

Міністерство освіти і науки України
Національний університет “Львівська політехніка”



Курсовий проект

З дисципліни «Системне програмування»
на тему: "Розробка системних програмних модулів
та компонент систем програмування."
Розробка транслятора з вхідної мови програмування"
Варіант №1

Виконав: ст. гр. КІ-309
Бодров А. Ю.
Перевірив:
Козак Н. Б.

Львів-2024

Анотація

Цей курсовий проект 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.....	6
1. Огляд методів та способів проектування трансляторів.....	7
2. Формальний опис вхідної мови програмування	10
2.1 Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура	10
2.2 Опис термінальних символів та ключових слів.....	12
3. Розробка транслятора вхідної мови програмування.....	14
3.1 Вибір технології програмування.....	14
3.2 Проектування таблиць транслятора.....	15
3.3 Розробка лексичного аналізатора.....	17
3.3.1 Розробка блок-схеми алгоритму.....	19
3.3.2 Опис програми реалізації лексичного аналізатора	19
3.4 Розробка синтаксичного та семантичного аналізатора	21
3.4.1 Опис програми реалізації синтаксичного та семантичного аналізатора	22
3.4.2 Опис програми реалізації семантичного аналізатора	23
3.4.3 Розробка блок-схеми роботи синтаксичного аналізатора	25
На рис. 3.2 зображена граф-схема синтаксичного аналізатора.....	25
3.5 Розробка генератора коду	26
3.5.1 Розробка блок-схеми роботи генератора коду.....	27
3.5.2 Опис програми реалізації генератора коду.....	28
4. Опис програми.....	29
4.1 Опис інтерфейсу та інструкція користувачеві.....	32
5. Відлагодження та тестування програми.....	33
5.1 Виявлення лексичних та синтаксичних помилок	33
5.2 Виявлення семантичних помилок.....	34
5.3 Загальна перевірка коректності роботи транслятора.....	34
5.4 Тестова програма №1.....	36
5.5 Тестова програма №2.....	37
5.6 Тестова програма №3.....	38
6. Верефікація тестових програм.....	40
Висновки.....	41
Список використаної літератури	42
Додатки.....	43

Завдання до курсового проекту

Варіант 1

Завдання на курсовий проект

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 .b01

Опис вхідної мови програмування:

- Тип даних: INT32_t
- Блок тіла програми: PROGRAM <name>; VAR...; START FINISH
- 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
- Регістр ідентифікаторів: Low8
- Операції арифметичні: ADD, SUB, MUL, DIV, MOD
- Операції порівняння: =, <>, GE, LE
- Операції логічні: !, &, |
- Коментар: \$\$...
- Ідентифікатори змінних, числові константи

- Оператор присвоения: \Leftarrow

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 (лексеми), 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 – кожну 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 проектування трансляторів

Термін "транслятор" визначає обслуговуючу програму, що проводить трансляцію вихідної програми, представленої на вхідній мові програмування, у робочу програму, яка відображена на іншій цільовій мові програмування, такий як С. Наведене визначення застосовне до різноманітних транслуючих програм. Однак кожна з таких програм може виявляти свої особливості в організації процесу трансляції. В сучасному контексті транслятори поділяються на три основні групи: асемблери, компілятори та інтерпретатори.

Компілятор - обслуговуюча програма, яка виконує трансляцію програми, написаної мовою оригіналу програмування, в іншу мову, наприклад, мову С. Схоже до асемблера, компілятор виконує перетворення програми з однієї мови в іншу, часто генеруючи код, який може бути виконаний іншими компіляторами.

Інтерпретатор - це програма чи пристрій, що виконує пооператорну трансляцію та виконання вихідної програми. Відмінно від компілятора, інтерпретатор не створює на виході нову програму мовою С. Розпізнавши команду вихідної мови, він негайно її виконує, забезпечуючи більшу гнучкість у процесі розробки та налагодження програм.

Процес трансляції включає фази лексичного аналізу, синтаксичного та семантичного аналізу, оптимізації коду та генерації коду. Лексичний аналіз розбиває вхідну програму на лексеми, що представляють слова відповідно до визначень мови. Синтаксичний аналіз визначає структуру програми, створюючи синтаксичне дерево. Семантичний аналіз виявляє залежності між частинами програми, недосяжні контекстно-вільним синтаксисом. Оптимізація коду та генерація коду спрямовані на створення коду мовою С, з урахуванням ефективності його виконання.

Зазначені фази можуть об'єднуватися або відсутні у трансляторах залежно від їхньої реалізації. Наприклад, у простих однопрохідних трансляторах може бути відсутня фаза генерації проміжного представлення та оптимізації, а інші фази можуть об'єднуватися.

Під час процесу виділення лексем лексичний аналізатор може виконувати дві основні функції: автоматичну побудову таблиць об'єктів (таких як ідентифікатори, рядки, числа тощо) і видачу значень для кожної лексеми при кожному новому зверненні до нього. У цьому контексті таблиці об'єктів формуються в подальших етапах, наприклад, під час синтаксичного аналізу.

На етапі лексичного аналізу виявляються деякі прості помилки, такі як неприпустимі символи або невірний формат чисел та ідентифікаторів.

Основним завданням синтаксичного аналізу є розбір структури програми. Зазвичай під структурою розуміється дерево, яке відповідає розбору в контекстно-вільній граматиці мови програмування. У сучасній практиці найчастіше використовуються методи аналізу, такі як LL(1) або LR(1) та їхні варіанти (рекурсивний спуск для LL(1) або LR(1), LR(0), SLR(1), LALR(1) та інші для LR(1)). Рекурсивний спуск застосовується частіше при ручному програмуванні синтаксичного аналізатора, тоді як LR(1) використовується при автоматичній генерації синтаксичних аналізаторів.

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

На етапі контекстного аналізу виявляються взаємозалежності між різними частинами програми, які не можуть бути адекватно описані за допомогою контекстно-вільної граматики. Ці взаємозалежності, зокрема, включають аналіз типів об'єктів, областей видимості, відповідності параметрів, міток та інших аспектів "опис-використання". У ході контекстного аналізу таблиці об'єктів доповнюються інформацією, пов'язаною з описами (властивостями) об'єктів.

В основі контекстного аналізу лежить апарат атрибутних граматик. Результатом цього аналізу є створення атрибутованого дерева програми, де інформація про об'єкти може бути розсіяна в самому дереві чи сконцентрована в окремих таблицях об'єктів. Під час контекстного аналізу також можуть бути виявлені помилки, пов'язані з неправильним використанням об'єктів.

Після завершення контекстного аналізу програма може бути перетворена у внутрішнє представлення. Це здійснюється з метою оптимізації та/або для полегшення генерації коду мовою C. Крім того, перетворення програми у внутрішнє представлення може бути використано для створення переносимого транслятора. У цьому випадку, тільки остання фаза (генерація коду) є залежною від конкретної архітектури. В якості внутрішнього представлення може використовуватися орієнтований граф, трійки, четвірки та інші формати.

Фаза оптимізації транслятора може включати декілька етапів, які спрямовані на покращення якості та ефективності згенерованого коду. Ці оптимізації часто розподіляються за двома головними критеріями: машинно-залежні та машинно-незалежні, а також локальні та глобальні.

Машинно-незалежні оптимізації орієнтовані на спрощення коду або видалення надлишкових обчислень, тоді як машинно-залежні оптимізації проводяться на етапі генерації коду.

Фінальна фаза трансляції - генерація коду мовою C. На цьому етапі можуть застосовуватися деякі локальні оптимізації для полегшення генерації ефективного та читабельного коду.

Важливо відзначити, що фази транслятора можуть бути відсутніми або об'єднаними залежно від конкретної реалізації. У простіших випадках, таких як однопрохідні транслятори, може бути відсутній окремий етап генерації проміжного представлення та оптимізації, а інші фази можуть бути об'єднані в одну, без створення явно побудованого синтаксичного дерева.

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 = "PROGRAM", identifier, ";", varsBlok, ";", "START", operators, "FINISH";
```

```
varsBlok = "VAR", "INT32_t", identifier, {commaAndIdentifier };
```

```
identifier = low_letter, low_letter, low_letter, low_letter, low_letter, low_letter, low_letter, low_letter;
```

```
commaAndIdentifier = ",", identifier;
```

```
statement = "START", write | read | assignment | ifStatement | goto_statement | labelRule | forToOrDownToDoRule | while | repeatUntil, "FINISH";
```

```
statement_for_while = {statement | whileContinue | whileExit};
```

```
statements_for_while = {statement_for_while};
```

```
statements = {statement};
```

```
read = "READ", identifier;
```

```
write = "WRITE", equation | stringRule;
```

```
assignment = identifier, "<==", equation;
```

```
cycle_counter = identifier;
```

```
cycle_counter_last_value = equation;
```

```
ifStatement = "IF", "(", equation, ")", statements_for_while, ";" [ "ELSE", statements_for_while, ";" ];
```

```
goto_statement = "GOTO", identifier;
```

```
labelRule = identifier, ":";
```

```
forToOrDownToDoRule = "FOR", cycle_counter, "<==", equation, "TO" | "DOWNT", cycle_counter_last_value, "DO", statements, ";";
```

```

while = "WHILE", "(", equation, ")", "START", statements_for_while, "END",
"WHILE";

whileContinue = "CONTINUE", "WHILE";

whileExit = "EXIT", "WHILE";

repeatUntil = "REPEAT", statements, "UNTIL", equation;

equation = signedNumber | identifier | notRule { operationAndIdentOrNumber
| equation };

notRule = notOperation, signedNumber | identifier | equation;

operationAndIdentOrNumber = mult | arithmetic | logic | compare
signedNumber | identifier | equation;

arithmetic = "ADD" | "SUB";

mult = "MUL" | "DIV" | "MOD";

logic = "&" | "|";

notOperation = "!";

compare = "=" | "<>" | "LE" | "GE";

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
PROGRAM	Початок програми
START	Початок тексту програми
VAR	Початок блоку опису змінних
FINISH	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
DOWNT0	Декремент циклу
DO	Початок тіла циклу
WHILE	Оператор циклу
CONTINUE	Оператор циклу
EXIT	Оператор циклу
REPEAT	Початок тіла циклу
UNTIL	Оператор циклу
ADD	Оператор додавання
SUB	Оператор віднімання
MUL	Оператор множення

