізації лексичного аналізатора	17
3.4.Розробка синтаксичного та семантичного аналізатора.....	19
3.4.1.Опис програми реалізації синтаксичного та семантичного аналізатора	20
3.4.2.Розробка граф-схеми алгоритму	23
3.5.Розробка генератора коду	24
3.5.1.Розробка граф-схеми алгоритму	25
3.5.2.Опис програми реалізації генератора коду.....	26
4.Опис програми.....	27
4.1.Опис інтерфейсу та інструкція користувачеві.....	30
5.Відлагодження та тестування програми	31
5.1.Виявлення лексичних та синтаксичних помилок.....	31
5.2.Виявлення семантичних помилок	32
5.3.Загальна перевірка коректності роботи транслятора.....	32
5.4.Тестова програма №1.....	34
5.5.Тестова програма №2.....	35
5.6.Тестова програма №3.....	36
6. Верифікація тестових програм.....	38
Висновки.....	39
Список використаної літератури.....	40
Додатки	42

Завдання до курсового проекту

Варіант 23

Завдання на курсовий проєкт

1. Цільова мова транслятора – мова програмування C.
2. Для отримання виконавчого файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio або будь-яким іншим.
3. Мова розробки транслятора: C++.
4. Реалізувати оболонку або інтерфейс з командного рядка.
5. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.
6. На виході розробленого транслятора мають створюватись такі файли:
 - *файл з лексемами;*
 - *файл з повідомленнями про помилки (або про їх відсутність);*
 - *файл на мові C;*
 - *об'єктний файл;*
 - *виконавчий файл.*
7. Назва вхідної мови програмування утворюється від першої букви у прізвищі студента та останніх двох цифр номера його варіанту. Саме таке розширення повинні мати текстові файли, написані на цій мові програмування.

В моєму випадку це .c23

Опис вхідної мови програмування:

- Тип даних: Longint
- Блок тіла програми: Name <name>; Data...; Body End
- Оператор вводу: Read ()
- Оператор виводу: Write ()
- Оператори: If Else (C)
Goto (C)
For-To-Do (Паскаль)
For-DownTo-Do (Паскаль)
While (Бейсік)
Repeat-Until (Паскаль)
- Регістр ключових слів: Up-Low перший символ Up
- Регістр ідентифікаторів: Low-Up16 перший символ _
- Операції арифметичні: ++, --, **, Div, Mod
- Операції порівняння: =, <>, >=, <=
- Операції логічні: !!, &&, ||
- Коментар: \\\...
- Ідентифікатори змінних, числові константи
- Оператор присвоєння: <==

Вступ

Термін "транслятор" визначає програму, яка виконує переклад (трансляцію) початкової програми, написаної на вхідній мові, у еквівалентну їй об'єктну програму. У випадку, коли мова високого рівня є вхідною, а мова асемблера або машинна – вихідною, такий транслятор отримує назву компілятора.

Транслятори можуть бути розділені на два основних типи: компілятори та інтерпретатори. Процес компіляції включає дві основні фази: аналіз та синтез. Під час аналізу вхідну програму розбивають на окремі елементи (лексми), перевіряють її відповідність граматичним правилам і створюють проміжне представлення програми. На етапі синтезу з проміжного представлення формується програма в машинних кодах, яку називають об'єктною програмою. Останню можна виконати на комп'ютері без додаткової трансляції.

У відмінну від компіляторів, інтерпретатор не створює нову програму; він лише виконує – інтерпретує – кожен інструкцію вхідної мови програмування. Подібно компілятору, інтерпретатор аналізує вхідну програму, створює проміжне представлення, але не формує об'єктну програму, а негайно виконує команди, передбачені вхідною програмою.

Компілятор виконує переклад програми з однієї мови програмування в іншу. На вхід компілятора надходить ланцюг символів, який представляє вхідну програму на певній мові програмування. На виході компілятора (об'єктна програма) також представляє собою ланцюг символів, що вже відповідає іншій мові програмування, наприклад, машинній мові конкретного комп'ютера. При цьому сам компілятор може бути написаний на третій мові.

1. Огляд методів та способів проєктування трансляторів

1.1. Методи побудови трансляторів

1.1.1. Лексичний аналіз

На цьому етапі текст програми розбивається на окремі елементи (лексеми): ключові слова, ідентифікатори, літерали, роздільники та оператори. Для реалізації лексичного аналізу зазвичай застосовують:

Регулярні вирази — для опису шаблонів лексем.

Скінченні автомати — для автоматизованого розпізнавання лексем.

1.2. Синтаксичний аналіз

Синтаксичний аналізатор перевіряє структуру програми відповідно до граматики мови та формує дерево розбору. Основні методи:

Зверху вниз (top-down parsing):

Рекурсивний спуск (Recursive Descent Parsing).

LL-аналізатори (табличні методи).

Знизу вгору (bottom-up parsing):

LR-аналізатори (типи LR(0), SLR, LALR, Canonical LR).

1.3. Семантичний аналіз

Цей етап спрямований на перевірку логічної коректності програми: відповідність типів даних, області видимості змінних, коректність викликів функцій. Для семантичного аналізу використовують:

Атрибутні граматики (Attribute Grammars).

Таблиці символів — структури для зберігання інформації про ідентифікатори.

1.4. Генерація коду

Генерація коду полягає у створенні проміжного або машинного коду. Методи включають:

Однопрохідну генерацію — для простих мов.

Двопрохідну генерацію — спочатку створюється проміжний код, який оптимізується перед перетворенням у машинний.

Багатопохідну генерацію — для складних мов із додатковими етапами оптимізації.

1.5. Оптимізація коду

Мета оптимізації — зменшити розмір і підвищити ефективність виконання програми. Популярні техніки:

- Локальні оптимізації — у межах одного блоку коду.
- Глобальні оптимізації — враховують взаємодію між кількома блоками.
- Машинно-незалежна оптимізація — спрощення обчислень.
- Машинно-залежна оптимізація — призначення регістрів, налаштування інструкцій.

1.6. Класифікація трансляторів

Транслятори можна класифікувати за різними критеріями:

- За способом роботи: компілятори, інтерпретатори, асемблери.
- За кількістю проходів: однопрохідні, багатопохідні.
- За цільовим кодом: машинний код, проміжний код (LLVM IR, байт-код).

1.7. Основні підходи до проєктування

Модульний підхід: розбиття транслятора на окремі компоненти (лексичний і синтаксичний аналізатори, генератор коду тощо).

Фазовий підхід: чітке розділення фаз із обміном результатами між ними.

Однопрохідний дизайн: підходить для невеликих або простих мов.

Багатопохідний дизайн: забезпечує глибоку оптимізацію та підтримує складні мови.

2. Формальний опис вхідної мови програмування

2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура

Однією з перших задач, що виникають при побудові компілятора, є визначення вхідної мови програмування. Для цього використовують різні способи формального опису, серед яких я застосував розширену нотацію Бекуса-Наура (extended Backus/Naur Form - EBNF).

```

topRule = "Name", identifier, ";", varsBlok, ";", "Body",
operators, "End";

varsBlok = "Data", "Longint", identifier, [{ commaAndIdentifier
}];

identifier = "_", low_letter, { up_letter | number } {16};
commaAndIdentifier = ",", identifier;

codeBlok = "Body", write | read | assignment | ifStatement
| goto_statement | labelRule | forToOrDownToDoRule |
while | repeatUntil, "End";

operators = write | read | assignment | ifStatement |
goto_statement | labelRule | forToOrDownToDoRule | while
| repeatUntil;

read = "Read", "(", identifier, ")";
write = "Write", "(", equation | stringRule, ")";
assignment = identifier, "<==", equation;
cycle_counter = identifier;
cycle_counter_last_value = equation;
ifStatement = "If", "(", equation, ")", codeBlok, ["Else",
codeBlok];
goto_statement = "Goto", ident ;
labelRule = identifier, ":";
forToOrDownToDoRule = "For", cycle_counter, "<==", equation ,
"To" | "Downto", cycle_counter_last_value, "Do", codeBlok;
while = "While", "(", equation, ")", "Body", operators |
whileContinue | whileExit, "End", "While";
whileContinue = "Continue", "While";

```



```

whileExit = "Exit", "While";
repeatUntil = "Repeat", operators, "Until", "(", equation, ")";
equation = signedNumber | identifier | notRule
[ { operationAndIdentOrNumber | equation } ];
notRule = notOperation, signedNumber | identifier |
equation;
operationAndIdentOrNumber = mult | arithmetic | logic |
compare signedNumber | identifier | equation;
arithmetic = "++" | "--";
mult = "**" | "Div" | "Mod";
logic = "&&" | "||";
notOperation = "!!";
compare = "=" | "<>" | "<=" | ">=";
comment = "LComment", text;
LComment = "\\\";
text = { low_letter | up_letter | number };
signedNumber = [ sign ] digit [{digit}];
sign = "+" | "-";
low_letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
"j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" |
"u" | "v" | "w" | "x" | "y" | "z";
up_letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
"J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
"U" | "V" | "W" | "X" | "Y" | "Z";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
"8" | "9";

```

2.2.Опис термінальних символів та ключових слів

Визначимо окремі термінальні символи та нерозривні набори термінальних символів (ключові слова):

Термінальний символ або ключове слово	Значення
Name	Початок програми
Body	Початок тексту програми
Data	Початок блоку опису змінних
End	Кінець розділу операторів
Read	Оператор вводу змінних
Write	Оператор виводу (змінних або рядкових констант)
<==	Оператор присвоєння
If	Оператор умови
Else	Оператор умови
Goto	Оператор переходу
Label	Мітка переходу
For	Оператор циклу
To	Інкремент циклу
DownTo	Декремент циклу
Do	Початок тіла циклу
While	Оператор циклу
Continue	Оператор циклу
Exit	Оператор циклу
Repeat	Початок тіла циклу
Until	Оператор циклу
++	Оператор додавання

--	Оператор віднімання
**	Оператор множення
Div	Оператор ділення
Mod	Оператор знаходження залишку від ділення
=	Оператор перевірки на рівність
<>	Оператор перевірки на нерівність
<=	Оператор перевірки чи менше
>=	Оператор перевірки чи більше
!!	Оператор логічного заперечення
&&	Оператор кон'юнкції
	Оператор диз'юнкції
Longint	32ох розрядні знакові цілі
\\...	Коментар
,	Розділювач
;	Ознака кінця оператора
(Відкриваюча дужка
)	Закриваюча дужка

Табл. 2.2.1. Опис термінальних символі та ключових слів

До термінальних символів віднесемо також усі цифри (0-9), латинські букви (a-z, A-Z), символи табуляції, символ переходу на нову стрічку, пробілу.

3. Розробка транслятора вхідної мови програмування

3.1. Вибір технології програмування

Для ефективної роботи створюваної програми важливу роль відіграє попереднє складення алгоритму роботи програми, алгоритму написання програми і вибір технології програмування.

Тому при складанні транслятора треба брати до уваги швидкість компіляції, якість об'єктної програми. Проект повинен давати можливість просто вносити зміни.

В реалізації мов високого рівня часто використовується специфічний тільки для компіляції засіб “розкрутки”. З кожним транслятором завжди зв'язані три мови програмування: X – початкова, Y – об'єктна та Z – інструментальна. Транслятор перекладає програми мовою X в програми, складені мовою Y , при цьому сам транслятор є програмою написаною мовою Z .

При розробці даного курсового проекту був використаний висхідний метод синтаксичного аналізу.

Також був обраний прямий метод лексичного аналізу. Характерною ознакою цього методу є те, що його реалізація відбувається без повернення назад. Його можна сприймати, як один спільний скінченний автомат. Такий автомат на кожному кроці читає один вхідний символ і переходить у наступний стан, що наближає його до розпізнавання поточної лексеми чи формування інформації про помилки. Для лексем, що мають однакові підланцюжки, автомат має спільні фрагменти, що реалізують єдину множину станів. Частини, що відрізняються, реалізуються своїми фрагментами

Для виконання поставленого завдання найбільш доцільно буде використати середовище програмування Microsoft Visual Studio 2022, та мову програмування C/C++.

Для якісного і зручного використання розробленої програми користувачем, було прийнято рішення створення консольного інтерфейсу.

3.2.Проектування таблиць транслятора

Використання таблиць значно полегшує створення трансляторів, тому у даному випадку використовуються наступне:

- 1) Таблиця лексем з елементами, які мають таку структуру:

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

- 2) Таблиця лексичних класів

```
enum TypeOfTokens
{
    Mainprogram,
    StartProgram,
    Variable,
    Type,
    EndProgram,
    Input,
    Output,

    If,
    Else,

    Goto,
    Label,

    For,
    To,
    DownTo,
    Do,

    While,
    Exit,
    Continue,
    End,

    Repeat,
    Until,

    Identifier,
    Number,
    Assign,
    Add,
    Sub,
    Mul,
```

```

Div,
Mod,
Equality,
NotEquality,
Greate,
Less,
Not,
And,
Or,
LBracket,
RBracket,
Semicolon,
Colon,
Comma,
Unknown
};

```

Якщо у стовпці «Значення» відсутня інформація про токен, то це вказує на те що його значення визначається користувачем.

Токен	Значення
Program	Name
Start	Body
Vars	Data
End	End
VarType	Longint
Read	Read
Write	Write
Assignment	<==
If	If
Else	Else
Goto	Goto
Colon	:
Label	
For	For
To	To
DownTo	Downto
Do	Do

While	While
Continue	Continue
Exit	Exit
Repeat	Repeat
Until	Until
Addition	++
Subtraction	--
Multiplication	**
Division	Div
Mod	Mod
Equal	=
NotEqual	<>
Less	<=
Greate	>=
Not	!!
And	&&
Or	
Identifier	
Number	
Unknown	
Comma	,
Semicolon	;
LBracket	(
RBracket)
LComment	\
Comment	

Табл. 3.2.1. Опис термінальних символі та ключових слів.

3.3.Розробка лексичного аналізатора

На фазі лексичного аналізу вхідна програма, що представляє собою потік літер, розбивається на лексеми - слова у відповідності з визначеннями мови. Лексичний аналізатор може працювати в двох основних режимах: або як підпрограма, що викликається синтаксичним аналізатором для отримання чергової лексеми, або як повний прохід, результатом якого є файл лексем.

Для нашої програми виберемо другий варіант. Тобто, спочатку буде виконуватись фаза лексичного аналізу. Результатом цієї фази буде файл з списком лексем. Але лексеми записуються у файл не як послідовність символів. Кожній лексемі присвоюється певний символ, тип, значення та рядок. Ці дані далі записуються у файл. Такий підхід дозволяє спростити роботу синтаксичного аналізатора.

Також на етапі лексичного аналізу виявляються деякі (найпростіші) помилки (неприпустимі символи, неправильний запис чисел, ідентифікаторів та ін.)

На вхід лексичного аналізатора надходить текст вихідної програми, а вихідна інформація передається для подальшої обробки компілятором на етапі синтаксичного аналізу.