DIV	Оператор ділення
MOD	Оператор знаходження залишку від ділення
=	Оператор перевірки на рівність
<>	Оператор перевірки на нерівність
LE	Оператор перевірки чи менше
GE	Оператор перевірки чи більше
!	Оператор логічного заперечення
&	Оператор кон'юнкції
	Оператор диз'юнкції
INT32_t	32-ох розрядні знакові цілі
\$\$...	Коментар
,	Розділювач
(Відкриваюча дужка
)	Закриваюча дужка

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

3.2 Проектування таблиць транслятора

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

- 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,
    LBraket,
    RBraket,
```

```
Semicolon,  
Colon,  
Comma,  
Unknown  
};
```

Якщо у стовпці «Значення» відсутня інформація про токен, то це означає що його значення визначається користувачем під час написання коду на створеній мові програмування.

Таблиця 2 Опис термінальних символів та ключових слів

Токен	Значення
Program	PROGRAM
Start	START
Vars	VAR
End	FINISH
VarType	INT32_t
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	ADD
Subtraction	SUB
Multiplication	MUL
Division	DIV
Mod	MOD
Equal	=
NotEqual	<>
Less	LE
Greate	GE
Not	!
And	&
Or	
Identifier	
Number	
Unknown	
Comma	,
Semicolon	;
LBracket	(
RBracket)
LComment	\$\$
Comment	

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

На даному рисунку зображена блок-схема роботи лексичного аналізатора, детальний опис аналізатора розписаний у пункті 3.3.2

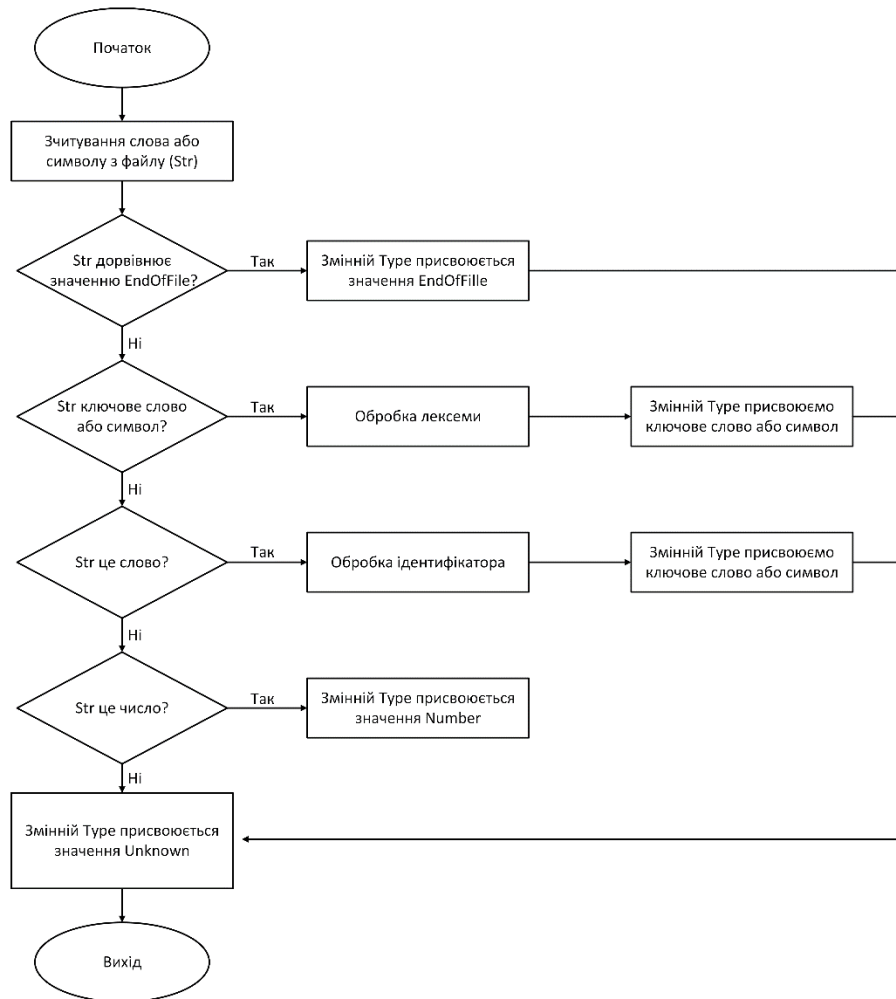


Рис. 3.1 Блок-схема роботи лексичного аналізатора

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 зарезервованою 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 вхідного 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 вхідний 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

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 Опис програми реалізації семантичного аналізатора

Семантичний аналізатор забезпечує перевірку смислової коректності програми, тобто перевіряє відповідність значень змінних, типів даних, ідентифікаторів та інших конструкцій заданим правилам мови програмування. Основою семантичного аналізатора є аналіз вмісту синтаксичного дерева, створеного на етапі синтаксичного аналізу.

Семантичний аналіз у нашому випадку буде реалізований у функції, яка опрацьовує оголошення і використання ідентифікаторів:

```
unsigned int IdIdentification(Id IdTable[], Token TokenTable[], unsigned int tokenCount, FILE* errFile)
{
    unsigned int idCount = 0;
    unsigned int i = 0;

    while (TokenTable[i++].type != Variable);

    if (TokenTable[i++].type == Type)
    {
        while (TokenTable[i].type != Semicolon)
        {
            if (TokenTable[i].type == Identifier)
            {
                int yes = 0;
                for (unsigned int j = 0; j < idCount; j++)
                {
                    if (!strcmp(TokenTable[i].name, IdTable[j].name))
                    {
                        yes = 1;
                        break;
                    }
                }
            }
        }
    }
}
```

```

    }
}
if (yes == 1)
{
    printf("\nidentifier \"%s\" is already declared !\n", TokenTable[i].name);
    return idCount;
}

if (idCount < MAX_IDENTIFIER)
{
    strcpy_s(IdTable[idCount++].name, TokenTable[i++].name);
}
else
{
    printf("\nToo many identifiers !\n");
    return idCount;
}
}
else
    i++;
}
}

for (; i < tokenCount; i++)
{
    if (TokenTable[i].type == Identifier && TokenTable[i + 1].type != Colon)
    {
        int yes = 0;
        for (unsigned int j = 0; j < idCount; j++)
        {
            if (!strcmp(TokenTable[i].name, IdTable[j].name))
            {
                yes = 1;
                break;
            }
        }
        if (yes == 0)
        {
            if (idCount < MAX_IDENTIFIER)
            {
                strcpy_s(IdTable[idCount++].name, TokenTable[i].name);
            }
            else
            {
                printf("\nToo many identifiers!\n");
                return idCount;
            }
        }
    }
}

return idCount;
}

```


На рис. 3.2 зображена граф-схема синтаксичного аналізатора.

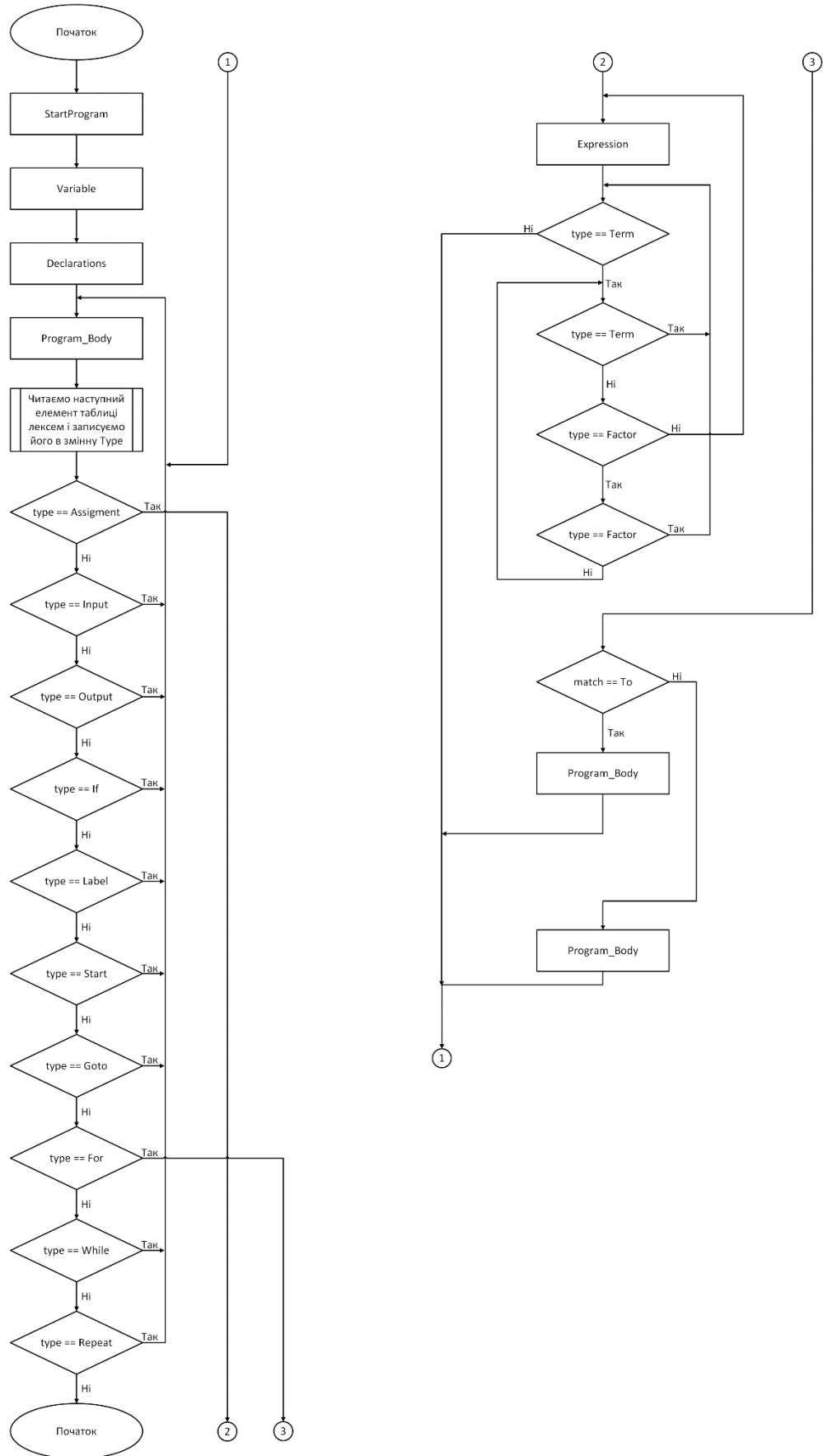


Рис. 3.2 Блок-схема роботи синтаксичного аналізатора

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 блок-схеми роботи генератора коду

На рис. 3.3 зображена граф-схема генератора коду.

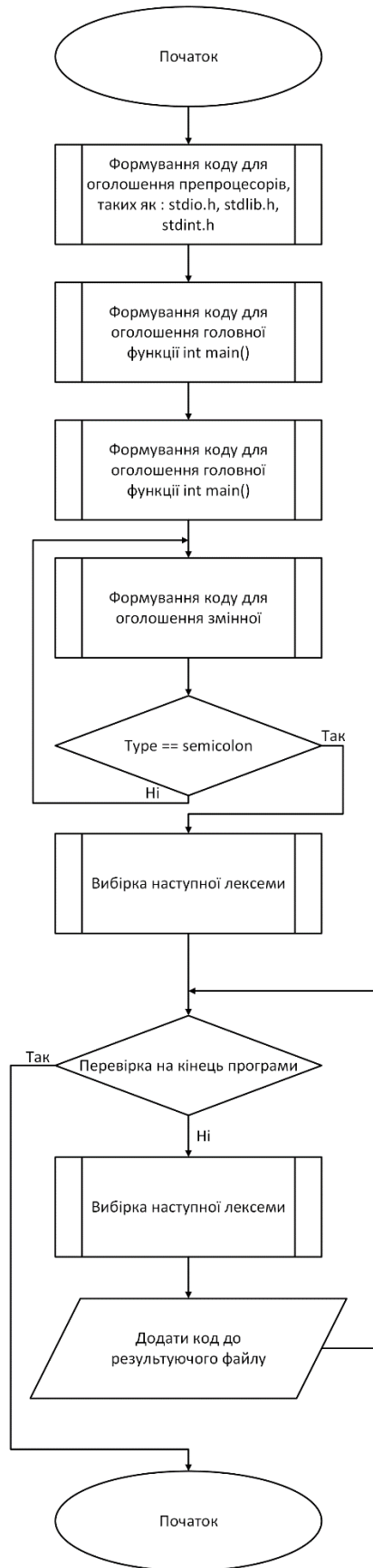


Рис. 3.3 Блок-схема роботи генератора код

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 b01. 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_b01.exe <ім'я програми>.b01"

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 перевірки 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
PROGRAM prog1;
VAR INT32_t  aaaaaa aa, bbbbbbbb, xxxxxxxx, yyyyyyyy;
START
READ  aaaaaaaa
READ  bbbbbbbb;
WRITE  aaaaaaaa ADD  bbbbbbbb;
WRITE  aaaaaaaa SUB  bbbbbbbb;
WRITE  aaaaaaaa MUL  bbbbbbbb;
WRITE  aaaaaaaa DIV  bbbbbbbb;
WRITE  aaaaaaaa MOD  bbbbbbbb;

xxxxxxx<==( aaaaaaaa SUB  bbbbbbbb) MUL 10 ADD ( aaaaaaaa ADD  bbbbbbbb) DIV 10;
yyyyyyy<== xxxxxxxx ADD ( xxxxxxxx MOD 10);
WRITE xxxxxxxx;
WRITE yyyyyyy;
```