Існує кілька причин, з яких до складу практично всіх компіляторів включають лексичний аналіз:

- застосування лексичного аналізатора спрощує роботу з текстом вихідної програми на етапі синтаксичного розбору;
- для виділення в тексті та розбору лексем можливо застосовувати просту, ефективну і теоретично добре пророблену техніку аналізу;

3.3.1.Розробка блок-схеми алгоритму

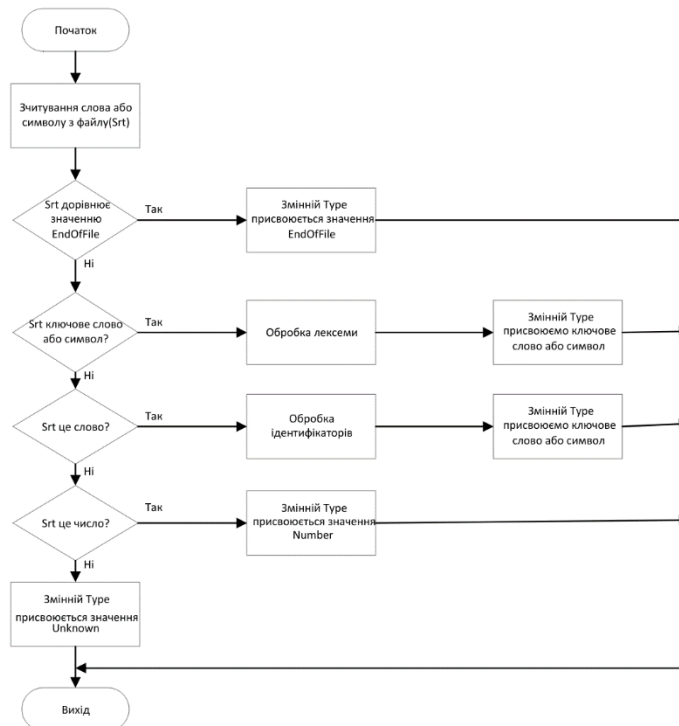


Рис. 3.1 Граф-схема алгоритму роботи лексичного аналізатора.

3.3.2.Опис програми реалізації лексичного аналізатора

Основна задача лексичного аналізу – розбити вихідний текст, що складається з послідовності одиночних символів, на послідовність слів, або лексем, тобто виділити ці слова з безперервної послідовності символів. Всі символи вхідної послідовності з цієї точки зору розділяються на символи, що належать яким-небудь лексемам, і символи, що розділяють лексеми. В цьому випадку використовуються звичайні засоби обробки рядків. Вхідна програма проглядається послідовно з початку до кінця. Базові елементи, або лексичні одиниці, розділяються пробілами, знаками операцій і спеціальними символами (новий рядок, знак табуляції), і таким чином виділяються та розпізнаються ідентифікатори, літерали і термінальні символи (операції, ключові слова).

Програма аналізує файл поки не досягне його кінця. Для вхідного файлу викликається функція `parser()`. Вона зчитує з файлу його вміст та кожну лексему порівнює з зарезервованими словами якщо є співпадіння то присвоює лексемі відповідний тип або значення, якщо це числова константа.

При виділенні лексеми вона розпізнається та записується у таблицю за допомогою відповідного типу лексеми, що є унікальним для кожної лексеми із усього можливого їх набору. Це дає можливість наступним фазам компіляції звертатись до лексеми не як до послідовності символів, а як до унікального типу лексеми, що значно спрощує роботу синтаксичного аналізатора: легко перевіряти належність лексеми до відповідної синтаксичної конструкції та є можливість легкого перегляду програми, як вгору, так і вниз, від поточної позиції аналізу. Також в таблиці лексем ведуться записи, щодо рядка відповідної лексеми – для місця помилки – та додаткова інформація.

При лексичному аналізі виявляються і відзначаються лексичні помилки (наприклад, недопустимі символи і неправильні ідентифікатори). Лексична фаза відкидає також коментарі, оскільки вони не мають ніякого впливу на виконання програми, отже й на синтаксичний розбір та генерацію коду.

В даному курсовому проекті реалізовано прямий лексичний аналізатор, який виділяє з вхідного тексту програми окремі лексеми і на основі цього формує таблицю.

3.4.Розробка синтаксичного та семантичного аналізатора

Синтаксичний аналізатор - частина компілятора, яка відповідає за виявлення основних синтаксичних конструкцій вхідної мови. У завдання синтаксичного аналізатора входить: знайти і виділити основні синтаксичні конструкції в тексті вхідної програми, встановити тип і перевірити правильність кожної синтаксичної конструкції у вигляді, зручному для подальшої генерації тексту результуючої програми.

В основі синтаксичного аналізатора лежить розпізнавач тексту вхідної програми на основі граматики вхідної мови. Як правило, синтаксичні конструкції мов програмування можуть бути описані за допомогою КС-грамматик, рідше зустрічаються мови, які можуть бути описані за допомогою регулярних граматик. Найчастіше регулярні граматики застосовні до мов асемблера, а мови високого рівня побудовані на основі КС-мов.

Синтаксичний розбір - це основна частина компіляції на етапі аналізу. Без виконання синтаксичного розбору робота компілятора безглузда, у той час як лексичний аналізатор є зовсім необов'язковим. Усі завдання з перевірки лексики вхідної мови можуть бути вирішені на етапі синтаксичного розбору. Сканер тільки дозволяє позбавити складний за структурою лексичний аналізатор від рішення примітивних завдань з виявлення та запам'ятовування лексем вхідної програми.

В даному курсовому проекті синтаксичний аналіз можна виконувати лише після виконання лексичного аналізу, він являється окремим етапом трансляції.

На вході даного аналізатора є файл лексем, який є результатом виконання лексичного аналізу, на базі цього файлу синтаксичний аналізатор формує таблицю ідентифікаторів та змінних.

3.4.1.Опис програми реалізації синтаксичного та семантичного аналізатора

Синтаксичний аналіз - це процес, що визначає, чи належить деяка послідовність лексем граматиці мови програмування. В принципі, для будь-якої граматики можна побудувати синтаксичний аналізатор, але граматики, які використовуються на практиці, мають спеціальну форму. Наприклад, відомо, що для будь-якої контекстно-вільної граматики може бути побудований аналізатор, складність якого не перевищує $O(n^3)$ для вхідного рядка довжиною n , але в більшості випадків для заданої мови програмування ми можемо побудувати таку граматику, що дозволить сконструювати і більш швидкий аналізатор.

Аналізатори реальних мов зазвичай мають лінійну складність; це досягається за рахунок перегляду вхідної програми зліва направо із загляданням уперед на один термінальний символ (лексичний клас).

Вхід синтаксичного аналізатора - це послідовність лексем і таблиці представлень, які є виходом лексичного аналізатора.

На виході синтаксичного аналізатора отримуємо дерево граматичного розбору і таблиці ідентифікаторів та типів, які є входом для наступного перегляду компілятора.

Семантичний аналіз перевіряє змістовну коректність програми, враховуючи правила мови програмування. Він визначає, чи відповідають операції типам даних та чи виконуються всі обмеження мови. Результатом є анотація дерева розбору додатковою інформацією для генерації коду.

На вхід синтаксичного аналізатора подіється таблиця лексем створена на етапі лексичного аналізу. Аналізатор проходить по ній і перевіряє чи набір лексем відповідає раніше описаним формам нотації Бекуса-Наура. І разі не відповідності у файл з помилками виводиться інформація про помилку і про рядок на якій вона знаходиться.

При знаходженні оператора присвоєння або математичних виразів здійснюється перевірка балансу дужок(кількість відкриваючих дужок має дорівнювати кількості закриваючих). Також здійснюється перевірка чи не йдуть підряд декілька лексем одного типу

Результатом синтаксичного аналізу є синтаксичне дерево з посиланнями на таблиці об'єктів. У процесі синтаксичного аналізу також виявляються помилки, пов'язані зі структурою програми.

В основі синтаксичного аналізатора лежить розпізнавач тексту вхідної програми на основі граматики вхідної мови.

Аналізатор працює за принципом рекурсивного спуску, де кожне правило граматики реалізується окремою функцією.

Основні етапи роботи аналізатора:

1. **Ініціалізація:** Виклик функції `Parser()`, яка починає аналіз програми.
2. **Аналіз програми:** Функція `program()` аналізує основну структуру програми, включаючи оголошення змінних та тіло програми.
3. **Аналіз операторів:** Функція `statement()` визначає тип оператора (ввід, вивід, умовний оператор, присвоєння тощо) та викликає відповідну функцію для його аналізу.
4. **Аналіз виразів:** Функції `arithmetic_expression()`, `term()`, `factor()` аналізують арифметичні вирази, включаючи операції додавання, віднімання, множення та ділення.
5. **Аналіз умов:** Функції `logical_expression()`, `and_expression()`, `comparison()` аналізують логічні вирази та операції порівняння.

Основні функції

- **`program()`:** Аналізує основну структуру програми.
- **`variable_declaration()`:** Аналізує оголошення змінних.
- **`variable_list()`:** Аналізує список змінних.
- **`program_body()`:** Аналізує тіло програми.
- **`statement()`:** Визначає тип оператора та викликає відповідну функцію для його аналізу.
- **`assignment()`:** Аналізує оператор присвоєння.
- **`arithmetic_expression()`:** Аналізує арифметичний вираз.
- **`term()`:** Аналізує доданок у виразі.
- **`factor()`:** Аналізує множник у виразі.
- **`input()`:** Аналізує оператор вводу.
- **`output()`:** Аналізує оператор виводу.
- **`conditional()`:** Аналізує умовний оператор.
- **`goto_statement()`:** Аналізує оператор переходу.

- **label_statement()**: Аналізує мітку.
- **for_to_do()**: Аналізує цикл for з інкрементом.
- **for_downto_do()**: Аналізує цикл for з декрементом.
- **while_statement()**: Аналізує цикл while.
- **repeat_until()**: Аналізує цикл repeat until.
- **logical_expression()**: Аналізує логічний вираз.
- **and_expression()**: Аналізує логічний вираз з операцією AND.
- **comparison()**: Аналізує операції порівняння.
- **compound_statement()**: Аналізує складений оператор.

Цей аналізатор забезпечує перевірку синтаксичної коректності програми та виявлення синтаксичних помилок. Якщо виявляється помилка, аналізатор виводить повідомлення про помилку та завершує роботу.

3.4.2. Розробка граф-схеми алгоритму

На рис. 3.2 зображена граф-схема алгоритму роботи синтаксичного аналізатора.

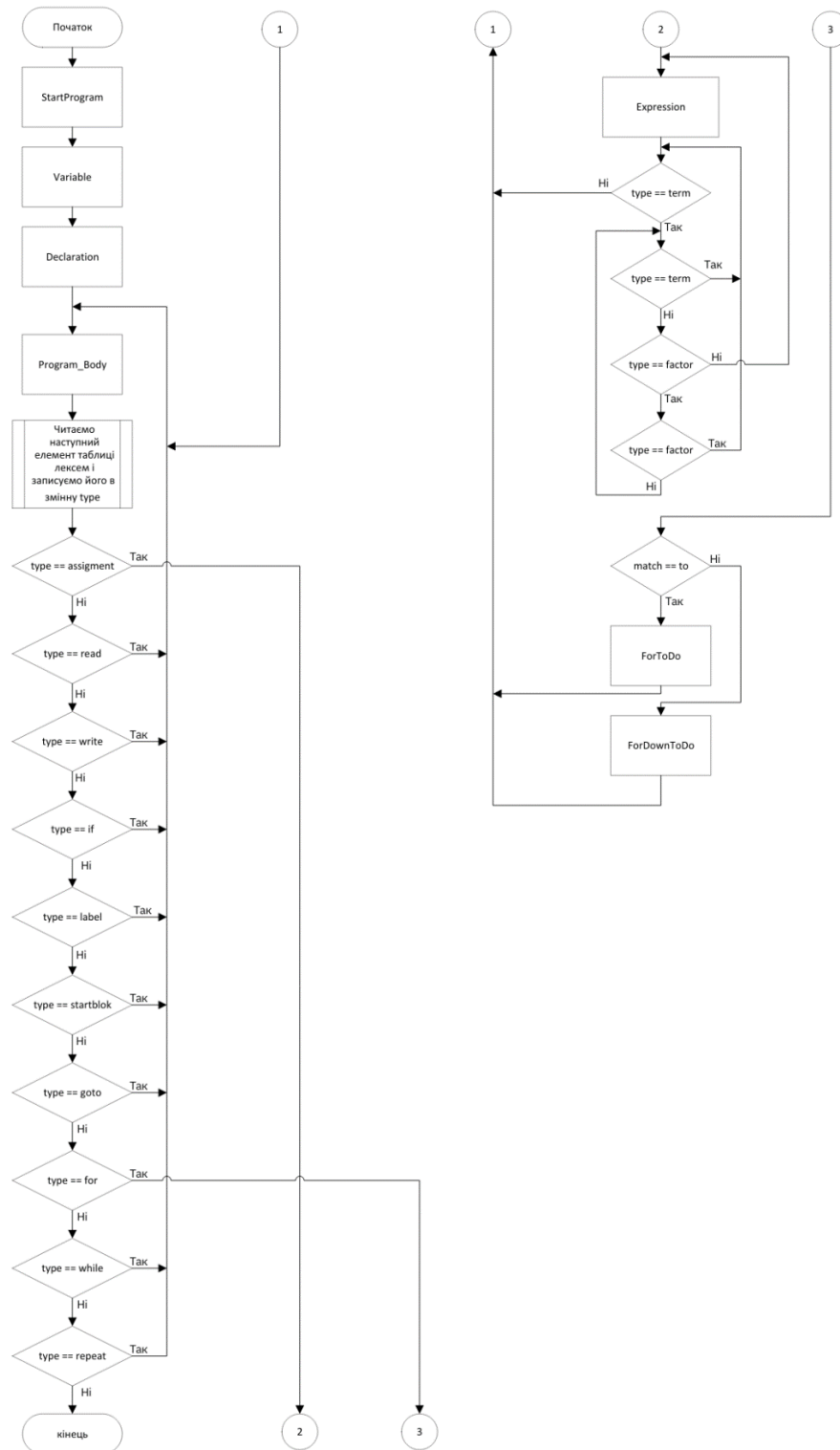


Рис. 3.2 Граф-схема алгоритму роботи синтаксичного аналізатора

3.5.Розробка генератора коду

Синтаксичне дерево в чистому вигляді несе тільки інформацію про структуру програми. Насправді в процесі генерації коду потрібна також інформація про змінні, операції, мітки і т.д. Для представлення цієї інформації можливі різні рішення. Найбільш поширені два:

- інформація зберігається у таблицях генератора коду;
- інформація зберігається у відповідних вершинах дерева.

Розглянемо, наприклад, структуру таблиць, які можуть бути використані в поєднанні з Лідер-представленням. Оскільки Лідер-представлення не містить інформації про адреси змінних, значить, цю інформацію потрібно формувати в процесі обробки оголошень і зберігати в таблицях. Це стосується і описів масивів, записів і т.д. Крім того, в таблицях також повинна міститися інформація про операції.

Генерація коду – це машинно-залежний етап компіляції, під час якого відбувається побудова машинного еквівалента вхідної програми. Зазвичай входом для генератора коду служить проміжна форма представлення програми, а на виході може з'являтися об'єктний код або модуль завантаження.

Генератор С коду приймає масив лексем без помилок. Якщо на двох попередніх етапах виявлено помилки, то ця фаза не виконується.

В даному курсовому проекті генерація коду реалізується як окремий етап. Можливість його виконання є лише за умови, що попередньо успішно виконався етап синтаксичного аналізу. І використовує результат виконання попереднього аналізу, тобто два файли: перший містить згенерований С код відповідно операторам які були в програмі, другий файл містить таблицю змінних.

3.5.1.Розробка граф-схеми алгоритму

На рис. 3.3 зображена граф-схема алгоритму роботи генератора коду.