FINISH

Текст файлу з повідомленнями про помилки

Lexical Error: line 3, lexem aaaaaa is Unknown

Lexical Error: line 3, lexem aa is Unknown

Syntax error in line 3 : another type of lexeme was expected.

Syntax error: type Unknown

Expected Type: Identifier

5.2 Виявлення семантичних помилок

Суттю виявлення семантичних помилок є перевірка числових констант на відповідність типу INT32_t, тобто знаковому цілому числу з відповідним діапазоном значень і перевірку на коректність використання змінних INT32_t у цілочисельних і логічних виразах.

5.3 Загальна перевірка коректності роботи транслятора

Для того щоб здійснити перевірку коректності роботи транслятора необхідно завантажити коректну до заданої вхідної мови програму.

Текст коректної програми

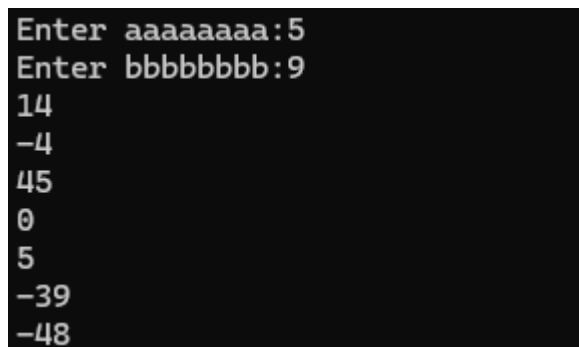
```
$$Prog1
PROGRAM prog1;
VAR INT32_t aaaaaaaa, bbbbbbbb, xxxxxxxx, yyyyyyyy;
START
READ aaaaaaaa;
READ bbbbbbbb;
WRITE aaaaaaaa ADD bbbbbbbb;
WRITE aaaaaaaa SUB bbbbbbbb;
WRITE aaaaaaaa MUL bbbbbbbb;
WRITE aaaaaaaa DIV bbbbbbbb;
WRITE aaaaaaaa MOD bbbbbbbb;

xxxxxxx<==( aaaaaaaa SUB bbbbbbbb) MUL 10 ADD ( aaaaaaaa ADD bbbbbbbb) DIV 10;
yyyyyyy<== xxxxxxxx ADD ( xxxxxxxx MOD 10);
WRITE xxxxxxxx;
WRITE yyyyyyyy;
FINISH
```

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 отримаємо наступний результат роботи програми:

A screenshot of a terminal window with a black background and white text. The text shows two input prompts followed by a series of numerical outputs.

```
Enter aaaaaaaa:5
Enter bbbbbbbb: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.

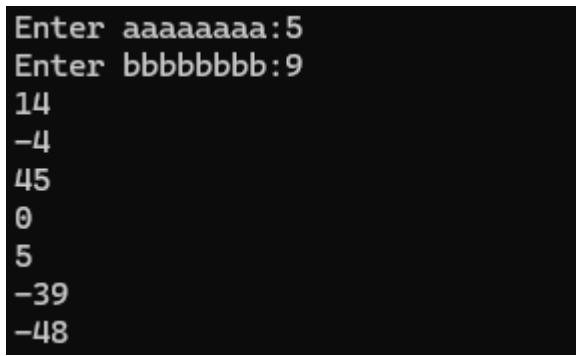
5.4 Тестова програма №1

Текст програми

```
$$Prog1
PROGRAM prog1;
VAR INT32_t aaaaaaaa, bbbbbbbb, xxxxxxxx, yyyyyyyy;
START
READ aaaaaaaa;
READ bbbbbbbb;
WRITE aaaaaaaa ADD bbbbbbbb;
WRITE aaaaaaaa SUB bbbbbbbb;
WRITE aaaaaaaa MUL bbbbbbbb;
WRITE aaaaaaaa DIV bbbbbbbb;
WRITE aaaaaaaa MOD bbbbbbbb;

xxxxxxx<==( aaaaaaaa SUB bbbbbbbb) MUL 10 ADD ( aaaaaaaa ADD bbbbbbbb) DIV 10;
yyyyyyy<== xxxxxxxx ADD ( xxxxxxxx MOD 10);
WRITE xxxxxxxx;
WRITE yyyyyyyy;
FINISH
```

Результат виконання



```
Enter aaaaaaaa:5
Enter bbbbbbbb:9
14
-4
45
0
5
-39
-48
```

Рис. 5.2 Результат виконання тестової програми №1

5.5 Тестова програма №2

Текст програми

```
$$Prog2
PROGRAM prog2;
VAR INT32_t aaaaaaaaa, bbbbbbbb, ccccccc;
START
READ aaaaaaaa;
READ bbbbbbbb;
READ ccccccc;
IF( aaaaaaaa GE bbbbbbbb)
START
    IF( aaaaaaaa GE ccccccc)
    START
        GOTO Abigger;
    FINISH
    ELSE
    START
        WRITE ccccccc;
        GOTO OutofIF;
        Abigger:
        WRITE aaaaaaaa;
        GOTO OutofIF;
    FINISH
FINISH
IF( bbbbbbbb LE ccccccc)
START
    WRITE ccccccc;
FINISH
ELSE
START
    WRITE bbbbbbbb;
FINISH
OutofIF:

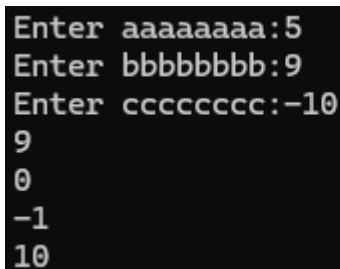
IF(( aaaaaaaa = bbbbbbbb) & ( aaaaaaaa = ccccccc) & ( bbbbbbbb = ccccccc))
START
    WRITE 1;
FINISH
ELSE
START
    WRITE 0;
FINISH
IF(( aaaaaaaa LE 0) | ( bbbbbbbb LE 0) | ( ccccccc LE 0))
START
    WRITE -1;
FINISH
```

```

ELSE
START
    WRITE 0;
FINISH
IF(!( aaaaaaaa LE ( bbbbbbbb ADD cccccccc)))
START
    WRITE(10);
FINISH
ELSE
START
    WRITE(0);
FINISH
FINISH

```

Результат виконання



```

Enter aaaaaaaa:5
Enter bbbbbbbb:9
Enter cccccccc:-10
9
0
-1
10

```

Рис. 5.3 Результат виконання тестової програми №2

5.6 Тестова програма №3

Текст програми

```

$$Prog3
PROGRAM prog3;
VAR INT32_t aaaaaaaa, aaaaaa2, bbbbbbbb, xxxxxxxx, ccccccc1, ccccccc2;
START
READ aaaaaaaa;
READ bbbbbbbb;
FOR aaaaaa2<== aaaaaaaa TO bbbbbbbb DO
    WRITE aaaaaa2 MUL aaaaaa2;

FOR aaaaaa2<== bbbbbbbb TO aaaaaaaa DO
    WRITE aaaaaa2 MUL aaaaaa2;

xxxxxxx<==0;
ccccccc1<==0;
WHILE ccccccc1 LE aaaaaaaa
START
    ccccccc2<==0;
    WHILE ccccccc2 LE bbbbbbbb
    START
        xxxxxxxx<== xxxxxxxx ADD 1;
        ccccccc2<== ccccccc2 ADD 1;
    END
    END
END

```