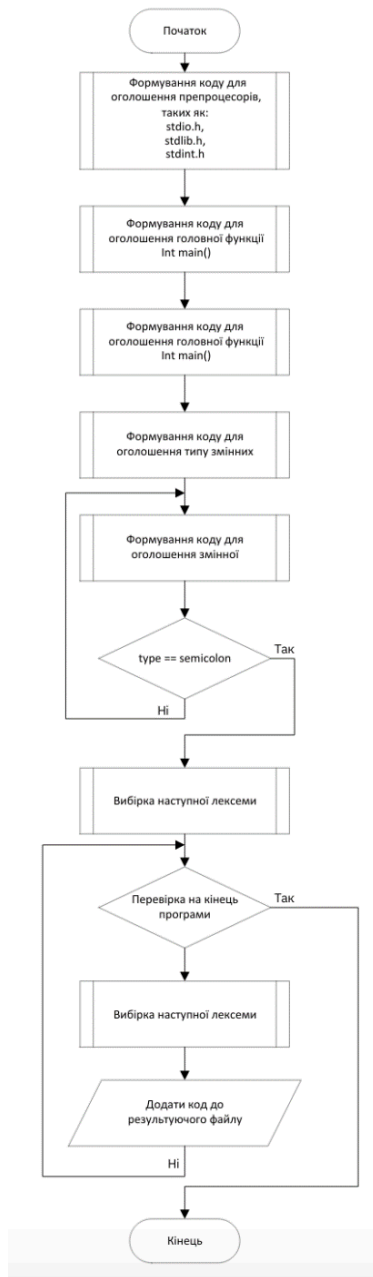


Рис. 3.3 Граф-схема алгоритму роботи генератора коду

3.5.2.Опис програми реалізації генератора коду

У компілятора, реалізованого в даному курсовому проєкті, вихідна мова - програма на мові C. Ця програма записується у файл, що має таку ж саму назву, як і файл з вхідним текстом, але розширення “.c”. Генерація коду відбувається одразу ж після синтаксичного аналізу.

В даному трансляторі генератор коду послідовно викликає окремі функції, які записують у вихідний файл частини коду.

Першим кроком генерації коду записується заголовки, необхідні для програми на C, та визначається основна функція `main()`. Далі виконується аналіз коду та визначаються змінні, які використовуються.

Проаналізувавши змінні, які є у програмі, генератор формує секцію оголошення змінних для програми на C. Для цього з таблиці лексем вибирається ім'я змінної (типи змінних відповідають типам у C, наприклад `int`), та записується її початкове значення, якщо воно задано.

Аналіз наявних операторів необхідний у зв'язку з тим, що введення/виведення, виконання арифметичних та логічних операцій виконуються як окремі конструкції, і у випадку їх відсутності немає сенсу записувати у вихідний файл зайву інформацію.

Після цього зчитується лексема з таблиці лексем. Також відбувається перевірка, чи це не остання лексема. Якщо це остання лексема, то функція завершується.

Наступним кроком є аналіз таблиці лексем та безпосередня генерація коду у відповідності до вхідної програми.

Генератор коду зчитує лексему та генерує відповідний код, який записується у файл. Наприклад, якщо це лексема виведення, то у основну програму записується виклик функції `printf`, яка формує вихідний текст. Якщо це арифметична операція, то у вихідний файл записується вираз, що відповідає правилам C, із врахуванням пріоритетів операцій.

Генератор закінчує свою роботу, коли зчитує лексему, що відповідає кінцю файлу.

В кінці своєї роботи генератор формує завершення програми на C, додаючи повернення значення 0 з основної функції.

4. Опис програми

Дана програма написана мовою C++ з використанням визначень нових типів та перелічень:

```
enum TypeOfTokens
{
    Mainprogram,
    StartProgram,
    Variable,
    Type,
    EndProgram,
    Input,
    Output,

    If,
    Else,

    Goto,
    Label,

    For,
    To,
    DownTo,
    Do,

    While,
    Exit,
    Continue,
    End,

    Repeat,
    Until,

    Identifier,
    Number,
    Assign,
    Add,
    Sub,
    Mul,
    Div,
    Mod,
    Equality,
    NotEquality,
    Greate,
    Less,
    Not,
    And,
    Or,
    LBraket,
    RBraket,
```

```

    Semicolon,
    Colon,
    Comma,
    Unknown
};

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

// структура для зберігання інформації про ідентифікатор
struct Id
{
    char name[16];
};

// 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
};

```

Спочатку вхідна програма за допомогою функції `unsigned int GetTokens(FILE* F, Token TokenTable[])` розбивається на відповідні токени для запису у таблицю та подальше їх використання в процесі синтаксичного аналізу та генерації коду.

Далі відбувається синтаксичний аналіз вхідної програми за допомогою функції `void Parser()`. Всі правила запису як різноманітних операцій так і програми в цілому відбувається за нотатками Бекуса-Наура, за допомогою яких можна легко описати синтаксис всіх операцій.

Нище наведено опис структури програми за допомогою нотаток Бекуса-Наура.

```
void program()
```

```

{
    match(Mainprogram);
    match(StartProgram);
    match(Variable);
    variable_declaration();
    match(Semicolon);
    program_body();
    match(EndProgram);
}

```

Наступним етапом є генерація С коду. Алгоритм генерації працює за принципом синтаксичного аналізу але при вибірці певної лексеми або операції генерує відповідний С код який записується у вихідний файл.

Нище наведено генерацію С коду на прикладі операції присвоєння.

```

void assignment(FILE* outFile)
{
    fprintf(outFile, "    ");
    fprintf(outFile, TokenTable[pos++].name);
    fprintf(outFile, " = ");
    pos++;
    arithmetic_expression(outFile);
    pos++;
    fprintf(outFile, ";\n");
}

```

Така структура програми дозволяє без проблем аналізувати великі програми, написані на вхідній мові програмування. Також використання правил Бекуса-Наура дозволяє ефективно аналізувати програми великого обсягу.

4.1.Опис інтерфейсу та інструкція користувачеві

Вхідним файлом для даної програми є звичайний текстовий файл з розширенням c23. У цьому файлі необхідно набрати бажану для трансляції програму та зберегти її. Синтаксис повинен відповідати вхідній мові.

Створений транслятор є консольною програмою, що запускається з командної стрічки з параметром: "CWork_c23.exe <ім'я програми>.c23"

Якщо обидва файли мають місце на диску та правильно сформовані, програма буде запущена на виконання.

Початковою фазою обробки є лексичний аналіз (розбиття на окремі лексеми). Результатом цього етапу є файл lexems.txt, який містить таблицю лексем. Вміст цього файлу складається з 4 полів – 1 – безпосередньо сама лексема; 2 – тип лексеми; 3 – значення лексеми (необхідне для чисел і ідентифікаторів); 4 – рядок, у якому лексема знаходиться. Наступним етапом є перевірка на правильність написання програми (вхідної). Інформацію про наявність чи відсутність помилок можна переглянути у файлі error.txt. Якщо граматичний розбір виконаний успішно, файл буде містити відповідне повідомлення. Інакше, у файлі будуть зазначені помилки з їх описом та вказанням їх місця у тексті програми.

Останнім етапом є генерація коду. Транслятор переходить до цього етапу, лише у випадку, коли відсутні граматичні помилки у вхідній програмі. Згенерований код записується у файлу <ім'я програми>.c.

5. Відлагодження та тестування програми

Тестування програмного забезпечення є важливим етапом розробки продукту. На цьому етапі знаходяться помилки допущені на попередніх етапах. Цей етап дозволяє покращити певні характеристики продукту, наприклад – інтерфейс. Дає можливість знайти та вподальшому виправити слабкі сторони, якщо вони є.

Відлагодження даної програми здійснюється за допомогою набору кількох програм, які відповідають заданій граматиці. Та перевірка коректності коду, що генерується, коректності знаходження помилок та розбивки на лексеми.

5.1. Виявлення лексичних та синтаксичних помилок

Виявлення лексичних помилок відбувається на стадії лексичного аналізу. Під час розбиття вхідної програми на окремі лексеми відбувається перевірка чи відповідає вхідна лексема граматиці. Якщо ця лексема є в граматиці то вона ідентифікується і в таблиці лексем визначається. У випадку неспівпадіння лексемі присвоюється тип "невпізнаної лексеми". Повідомлення про такі помилки можна побачити лише після виконання процедури перевірки таблиці лексем, яка знаходиться в файлі.

Виявлення синтаксичних помилок відбувається на стадії перевірки програми на коректність окремо від синтаксичного аналізу. При цьому перевіряється окремо кожне твердження яке може бути або виразом, або оператором (циклу, вводу/виводу), або оголошенням, та перевіряється структура програми в цілому.

Приклад виявлення:

Текст програми з помилками

```
\\Prog1
Name prog1;
Data Longint _aAAAAA
AAAAAAAAAAAAA, _bBBBBBBBBBBBBBBBBB, _xxxxxxxxxxxxxxxxxx, _yyyyyyyyy
yyyyyyyyy;
Body
Read _aAAAAAAAAAAAAAAAAA
Read _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA ++ _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA -- _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA ** _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA Div _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA Mod _bBBBBBBBBBBBBBBBBB;
```



```

_y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y < == _x X X X X X X X X X X X X X X X X ++
(_xXXXXXXXXXXXXXXXXXXXXX Mod 10);
Write _xXXXXXXXXXXXXXXXXXXXXX;
Write _yYYYYYYYYYYYYYYYYYYY;
End

```

Оскільки дана програма відповідає граматиці то результати виконання лексичного, синтаксичного аналізів, а також генератора коду будуть позитивними.

В результаті буде отримано с файл, який є результатом виконання трансляції з заданої вхідної мови на мову С даної програми (його вміст наведений в Додатку А).

Після виконання компіляції даного файлу на виході отримаєм наступний результат роботи програми:

```

Enter _aAAAAAAAAAAAAAAAAA:5
Enter _bBBBBBBBBBBBBBBBBB:9
14
-4
45
0
5
-39
-48

```

Рис. 5.1 Результат виконання коректної програми

При перевірці отриманого результату, можна зробити висновок про правильність роботи програми, а отже і про правильність роботи транслятора.

5.4.Тестова програма №1

Текст програми

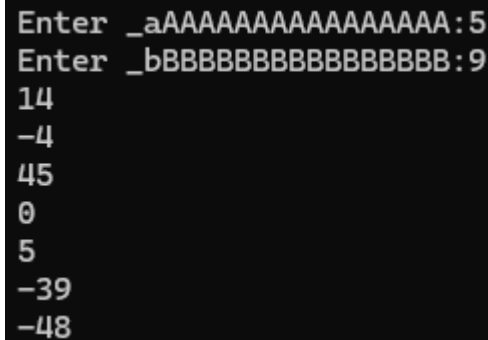
```

\\Prog1
Name prog1;
Data Longint
_aAAAAAAAAAAAAAAAAA,_BBBBBBBBBBBBBBBBBB,_XXXXXXXXXXXXXXXXXX,_YY
YYYYYYYYYYYYYYY;
Body
Read _aAAAAAAAAAAAAAAAAA;
Read _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA ++ _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA -- _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA ** _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA Div _bBBBBBBBBBBBBBBBBB;
Write _aAAAAAAAAAAAAAAAAA Mod _bBBBBBBBBBBBBBBBBB;

_xXXXXXXXXXXXXXXXXX<==(_aAAAAAAAAAAAAAAAAA --
_bBBBBBBBBBBBBBBBBB) ** 10 ++ (_aAAAAAAAAAAAAAAAAA ++
_bBBBBBBBBBBBBBBBBB) Div 10;
_yYYYYYYYYYYYYYYY<==_xXXXXXXXXXXXXXXXXX ++
(_XXXXXXXXXXXXXXXXX Mod 10);
Write _xXXXXXXXXXXXXXXXXX;
Write _yYYYYYYYYYYYYYYY;
End

```

Результат виконання



```

Enter _aAAAAAAAAAAAAAAAAA: 5
Enter _bBBBBBBBBBBBBBBBBB: 9
14
-4
45
0
5
-39
-48

```

Рис. 5.2 Результат виконання тестової програми №1

5.5.Тестова програма №2

Текст програми

```

\\Prog2
Name prog2;
Data Longint
_aAAAAAAAAAAAAAAAAA,_BBBBBBBBBBBBBBBBB,_cCCCCCCCCCCCCCCC;
Body
Read _aAAAAAAAAAAAAAAAAA;
Read _BBBBBBBBBBBBBBBBB;
Read _cCCCCCCCCCCCCCCC;
If(_aAAAAAAAAAAAAAAAAA >= _BBBBBBBBBBBBBBBBB)
Body
    If(_aAAAAAAAAAAAAAAAAA >= _cCCCCCCCCCCCCCCC)
    Body
        Goto Abigger;
    End
    Else
    Body
        Write _cCCCCCCCCCCCCCCC;
        Goto Outofif;
        Abigger:
        Write _aAAAAAAAAAAAAAAAAA;
        Goto Outofif;
    End
End
If(_BBBBBBBBBBBBBBBBB <= _cCCCCCCCCCCCCCCC)
Body
    Write _cCCCCCCCCCCCCCCC;
End
Else
Body
    Write _BBBBBBBBBBBBBBBBB;
End
Outofif:

If(( _aAAAAAAAAAAAAAAAAA = _BBBBBBBBBBBBBBBBB) &&
( _aAAAAAAAAAAAAAAAAA = _cCCCCCCCCCCCCCCC) && ( _BBBBBBBBBBBBBBBBB
= _cCCCCCCCCCCCCCCC))
Body
    Write 1;
End
Else
Body

```

```

        Write 0;
End
If((_aAAAAAAAAAAAAAAAAA <= 0) || (_bBBBBBBBBBBBBBBBBB <= 0) ||
(_cCCCCCCCCCCCCCCCCC <= 0))
Body
    Write -1;
End
Else
Body
    Write 0;
End
If(!( (_aAAAAAAAAAAAAAAAAA <= (_bBBBBBBBBBBBBBBBBB ++
_cCCCCCCCCCCCCCCCCC)))
Body
    Write(10);
End
Else
Body
    Write(0);
End
End

```

Результат виконання

```

Enter _aAAAAAAAAAAAAAAAAA: 5
Enter _bBBBBBBBBBBBBBBBBB: 9
Enter _cCCCCCCCCCCCCCCCCC: -10
9
0
-1
10

```

Рис. 5.3 Результат виконання тестової програми №2

5.6. Тестова програма №3

Текст програми

```

\\Prog3
Name prog3;
Data Longint
_aAAAAAAAAAAAAAAAAA, _aAAAAAAAAAAAAAAAAA2, _bBBBBBBBBBBBBBBBBB, _xXX
XXXXXXXXXXXXXXXXXX, _cCCCCCCCCCCCCCCC1, _cCCCCCCCCCCCCCCC2;
Body
Read _aAAAAAAAAAAAAAAAAA;
Read _bBBBBBBBBBBBBBBBBB;

```

```

For _aAAAAAAAAAAAAAAAAA2<==_aAAAAAAAAAAAAAAAAA To
_bBBBBBBBBBBBBBBBBB Do
    Write _aAAAAAAAAAAAAAAAAA2 ** _aAAAAAAAAAAAAAAAAA2;

For _aAAAAAAAAAAAAAAAAA2<==_bBBBBBBBBBBBBBBBBB To
_aAAAAAAAAAAAAAAAAA Do
    Write _aAAAAAAAAAAAAAAAAA2 ** _aAAAAAAAAAAAAAAAAA2;

_xXXXXXXXXXXXXXXXXX<==0;
_cCCCCCCCCCCCCCCC1<==0;
While _cCCCCCCCCCCCCCCC1 <= _aAAAAAAAAAAAAAAAAA
Body
    _cCCCCCCCCCCCCCCC2<==0;
    While _cCCCCCCCCCCCCCCC2 <= _bBBBBBBBBBBBBBBBBB
    Body
        _xXXXXXXXXXXXXXXXXX<==_xXXXXXXXXXXXXXXXXX ++ 1;
        _cCCCCCCCCCCCCCCC2<==_cCCCCCCCCCCCCCCC2 ++ 1;
    End
    End While
    _cCCCCCCCCCCCCCCC1<==_cCCCCCCCCCCCCCCC1 ++ 1;
End
End While
Write _xXXXXXXXXXXXXXXXXX;

_xXXXXXXXXXXXXXXXXX<==0;
_cCCCCCCCCCCCCCCC1<==1;
Repeat
Body
    _cCCCCCCCCCCCCCCC2<==1;
    Repeat
    Body
        _xXXXXXXXXXXXXXXXXX<==_xXXXXXXXXXXXXXXXXX++1;
        _cCCCCCCCCCCCCCCC2<==_cCCCCCCCCCCCCCCC2++1;
    End
    Until !!( _cCCCCCCCCCCCCCCC2 >= _bBBBBBBBBBBBBBBBBB)
    _cCCCCCCCCCCCCCCC1<==_cCCCCCCCCCCCCCCC1++1;
End
Until !!( _cCCCCCCCCCCCCCCC1 >= _aAAAAAAAAAAAAAAAAA)
Write _xXXXXXXXXXXXXXXXXX;

End

```

Результат виконання

```

Enter _aAAAAAAAAAAAAAAAAA:5
Enter _bBBBBBBBBBBBBBBBBB:9
25
36
49
64
81
45
45

```

Рис. 5.4 Результат виконання тестової програми №3

6. Верифікація тестових програм

Для верифікації наших тестових програм ми використовували Boost.Spirit. Boost.Spirit — це бібліотека, яка є частиною набору бібліотек Boost для мови програмування C++. Вона призначена для створення парсерів (аналізаторів) і граматик, які можуть використовуватися для аналізу текстових даних. Основною перевагою Boost.Spirit є те, що вона дозволяє писати парсери безпосередньо в C++ коді, використовуючи декларативний підхід, схожий на формальні граматики.

Основні можливості Boost.Spirit:

- Побудова граматик: Використовуючи Boost.Spirit, можна створювати граматики з використанням C++ виразів, які виглядають схоже на контекстно-вільні граматики (CFG).
- Компактність: Граматики визначаються безпосередньо у коді, без необхідності використовувати зовнішні файли.
- Підтримка синтаксичних та семантичних дій: Можна виконувати додаткові операції під час розбору тексту (наприклад, обробка даних чи збереження результатів).
- Модульність: Граматики можуть бути розбиті на менші компоненти, що спрощує їх повторне використання.
- Компоненти Boost.Spirit:
- Qi (Query Interface): Використовується для створення парсерів, які аналізують вхідні дані та перевіряють їх на відповідність заданій граматиці.
- Lex: Інструмент для токенизації (розбиття тексту на окремі елементи — токени).

Висновки

У межах курсового проекту було розроблено транслятор вхідної мови програмування, який виконує такі завдання:

1. Лексичний аналіз

- Текст програми розбивається на лексеми з подальшим формуванням таблиці, що містить тип, значення та номер рядка кожної лексеми.
- Лексичний аналізатор працює за принципом скінченного автомату, розпізнаючи ключові слова, ідентифікатори, константи, оператори та розділювачі.

2. Синтаксичний і семантичний аналіз

- **Синтаксичний аналізатор** перевіряє відповідність структури програми заданій граматиці, будує дерево розбору, а також формує таблиці ідентифікаторів і типів.
- **Семантичний аналізатор** аналізує логічну коректність програми: перевіряє відповідність типів даних, області видимості змінних і коректність викликів функцій.

3. Генерація коду

- На основі абстрактного дерева генератор коду створює вихідний текст на мові C, здійснюючи обхід дерева та генеруючи відповідний код для кожного вузла.

4. Тестування

- Було проведено тестування на різноманітних програмах (лінійні алгоритми, конструкції з розгалуженнями та циклами).
- Виявлені лексичні, синтаксичні й семантичні помилки успішно усунуті.
- Транслятор генерує коректний код на основі введених програм.

Висновок:

Розроблений під час виконання курсового проекту транслятор демонструє базову функціональність і може слугувати основою для подальшого розвитку та вдосконалення.

Список використаної літератури

1. Language Processors: Assembler, Compiler and Interpreter URL: [Language Processors: Assembler, Compiler and Interpreter - GeeksforGeeks](#)
2. Error Handling in Compiler Design URL: [Error Handling in Compiler Design - GeeksforGeeks](#)
3. Symbol Table in Compiler URL: [Symbol Table in Compiler - GeeksforGeeks](#)
4. Stack Overflow URL: [Stack Overflow - Where Developers Learn, Share, & Build Careers](#)
5. Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 – «Комп’ютерна інженерія» / О. І. Марченко ; КПП ім. Ігоря Сікорського. – Київ: КПП ім. Ігоря Сікорського, 2021. – 108 с.
6. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. – Харків: НТУ «ХП», 2021. – 133 с.
7. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії формальних мов:
8. Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 с.
9. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 с.
10. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. – 1038 с.
11. Системне програмування (курсний проєкт) [Електронний ресурс] – Режим доступу до ресурсу: <https://vns.lpnu.ua/course/view.php?id=11685>.
12. MIT OpenCourseWare. Computer Language Engineering [Електронний ресурс] – Режим доступу до ресурсу: <https://ocw.mit.edu/courses/6-035-computer-language-engineering-spring-2010>.
13. Рисований О.М. Системне програмування: підручник для студентів напрямку “Комп’ютерна інженерія” вищих навчальних закладів в 2-х томах. Том 1. – Видання четверте: виправлено та доповнено – Х.: “Слово”, 2015. – 576 с.
14. Рисований О.М. Системне програмування: підручник для студентів напрямку “Комп’ютерна інженерія” вищих навчальних закладів в 2-х

томах. Том 2. – Видання четверте: виправлено та доповнено – Х.:
“Слово”, 2015. – 378 с.

Додатки

Додаток А (Код на мові С, отриманий на виході транслятора для тестових прикладів)

Prog1.c

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

int main()
{
    int _aAAAAAAAAAAAAAAAAA, _bBBBBBBBBBBBBBBBBB,
    _xXXXXXXXXXXXXXXXXX, _yYYYYYYYYYYYYYYYYY;
    printf("Enter _aAAAAAAAAAAAAAAAAA:");
    scanf("%d", &_aAAAAAAAAAAAAAAAAA);
    printf("Enter _bBBBBBBBBBBBBBBBBB:");
    scanf("%d", &_bBBBBBBBBBBBBBBBBB);
    printf("%d\n", _aAAAAAAAAAAAAAAAAA + _bBBBBBBBBBBBBBBBBB);
    printf("%d\n", _aAAAAAAAAAAAAAAAAA - _bBBBBBBBBBBBBBBBBB);
    printf("%d\n", _aAAAAAAAAAAAAAAAAA * _bBBBBBBBBBBBBBBBBB);
    printf("%d\n", _aAAAAAAAAAAAAAAAAA / _bBBBBBBBBBBBBBBBBB);
    printf("%d\n", _aAAAAAAAAAAAAAAAAA % _bBBBBBBBBBBBBBBBBB);
    _xXXXXXXXXXXXXXXXXX = (_aAAAAAAAAAAAAAAAAA -
    _bBBBBBBBBBBBBBBBBB) * 10 + (_aAAAAAAAAAAAAAAAAA +
    _bBBBBBBBBBBBBBBBBB) / 10;
    _yYYYYYYYYYYYYYYYYY = _xXXXXXXXXXXXXXXXXX +
    (_xXXXXXXXXXXXXXXXXX % 10);
    printf("%d\n", _xXXXXXXXXXXXXXXXXX);
    printf("%d\n", _yYYYYYYYYYYYYYYYYY);
    system("pause");
    return 0;
}
```

Prog2.c

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

int main()
{
    int _aAAAAAAAAAAAAAAAAA, _bBBBBBBBBBBBBBBBBB,
    _cCCCCCCCCCCCCCCC;
    printf("Enter _aAAAAAAAAAAAAAAAAA:");
    scanf("%d", &_aAAAAAAAAAAAAAAAAA);
    printf("Enter _bBBBBBBBBBBBBBBBBB:");
    scanf("%d", &_bBBBBBBBBBBBBBBBBB);
    printf("Enter _cCCCCCCCCCCCCCCC:");
    scanf("%d", &_cCCCCCCCCCCCCCCC);
    if ((_aAAAAAAAAAAAAAAAAA > _bBBBBBBBBBBBBBBBBB))
```

```

{
if ((_aAAAAAAAAAAAAAAAAA > _cCCCCCCCCCCCCCCCCC))
{
goto Abigger;
}
else
{
printf("%d\n", _cCCCCCCCCCCCCCCCCC);
goto Outofif;
}
Abigger:
printf("%d\n", _aAAAAAAAAAAAAAAAAA);
goto Outofif;
}
}
if ((_bBBBBBBBBBBBBBBBBBB < _cCCCCCCCCCCCCCCCCC))
{
printf("%d\n", _cCCCCCCCCCCCCCCCCC);
}
else
{
printf("%d\n", _bBBBBBBBBBBBBBBBBBB);
}
Outofif:
if (((_aAAAAAAAAAAAAAAAAA == _bBBBBBBBBBBBBBBBBBB) &&
(_aAAAAAAAAAAAAAAAAA == _cCCCCCCCCCCCCCCCCC) &&
(_bBBBBBBBBBBBBBBBBBB == _cCCCCCCCCCCCCCCCCC)))
{
printf("%d\n", 1);
}
else
{
printf("%d\n", 0);
}
if (((_aAAAAAAAAAAAAAAAAA < 0) || (_bBBBBBBBBBBBBBBBBBB < 0) ||
(_cCCCCCCCCCCCCCCCCC < 0)))
{
printf("%d\n", -1);
}
else
{
printf("%d\n", 0);
}
if (!((_aAAAAAAAAAAAAAAAAA < (_bBBBBBBBBBBBBBBBBBB +
_cCCCCCCCCCCCCCCCCC))))
{
printf("%d\n", (10));
}
}

```



```

do
{
_cCCCCCCCCCCCCCCCCC2 = 1;
do
{
_xXXXXXXXXXXXXXXXXX = _xXXXXXXXXXXXXXXXXX + 1;
_cCCCCCCCCCCCCCCCCC2 = _cCCCCCCCCCCCCCCCCC2 + 1;
}
while (!(_cCCCCCCCCCCCCCCCCC2 > _bBBBBBBBBBBBBBBBBB));
_cCCCCCCCCCCCCCCCCC1 = _cCCCCCCCCCCCCCCCCC1 + 1;
}
while (!(_cCCCCCCCCCCCCCCCCC1 > _aAAAAAAAAAAAAAAAAA));
printf("%d\n", _xXXXXXXXXXXXXXXXXX);
system("pause");
return 0;
}

```

Додаток Б Документований текст програмних модулів (лістинги):

main.cpp

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "translator.h"

// таблиця лексем
Token* TokenTable;
// кількість лексем
unsigned int TokensNum;

// таблиця ідентифікаторів
Id* IdTable;
// кількість ідентифікаторів
unsigned int IdNum;

// Function to validate file extension
int isValidExtension(const char* fileName, const char*
extension)
{
    const char* dot = strrchr(fileName, '.');
    if (!dot || dot == fileName) return 0; // No extension found
    return strcmp(dot, extension) == 0;
}

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

```

```

// виділення пам'яті під таблицю лексем
TokenTable = new Token[MAX_TOKENS];

// виділення пам'яті під таблицю ідентифікаторів
IdTable = new Id[MAX_IDENTIFIER];

char InputFile[32] = "";

FILE* InFile;

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

// Check if the input file has the correct extension
if (!isValidExtension(InputFile, ".c23"))
{
    printf("Error: Input file has invalid extension.\n");
    return 1;
}

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

char NameFile[32] = "";
int i = 0;
while (InputFile[i] != '.' && InputFile[i] != '\0')
{
    NameFile[i] = InputFile[i];
    i++;
}
NameFile[i] = '\0';

char TokenFile[32];
strcpy_s(TokenFile, NameFile);
strcat_s(TokenFile, ".token");

char ErrFile[32];
strcpy_s(ErrFile, NameFile);
strcat_s(ErrFile, "_errors.txt");

```

```

FILE* errFile;
if (fopen_s(&errFile, ErrFile, "w") != 0)
{
    printf("Error: Cannot open file for writing: %s\n",
ErrFile);
    return 1;
}

TokensNum = GetTokens(InFile, TokenTable, errFile);

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

printf("\nLexical analysis completed: %d tokens. List of
tokens in the file %s\n", TokensNum, TokenFile);
printf("\nList of errors in the file %s\n", ErrFile);

Parser(errFile);
fclose(errFile);
ASTNode* ASTree = ParserAST();

char AST[32];
strcpy_s(AST, NameFile);
strcat_s(AST, ".ast");
// Open output file
FILE* ASTFile;
fopen_s(&ASTFile, AST, "w");
if (!ASTFile)
{
    printf("Failed to open output file.\n");
    exit(1);
}
PrintASTToFile(ASTree, 0, ASTFile);
printf("\nAST has been created and written to %s.\n", AST);

char OutputFile[32];
strcpy_s(OutputFile, NameFile);
strcat_s(OutputFile, ".c");

FILE* outFile;
fopen_s(&outFile, OutputFile, "w");
if (!outFile)
{
    printf("Failed to open output file.\n");
    exit(1);
}
// генерація вихідного C коду
generateCCode(outFile);
printf("\nC code has been generated and written to %s.\n",
OutputFile);

```

```

fclose(outFile);

fopen_s(&outFile, OutputFile, "r");
char ExecutableFile[32];
strcpy_s(ExecutableFile, NameFile);
strcat_s(ExecutableFile, ".exe");
compile_to_exe(OutputFile, ExecutableFile);

char OutputFileFromAST[32];
strcpy_s(OutputFileFromAST, NameFile);
strcat_s(OutputFileFromAST, "_fromAST.c");

FILE* outFileFromAST;
fopen_s(&outFileFromAST, OutputFileFromAST, "w");
if (!outFileFromAST)
{
    printf("Failed to open output file.\n");
    exit(1);
}
generateCodefromAST(ASTree, outFileFromAST);
printf("\nC code has been generated and written to %s.\n",
OutputFileFromAST);

fclose(outFileFromAST);

fopen_s(&outFileFromAST, OutputFileFromAST, "r");
char ExecutableFileFromAST[32];
strcpy_s(ExecutableFileFromAST, NameFile);
strcat_s(ExecutableFileFromAST, "_fromAST.exe");
compile_to_exe(OutputFileFromAST, ExecutableFileFromAST);

// Close the file
_fcloseall();

destroyTree(ASTree);

delete[] TokenTable;
delete[] IdTable;

return 0;
}

```

translator.h

```

#pragma once

#define MAX_TOKENS 1000
#define MAX_IDENTIFIER 10

```



```
// перерахування, яке описує всі можливі типи лексем
enum TypeOfTokens
{
    Mainprogram,
    ProgramName,
    StartProgram,
    Variable,
    Type,
    EndProgram,
    Input,
    Output,

    If,
    Then,
    Else,

    Goto,
    Label,

    For,
    To,
    DownTo,
    Do,

    While,
    Exit,
    Continue,
    End,

    Repeat,
    Until,

    Identifier,
    Number,
    Assign,
    Add,
    Sub,
    Mul,
    Div,
    Mod,
    Equality,
    NotEquality,
    Greate,
    Less,
    Not,
    And,
    Or,
    LBraket,
    RBraket,
```

```

    Semicolon,
    Colon,
    Comma,
    Minus,
    Unknown
};

// структура для зберігання інформації про лексему
struct Token
{
    char name[32];          // ім'я лексеми
    int value;              // значення лексеми (для цілих констант)
    int line;              // номер рядка
    TokenType type;        // тип лексеми
};

// структура для зберігання інформації про ідентифікатор
struct Id
{
    char name[32];
};

// 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
};

// перерахування, яке описує всі можливі вузли абстрактного
синтаксичного дерева
enum TokenType
{
    program_node,
    var_node,
    input_node,
    output_node,

    if_node,
    then_node,

```

```

goto_node,
label_node,

for_to_node,
for_downto_node,

while_node,
exit_while_node,
continue_while_node,

repeat_until_node,

id_node,
num_node,
assign_node,
add_node,
sub_node,
mul_node,
div_node,
mod_node,
or_node,
and_node,
not_node,
cmp_node,
statement_node,
compount_node
};

// структура, яка описує вузол абстрактного синтаксичного дерева
// (AST)
struct ASTNode
{
    TypeOfNodes nodetype;    // Тип вузла
    char name[32];           // Ім'я вузла
    struct ASTNode* left;    // Лівий нащадок
    struct ASTNode* right;   // Правий нащадок
};