```

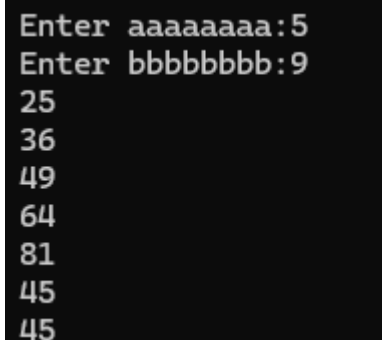
    FINISH
  END WHILE
  ccccccc1<== ccccccc1 ADD 1;
FINISH
END WHILE
WRITE xxxxxxxx;

xxxxxxx<==0;
cccccc1<==1;
REPEAT
START
  ccccccc2<==1;
  REPEAT
  START
    xxxxxxxx<== xxxxxxxx ADD 1;
    ccccccc2<== ccccccc2 ADD 1;
    FINISH
  UNTIL !( ccccccc2 GE bbbbbb)
  ccccccc1<== ccccccc1 ADD 1;
  FINISH
UNTIL !( ccccccc1 GE aaaaaaa)
WRITE xxxxxxxx;

FINISH

```

Результат виконання



```

Enter aaaaaaaa:5
Enter bbbbbbbb:9
25
36
49
64
81
45
45

```

Рис. 5.4 Результат виконання тестової програми №3

6. Верефікація 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

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування b01, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створено компілятор мови програмування b01, а саме:

2.1. Розроблено прямий лексичний аналізатор, орієнтований на розпізнавання лексем, що є заявлені в формальному описі мови програмування.

2.2. Розроблено синтаксичний аналізатор на основі низхідного методу. Складено деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура

2.3. Розроблено генератор коду, відповідні процедури якого викликаються після перевірки синтаксичним аналізатором коректності запису чергового оператора, мови програмування b01. Вихідним кодом генератора є програма на мові C.

3. Проведене тестування компілятора на тестових програмах за наступними пунктами:

3.1. На виявлення лексичних помилок.

3.2. На виявлення синтаксичних помилок.

3.3. Загальна перевірка роботи компілятора.

Тестування не виявило помилок в роботі компілятор, і всі помилки в тестових програмах на мові b01 були успішно виявлені і відповідно оброблені.

В результаті виконання даної курсового проекту було засвоєно методи розробки та реалізації компонент систем програмування.

Список використаної літератури

1. C Programming Language Tutorial - GeeksforGeeks
URL: [C Programming Language Tutorial - 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. Вікіпедія
URL: [Wikipedia](#)
5. Stack Overflow
URL: [Stack Overflow - Where Developers Learn, Share, & Build Careers](#)
6. Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 - «Комп'ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. - Київ: КПІ ім. Ігоря Сікорського, 2021. - 108 с.
7. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. - Харків: НТУ «ХПІ», 2021. - 133 с.
8. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії формальних мов: 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.

Додатки

Додаток А. Таблиці лексем для тестових прикладів

Тестова програма “Лінійний алгоритм”

TOKEN TABLE				
line number	token	value	token code	type of token
2	PROGRAM	0	0	MainProgram
3	VAR	0	3	Variable
3	INT32_t	0	4	Integer
3	aaaaaaa	0	23	Identifier
3	,	0	42	Comma
3	bbbbbbb	0	23	Identifier
3	,	0	42	Comma
3	xxxxxxx	0	23	Identifier
3	,	0	42	Comma
3	yyyyyyyy	0	23	Identifier
3	;	0	40	Semicolon
4	START	0	2	StartProgram
5	READ	0	6	Input
5	aaaaaaa	0	23	Identifier
5	;	0	40	Semicolon
6	READ	0	6	Input
6	bbbbbbb	0	23	Identifier
6	;	0	40	Semicolon
7	WRITE	0	7	Output

	7	aaaaaaa	0	23	Identifier

	7	ADD	0	26	Add

	7	bbbbbbb	0	23	Identifier

	7	;;	0	40	Semicolon

	8	WRITE	0	7	Output

	8	aaaaaaa	0	23	Identifier

	8	SUB	0	27	Sub

	8	bbbbbbb	0	23	Identifier

	8	;;	0	40	Semicolon

	9	WRITE	0	7	Output

	9	aaaaaaa	0	23	Identifier

	9	MUL	0	28	Mul

	9	bbbbbbb	0	23	Identifier

	9	;;	0	40	Semicolon

	10	WRITE	0	7	Output

	10	aaaaaaa	0	23	Identifier

	10	DIV	0	29	Div

	10	bbbbbbb	0	23	Identifier

	10	;;	0	40	Semicolon

	11	WRITE	0	7	Output

	11	aaaaaaa	0	23	Identifier

	11	MOD	0	30	Mod

	11	bbbbbbb	0	23	Identifier

	11	;;	0	40	Semicolon

```

-----
| 13 | xxxxxxxx | 0 | 23 | Identifier |
-----
| 13 | <== | 0 | 25 | Assign |
-----
| 13 | ( | 0 | 38 | LBraket |
-----
| 13 | aaaaaaaa | 0 | 23 | Identifier |
-----
| 13 | SUB | 0 | 27 | Sub |
-----
| 13 | bbbbbbbb | 0 | 23 | Identifier |
-----
| 13 | ) | 0 | 39 | RBraket |
-----
| 13 | MUL | 0 | 28 | Mul |
-----
| 13 | 10 | 10 | 24 | Number |
-----
| 13 | ADD | 0 | 26 | Add |
-----
| 13 | ( | 0 | 38 | LBraket |
-----
| 13 | aaaaaaaa | 0 | 23 | Identifier |
-----
| 13 | ADD | 0 | 26 | Add |
-----
| 13 | bbbbbbbb | 0 | 23 | Identifier |
-----
| 13 | ) | 0 | 39 | RBraket |
-----
| 13 | DIV | 0 | 29 | Div |
-----
| 13 | 10 | 10 | 24 | Number |
-----
| 13 | ; | 0 | 40 | Semicolon |
-----
| 14 | yyyyyyyy | 0 | 23 | Identifier |
-----
| 14 | <== | 0 | 25 | Assign |
-----
| 14 | xxxxxxxx | 0 | 23 | Identifier |
-----
| 14 | ADD | 0 | 26 | Add |
-----
| 14 | ( | 0 | 38 | LBraket |
-----
| 14 | xxxxxxxx | 0 | 23 | Identifier |

```

	14	MOD	0	30 Mod	

	14	10	10	24 Number	

	14)	0	39 RBracket	

	14	;;	0	40 Semicolon	

	15	WRITE	0	7 Output	

	15	xxxxxxx	0	23 Identifier	

	15	;;	0	40 Semicolon	

	16	WRITE	0	7 Output	

	16	yyyyyyyy	0	23 Identifier	

	16	;;	0	40 Semicolon	

	17	FINISH	0	5 EndProgram	

Тестова програма “Алгоритм з розгалуженням”

	TOKEN TABLE				

	line number		token		value token code type of token

	2		PROGRAM		0 0 MainProgram

	3		VAR		0 3 Variable

	3		INT32_t		0 4 Integer

	3		aaaaaaaa		0 23 Identifier

	3		,		0 42 Comma

	3		bbbbbbbb		0 23 Identifier

	3		,		0 42 Comma

	3		cccccccc		0 23 Identifier

	3		;		0 40 Semicolon

	4		START		0 2 StartProgram

	5	READ	0	6 Input	

	5	aaaaaaa	0	23 Identifier	

	5	;;	0	40 Semicolon	

	6	READ	0	6 Input	

	6	bbbbbbbb	0	23 Identifier	

	6	;;	0	40 Semicolon	

	7	READ	0	6 Input	

	7	ccccccc	0	23 Identifier	

	7	;;	0	40 Semicolon	

	8	IF	0	8 If	

	8	(0	38 LBraket	

	8	aaaaaaa	0	23 Identifier	

	8	GE	0	33 Greate	

	8	bbbbbbbb	0	23 Identifier	

	8)	0	39 RBraket	

	9	START	0	2 StartProgram	

	10	IF	0	8 If	

	10	(0	38 LBraket	

	10	aaaaaaa	0	23 Identifier	

	10	GE	0	33 Greate	

	10	ccccccc	0	23 Identifier	

	10)	0	39 RBraket	

	11	START	0	2 StartProgram	

	12	GOTO	0	11 Goto	

```

-----
| 12 | Abigger | 0 | 23 | Identifier |
-----
| 12 | ; | 0 | 40 | Semicolon |
-----
| 13 | FINISH | 0 | 5 | EndProgram |
-----
| 14 | ELSE | 0 | 10 | Else |
-----
| 15 | START | 0 | 2 | StartProgram |
-----
| 16 | WRITE | 0 | 7 | Output |
-----
| 16 | cccceccc | 0 | 23 | Identifier |
-----
| 16 | ; | 0 | 40 | Semicolon |
-----
| 17 | GOTO | 0 | 11 | Goto |
-----
| 17 | OutoIF | 0 | 23 | Identifier |
-----
| 17 | ; | 0 | 40 | Semicolon |
-----
| 18 | Abigger | 0 | 12 | Label |
-----
| 19 | WRITE | 0 | 7 | Output |
-----
| 19 | aaaaaaaa | 0 | 23 | Identifier |
-----
| 19 | ; | 0 | 40 | Semicolon |
-----
| 20 | GOTO | 0 | 11 | Goto |
-----
| 20 | OutoIF | 0 | 23 | Identifier |
-----
| 20 | ; | 0 | 40 | Semicolon |
-----
| 21 | FINISH | 0 | 5 | EndProgram |
-----
| 22 | FINISH | 0 | 5 | EndProgram |
-----
| 23 | IF | 0 | 8 | If |
-----
| 23 | ( | 0 | 38 | LBraket |
-----
| 23 | bbbbbbbb | 0 | 23 | Identifier |
-----
| 23 | LE | 0 | 34 | Less |

```


	23	cccccccc	0	23	Identifier	
	23)	0	39	RBracket	
	24	START	0	2	StartProgram	
	25	WRITE	0	7	Output	
	25	cccccccc	0	23	Identifier	
	25	;	0	40	Semicolon	
	26	FINISH	0	5	EndProgram	
	27	ELSE	0	10	Else	
	28	START	0	2	StartProgram	
	29	WRITE	0	7	Output	
	29	bbbbbbbbb	0	23	Identifier	
	29	;	0	40	Semicolon	
	30	FINISH	0	5	EndProgram	
	31	Outoff	0	12	Label	
	33	IF	0	8	If	
	33	(0	38	LBracket	
	33	(0	38	LBracket	
	33	aaaaaaaa	0	23	Identifier	
	33	=	0	31	Equality	
	33	bbbbbbbbb	0	23	Identifier	
	33)	0	39	RBracket	
	33	&	0	36	And	
	33	(0	38	LBracket	
	33	aaaaaaaa	0	23	Identifier	

	33		=	

	33		cccccccc	

	33)	

	33		&	

	33		(

	33		bbbbbbbbb	

	33		=	

	33		cccccccc	

	33)	

	33)	

	34		START	

	35		WRITE	

	35		1	

	35		;	

	36		FINISH	

	37		ELSE	

	38		START	

	39		WRITE	

	39		0	

	39		;	

	40		FINISH	

	41		IF	

	41		(

	41		(

	41	aaaaaaaa	0	23 Identifier

	41	LE	0	34 Less

	41	0	0	24 Number

	41)	0	39 RBracket

	41		0	37 Or

	41	(0	38 LBracket

	41	bbbbbbbb	0	23 Identifier

	41	LE	0	34 Less

	41	0	0	24 Number

	41)	0	39 RBracket

	41		0	37 Or

	41	(0	38 LBracket

	41	cccccccc	0	23 Identifier

	41	LE	0	34 Less

	41	0	0	24 Number

	41)	0	39 RBracket

	41)	0	39 RBracket

	42	START	0	2 StartProgram

	43	WRITE	0	7 Output

	43	-	0	43 Unknown

	43	1	1	24 Number

	43	;	0	40 Semicolon

	44	FINISH	0	5 EndProgram

	45	ELSE	0	10 Else

	46		START	0 2 StartProgram

	47		WRITE	0 7 Output

	47		0	0 24 Number

	47		;	0 40 Semicolon

	48		FINISH	0 5 EndProgram

	49		IF	0 8 If

	49		(0 38 LBracket

	49		!	0 35 Not

	49		(0 38 LBracket

	49		aaaaaaa	0 23 Identifier

	49		LE	0 34 Less

	49		(0 38 LBracket

	49		bbbbbbbb	0 23 Identifier

	49		ADD	0 26 Add

	49		cececece	0 23 Identifier

	49)	0 39 RBracket

	49)	0 39 RBracket

	49)	0 39 RBracket

	50		START	0 2 StartProgram

	51		WRITE	0 7 Output

	51		(0 38 LBracket

	51		10	10 24 Number

	51)	0 39 RBracket

	51		;	0 40 Semicolon

52	FINISH	0	5	EndProgram
53	ELSE	0	10	Else
54	START	0	2	StartProgram
55	WRITE	0	7	Output
55	(0	38	LBracket
55		0	24	Number
55)	0	39	RBracket
55	;	0	40	Semicolon
56	FINISH	0	5	EndProgram
57	FINISH	0	5	EndProgram

Тестова програма “Циклічний алгоритм”

TOKEN TABLE				
line number	token	value	token code	type of token
2	PROGRAM	0	0	MainProgram
2	prog3	0	1	Unknown
3	VAR	0	3	Variable
3	INT32_t	0	4	Integer
3	aaaaaaaa	0	23	Identifier
3	,	0	42	Comma
3	aaaaaaa2	0	23	Identifier
3	,	0	42	Comma
3	bbbbbbbbb	0	23	Identifier
3	,	0	42	Comma

3	xxxxxxx	0	23	Identifier
3	,	0	42	Comma
3	cccccc1	0	23	Identifier
3	,	0	42	Comma
3	cccccc2	0	23	Identifier
3	;	0	40	Semicolon
4	START	0	2	StartProgram
5	READ	0	6	Input
5	aaaaaaa	0	23	Identifier
5	;	0	40	Semicolon
6	READ	0	6	Input
6	bbbbbbbb	0	23	Identifier
6	;	0	40	Semicolon
7	FOR	0	13	For
7	aaaaaaa2	0	23	Identifier
7	<==	0	25	Assign
7	aaaaaaa	0	23	Identifier
7	TO	0	14	To
7	bbbbbbbb	0	23	Identifier
7	DO	0	16	Do
8	WRITE	0	7	Output
8	aaaaaaa2	0	23	Identifier
8	MUL	0	28	Mul
8	aaaaaaa2	0	23	Identifier

	8	;	0	40 Semicolon	

	10	FOR	0	13 For	

	10	aaaaaaa2	0	23 Identifier	

	10	<==	0	25 Assign	

	10	bbbbbbbb	0	23 Identifier	

	10	TO	0	14 To	

	10	aaaaaaaa	0	23 Identifier	

	10	DO	0	16 Do	

	11	WRITE	0	7 Output	

	11	aaaaaaa2	0	23 Identifier	

	11	MUL	0	28 Mul	

	11	aaaaaaa2	0	23 Identifier	

	11	;	0	40 Semicolon	

	13	xxxxxxx	0	23 Identifier	

	13	<==	0	25 Assign	

	13	0	0	24 Number	

	13	;	0	40 Semicolon	

	14	cececcc1	0	23 Identifier	

	14	<==	0	25 Assign	

	14	0	0	24 Number	

	14	;	0	40 Semicolon	

	15	WHILE	0	17 While	

	15	cececcc1	0	23 Identifier	

	15	LE	0	34 Less	

	15	aaaaaaa	0	23	Identifier

	16	START	0	2	StartProgram

	17	ccccccc2	0	23	Identifier

	17	<==	0	25	Assign

	17	0	0	24	Number

	17	;;	0	40	Semicolon

	18	WHILE	0	17	While

	18	ccccccc2	0	23	Identifier

	18	LE	0	34	Less

	18	bbbbbbbb	0	23	Identifier

	19	START	0	2	StartProgram

	20	xxxxxxx	0	23	Identifier

	20	<==	0	25	Assign

	20	xxxxxxx	0	23	Identifier

	20	ADD	0	26	Add

	20	1	1	24	Number

	20	;;	0	40	Semicolon

	21	ccccccc2	0	23	Identifier

	21	<==	0	25	Assign

	21	ccccccc2	0	23	Identifier

	21	ADD	0	26	Add

	21	1	1	24	Number

	21	;;	0	40	Semicolon

	22	FINISH	0	5	EndProgram

23	END	0	20	End
23	WHILE	0	17	While
24	ccccccc1	0	23	Identifier
24	<==	0	25	Assign
24	ccccccc1	0	23	Identifier
24	ADD	0	26	Add
24	1	1	24	Number
24	;	0	40	Semicolon
25	FINISH	0	5	EndProgram
26	END	0	20	End
26	WHILE	0	17	While
27	WRITE	0	7	Output
27	xxxxxxx	0	23	Identifier
27	;	0	40	Semicolon
29	xxxxxxx	0	23	Identifier
29	<==	0	25	Assign
29	0	0	24	Number
29	;	0	40	Semicolon
30	ccccccc1	0	23	Identifier
30	<==	0	25	Assign
30	1	1	24	Number
30	;	0	40	Semicolon
31	REPEAT	0	21	Repeat
32	START	0	2	StartProgram

	33	ccccccc2		0	23 Identifier

	33	<==		0	25 Assign

	33	1		1	24 Number

	33	;		0	40 Semicolon

	34	REPEAT		0	21 Repeat

	35	START		0	2 StartProgram

	36	xxxxxxx		0	23 Identifier

	36	<==		0	25 Assign

	36	xxxxxxx		0	23 Identifier

	36	ADD		0	26 Add

	36	1		1	24 Number

	36	;		0	40 Semicolon

	37	ccccccc2		0	23 Identifier

	37	<==		0	25 Assign

	37	ccccccc2		0	23 Identifier

	37	ADD		0	26 Add

	37	1		1	24 Number

	37	;		0	40 Semicolon

	38	FINISH		0	5 EndProgram

	39	UNTIL		0	22 Until

	39	!		0	35 Not

	39	(0	38 LBracket

	39	ccccccc2		0	23 Identifier

	39	GE		0	33 Greater

	39	bbbbbbbb	0	23 Identifier

	39)	0	39 RBraket

	40	ccccccc1	0	23 Identifier

	40	<==	0	25 Assign

	40	ccccccc1	0	23 Identifier

	40	ADD	0	26 Add

	40	1	1	24 Number

	40	;	0	40 Semicolon

	41	FINISH	0	5 EndProgram

	42	UNTIL	0	22 Until

	42	!	0	35 Not

	42	(0	38 LBraket

	42	ccccccc1	0	23 Identifier

	42	GE	0	33 Greate

	42	aaaaaaaa	0	23 Identifier

	42)	0	39 RBraket

	43	WRITE	0	7 Output

	43	xxxxxxxx	0	23 Identifier

	43	;	0	40 Semicolon

	45	FINISH	0	5 EndProgram

Додаток Б. Код на мові С, отриманий на виході транслятора для тестових прикладів

Prog1.c

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

int main()
{
    int aaaaaaaa, bbbbbbbb, xxxxxxxx, yyyyyyyy;
    printf("Enter aaaaaaaa:");
    scanf("%d", &aaaaaaa);
    printf("Enter bbbbbbbb:");
    scanf("%d", &bbbbbbb);
    printf("%d\n", aaaaaaaa + bbbbbbbb);
    printf("%d\n", aaaaaaaa - bbbbbbbb);
    printf("%d\n", aaaaaaaa * bbbbbbbb);
    printf("%d\n", aaaaaaaa / bbbbbbbb);
    printf("%d\n", aaaaaaaa % bbbbbbbb);
    xxxxxxxx = (aaaaaaa - bbbbbbbb) * 10 + (aaaaaaa + bbbbbbbb) / 10;
    yyyyyyyy = xxxxxxxx + (xxxxxxx % 10);
    printf("%d\n", xxxxxxxx);
    printf("%d\n", yyyyyyyy);
    system("pause");
    return 0;
}
```