// 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[], FILE*
errFile);

// функція друкує таблицю лексем на екран
void PrintTokens(Token TokenTable[], unsigned int TokensNum);

// функція друкує таблицю лексем у файл

```

```

void PrintTokensToFile(char* FileName, Token TokenTable[],
unsigned int TokensNum);

// синтаксичний аналіз методом рекурсивного спуску
// вхідні дані – глобальна таблиця лексем TokenTable
void Parser(FILE* errFile);

// функція синтаксичного аналізу і створення абстрактного
синтаксичного дерева
ASTNode* ParserAST();

// функція знищення дерева
void destroyTree(ASTNode* root);

// функція для друку AST у вигляді дерева на екран
void PrintAST(ASTNode* node, int level);

// функція для друку AST у вигляді дерева у файл
void PrintASTToFile(ASTNode* node, int level, FILE* outFile);

// Рекурсивна функція для генерації коду з AST
void generateCodefromAST(ASTNode* node, FILE* output);

// функція для генерації коду
void generateCCode(FILE* outFile);

void compile_to_exe(const char* source_file, const char*
output_file);

```

ast.cpp

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "translator.h"
#include <iostream>

// таблиця лексем
extern Token* TokenTable;
// кількість лексем
extern unsigned int TokensNum;

static int pos = 0;

// функція створення вузла AST
ASTNode* createNode(TypeOfNodes type, const char* name, ASTNode*
left, ASTNode* right)
{
    ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));

```

```

        node->nodetype = type;
        strcpy_s(node->name, name);
        node->left = left;
        node->right = right;
        return node;
    }

    // функція знищення дерева
    void destroyTree(ASTNode* root)
    {
        if (root == NULL)
            return;

        // Рекурсивно знищуємо ліве і праве піддерево
        destroyTree(root->left);
        destroyTree(root->right);

        // Звільняємо пам'ять для поточного вузла
        free(root);
    }

    // набір функцій для рекурсивного спуску
    // на кожне правило – окрема функція
    ASTNode* program();
    ASTNode* variable_declaration();
    ASTNode* variable_list();
    ASTNode* program_body();
    ASTNode* statement();
    ASTNode* assignment();
    ASTNode* arithmetic_expression();
    ASTNode* term();
    ASTNode* factor();
    ASTNode* input();
    ASTNode* output();
    ASTNode* conditional();

    ASTNode* goto_statement();
    ASTNode* label_statement();
    ASTNode* for_to_do();
    ASTNode* for_downto_do();
    ASTNode* while_statement();
    ASTNode* repeat_until();

    ASTNode* logical_expression();
    ASTNode* and_expression();
    ASTNode* comparison();
    ASTNode* compound_statement();

    // функція синтаксичного аналізу і створення абстрактного
    синтаксичного дерева

```

```

ASTNode* ParserAST()
{
    ASTNode* tree = program();

    printf("\nParsing completed. AST created.\n");

    return tree;
}

static void match(TypeOfTokens expectedType)
{
    if (TokenTable[pos].type == expectedType)
        pos++;
    else
    {
        printf("\nSyntax error in line %d: Expected another type
of lexeme.\n", TokenTable[pos].line);
        std::cout << "AST Type: " << TokenTable[pos].type <<
std::endl;
        std::cout << "AST Expected type:" << expectedType <<
std::endl;
        exit(10);
    }
}

// <програма> = 'start' 'var' <оголошення змінних> ';' <тіло
програми> 'stop'
ASTNode* program()
{
    match(Mainprogram);
    match(ProgramName);
    match(Variable);
    ASTNode* declarations = variable_declaration();
    match(Semicolon);
    match(StartProgram);
    ASTNode* body = program_body();
    match(EndProgram);
    return createNode(program_node, "program", declarations,
body);
}

// <оголошення змінних> = [<тип даних> <список змінних>]
ASTNode* variable_declaration()
{
    if (TokenTable[pos].type == Type)
    {
        pos++;
        return variable_list();
    }
    return NULL;
}

```

```

}

// <список змінних> = <ідентифікатор> { ',' <ідентифікатор> }
ASTNode* variable_list()
{
    match(Identifier);
    ASTNode* id = createNode(id_node, TokenTable[pos - 1].name,
NULL, NULL);
    ASTNode* list = list = createNode(var_node, "var", id,
NULL);
    while (TokenTable[pos].type == Comma)
    {
        match(Comma);
        match(Identifier);
        id = createNode(id_node, TokenTable[pos - 1].name, NULL,
NULL);
        list = createNode(var_node, "var", id, list);
    }
    return list;
}

// <тіло програми> = <оператор> ';' { <оператор> ';' }
ASTNode* program_body()
{
    ASTNode* stmt = statement();
    //match(Semicolon);
    ASTNode* body = stmt;
    while (TokenTable[pos].type != EndProgram)
    {
        ASTNode* nextStmt = statement();
        body = createNode(statement_node, "statement", body,
nextStmt);
    }
    return body;
}

// <оператор> = <присвоєння> | <ввід> | <вивід> | <умовний
оператор> | <складений оператор>
ASTNode* statement()
{
    switch (TokenTable[pos].type)
    {
        case Input: return input();
        case Output: return output();
        case If: return conditional();
        case StartProgram: return compound_statement();
        case Goto: return goto_statement();
        case Label: return label_statement();
        case For:
        {

```

```

        int temp_pos = pos + 1;
        while (TokenTable[temp_pos].type != To &&
TokenTable[temp_pos].type != DownTo && temp_pos < TokensNum)
        {
            temp_pos++;
        }
        if (TokenTable[temp_pos].type == To)
        {
            return for_to_do();
        }
        else if (TokenTable[temp_pos].type == DownTo)
        {
            return for_downto_do();
        }
        else
        {
            printf("Error: Expected 'To' or 'DownTo' after
'For'\n");
            exit(1);
        }
    }
    case While: return while_statement();
    case Exit:
        match(Exit);
        match(While);
        return createNode(exit_while_node, "exit-while", NULL,
NULL);
    case Continue:
        match(Continue);
        match(While);
        return createNode(continue_while_node, "continue-while",
NULL, NULL);
    case Repeat: return repeat_until();
    default: return assignment();
    }
}

// <присвоєння> = <ідентифікатор> '[:=' <арифметичний вираз>
ASTNode* assignment()
{
    ASTNode* id = createNode(id_node, TokenTable[pos].name,
NULL, NULL);
    match(Identifier);
    match(Assign);
    ASTNode* expr = arithmetic_expression();
    match(Semicolon);
    return createNode(assign_node, "<==", id, expr);
}

```



```

// <арифметичний вираз> = <доданок> { ('+' | '-') <доданок> }
ASTNode* arithmetic_expression()
{
    ASTNode* left = term();
    while (TokenTable[pos].type == Add || TokenTable[pos].type
== Sub)
    {
        TypeOfTokens op = TokenTable[pos].type;
        match(op);
        ASTNode* right = term();
        if (op == Add)
            left = createNode(add_node, "+", left, right);
        else
            left = createNode(sub_node, "-", left, right);
    }
    return left;
}

// <доданок> = <множник> { ('*' | '/') <множник> }
ASTNode* term()
{
    ASTNode* left = factor();
    while (TokenTable[pos].type == Mul || TokenTable[pos].type
== Div || TokenTable[pos].type == Mod)
    {
        TypeOfTokens op = TokenTable[pos].type;
        match(op);
        ASTNode* right = factor();
        if (op == Mul)
            left = createNode(mul_node, "*", left, right);
        if (op == Div)
            left = createNode(div_node, "/", left, right);
        if (op == Mod)
            left = createNode(mod_node, "%", left, right);
    }
    return left;
}

// <множник> = <ідентифікатор> | <число> | '(' <арифметичний
вираз> ') '
ASTNode* factor()
{
    if (TokenTable[pos].type == Identifier)
    {
        ASTNode* id = createNode(id_node, TokenTable[pos].name,
NULL, NULL);
        match(Identifier);
        return id;
    }
    else

```

```

        if (TokenTable[pos].type == Number)
        {
            ASTNode* num = createNode(num_node,
TokenTable[pos].name, NULL, NULL);
            match(Number);
            return num;
        }
        else
        {
            if (TokenTable[pos].type == LBraket)
            {
                match(LBraket);
                ASTNode* expr = arithmetic_expression();
                match(RBraket);
                return expr;
            }
            else
            {
                printf("\nSyntax error in line %d: A multiplier
was expected.\n", TokenTable[pos].line);
                exit(11);
            }
        }
    }

    // <ввїд> = 'input' <їдентифїкатор>
    ASTNode* input()
    {
        match(Input);
        ASTNode* id = createNode(id_node, TokenTable[pos].name,
NULL, NULL);
        match(Identifier);
        match(Semicolon);
        return createNode(input_node, "input", id, NULL);
    }

    // <вивїд> = 'output' <їдентифїкатор>
    ASTNode* output()
    {
        match(Output); // Match the "Output" token

        ASTNode* expr = NULL;
        // Check for a negative number
        if (TokenTable[pos].type == Minus && TokenTable[pos +
1].type == Number)
        {
            pos++; // Skip the 'Sub' token
            expr = createNode(sub_node, "-", createNode(num_node,
"0", NULL, NULL),
                createNode(num_node, TokenTable[pos].name, NULL,
NULL));
            match(Number); // Match the number token

```

```

    }
    else
    {
        // Parse the arithmetic expression
        expr = arithmetic_expression();
    }
    match(Semicolon); // Ensure the statement ends with a
    semicolon

    // Create the output node with the parsed expression as its
    left child
    return createNode(output_node, "output", expr, NULL);
}

```

```

// <умовний оператор> = 'if' <логічний вираз> <оператор>
[ 'else' <оператор> ]
ASTNode* conditional()
{
    match(If);
    ASTNode* condition = logical_expression();
    ASTNode* ifBranch = statement();
    ASTNode* elseBranch = NULL;
    if (TokenTable[pos].type == Else)
    {
        match(Else);
        elseBranch = statement();
    }
    return createNode(if_node, "if", condition,
createNode(statement_node, "branches", ifBranch, elseBranch));
}

```

```

ASTNode* goto_statement()
{
    match(Goto);
    if (TokenTable[pos].type == Identifier)
    {
        ASTNode* label = createNode(label_node,
TokenTable[pos].name, NULL, NULL);
        match(Identifier);
        match(Semicolon);
        return createNode(goto_node, "goto", label, NULL);
    }
    else
    {
        printf("Syntax error: Expected a label after 'goto' at
line %d.\n", TokenTable[pos].line);
        exit(1);
    }
}

```

```

    }
}

ASTNode* label_statement()
{
    match(Label);
    ASTNode* label = createNode(label_node, TokenTable[pos -
1].name, NULL, NULL);
    return label;
}

ASTNode* for_to_do()
{
    match(For);

    if (TokenTable[pos].type != Identifier)
    {
        printf("Syntax error: Expected variable name after 'for'
at line %d.\n", TokenTable[pos].line);
        exit(1);
    }
    ASTNode* var = createNode(id_node, TokenTable[pos].name,
NULL, NULL);
    match(Identifier);
    match(Assign);
    ASTNode* start = arithmetic_expression();
    match(To);
    ASTNode* end = arithmetic_expression();
    match(Do);
    ASTNode* body = statement();
    // Повертаємо вузол циклу for-to
    return createNode(for_to_node, "for-to",
        createNode(assign_node, "<==", var, start),
        createNode(statement_node, "body", end, body));
}

ASTNode* for_downto_do()
{
    // Очікуємо "for"
    match(For);

    // Очікуємо ідентифікатор змінної циклу
    if (TokenTable[pos].type != Identifier)
    {
        printf("Syntax error: Expected variable name after 'for'
at line %d.\n", TokenTable[pos].line);
        exit(1);
    }

```

```

    }
    ASTNode* var = createNode(id_node, TokenTable[pos].name,
NULL, NULL);
    match(Identifier);
    match(Assign);
    ASTNode* start = arithmetic_expression();
    match(DownTo);
    ASTNode* end = arithmetic_expression();
    match(Do);
    ASTNode* body = statement();
    // Повертаємо вузол циклу for-to
    return createNode(for_downto_node, "for-downto",
        createNode(assign_node, "<==", var, start),
        createNode(statement_node, "body", end, body));
}

ASTNode* while_statement()
{
    match(While);
    ASTNode* condition = logical_expression();

    // Parse the body of the While loop
    ASTNode* body = NULL;
    while (1) // Process until "End While"
    {
        if (TokenTable[pos].type == End)
        {
            match(End);
            match(While);
            break; // End of the While loop
        }
        else
        {
            // Delegate to the `statement` function
            ASTNode* stmt = statement();
            body = createNode(statement_node, "statement", body,
stmt);
        }
    }

    return createNode(while_node, "while", condition, body);
}

// Updated variable validation logic
ASTNode* validate_identifier()
{
    const char* identifierName = TokenTable[pos].name;

    // Check if the identifier was declared

```

```

    bool declared = false;
    for (unsigned int i = 0; i < TokensNum; i++)
    {
        if (TokenTable[i].type == Variable && !
strcmp(TokenTable[i].name, identifierName))
        {
            declared = true;
            break;
        }
    }

    if (!declared && (pos == 0 || TokenTable[pos - 1].type !=
Goto))
    {
        printf("Syntax error: Undeclared identifier '%s' at line
%d.\n", identifierName, TokenTable[pos].line);
        exit(1);
    }

    match(Identifier);
    return createNode(id_node, identifierName, NULL, NULL);
}

```

```

ASTNode* repeat_until()
{
    match(Repeat);
    ASTNode* body = NULL;
    ASTNode* stmt = statement();
    body = createNode(statement_node, "body", body, stmt);
    //pos++;
    match(Until);
    ASTNode* condition = logical_expression();
    return createNode(repeat_until_node, "repeat-until", body,
condition);
}

```

// <логічний вираз> = <вираз I> { '|' <вираз I> }

```

ASTNode* logical_expression()
{
    ASTNode* left = and_expression();
    while (TokenTable[pos].type == Or)
    {
        match(Or);
        ASTNode* right = and_expression();
        left = createNode(or_node, "|", left, right);
    }
    return left;
}

```

```

// <вираз I> = <порівняння> { '&' <порівняння> }
ASTNode* and_expression()
{
    ASTNode* left = comparison();
    while (TokenTable[pos].type == And)
    {
        match(And);
        ASTNode* right = comparison();
        left = createNode(and_node, "&", left, right);
    }
    return left;
}

// <порівняння> = <операція порівняння> | '!' '(' <логічний вираз> ')' | '(' <логічний вираз> ')'
// <операція порівняння> = <арифметичний вираз> <менше-більше>
<арифметичний вираз>
// <менше-більше> = '>' | '<' | '=' | '<>'
ASTNode* comparison()
{
    if (TokenTable[pos].type == Not)
    {
        // Варіант: ! (<логічний вираз>)
        match(Not);
        match(LBraket);
        ASTNode* expr = logical_expression();
        match(RBraket);
        return createNode(not_node, "!", expr, NULL);
    }
    else
    {
        if (TokenTable[pos].type == LBraket)
        {
            // Варіант: ( <логічний вираз> )
            match(LBraket);
            ASTNode* expr = logical_expression();
            match(RBraket);
            return expr; // Повертаємо вираз у дужках як
піддерево
        }
        else
        {
            // Варіант: <арифметичний вираз> <менше-більше>
<арифметичний вираз>
            ASTNode* left = arithmetic_expression();
            if (TokenTable[pos].type == Greater ||
TokenTable[pos].type == Less ||
TokenTable[pos].type == Equality ||
TokenTable[pos].type == NotEquality)
            {