Prog2.c

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

int main()
{
    int aaaaaaaa, bbbbbbbb, cccccccc;
    printf("Enter aaaaaaaa:");
    scanf("%d", &aaaaaaa);
    printf("Enter bbbbbbbb:");
    scanf("%d", &bbbbbbb);
    printf("Enter cccccccc:");
    scanf("%d", &ccccccc);
    if ((aaaaaaa > bbbbbbbb))
    {
        if ((aaaaaaa > cccccccc))
        {
            goto Abigger;
        }
        else
        {
            printf("%d\n", cccccccc);
            goto OutofIF;
        }
    }
    Abigger:
    printf("%d\n", aaaaaaaa);
    goto OutofIF;
}
if ((bbbbbbb < cccccccc))
{
    printf("%d\n", cccccccc);
}
else
{
    printf("%d\n", bbbbbbbb);
}
OutofIF:
if (((aaaaaaa == bbbbbbbb) && (aaaaaaa == cccccccc) && (bbbbbbb == cccccccc)))
{
    printf("%d\n", 1);
}
else
{
    printf("%d\n", 0);
}
if (((aaaaaaa < 0) || (bbbbbbb < 0) || (ccccccc < 0)))
{
    printf("%d\n", -1);
}
else
{
    printf("%d\n", 0);
}
if (!(aaaaaaa < (bbbbbbb + cccccccc)))
{
    printf("%d\n", (10));
}
else
{
    printf("%d\n", (0));
}
system("pause");
return 0;
}
```

Prog3.c

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

int main()
{
    int aaaaaaaa, aaaaaaa2, bbbbbbbb, xxxxxxxx, ccccccc1, ccccccc2;
    printf("Enter aaaaaaaa:");
    scanf("%d", &aaaaaaa);
    printf("Enter bbbbbbbb:");
    scanf("%d", &bbbbbbb);
    for (int aaaaaaa2 = aaaaaaaa; aaaaaaa2 <= bbbbbbbb; aaaaaaa2++)
    printf("%d\n", aaaaaaa2 * aaaaaaa2);
    for (int aaaaaaa2 = bbbbbbbb; aaaaaaa2 <= aaaaaaaa; aaaaaaa2++)
    printf("%d\n", aaaaaaa2 * aaaaaaa2);
    xxxxxxxx = 0;
    ccccccc1 = 0;
    while (cccccc1 < aaaaaaaa)
    {
        {
            ccccccc2 = 0;
            while (cccccc2 < bbbbbbbb)
            {
                xxxxxxxx = xxxxxxxx + 1;
                ccccccc2 = ccccccc2 + 1;
            }
        }
        ccccccc1 = ccccccc1 + 1;
    }
}
```

```

}
}
ccccccc1 = ccccccc1 + 1;
}
}
printf("%d\n", xxxxxxx);
xxxxxxx = 0;
ccccccc1 = 1;
do
{
ccccccc2 = 1;
do
{
xxxxxxx = xxxxxxx + 1;
ccccccc2 = ccccccc2 + 1;
}
while (!(ccccccc2 > bbbbbbbb));
ccccccc1 = ccccccc1 + 1;
}
while (!(ccccccc1 > aaaaaaaa));
printf("%d\n", xxxxxxx);
system("pause");
return 0;
}

```

Додаток В. Абстрактне синтаксичне дерево для тестових прикладів

Тестова програма «Лінійний алгоритм»

```

|-- program
|  |-- var
|  |  |-- yyyyyyyy
|  |  |-- var
|  |  |  |-- xxxxxxxx
|  |  |  |-- var
|  |  |  |  |-- bbbbbbbb
|  |  |  |  |-- var
|  |  |  |  |  |-- aaaaaaaa
|  |-- statement
|  |  |-- statement
|  |  |  |-- statement
|  |  |  |  |-- statement
|  |  |  |  |  |-- statement
|  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |  |  |  |-- input
|  |  |  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |  |  |-- input
|  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |  |-- +
|  |  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |  |-- -
|  |  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |  |-- *
|  |  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |  |-- /
|  |  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |  |-- %
|  |  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |  |  |  |  |-- <==
|  |  |  |  |  |  |  |  |  |  |  |-- xxxxxxxx
|  |  |  |  |  |  |  |  |  |  |  |-- +
|  |  |  |  |  |  |  |  |  |  |  |-- *
|  |  |  |  |  |  |  |  |  |  |  |-- -
|  |  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |  |  |  |  |-- 10
|  |  |  |  |  |  |  |  |  |  |  |-- /
|  |  |  |  |  |  |  |  |  |  |  |-- +
|  |  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |  |  |  |  |-- 10
|  |  |  |  |  |  |  |  |  |  |  |-- <==
|  |  |  |  |  |  |  |  |  |  |  |-- yyyyyyyy

```

```

| | | | | |-- +
| | | | | |-- xxxxxxxx
| | | | | |-- %
| | | | | |-- xxxxxxxx
| | | | | |-- 10
| | | |-- output
| | | |-- xxxxxxxx
| | |-- output
| | |-- уууууууу

```

Тестова програма «Алгоритм з розгалуженням»

```

|-- program
| |-- var
| | |-- cccccccc
| | |-- var
| | | |-- bbbbbbbb
| | | |-- var
| | | | |-- aaaaaaaa
| |-- statement
| | |-- statement
| | | |-- statement
| | | | |-- statement
| | | | | |-- statement
| | | | | |-- statement
| | | | | |-- statement
| | | | | |-- input
| | | | | | |-- aaaaaaaa
| | | | | |-- input
| | | | | | |-- bbbbbbbb
| | | | | |-- input
| | | | | |-- cccccccc
| | | | |-- if
| | | | | |-- GE
| | | | | | |-- aaaaaaaa
| | | | | | |-- bbbbbbbb
| | | | | |-- branches
| | | | | |-- compound
| | | | | | |-- if
| | | | | | | |-- GE
| | | | | | | | |-- aaaaaaaa
| | | | | | | | |-- cccccccc
| | | | | | |-- branches
| | | | | | |-- compound
| | | | | | | |-- goto
| | | | | | | | |-- Abigger
| | | | | |-- compound
| | | | | |-- statement
| | | | | |-- statement
| | | | | |-- statement
| | | | | |-- output
| | | | | | |-- cccccccc
| | | | | |-- goto
| | | | | | |-- OutofIF
| | | | | | |-- Abigger
| | | | | |-- output
| | | | | | |-- aaaaaaaa
| | | | | |-- goto
| | | | | |-- OutofIF

```

```
-- if
-- LE
| | | |-- bbbbbbbb
| | | |-- cccccccc
| | | |-- branches
| | | |-- compound
| | | | |-- output
| | | | | |-- cccccccc
| | | |-- compound
| | | | |-- output
| | | | | |-- bbbbbbbb
    |-- OutofIF
-- if
-- &
| | | |-- &
| | | | |-- =
| | | | | |-- aaaaaaaa
| | | | | |-- bbbbbbbb
| | | | |-- =
| | | | | |-- aaaaaaaa
| | | | | |-- cccccccc
| | | | |-- =
| | | | | |-- bbbbbbbb
| | | | | |-- cccccccc
    |-- branches
        |-- compound
            |-- output
                |-- 1
        |-- compound
            |-- output
                |-- 0
-- if
    |-- |
        |-- |
            |-- LE
                |-- aaaaaaaa
                |-- 0
            |-- LE
                |-- bbbbbbbb
                |-- 0
        |-- LE
            |-- cccccccc
            |-- 0
    |-- branches
        |-- compound
            |-- output
                |-- -
                |-- 0
                |-- 1
        |-- compound
            |-- output
                |-- 0
-- if
    |-- !
        |-- LE
            |-- aaaaaaaa
            |-- +
                |-- bbbbbbbb
                |-- cccccccc
    |-- branches
        |-- compound
```



```
| | | | | |-- output
| | | | | |-- 10
| | | | |-- compound
| | | | | |-- output
| | | | | |-- 0
```

Тестова програма «Циклічний алгоритм»

```
-- program
|  |-- var
|  |  |-- cccccc2
|  |  |-- var
|  |  |  |-- cccccc1
|  |  |  |-- var
|  |  |  |  |-- xxxxxxxx
|  |  |  |  |-- var
|  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |-- var
|  |  |  |  |  |  |-- aaaaaaa2
|  |  |  |  |  |  |-- var
|  |  |  |  |  |  |-- aaaaaaaa
|
|  |-- statement
|  |  |-- statement
|  |  |  |-- statement
|  |  |  |  |-- statement
|  |  |  |  |  |-- statement
|  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |-- input
|  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |-- input
|  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |-- for-to
|  |  |  |  |  |  |  |-- <==
|  |  |  |  |  |  |  |  |-- aaaaaaa2
|  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |-- body
|  |  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |-- *
|  |  |  |  |  |  |  |  |  |  |-- aaaaaaa2
|  |  |  |  |  |  |  |  |  |  |-- aaaaaaa2
|  |  |  |  |  |  |-- for-to
|  |  |  |  |  |  |  |-- <==
|  |  |  |  |  |  |  |  |-- aaaaaaa2
|  |  |  |  |  |  |  |  |-- bbbbbbbb
|  |  |  |  |  |  |  |-- body
|  |  |  |  |  |  |  |  |-- aaaaaaaa
|  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |-- *
|  |  |  |  |  |  |  |  |  |  |-- aaaaaaa2
|  |  |  |  |  |  |  |  |  |  |-- aaaaaaa2
|  |  |  |  |  |  |-- <==
|  |  |  |  |  |  |  |-- xxxxxxxx
|  |  |  |  |  |  |  |-- 0
|  |  |  |  |  |-- <==
|  |  |  |  |  |  |-- cccccc1
```

```

-- 0
-- while
-- LE
-- cccccc1
-- aaaaaaaa
-- statement
-- compound
-- statement
-- statement
-- <==
-- cccccc2
-- 0
-- while
-- LE
-- cccccc2
-- bbbbbbbb
-- statement
-- compound
-- statement
-- <==
-- xxxxxxxx
-- +
-- xxxxxxxx
-- 1
-- <==
-- cccccc2
-- +
-- cccccc2
-- 1
-- <==
-- cccccc1
-- +
-- cccccc1
-- 1
-- output
-- xxxxxxxx
-- <==
-- xxxxxxxx
-- 0
-- <==
-- cccccc1
-- 1
-- repeat-until
-- body
-- compound
-- statement
-- statement
-- <==
-- cccccc2
-- 1
-- repeat-until
-- body
-- compound
-- statement
-- <==
-- xxxxxxxx
-- +
-- xxxxxxxx
-- 1
-- <==
-- cccccc2

```


Додаток Г. Документований текст програмних модулів (лістинги)

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,
    Greater,
    Less,
    Not,
    And,
    Or,
    LBracket,
    RBracket,
    Semicolon,
    Colon,
    Comma,
    Minus,
    Unknown
};

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

// 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
};

// перерахування, яке описує всі можливі вузли абстрактного синтаксичного дерева
enum TypeOfNodes
{
    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[16]; // 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(TokenType 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);
            }
        }
    }
}

// <ввід> = 'input' <ідентифікатор>
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);
}

// <вивід> = 'output' <ідентифікатор>
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();
    // Перевіряємо вираз 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();
    // Перевіряємо вираз 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(compound_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 піддерев
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");
    fprintf(outFile, "int main() {\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 + 1].name);
    while (TokenTable[pos].type == Comma)
    {
        fprintf(outFile, ", ");
        pos++;
        fprintf(outFile, TokenTable[pos + 1].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> = <присвоєння> | <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);
}
}

// <присвоєння> = <ідентифікатор> '=' <арифметичний вираз>
void gen_assignment(FILE* outFile)
{
fprintf(outFile, " ");
fprintf(outFile, TokenTable[pos+1].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> = <ідентифікатор> | <число> | '(' <арифметичний вираз> ')'
void gen_factor(FILE* outFile)
{
if (TokenTable[pos].type == Identifier || TokenTable[pos].type == Number)
{
fprintf(outFile, TokenTable[pos+1].name);
}
else
{
if (TokenTable[pos].type == LBracket)
{
{
fprintf(outFile, "(");
pos++;
gen_arithmetic_expression(outFile);
fprintf(outFile, ")");
pos++;
}
}
}
}

// <вв> <д> = 'input' <ідентифікатор>
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+1].name);
fprintf(outFile, ");\n");
pos++;
}
}

// <вв> <д> = 'output' <ідентифікатор>
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' <логічний 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 {");

    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 выражение> = <выражение> { '|' <выражение> }
void gen_logical_expression(FILE* outFile)
{
    gen_and_expression(outFile);
    while (TokenTable[pos].type == Or)
    {
        fprintf(outFile, " || ");
        pos++;
        gen_and_expression(outFile);
    }
}

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

// <порядоченное> = <оператор> <порядоченное> | C!C C(<логический выражение> C)C | C(C <логический выражение> C)C
// <оператор> <порядоченное> = <арифметический выражение> <меньше-большее> <арифметический выражение>
// <меньше-большее> = C>C | C<C | C=C | C<>C
void gen_comparison(FILE* outFile)
{
    if (TokenTable[pos].type == Not)
    {
        // ~аргумент: !(<логический выражение>)
        fprintf(outFile, "!(");
        pos++;
        pos++;
        gen_logical_expression(outFile);
        fprintf(outFile, ")");
        pos++;
    }
    else
    {
        if (TokenTable[pos].type == LBracket)
        {
            // ~аргумент: (<логический выражение>)
            fprintf(outFile, "(");
            pos++;
            gen_logical_expression(outFile);
            fprintf(outFile, ")");
            pos++;
        }
        else
        {
            // ~аргумент: <арифметический выражение> <меньше-большее> <арифметический выражение>
            gen_arithmetic_expression(outFile);
            if (TokenTable[pos].type == Greater || TokenTable[pos].type == Less ||
                TokenTable[pos].type == Equality || TokenTable[pos].type == NotEquality)
            {
                switch (TokenTable[pos].type)
                {

```

```

        case Greater: fprintf(outFile, "> "); break;
        case Less: fprintf(outFile, "< "); break;
        case Equality: fprintf(outFile, "== "); break;
        case NotEquality: fprintf(outFile, "!="); break;
    }
    pos++;
    gen_arithmetic_expression(outFile);
}
}

// <сложенный оператор> = 'start' <т.е. по программе> 'stop'
void gen_compound_statement(FILE* outFile)
{
    fprintf(outFile, "{\n");
    pos++;
    gen_program_body(outFile);
    fprintf(outFile, "}\n");
    pos++;
}

```

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
    {
    public:
        using A = typename std::decay_t<F>;

        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
    {
    public:
        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\\WinGW-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 %s. 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);
}
}

```

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[16];
    int line = 1;

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

    // пошук лексем
    while (1)
    {
        switch (state)
        {
            // стан Start - початок виділення чергової лексеми
            // якщо поточний символ маленька літера, то переходимо до стану Letter
            // якщо поточний символ цифра, то переходимо до стану Digit
            // якщо поточний символ пробіл, символ табуляції або переходу на новий рядок, то переходимо до стану Separators
            // якщо поточний символ 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 == ':' || ch == '-') && j < 15)
                {
                    buf[j++] = ch;
                    ch =getc(F);
                }
                buf[j] = '\0';

                TokenType 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 < 15)
    {
        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 if (!strcmp(buf, "PROGRAM"))
{
    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, "START")) temp_type = StartProgram;
else if (!strcmp(buf, "VAR")) temp_type = Variable;
else if (!strcmp(buf, "INT32_t")) temp_type = Type;
else if (!strcmp(buf, "FINISH")) temp_type = EndProgram;
else if (!strcmp(buf, "READ")) temp_type = Input;
else if (!strcmp(buf, "WRITE")) temp_type = Output;

else if (!strcmp(buf, "ADD")) temp_type = Add;
else if (!strcmp(buf, "SUB")) temp_type = Sub;
else if (!strcmp(buf, "MUL")) temp_type = Mul;
else if (!strcmp(buf, "DIV")) temp_type = Div;
else if (!strcmp(buf, "MOD")) temp_type = Mod;

else if (!strcmp(buf, "LE")) temp_type = Less;
else if (!strcmp(buf, "GE")) temp_type = Greate;

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, "DOWNT0")) 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;
else 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] >= 'a' && buf[0] <= 'z') && (strlen(buf) == 8))
{
    bool valid = true;

    for (int i = 1; i < 8; 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(stderr, "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 '-':
    {
        strcpy_s(TempToken.name, "-");
        TempToken.type = Minus;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
    }

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

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

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

    case '<':
    {
        ch = getc(F);
        if (ch == '=')
        {
            ch = getc(F);
            if (ch == '=')
            {
                strcpy_s(TempToken.name, "<==");
                TempToken.type = Assign;
                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;
            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("line number | token | value | token code | type of token\n\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 Greater:
    strcpy_s(type_tokens, "Greater");
    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("-----");
}
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, "-----");
    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 Greater:
                strcpy_s(type_tokens, "Greater");
                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);
}

```

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 hasValidExtension(const char* fileName, const char* extension)
{
    const char* dot = strchr(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 (!hasValidExtension(InputFile, ".m14"))
    {
        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] != '\0' && InputFile[i] != '\n')
    {
        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* ASTTree = 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(ASTTree, 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(ASTTree, 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(ASTTree);

delete[] TokenTable;
delete[] IdTable;

return 0;
}

```

parser.cpp

```

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

// 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 = 0;

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

void goto_statement(FILE* errFile);

```

```

void label_statement(FILE* errFile);
void for_to_do(FILE* errFile);
void for_downto_do(FILE* errFile);
void while_statement(FILE* errFile);
void repeat_until(FILE* errFile);

void logical_expression(FILE* errFile);
void and_expression(FILE* errFile);
void comparison(FILE* errFile);
void compound_statement(FILE* errFile);
std::string TokenTypeToString(TokenType type);

unsigned int Identification(Id IdTable[], Token TokenTable[], unsigned int tokenCount, FILE* errFile);

void Parser(FILE* errFile)
{
    program(errFile);
    fprintf(errFile, "\nNo errors found.\n");
}

void match(TokenType expectedType, FILE* errFile)
{
    if (TokenTable[pos].type == expectedType)
        pos++;
    else
    {
        fprintf(errFile, "\nSyntax error in line %d : another type of lexeme was expected.\n", TokenTable[pos].line);
        fprintf(errFile, "\nSyntax error: type %s\n", TokenTypeToString(TokenTable[pos].type).c_str());
        fprintf(errFile, "Expected Type: %s ", TokenTypeToString(expectedType).c_str());
        exit(10);
    }
}

void program(FILE* errFile)
{
    match(Mainprogram, errFile);
    match(ProgramName, errFile);
    match(Variable, errFile);
    variable_declaration(errFile);
    match(Semicolon, errFile);
    match(StartProgram, errFile);
    program_body(errFile);
    match(EndProgram, errFile);
}

void variable_declaration(FILE* errFile)
{
    if (TokenTable[pos].type == Type)
    {
        pos++;
        variable_list(errFile);
    }
}

void variable_list(FILE* errFile)
{
    match(Identifier, errFile);
    while (TokenTable[pos].type == Comma)
    {
        pos++;
        match(Identifier, errFile);
    }
}

void program_body(FILE* errFile)
{
    do
    {
        statement(errFile);
    } while (TokenTable[pos].type != EndProgram);
}

void statement(FILE* errFile)
{
    switch (TokenTable[pos].type)
    {
        case Input: input(errFile); break;
        case Output: output(errFile); break;
        case IF: conditional(errFile); break;
        case Label: label_statement(errFile); break;
        case StartProgram: compound_statement(errFile); break;
        case Goto: goto_statement(errFile); 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)
            {
                for_to_do(errFile);
            }
            else if (TokenTable[temp_pos].type == DownTo)
            {
                for_downto_do(errFile);
            }
            else
            {
                printf("Error: Expected 'To' or 'DownTo' after 'For'\n");
            }
            break;
        }
        case While: while_statement(errFile); break;
        case Exit: pos += 2; break;
        case Continue: pos += 2; break;
        case Repeat: repeat_until(errFile); break;
        default: assignment(errFile); break;
    }
}

void assignment(FILE* errFile)
{
    match(Identifier, errFile);
    match(Assign, errFile);
    arithmetic_expression(errFile);
    match(Semicolon, errFile);
}

void arithmetic_expression(FILE* errFile)
{
    term(errFile);
}

```

```

while (TokenTable[pos].type == Add || TokenTable[pos].type == Sub)
{
    pos++;
    term(errFile);
}

void term(FILE* errFile)
{
    factor(errFile);
    while (TokenTable[pos].type == Mul || TokenTable[pos].type == Div || TokenTable[pos].type == Mod)
    {
        pos++;
        factor(errFile);
    }
}

void factor(FILE* errFile)
{
    if (TokenTable[pos].type == Identifier)
    {
        match(Identifier, errFile);
    }
    else
    {
        if (TokenTable[pos].type == Number)
        {
            match(Number, errFile);
        }
        else
        {
            if (TokenTable[pos].type == LBraket)
            {
                match(LBraket, errFile);
                arithmetic_expression(errFile);
                match(RBraket, errFile);
            }
            else
            {
                printf("\nSyntax error in line %d : A multiplier was expected.\n", TokenTable[pos].line);
                exit(11);
            }
        }
    }
}

void input(FILE* errFile)
{
    match(Input, errFile);
    match(Identifier, errFile);
    match(Semicolon, errFile);
}

void output(FILE* errFile)
{
    match(Output, errFile);
    if (TokenTable[pos].type == Minus)
    {
        pos++;
        if (TokenTable[pos].type == Number)
        {
            match(Number, errFile);
        }
    }
    else
    {
        arithmetic_expression(errFile);
    }
    match(Semicolon, errFile);
}

void conditional(FILE* errFile)
{
    match(If, errFile);
    logical_expression(errFile);
    statement(errFile);
    if (TokenTable[pos].type == Else)
    {
        pos++;
        statement(errFile);
    }
}

void goto_statement(FILE* errFile)
{
    match(Goto, errFile);
    if (TokenTable[pos].type == Identifier)
    {
        pos++;
        match(Semicolon, errFile);
    }
    else
    {
        printf("Error: Expected a label after 'goto' at line %d.\n", TokenTable[pos].line);
        exit(1);
    }
}

void label_statement(FILE* errFile)
{
    match(Label, errFile);
}

void for_to_do(FILE* errFile)
{
    match(For, errFile);
    match(Identifier, errFile);
    match(Assign, errFile);
    arithmetic_expression(errFile);
    match(To, errFile);
    arithmetic_expression(errFile);
    match(Do, errFile);
    statement(errFile);
}

void for_downto_do(FILE* errFile)
{
    match(For, errFile);
    match(Identifier, errFile);
    match(Assign, errFile);
    arithmetic_expression(errFile);
    match(DownTo, errFile);
    arithmetic_expression(errFile);