```

```

        TypeOfTokens op = TokenTable[pos].type;
        char operatorName[16];
        strcpy_s(operatorName, TokenTable[pos].name);
        match(op);
        ASTNode* right = arithmetic_expression();
        return createNode(cmp_node, operatorName, left,
right);
    }
    else
    {
        printf("\nSyntax error: A comparison operation
is expected.\n");
        exit(12);
    }
}

// <складений оператор> = 'start' <тіло програми> 'stop'
ASTNode* compound_statement()
{
    match(StartProgram);
    ASTNode* body = program_body();
    match(EndProgram);
    return createNode(compount_node, "compound", body, NULL);
}

// функція для друку AST у вигляді дерева на екран
void PrintAST(ASTNode* node, int level)
{
    if (node == NULL)
        return;

    // Відступи для позначення рівня вузла
    for (int i = 0; i < level; i++)
        printf("|   ");

    // Виводимо інформацію про вузол
    printf("|-- %s", node->name);
    printf("\n");

    // Рекурсивний друк лівого та правого піддерева
    if (node->left || node->right)
    {
        PrintAST(node->left, level + 1);
        PrintAST(node->right, level + 1);
    }
}

// функція для друку AST у вигляді дерева у файл

```



```

void PrintASTToFile(ASTNode* node, int level, FILE* outFile)
{
    if (node == NULL)
        return;

    // Відступи для позначення рівня вузла
    for (int i = 0; i < level; i++)
        fprintf(outFile, "|   ");

    // Виводимо інформацію про вузол
    fprintf(outFile, "|-- %s", node->name);
    fprintf(outFile, "\n");

    // Рекурсивний друк лівого та правого піддерева
    if (node->left || node->right)
    {
        PrintASTToFile(node->left, level + 1, outFile);
        PrintASTToFile(node->right, level + 1, outFile);
    }
}

```

codegen.cpp

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "translator.h"

// таблиця лексем
extern Token* TokenTable;
// кількість лексем
extern unsigned int TokensNum;

// таблиця ідентифікаторів
extern Id* IdTable;
// кількість ідентифікаторів
extern unsigned int IdNum;

static int pos = 2;

// набір функцій для рекурсивного спуску
// на кожне правило – окрема функція

void gen_variable_declaration(FILE* outFile);
void gen_variable_list(FILE* outFile);
void gen_program_body(FILE* outFile);
void gen_statement(FILE* outFile);
void gen_assignment(FILE* outFile);
void gen_arithmetic_expression(FILE* outFile);
void gen_term(FILE* outFile);

```

```

void gen_factor(FILE* outFile);
void gen_input(FILE* outFile);
void gen_output(FILE* outFile);
void gen_conditional(FILE* outFile);

void gen_goto_statement(FILE* outFile);
void gen_label_statement(FILE* outFile);
void gen_for_to_do(FILE* outFile);
void gen_for_downto_do(FILE* outFile);
void gen_while_statement(FILE* outFile);
void gen_repeat_until(FILE* outFile);

void gen_logical_expression(FILE* outFile);
void gen_and_expression(FILE* outFile);
void gen_comparison(FILE* outFile);
void gen_compound_statement(FILE* outFile);

void generateCCode(FILE* outFile)
{
    fprintf(outFile, "#include <stdio.h>\n");
    fprintf(outFile, "#include <stdlib.h>\n\n");
    fprintf(outFile, "int main() \n{\n");
    gen_variable_declaration(outFile);
    fprintf(outFile, ";\n");
    pos++;
    pos++;
    gen_program_body(outFile);
    fprintf(outFile, "    system(\"pause\");\n ");
    fprintf(outFile, "    return 0;\n");
    fprintf(outFile, "}\n");
}

// <оголошення змінних> = [<тип даних> <список змінних>]
void gen_variable_declaration(FILE* outFile)
{
    if (TokenTable[pos + 1].type == Type)
    {
        fprintf(outFile, "    int ");
        pos++;
        pos++;
        gen_variable_list(outFile);
    }
}

// <список змінних> = <ідентифікатор> { ',', <ідентифікатор> }
void gen_variable_list(FILE* outFile)
{
    fprintf(outFile, TokenTable[pos++].name);
    while (TokenTable[pos].type == Comma)

```

```

    {
        fprintf(outFile, ", ");
        pos++;
        fprintf(outFile, TokenTable[pos++].name);
    }
}

// <тіло програми> = <оператор> ';' { <оператор> ';' }
void gen_program_body(FILE* outFile)
{
    while (pos < TokensNum && TokenTable[pos].type !=
EndProgram)
    {
        gen_statement(outFile);
    }

    if (pos >= TokensNum || TokenTable[pos].type != EndProgram)
    {
        printf("Error: 'EndProgram' token not found or
unexpected end of tokens.\n");
        exit(1);
    }
}

// <оператор> = <присвоїння> | <ввід> | <вивід> | <умовний
оператор> | <складений оператор>
void gen_statement(FILE* outFile)
{
    switch (TokenTable[pos].type)
    {
    case Input: gen_input(outFile); break;
    case Output: gen_output(outFile); break;
    case If: gen_conditional(outFile); break;
    case StartProgram: gen_compound_statement(outFile); break;
    case Goto: gen_goto_statement(outFile); break;
    case Label: gen_label_statement(outFile); break;
    case For:
    {
        int temp_pos = pos + 1;

        while (TokenTable[temp_pos].type != To &&
TokenTable[temp_pos].type != DownTo && temp_pos < TokensNum)
        {
            temp_pos++;
        }

        if (TokenTable[temp_pos].type == To)
        {
            gen_for_to_do(outFile);
        }
    }
    }
}

```

```

        else if (TokenTable[temp_pos].type == DownTo)
        {
            gen_for_downto_do(outFile);
        }
        else
        {
            printf("Error: Expected 'To' or 'DownTo' after
'For'\n");
        }
    }
    break;
case While: gen_while_statement(outFile); break;
case Exit:
    fprintf(outFile, "        break;\n");
    pos += 2;
    break;

case Continue:
    fprintf(outFile, "        continue;\n");
    pos += 2;
    break;
case Repeat: gen_repeat_until(outFile); break;
default: gen_assignment(outFile);
}
}

// <присвоїння> = <ідентифікатор> ':' '=' <арифметичний вираз>
void gen_assignment(FILE* outFile)
{
    fprintf(outFile, "    ");
    fprintf(outFile, TokenTable[pos++].name);
    fprintf(outFile, " = ");
    pos++;
    gen_arithmetic_expression(outFile);
    pos++;
    fprintf(outFile, ";\n");
}

// <арифметичний вираз> = <доданок> { ('+' | '-') <доданок> }
void gen_arithmetic_expression(FILE* outFile)
{
    gen_term(outFile);
    while (TokenTable[pos].type == Add || TokenTable[pos].type
== Sub)
    {
        if (TokenTable[pos].type == Add)
            fprintf(outFile, " + ");
        else
            fprintf(outFile, " - ");
        pos++;
    }
}

```

```

        gen_term(outFile);
    }
}

// <доданок> = <множник> { ('*' | '/') <множник> }
void gen_term(FILE* outFile)
{
    gen_factor(outFile);
    while (TokenTable[pos].type == Mul || TokenTable[pos].type
== Div || TokenTable[pos].type == Mod)
    {
        if (TokenTable[pos].type == Mul)
            fprintf(outFile, " * ");
        if (TokenTable[pos].type == Div)
            fprintf(outFile, " / ");
        if (TokenTable[pos].type == Mod)
            fprintf(outFile, " %% ");
        pos++;
        gen_factor(outFile);
    }
}

// <множник> = <≥дентиф≥катор> | <число> | '(' <арифметичний
вираз> ')'
void gen_factor(FILE* outFile)
{
    if (TokenTable[pos].type == Identifier ||
TokenTable[pos].type == Number)
        fprintf(outFile, TokenTable[pos++].name);
    else
        if (TokenTable[pos].type == LBraket)
        {
            fprintf(outFile, "(");
            pos++;
            gen_arithmetic_expression(outFile);
            fprintf(outFile, ")");
            pos++;
        }
}

// <вв≥д> = 'input' <≥дентиф≥катор>
void gen_input(FILE* outFile)
{
    fprintf(outFile, "    printf(\"Enter \");
fprintf(outFile, TokenTable[pos + 1].name);
fprintf(outFile, ":\");\n");
fprintf(outFile, "    scanf(\"%d\", &");
pos++;
fprintf(outFile, TokenTable[pos++].name);
fprintf(outFile, ");\n");
}

```

```

        pos++;
    }

    // <вив≥д> = 'output' <≥дентиф≥катор>
    void gen_output(FILE* outFile)
    {
        pos++;

        if (TokenTable[pos].type == Minus && TokenTable[pos +
1].type == Number)
        {
            fprintf(outFile, "    printf(\"%%d\\n\", -%s);\\n",
TokenTable[pos + 1].name);
            pos += 2;
        }
        else
        {
            fprintf(outFile, "    printf(\"%%d\\n\", ");
            gen_arithmetic_expression(outFile);
            fprintf(outFile, ");\\n");
        }

        if (TokenTable[pos].type == Semicolon)
        {
            pos++;
        }
        else
        {
            printf("Error: Expected a semicolon at the end of
'Output' statement.\\n");
            exit(1);
        }
    }
}

```

```

// <умовний оператор> = 'if' <лог≥чний вираз> 'then' <оператор>
[ 'else' <оператор> ]
void gen_conditional(FILE* outFile)
{
    fprintf(outFile, "    if (");
    pos++;
    gen_logical_expression(outFile);
    fprintf(outFile, ")\n");
    gen_statement(outFile);
    if (TokenTable[pos].type == Else)
    {
        fprintf(outFile, "    else\n");
        pos++;
    }
}

```

```

        gen_statement(outFile);
    }
}

void gen_goto_statement(FILE* outFile)
{
    fprintf(outFile, "    goto %s;\n", TokenTable[pos + 1].name);
    pos += 3;
}

void gen_label_statement(FILE* outFile)
{
    fprintf(outFile, "%s:\n", TokenTable[pos].name);
    pos++;
}

void gen_for_to_do(FILE* outFile)
{
    int temp_pos = pos + 1;

    const char* loop_var = TokenTable[temp_pos].name;
    temp_pos += 2;

    fprintf(outFile, "    for (int %s = ", loop_var);
    pos = temp_pos;
    gen_arithmetic_expression(outFile);
    fprintf(outFile, "; ");

    while (TokenTable[pos].type != To && pos < TokensNum)
    {
        pos++;
    }

    if (TokenTable[pos].type == To)
    {
        pos++;
        fprintf(outFile, "%s <= ", loop_var);
        gen_arithmetic_expression(outFile);
    }
    else
    {
        printf("Error: Expected 'To' in For-To loop\n");
        return;
    }

    fprintf(outFile, "; %s++)\n", loop_var);

    if (TokenTable[pos].type == Do)
    {
        pos++;

```

```

    }
    else
    {
        printf("Error: Expected 'Do' after 'To' clause\n");
        return;
    }

    gen_statement(outFile);
}
void gen_for_downto_do(FILE* outFile)
{
    int temp_pos = pos + 1;

    const char* loop_var = TokenTable[temp_pos].name;
    temp_pos += 2;

    fprintf(outFile, "    for (int %s = ", loop_var);
    pos = temp_pos;
    gen_arithmetic_expression(outFile);
    fprintf(outFile, "; ");

    while (TokenTable[pos].type != DownTo && pos < TokensNum)
    {
        pos++;
    }

    if (TokenTable[pos].type == DownTo)
    {
        pos++;

        fprintf(outFile, "%s >= ", loop_var);
        gen_arithmetic_expression(outFile);
    }
    else
    {
        printf("Error: Expected 'Downto' in For-Downto loop\n");
        return;
    }

    fprintf(outFile, "; %s--)\n", loop_var);

    if (TokenTable[pos].type == Do)
    {
        pos++;
    }
    else
    {
        printf("Error: Expected 'Do' after 'Downto' clause\n");
        return;
    }
}

```



```

    gen_statement(outFile);
}

void gen_while_statement(FILE* outFile)
{
    fprintf(outFile, "    while (");
    pos++;
    gen_logical_expression(outFile);
    fprintf(outFile, ")\n    {\n");

    while (pos < TokensNum)
    {
        if (TokenTable[pos].type == End && TokenTable[pos +
1].type == While)
        {
            pos += 2;
            break;
        }
        else
        {
            gen_statement(outFile);
            if (TokenTable[pos].type == Semicolon)
            {
                pos++;
            }
        }
    }

    fprintf(outFile, "    }\n");
}

```

```

void gen_repeat_until(FILE* outFile)
{
    fprintf(outFile, "    do\n");
    pos++;
    do
    {
        gen_statement(outFile);
    } while (TokenTable[pos].type != Until);
    fprintf(outFile, "    while (");
    pos++;
    gen_logical_expression(outFile);
    fprintf(outFile, ");\n");
}

```

```

// <логический вираз> = <вираз <= { '|' <вираз <= }
void gen_logical_expression(FILE* outFile)

```

```

{
    gen_and_expression(outFile);
    while (TokenTable[pos].type == Or)
    {
        fprintf(outFile, " || ");
        pos++;
        gen_and_expression(outFile);
    }
}

// <вираз ≤> = <пор≥внєєє> { '&' <пор≥внєєє> }
void gen_and_expression(FILE* outFile)
{
    gen_comparison(outFile);
    while (TokenTable[pos].type == And)
    {
        fprintf(outFile, " && ");
        pos++;
        gen_comparison(outFile);
    }
}

// <пор≥внєєє> = <операц≥є пор≥внєєє> | C!C C(C <лог≥чний
вираз> C)C | C(C <лог≥чний вираз> C)C
// <операц≥є пор≥внєєє> = <арифметичний вираз> <менше-б≥льше>
<арифметичний вираз>
// <менше-б≥льше> = C>C | C<C | C=C | C<>C
void gen_comparison(FILE* outFile)
{
    if (TokenTable[pos].type == Not)
    {
        // ¬ар≥єєє: ! (<лог≥чний вираз>)
        fprintf(outFile, "!(");
        pos++;
        pos++;
        gen_logical_expression(outFile);
        fprintf(outFile, ")");
        pos++;
    }
    else
        if (TokenTable[pos].type == LBracket)
        {
            // ¬ар≥єєє: ( <лог≥чний вираз> )
            fprintf(outFile, "(");
            pos++;
            gen_logical_expression(outFile);
            fprintf(outFile, ")");
            pos++;
        }
    else

```

```

        {
            // -ар≥ант: <арифметичний вираз> <менше-б≥льше>
<арифметичний вираз>
            gen_arithmetic_expression(outFile);
            if (TokenTable[pos].type == Greate ||
TokenTable[pos].type == Less ||
            TokenTable[pos].type == Equality ||
TokenTable[pos].type == NotEquality)
            {
                switch (TokenTable[pos].type)
                {
                    case Greate: fprintf(outFile, " > "); break;
                    case Less: fprintf(outFile, " < "); break;
                    case Equality: fprintf(outFile, " == "); break;
                    case NotEquality: fprintf(outFile, " != ");
break;
                }
                pos++;
                gen_arithmetic_expression(outFile);
            }
        }

// <складений оператор> = 'start' <т≥ло програми> 'stop'
void gen_compound_statement(FILE* outFile)
{
    fprintf(outFile, "    {\n");
    pos++;
    gen_program_body(outFile);
    fprintf(outFile, "    }\n");
    pos++;
}

```

codegenfromast.cpp

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "translator.h"

// Рекурсивна функція для генерації коду з AST
void generateCodefromAST(ASTNode* node, FILE* outFile)
{
    if (node == NULL)
        return;

    switch (node->nodetype)
    {