```



```

    match(Do, errFile);
    statement(errFile);
}

void while_statement(FILE* errFile)
{
    match(While, errFile);
    logical_expression(errFile);

    while (1)
    {
        if (TokenTable[pos].type == End)
        {
            pos++;
            match(While, errFile);
            break;
        }
        else
        {
            statement(errFile);
            if (TokenTable[pos].type == Semicolon)
            {
                pos++;
            }
        }
    }
}

void repeat_until(FILE* errFile)
{
    match(Repeat, errFile);
    statement(errFile);
    match(Until, errFile);
    logical_expression(errFile);
}

void logical_expression(FILE* errFile)
{
    and_expression(errFile);
    while (TokenTable[pos].type == Or)
    {
        pos++;
        and_expression(errFile);
    }
}

void and_expression(FILE* errFile)
{
    comparison(errFile);
    while (TokenTable[pos].type == And)
    {
        pos++;
        comparison(errFile);
    }
}

void comparison(FILE* errFile)
{
    if (TokenTable[pos].type == Not)
    {
        pos++;
        match(LBraket, errFile);
        logical_expression(errFile);
        match(RBraket, errFile);
    }
    else
    {
        if (TokenTable[pos].type == LBraket)
        {
            pos++;
            logical_expression(errFile);
            match(RBraket, errFile);
        }
        else
        {
            arithmetic_expression(errFile);
            if (TokenTable[pos].type == Greater || TokenTable[pos].type == Less ||
                TokenTable[pos].type == Equality || TokenTable[pos].type == NotEquality)
            {
                pos++;
                arithmetic_expression(errFile);
            }
            else
            {
                fprintf(errFile, "\nSyntax error in line %d : A comparison operation is expected.\n", TokenTable[pos].line);
                exit(12);
            }
        }
    }
}

void compound_statement(FILE* errFile)
{
    match(StartProgram, errFile);
    program_body(errFile);
    match(EndProgram, errFile);
}

unsigned int Identification(Id IdTable[], Token TokenTable[], unsigned int tokenCount, FILE* errFile)
{
    unsigned int idCount = 0;
    unsigned int i = 0;

    while (TokenTable[i++].type != Variable);

    if (TokenTable[i++].type == Type)
    {
        while (TokenTable[i].type != Semicolon)
        {
            if (TokenTable[i].type == Identifier)
            {
                int yes = 0;
                for (unsigned int j = 0; j < idCount; j++)
                {
                    if (!strcmp(TokenTable[i].name, IdTable[j].name))
                    {
                        yes = 1;
                        break;
                    }
                }
                if (yes == 1)
                {
                    printf("\nidentifier \"%s\" is already declared !\n", TokenTable[i].name);
                }
            }
        }
    }
}

```

```

        return idCount;
    }

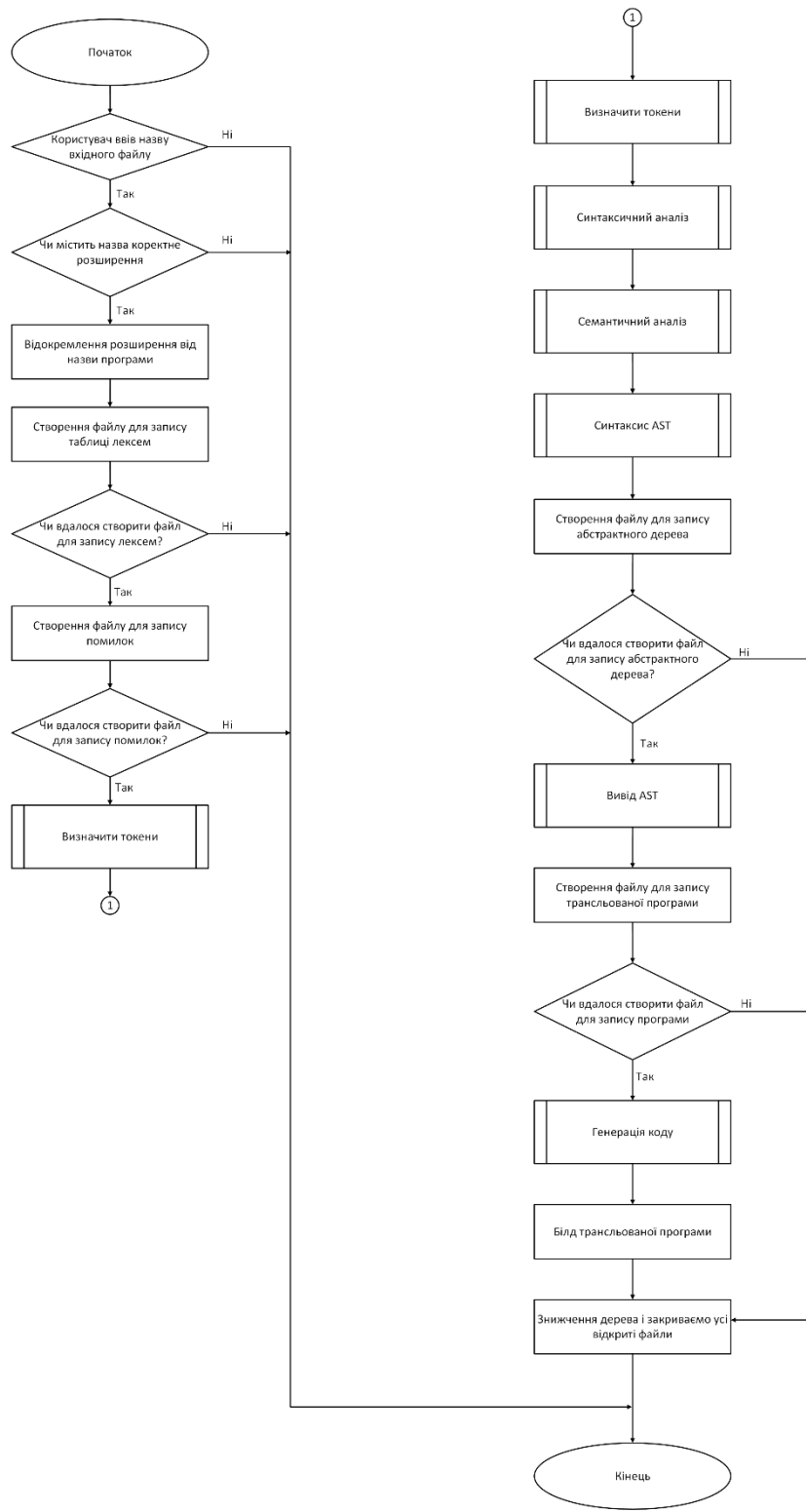
    if (idCount < MAX_IDENTIFIER)
    {
        strcpy_s(IdTable[idCount++].name, TokenTable[i++].name);
    }
    else
    {
        printf("\nToo many identifiers !\n");
        return idCount;
    }
}
else
    i++;
}

for (; i < tokenCount; i++)
{
    if (TokenTable[i].type == Identifier && TokenTable[i + 1].type != Colon)
    {
        int yes = 0;
        for (unsigned int j = 0; j < idCount; j++)
        {
            if (!strcmp(TokenTable[i].name, IdTable[j].name))
            {
                yes = 1;
                break;
            }
        }
        if (yes == 0)
        {
            if (idCount < MAX_IDENTIFIER)
            {
                strcpy_s(IdTable[idCount++].name, TokenTable[i].name);
            }
            else
            {
                printf("\nToo many identifiers!\n");
                return idCount;
            }
        }
    }
}

return idCount;
}

std::string TokenTypeToString(TokenType type)
{
    switch (type)
    {
        case Mainprogram: return "Mainprogram";
        case StartProgram: return "StartProgram";
        case Variable: return "Variable";
        case Type: return "Type";
        case EndProgram: return "EndProgram";
        case Input: return "Input";
        case Output: return "Output";
        case If: return "If";
        case Else: return "Else";
        case Goto: return "Goto";
        case Label: return "Label";
        case For: return "For";
        case To: return "To";
        case DownTo: return "DownTo";
        case Do: return "Do";
        case While: return "While";
        case Exit: return "Exit";
        case Continue: return "Continue";
        case End: return "End";
        case Repeat: return "Repeat";
        case Until: return "Until";
        case Identifier: return "Identifier";
        case Number: return "Number";
        case Assign: return "Assign";
        case Add: return "Add";
        case Sub: return "Sub";
        case Mul: return "Mul";
        case Div: return "Div";
        case Mod: return "Mod";
        case Equality: return "Equality";
        case NotEquality: return "NotEquality";
        case Greater: return "Greater";
        case Less: return "Less";
        case Not: return "Not";
        case And: return "And";
        case Or: return "Or";
        case LBracket: return "LBracket";
        case RBracket: return "RBracket";
        case Semicolon: return "Semicolon";
        case Colon: return "Colon";
        case Comma: return "Comma";
        case Unknown: return "Unknown";
        default: return "InvalidType";
    }
}

```



Міністерство освіти і науки України					Курсовий проєкт			
					Розробка системних програмних модулів так компонентів систем програмування			
					Алгоритм Транслятора М14			
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив	Бодров А.Ю.							
Керівник	Козак Н.Б.							
Консульт.								
Н.контр.					Додаток Д			
Зав.каф.								
Рецензент					НУ «ЛП», Каф. ЕОМ, гр. КІ-309			
					Літера	Маса	Масштаб	
					у			
					Аркуш		Аркушів 1	