```

```

    case program_node:
        fprintf(outFile, "#include <stdio.h>\n#include
<stdlib.h>\n\nint main() \n{\n");
        generateCodefromAST(node->left, outFile); // Оголошення
змінних
        generateCodefromAST(node->right, outFile); // Тіло
програми
        fprintf(outFile, "    system(\"pause\");\n ");
        fprintf(outFile, "    return 0;\n}\n");
        break;

    case var_node:
        // Якщо є права частина (інші змінні), додаємо коми і
генеруємо для них код
        if (node->right != NULL)
        {
            //fprintf(outFile, ", ");
            generateCodefromAST(node->right, outFile); //
Рекурсивно генеруємо код для інших змінних
        }
        fprintf(outFile, "    int "); // Виводимо тип змінних (в
даному випадку int)
        generateCodefromAST(node->left, outFile);
        fprintf(outFile, ";\n"); // Завершуємо оголошення
змінних
        break;

    case id_node:
        fprintf(outFile, "%s", node->name);
        break;

    case num_node:
        fprintf(outFile, "%s", node->name);
        break;

    case assign_node:
        fprintf(outFile, "    ");
        generateCodefromAST(node->left, outFile);
        fprintf(outFile, " = ");
        generateCodefromAST(node->right, outFile);
        fprintf(outFile, ";\n");
        break;

    case add_node:
        fprintf(outFile, "(");
        generateCodefromAST(node->left, outFile);
        fprintf(outFile, " + ");
        generateCodefromAST(node->right, outFile);
        fprintf(outFile, ")");
        break;

```

```

case sub_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " - ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case mul_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " * ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case mod_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " %% ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case div_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " / ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case input_node:
    fprintf(outFile, "    printf(\"Enter \");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ":\");\n");
    fprintf(outFile, "    scanf(\"%d\", &");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ");\n");
    break;

case output_node:
    fprintf(outFile, "    printf(\"%d\\n\", ");
    generateCodefromAST(node->left, outFile);

    fprintf(outFile, ");\n");
    break;

```

```

case if_node:
    fprintf(outFile, "    if (");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ") \n");
    generateCodefromAST(node->right->left, outFile);
    if (node->right->right != NULL)
    {
        fprintf(outFile, "    else\n");
        generateCodefromAST(node->right->right, outFile);
    }
    break;

case goto_node:
    fprintf(outFile, "    goto %s;\n", node->left->name);
    break;

case label_node:
    fprintf(outFile, "%s:\n", node->name);
    break;

case for_to_node:
    fprintf(outFile, "    for (int ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, " = ");
    generateCodefromAST(node->left->right, outFile);
    fprintf(outFile, "; ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, " <= ");
    generateCodefromAST(node->right->left, outFile);
    fprintf(outFile, "; ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, "++)\n");
    generateCodefromAST(node->right->right, outFile);
    break;

case for_downto_node:
    fprintf(outFile, "    for (int ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, " = ");
    generateCodefromAST(node->left->right, outFile);
    fprintf(outFile, "; ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, " >= ");
    generateCodefromAST(node->right->left, outFile);
    fprintf(outFile, "; ");
    generateCodefromAST(node->left->left, outFile);

```

```

        fprintf(outFile, "--)\n");
        generateCodefromAST(node->right->right, outFile);
        break;

case while_node:
    fprintf(outFile, "    while (");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ")\n");
    fprintf(outFile, "    {\n");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, "    }\n");
    break;

case exit_while_node:
    fprintf(outFile, "    break;\n");
    break;

case continue_while_node:
    fprintf(outFile, "    continue;\n");
    break;

case repeat_until_node:
    fprintf(outFile, "    do\n");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, "    while (");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ");\n");
    break;

case or_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " || ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case and_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " && ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case not_node:
    fprintf(outFile, "!(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ")");

```

```

        break;

    case cmp_node:
        generateCodefromAST(node->left, outFile);
        if (!strcmp(node->name, "="))
            fprintf(outFile, " == ");
        else if (!strcmp(node->name, "<>"))
            fprintf(outFile, " != ");
        else if (!strcmp(node->name, ">="))
            fprintf(outFile, " > ");
        else if (!strcmp(node->name, "<="))
            fprintf(outFile, " < ");
        else
            fprintf(outFile, " %s ", node->name);
        generateCodefromAST(node->right, outFile);
        break;

    case statement_node:
        generateCodefromAST(node->left, outFile);
        if (node->right != NULL)
            generateCodefromAST(node->right, outFile);
        break;

    case compount_node:
        fprintf(outFile, " {\n");
        generateCodefromAST(node->left, outFile);
        fprintf(outFile, " }\n");
        break;

    default:
        fprintf(stderr, "Unknown node type: %d\n", node->nodetype);
        break;
    }
}

```

lexer.cpp

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "translator.h"
#include <locale>

// 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[], FILE*
errFile)
{

```



```

States state = Start;
Token TempToken;
// кількість лексем
unsigned int NumberOfTokens = 0;
char ch, buf[32];
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 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') || (ch <= 'Z' &&
ch >= 'A') || ch == '_')
                    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 >= 'a' && ch <= 'z') || (ch >= 'A' &&
ch <= 'Z')) ||
        (ch >= '0' && ch <= '9') || ch == '_' || ch ==
':') || ch == '-') && j < 32)
    {
        buf[j++] = ch;
        ch = getc(F);
    }
    buf[j] = '\0';

    TypeOfTokens temp_type = Unknown;

    if (!strcmp(buf, "End"))
    {
```

```

char next_buf[16];
int next_j = 0;

while (ch == ' ' || ch == '\t')
{
    ch = getc(F);
}

while (((ch >= 'a' && ch <= 'z') || (ch >= 'A'
&& ch <= 'Z')) && next_j < 32)
{
    next_buf[next_j++] = ch;
    ch = getc(F);
}
next_buf[next_j] = '\0';

if (!strcmp(next_buf, "While"))
{
    temp_type = End;
    strcpy_s(TempToken.name, buf);
    TempToken.type = temp_type;
    TempToken.value = 0;
    TempToken.line = line;
    TokenTable[NumberOfTokens++] = TempToken;

    temp_type = While;
    strcpy_s(TempToken.name, next_buf);
    TempToken.type = temp_type;
    TempToken.value = 0;
    TempToken.line = line;
    TokenTable[NumberOfTokens++] = TempToken;

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

    for (int k = next_j - 1; k >= 0; k--)
    {
        ungetc(next_buf[k], F);
    }
    break;
}

```

```

    }

    else if(!strcmp(buf, "Name"))
    {
        char next_buf[32];
        int next_j = 0;

        while (ch == ' ' || ch == '\t')
        {
            ch = getc(F);
        }

        while (((ch >= 'a' && ch <= 'z') || (ch >= 'A'
&& ch <= 'Z')) || (ch >= '0' && ch <= '9' || ch == ';')) && next_j
< 31)
        {
            next_buf[next_j++] = ch;
            ch = getc(F);
        }
        next_buf[next_j] = '\0';

        if (next_buf[strlen(next_buf) - 1] == ';')
        {
            temp_type = Mainprogram;
            strcpy_s(TempToken.name, buf);
            TempToken.type = temp_type;
            TempToken.value = 0;
            TempToken.line = line;
            TokenTable[NumberOfTokens++] = TempToken;

            next_buf[strlen(next_buf) - 1] = '\0';
            temp_type = ProgramName;
            strcpy_s(TempToken.name, next_buf);
            TempToken.type = temp_type;
            TempToken.value = 0;
            TempToken.line = line;
            TokenTable[NumberOfTokens++] = TempToken;

            state = Start;
            break;
        }
    }

    else if (!strcmp(buf, "Name"))        temp_type =
Mainprogram;
    else if (!strcmp(buf, "Body"))        temp_type =
StartProgram;
    else if (!strcmp(buf, "Data"))        temp_type =
Variable;

```

```

else if (!strcmp(buf, "Longint")) temp_type = Type;
else if (!strcmp(buf, "Read"))      temp_type =
Input;
else if (!strcmp(buf, "Write"))      temp_type =
Output;

else if (!strcmp(buf, "Div"))        temp_type = Div;
else if (!strcmp(buf, "Mod"))        temp_type = Mod;

else if (!strcmp(buf, "If"))         temp_type = If;
else if (!strcmp(buf, "Else"))       temp_type =
Else;
else if (!strcmp(buf, "Goto"))       temp_type =
Goto;
else if (!strcmp(buf, "For"))        temp_type = For;
else if (!strcmp(buf, "To"))         temp_type = To;
else if (!strcmp(buf, "Downto"))    temp_type =
Downto;
else if (!strcmp(buf, "Do"))         temp_type = Do;
else if (!strcmp(buf, "Exit"))       temp_type =
Exit;
else if (!strcmp(buf, "While"))      temp_type =
While;
else if (!strcmp(buf, "Continue"))   temp_type =
Continue;
else if (!strcmp(buf, "Repeat"))     temp_type =
Repeat;
else if (!strcmp(buf, "Until"))      temp_type =
Until;
    if (temp_type == Unknown &&
TokenTable[NumberOfTokens - 1].type == Goto)
    {
        temp_type = Identifier;
    }
else if (buf[strlen(buf) - 1] == ':')
{
    buf[strlen(buf) - 1] = '\0';
    temp_type = Label;
}
else if (buf[0] == '_' && (strlen(buf) == 18))
{
    bool valid = true;

    if (!(buf[1] >= 'a' && buf[1] <= 'z')) valid =
false;

    for (int i = 2; i < 18; i++)
    {
        if (!(buf[i] >= 'A' && buf[i] <= 'Z') && !
(buf[i] >= '0' && buf[i] <= '9'))
        {

```

```

        valid = false;
        break;
    }
}
if (valid)
{
    temp_type = Identifier;
}
}
strcpy_s(TempToken.name, buf);
TempToken.type = temp_type;
TempToken.value = 0;
TempToken.line = line;
if (temp_type == Unknown)
{
    fprintf(errFile, "Lexical Error: line %d, lexem
%s is Unknown\n", line, TempToken.name);
}
state = Finish;
break;
}

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;
}

case Separators:
{
    if (ch == '\n')
        line++;

    ch = getc(F);

```

```

        state = Start;
        break;
    }

    case SComment:
    {
        ch = getc(F);
        if (ch == '\\')
            state = Comment;
        break;
    }

    case Comment:
    {
        while (ch != '\n' && ch != EOF)
        {
            ch = getc(F);
        }
        if (ch == EOF)
        {
            state = EndOfFile;
            break;
        }
        state = Start;
        break;
    }

    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 ':':
    {
        char next = getc(F);

        strcpy_s(TempToken.name, ":");
        TempToken.type = Colon;
        ungetc(next, F);

        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
    }

    case '+':
    {
        ch = getc(F);
        if (ch == '+')
        {
            strcpy_s(TempToken.name, "++");
            TempToken.type = Add;
            TempToken.value = 0;
            TempToken.line = line;
            ch = getc(F);

```



```

        state = Finish;
        break;
    }
}

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

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

case '&':
{
    ch = getc(F);
    if (ch == '&')
    {
        strcpy_s(TempToken.name, "&&");

```

```

        TempToken.type = And;
        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;
        fprintf(errFile, "Lexical Error: line %d,
lexem %s is Unknown\n", line, TempToken.name);
        state = Finish;
    }
    break;
}

case '|':
{
    ch = getc(F);
    if (ch == '|')
    {
        strcpy_s(TempToken.name, "||");
        TempToken.type = Or;
        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;
        fprintf(errFile, "Lexical Error: line %d,
lexem %s is Unknown\n", line, TempToken.name);
        state = Finish;
    }

    break;
}

case '!':
{
    ch = getc(F);
    if (ch == '!')
    {
        strcpy_s(TempToken.name, "!!");

```

```

        TempToken.type = Not;
        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;
        fprintf(errFile, "Lexical Error: line %d,
lexem %s is Unknown\n", line, TempToken.name);
        state = Finish;
    }
    break;
}

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

```



```

{
    char type_tokens[16];

printf("\n\n-----\n");
printf("|          TOKEN TABLE\n");
printf("-----\n");
printf("| line number |      token      |      value      | token\n");
printf("code | type of token |\n");

printf("-----\n");
for (unsigned int i = 0; i < TokensNum; i++)
{
    switch (TokenTable[i].type)
    {
    case Mainprogram:
        strcpy_s(type_tokens, "MainProgram");
        break;
    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 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 Mod:
    strcpy_s(type_tokens, "Mod");
    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 Goto:
    strcpy_s(type_tokens, "Goto");
    break;
case For:
    strcpy_s(type_tokens, "For");
    break;
case To:
    strcpy_s(type_tokens, "To");
    break;
case DownTo:
    strcpy_s(type_tokens, "DownTo");
    break;
case Do:
    strcpy_s(type_tokens, "Do");
    break;
case While:
    strcpy_s(type_tokens, "While");
    break;
case Exit:
    strcpy_s(type_tokens, "Exit");
    break;
case Continue:
    strcpy_s(type_tokens, "Continue");
    break;
case End:
    strcpy_s(type_tokens, "End");
    break;
case Repeat:
    strcpy_s(type_tokens, "Repeat");
    break;
case Until:
    strcpy_s(type_tokens, "Until");
    break;
case Label:
    strcpy_s(type_tokens, "Label");
    break;
case Unknown:
default:
    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("-----\n");
    }
    printf("\n");
}

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      |\n");
    fprintf(F,
"-----\n");
    fprintf(F, "token code | type of token |\n");
    fprintf(F,
"-----\n");
    for (unsigned int i = 0; i < TokensNum; i++)
    {
        switch (TokenTable[i].type)
        {
        case Mainprogram:
            strcpy_s(type_tokens, "MainProgram");
            break;
        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 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 Mod:
    strcpy_s(type_tokens, "Mod");
    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 LBracket:
    strcpy_s(type_tokens, "LBracket");
    break;
case RBracket:
    strcpy_s(type_tokens, "RBracket");
    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 Goto:
    strcpy_s(type_tokens, "Goto");
    break;
case For:
    strcpy_s(type_tokens, "For");
    break;
case To:
    strcpy_s(type_tokens, "To");
    break;
case DownTo:
    strcpy_s(type_tokens, "DownTo");
    break;
case Do:
    strcpy_s(type_tokens, "Do");
    break;
case While:
    strcpy_s(type_tokens, "While");
    break;
case Exit:
    strcpy_s(type_tokens, "Exit");
    break;
case Continue:
    strcpy_s(type_tokens, "Continue");
    break;
case End:
    strcpy_s(type_tokens, "End");
    break;
case Repeat:
    strcpy_s(type_tokens, "Repeat");
    break;
case Until:
```

```

        strcpy_s(type_tokens, "Until");
        break;
    case Label:
        strcpy_s(type_tokens, "Label");
        break;
    case Unknown:
    default:
        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);
}

```

parser.cpp

```

#include <stdlib.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "translator.h"
#include <locale>

// 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[], FILE*
errFile)
{
    States state = Start;
    Token TempToken;
    // кількість лексем
    unsigned int NumberOfTokens = 0;
    char ch, buf[32];
    int line = 1;

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

```



```

    }
    else
    {
        printf("\n\t\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 >= 'a' && ch <= 'z') || (ch >= 'A' &&
ch <= 'Z')) ||
        (ch >= '0' && ch <= '9') || ch == '_' || ch ==
':') || ch == '-') && j < 32)
    {
        buf[j++] = ch;
        ch = getc(F);
    }
    buf[j] = '\0';

    TypeOfTokens temp_type = Unknown;

    if (!strcmp(buf, "End"))
    {
        char next_buf[16];
        int next_j = 0;

        while (ch == ' ' || ch == '\t')
        {
            ch = getc(F);
        }

        while (((ch >= 'a' && ch <= 'z') || (ch >= 'A'
&& ch <= 'Z')) && next_j < 32)
        {

```

```

        next_buf[next_j++] = ch;
        ch = getc(F);
    }
    next_buf[next_j] = '\0';

    if (!strcmp(next_buf, "While"))
    {
        temp_type = End;
        strcpy_s(TempToken.name, buf);
        TempToken.type = temp_type;
        TempToken.value = 0;
        TempToken.line = line;
        TokenTable[NumberOfTokens++] = TempToken;

        temp_type = While;
        strcpy_s(TempToken.name, next_buf);
        TempToken.type = temp_type;
        TempToken.value = 0;
        TempToken.line = line;
        TokenTable[NumberOfTokens++] = TempToken;

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

        for (int k = next_j - 1; k >= 0; k--)
        {
            ungetc(next_buf[k], F);
        }
        break;
    }
}

else if(!strcmp(buf, "Name"))
{
    char next_buf[32];
    int next_j = 0;

    while (ch == ' ' || ch == '\t')
    {
        ch = getc(F);
    }

```

```

        while (((ch >= 'a' && ch <= 'z') || (ch >= 'A'
&& ch <= 'Z')) || (ch >= '0' && ch <= '9' || ch == ';')) && next_j
< 31)
        {
            next_buf[next_j++] = ch;
            ch = getc(F);
        }
        next_buf[next_j] = '\0';

        if (next_buf[strlen(next_buf) - 1] == ';')
        {
            temp_type = Mainprogram;
            strcpy_s(TempToken.name, buf);
            TempToken.type = temp_type;
            TempToken.value = 0;
            TempToken.line = line;
            TokenTable[NumberOfTokens++] = TempToken;

            next_buf[strlen(next_buf) - 1] = '\0';
            temp_type = ProgramName;
            strcpy_s(TempToken.name, next_buf);
            TempToken.type = temp_type;
            TempToken.value = 0;
            TempToken.line = line;
            TokenTable[NumberOfTokens++] = TempToken;

            state = Start;
            break;
        }
    }

    else if (!strcmp(buf, "Name"))        temp_type =
Mainprogram;
    else if (!strcmp(buf, "Body"))        temp_type =
StartProgram;
    else if (!strcmp(buf, "Data"))        temp_type =
Variable;
    else if (!strcmp(buf, "Longint"))    temp_type = Type;
    else if (!strcmp(buf, "Read"))        temp_type =
Input;
    else if (!strcmp(buf, "Write"))        temp_type =
Output;

    else if (!strcmp(buf, "Div"))        temp_type = Div;
    else if (!strcmp(buf, "Mod"))        temp_type = Mod;

    else if (!strcmp(buf, "If"))        temp_type = If;

```

```

else if (!strcmp(buf, "Else"))          temp_type =
Else;
else if (!strcmp(buf, "Goto"))          temp_type =
Goto;
else if (!strcmp(buf, "For"))           temp_type = For;
else if (!strcmp(buf, "To"))            temp_type = To;
else if (!strcmp(buf, "Downto"))        temp_type =
DownTo;
else if (!strcmp(buf, "Do"))            temp_type = Do;
else if (!strcmp(buf, "Exit"))          temp_type =
Exit;
else if (!strcmp(buf, "While"))         temp_type =
While;
else if (!strcmp(buf, "Continue"))      temp_type =
Continue;
else if (!strcmp(buf, "Repeat"))        temp_type =
Repeat;
else if (!strcmp(buf, "Until"))         temp_type =
Until;
    if (temp_type == Unknown &&
TokenTable[NumberOfTokens - 1].type == Goto)
    {
        temp_type = Identifier;
    }
    else if (buf[strlen(buf) - 1] == ':')
    {
        buf[strlen(buf) - 1] = '\0';
        temp_type = Label;
    }
    else if (buf[0] == '_' && (strlen(buf) == 18))
    {
        bool valid = true;

        if (!(buf[1] >= 'a' && buf[1] <= 'z')) valid =
false;
        for (int i = 2; i < 18; i++)
        {
            if (!(buf[i] >= 'A' && buf[i] <= 'Z') && !
(buf[i] >= '0' && buf[i] <= '9'))
            {
                valid = false;
                break;
            }
        }
        if (valid)
        {
            temp_type = Identifier;
        }
    }
    strcpy_s(TempToken.name, buf);

```



```

        TempToken.type = temp_type;
        TempToken.value = 0;
        TempToken.line = line;
        if (temp_type == Unknown)
        {
            fprintf(errFile, "Lexical Error: line %d, lexem
%s is Unknown\n", line, TempToken.name);
        }
        state = Finish;
        break;
    }

    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;
    }

    case Separators:
    {
        if (ch == '\n')
            line++;

        ch = getc(F);

        state = Start;
        break;
    }

    case SComment:
    {
        ch = getc(F);
        if (ch == '\\')
            state = Comment;
    }

```

```

        break;
    }

    case Comment:
    {
        while (ch != '\n' && ch != EOF)
        {
            ch = getc(F);
        }
        if (ch == EOF)
        {
            state = EndOfFile;
            break;
        }
        state = Start;
        break;
    }

    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 ':':
    {
        char next = getc(F);

        strcpy_s(TempToken.name, ":");
        TempToken.type = Colon;
        ungetc(next, F);

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

    case '-':
    {
        ch = getc(F);
        if (ch == '-')
        {

```

```

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

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

case '&':
{
    ch = getc(F);
    if (ch == '&')
    {
        strcpy_s(TempToken.name, "&&");
        TempToken.type = And;
        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;
        fprintf(errFile, "Lexical Error: line %d,
lexem %s is Unknown\n", line, TempToken.name);
        state = Finish;
    }
    break;
}

case '|':
{
    ch = getc(F);
    if (ch == '|')
    {
        strcpy_s(TempToken.name, "||");
        TempToken.type = Or;
        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;
        fprintf(errFile, "Lexical Error: line %d,
lexem %s is Unknown\n", line, TempToken.name);
        state = Finish;
    }

    break;
}

case '!':
{
    ch = getc(F);
    if (ch == '!')
    {
        strcpy_s(TempToken.name, "!!");
        TempToken.type = Not;
        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;
        fprintf(errFile, "Lexical Error: line %d,
lexem %s is Unknown\n", line, TempToken.name);
        state = Finish;
    }
    break;
}

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

case '>':
{
    ch = getc(F);
    if (ch == '=')

```

```

        {
            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;
    }

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

void PrintTokens(Token TokenTable[], unsigned int TokensNum)
{
    char type_tokens[16];

    printf("\n\n-----\n\n");
    printf("|          TOKEN TABLE\n|\n");
    printf("-----\n\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 Mainprogram:
                strcpy_s(type_tokens, "MainProgram");
                break;
            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 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 Mod:
    strcpy_s(type_tokens, "Mod");
    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 LBracket:
    strcpy_s(type_tokens, "LBracket");
    break;
case RBracket:
    strcpy_s(type_tokens, "RBracket");
    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 Goto:
    strcpy_s(type_tokens, "Goto");
    break;
case For:
    strcpy_s(type_tokens, "For");
    break;
case To:
```

```

        strcpy_s(type_tokens, "To");
        break;
    case DownTo:
        strcpy_s(type_tokens, "DownTo");
        break;
    case Do:
        strcpy_s(type_tokens, "Do");
        break;
    case While:
        strcpy_s(type_tokens, "While");
        break;
    case Exit:
        strcpy_s(type_tokens, "Exit");
        break;
    case Continue:
        strcpy_s(type_tokens, "Continue");
        break;
    case End:
        strcpy_s(type_tokens, "End");
        break;
    case Repeat:
        strcpy_s(type_tokens, "Repeat");
        break;
    case Until:
        strcpy_s(type_tokens, "Until");
        break;
    case Label:
        strcpy_s(type_tokens, "Label");
        break;
    case Unknown:
    default:
        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("-----\n");
}
printf("\n");
}

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,
"-----\n");
    for (unsigned int i = 0; i < TokensNum; i++)
    {
        switch (TokenTable[i].type)
        {
            case Mainprogram:
                strcpy_s(type_tokens, "MainProgram");
                break;
            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 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 Mod:
        strcpy_s(type_tokens, "Mod");
        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 Goto:
    strcpy_s(type_tokens, "Goto");
    break;
case For:
    strcpy_s(type_tokens, "For");
    break;
case To:
    strcpy_s(type_tokens, "To");
    break;
case DownTo:
    strcpy_s(type_tokens, "DownTo");
    break;
case Do:
    strcpy_s(type_tokens, "Do");
    break;
case While:
    strcpy_s(type_tokens, "While");
    break;
case Exit:
    strcpy_s(type_tokens, "Exit");
    break;
case Continue:
    strcpy_s(type_tokens, "Continue");
    break;
case End:
    strcpy_s(type_tokens, "End");
    break;
case Repeat:
    strcpy_s(type_tokens, "Repeat");
    break;
case Until:
    strcpy_s(type_tokens, "Until");
    break;
case Label:
    strcpy_s(type_tokens, "Label");
    break;
case Unknown:
default:
    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);
}

```

compile.cpp

```

#include <Windows.h>
#include <stdio.h>
#include <string>
#include <fstream>

#define SCOPE_EXIT_CAT2(x, y) x##y
#define SCOPE_EXIT_CAT(x, y) SCOPE_EXIT_CAT2(x, y)
#define SCOPE_EXIT auto SCOPE_EXIT_CAT(scopeExit_, __COUNTER__)
= Safe::MakeScopeExit() += [&

namespace Safe
{
    template <typename F>
    class ScopeExit
    {
        using A = typename std::decay_t<F>;

    public:
        explicit ScopeExit(A&& action) :
        _action(std::move(action)) {}
        ~ScopeExit() { _action(); }

        ScopeExit() = delete;
        ScopeExit(const ScopeExit&) = delete;
        ScopeExit& operator=(const ScopeExit&) = delete;
        ScopeExit(ScopeExit&&) = delete;
        ScopeExit& operator=(ScopeExit&&) = delete;
        ScopeExit(const A&) = delete;
        ScopeExit(A&) = delete;

    private:
        A _action;
    };
}

```

```

struct MakeScopeExit
{
    template <typename F>
    ScopeExit<F> operator+=(F&& f)
    {
        return ScopeExit<F>(std::forward<F>(f));
    }
};

bool is_file_accessible(const char* file_path)
{
    std::ifstream file(file_path);
    return file.is_open();
}

void compile_to_exe(const char* source_file, const char*
output_file)
{
    if (!is_file_accessible(source_file))
    {
        printf("Error: Source file %s is not accessible.\n",
source_file);
        return;
    }

    wchar_t current_dir[MAX_PATH];
    if (!GetCurrentDirectoryW(MAX_PATH, current_dir))
    {
        printf("Error retrieving current directory. Error code:
%lu\n", GetLastError());
        return;
    }

    //wprintf(L"CurrentDirectory: %s\n", current_dir);

    wchar_t command[512];
    _snwprintf_s(
        command,
        std::size(command),
        L"compiler\\MinGW-master\\MinGW\\bin\\gcc.exe -std=c11
\\\"%s\\\"%S\" -o \\\"%s\\\"%S\\\"",
        current_dir, source_file, current_dir, output_file
    );

    //wprintf(L"Command: %s\n", command);

    STARTUPINFO si = { 0 };
    PROCESS_INFORMATION pi = { 0 };

```

```

    si.cb = sizeof(si);

    if (CreateProcessW(
        NULL,
        command,
        NULL,
        NULL,
        FALSE,
        0,
        NULL,
        current_dir,
        &si,
        &pi
    ))
    {
        WaitForSingleObject(pi.hProcess, INFINITE);

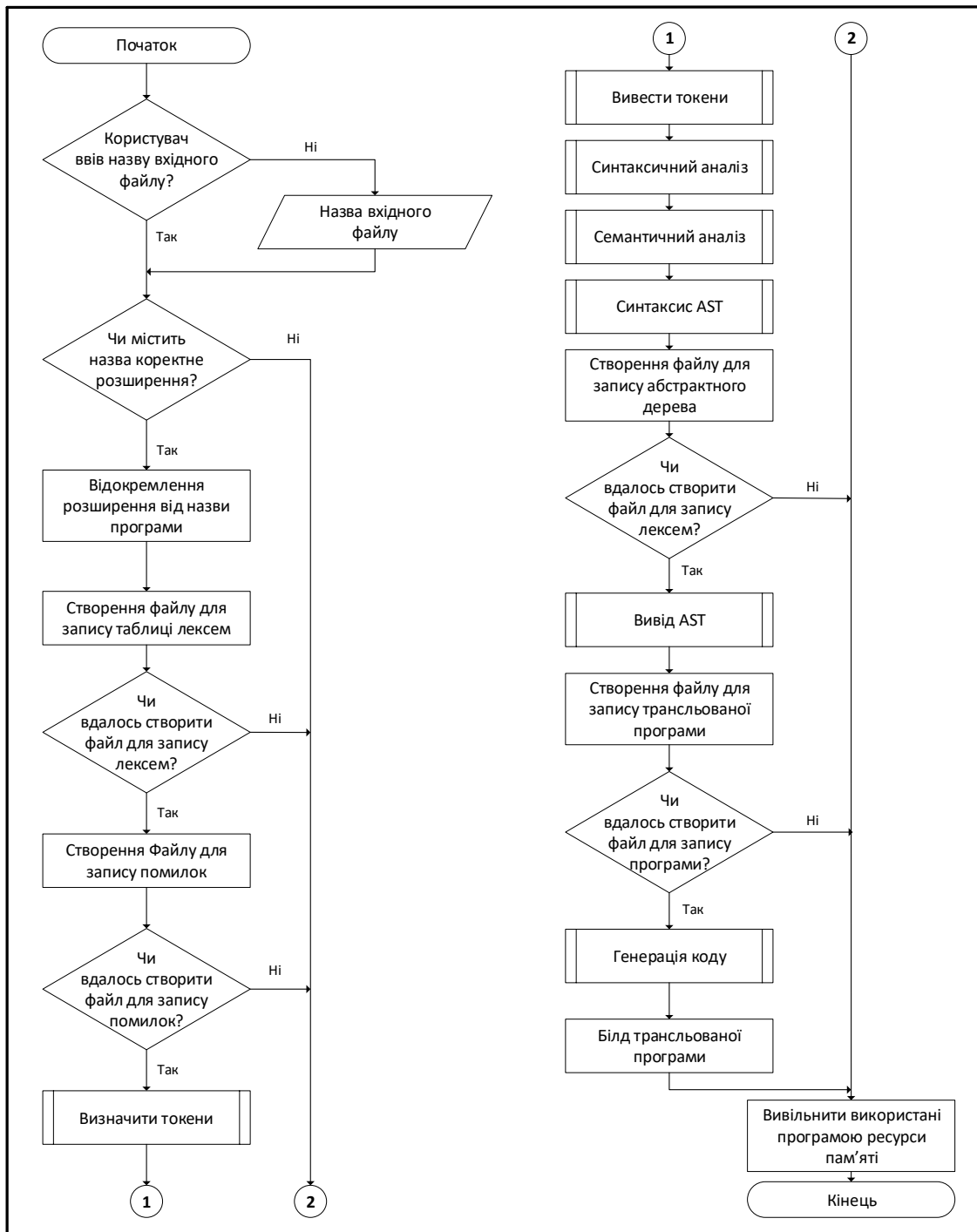
        DWORD exit_code;
        GetExitCodeProcess(pi.hProcess, &exit_code);

        if (exit_code == 0)
        {
            wprintf(L"File successfully compiled into %s\\%S\n",
current_dir, output_file);
        }
        else
        {
            wprintf(L"Compilation error for %. Exit code:
%lu\n", source_file, exit_code);
        }

        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }
    else
    {
        DWORD error_code = GetLastError();
        wprintf(L"Failed to start compiler process. Error code:
%lu\n", error_code);
    }
}

```


Додаток Г. Схема транслятора.



Міністерство освіти і науки України					КУРСОВИЙ ПРОЄКТ			
					Розробка системних програмних модулів та компонент систем програмування			
Зм.	Арк.	№ докум.	Підпис	Дата	Алгоритм транслятора С23		Літера	Маса
Виконав		Чорноморд Я. С					у	
Керівник		Козак Н.Б.						
Консульт.								
Консульт.								
Зав. каф.		Дунець Р. Б.			Додаток Г		Аркуш	Аркушів 1
Реценз.								
					НУ «ЛП», ІКТА, каф. ЕОМ, гр КІ-309			