

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 №16

Виконала: ст. гр. КІ-308
Паньків А.М.
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.

Анотація	2
Завдання до курсового проекту	4
Деталізація завдання на проектування:.....	5
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	18
3.3.2. Опис програми реалізації лексичного аналізатора	20
3.4. Розробка синтаксичного та семантичного аналізатора	22
3.4.1. Опис програми реалізації синтаксичного та семантичного аналізатора	22
3.4.2. Розробка граф-схеми алгоритму	26
3.5. Розробка генератора коду	27
3.5.1. Розробка граф-схеми алгоритму	28
3.5.2. Опис програми реалізації генератора коду.....	28
4. Опис програми	30
4.1. Опис інтерфейсу та інструкція користувачеві.....	33
5. Відлагодження та тестування програми	35
5.1. Виявлення лексичних та синтаксичних помилок	35
5.2. Виявлення семантичних помилок.....	36
5.3. Загальна перевірка коректності роботи транслятора.....	36
5.4. Тестова програма №1.....	37
5.5. Тестова програма №2.....	38
5.6. Тестова програма №3.....	39
Висновки.....	41
Список використаної літератури	42
Додатки.....	43
Додаток А. (Код на мові С)	43
Додаток Б. (Тестові програми).....	46
Додаток В. (Таблиці лексем для тестових прикладів)	49
Додаток Г (Абстрактне синтаксичне дерево для тестових прикладів).....	64
Додаток Д (Документований текст програмних модулів (лістинги)	70
Додаток Е.(Алгоритм транслятора П16).....	122

Завдання до курсового проекту

Варіант 16

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 .p16

Опис вхідної мови програмування:

- Тип даних: longint
- Блок тіла програми: startprogram startblok variable...; endblok
- 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: Low
- Регістр ідентифікаторів: Low12
- Операції арифметичні: add, sub, *, /, %
- Операції порівняння: ==, <>, >>, <<
- Операції логічні: !, &, |
- Коментар: @@...
- Ідентифікатори змінних, числові константи
- Оператор присвоєння: ==>

Деталізація завдання на проєктування:

1. В кожному завданні передбачається блок оголошення змінних; змінні зберігають значення цілих чисел і, в залежності від варіанту, можуть бути 16/32 розрядними. За потребою можна реалізувати логічний тип даних.
2. Необхідно реалізувати арифметичні операції – додавання, віднімання, множення, ділення, залишок від ділення; операції порівняння – перевірка на рівність і нерівність, більше і менше; логічні операції – заперечення, “логічне І” і “логічне АБО”. Пріоритет операцій наступний – круглі дужки (), логічне заперечення, мультиплікативні (множення, ділення, залишок від ділення), адитивні (додавання, віднімання), відношення (більше, менше), перевірка на рівність і нерівність, логічне І, логічне АБО.
3. За допомогою оператора вводу можна зчитати з клавіатури значення змінної; за допомогою оператора виводу можна вивести на екран значення змінної, виразу чи цілої константи.
4. В кожному завданні обов’язковим є оператор присвоєння за допомогою якого можна реалізувати обчислення виразів з використанням заданих операцій і операції круглі дужки (); у якості операндів можуть бути цілі константи, змінні, а також інші вирази.
5. В кожному завданні обов’язковим є оператор типу “блок” (складений оператор), його вигляд має бути таким, як і блок тіла програми
6. Необхідно реалізувати задані варіантом оператори, синтаксис операторів наведено у таблиці 1.1. Синтаксис вхідної мови має забезпечити реалізацію обчислень лінійних алгоритмів, алгоритмів з розгалуженням і циклічних алгоритмів. Опис формальної мови студент погоджує з викладачем.
8. Для перевірки роботи розробленого транслятора, необхідно написати три тестові програми на вхідній мові програмування.

Вступ

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

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

У відмінну від компіляторів, інтерпретатор не створює нову програму; він лише виконує – інтерпретує – 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).

```
program = "startprogram", "startblok", {"variable", variable_declaration,
";"}, {"statement, ";"}, "endblok";
```

```
variable_declaration = "longint", variable_list;
```

```
variable_list = identifier, {"", identifier};
```

```
identifier = low, low, low, low, low, low, low, low, low, low, low, low;
```

```
up_low = up | low | digit;
```

```
up = "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";
```

```
low = 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" ;
```

```
statement = input_statement | output_statement | assign_statement |
if_else_statement | goto_statement | label_point | for_statement |
while_statement | repeat_until_statement | compound_statement;
```

```
input_statement = "read", identifier;
```

```
output_statement = "write", arithmetic_expression;
```

```
arithmetic_expression = low_priority_expression {low_priority_operator,
low_priority_expression};
```

```
low_priority_operator = "add" | "sub";
```

```
low_priority_expression = middle_priority_expression {middle_priority_operator,
```

```

middle_priority_expression};
middle_priority_operator = "*" | "/" | "%";
middle_priority_expression = identifier | number | "(", arithmetic_expression,
");";
number = ["-"], (nonzero_digit, {digit} | "0") ;
nonzero_digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
assign_statement = arithmetic_expression, "==>", identifier;
if_else_statement = "if", "(", logical_expression, ")", statement, [";",
"else", statement];
logical_expression = and_expression {or_operator, and_expression};
or_operator = "|";
and_expression = comparison {and_operator, and_expression};
and_operator = "&";
comparison = comparison_expression | [not_operator] "(", logical_expression,
");";
not_operator = "!";
comparison_expression = arithmetic_expression comparison_operator
arithmetic_expression;
comparison_operator = "==" | "<>" | "<<" | ">>";
goto_statement = "goto", identifier;
label_point = identifier, ":";
for_to_statement = "for", assign_statement, "to" | "downto",
arithmetic_expression, "do", statement;
statement_in_while = statement | ("continue", "while") | ("exit", "while");
while_statement = "while", logical_expression, {statement_in_while}, "endblok",
"while";
repeat_until_statement = "repeat", {statement, ";"}, "until", "(",
logical_expression, ")";
compoundStatement = "startblok", {statement, ";"}, "endblok";

```

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
startprogram	Початок програми
startblok	Початок тексту програми
variable	Початок блоку опису змінних
endblok	Кінець розділу операторів
read	Оператор вводу змінних
write	Оператор виводу (змінних або рядкових констант)
==>	Оператор присвоєння
if	Оператор умови
else	Оператор умови
goto	Оператор переходу
label	Мітка переходу
for	Оператор циклу
to	Інкремент циклу
downto	Декремент циклу
do	Початок тіла циклу
while	Оператор циклу
continue	Оператор циклу
exit	Оператор циклу
repeat	Початок тіла циклу
until	Оператор циклу
add	Оператор додавання
sub	Оператор віднімання
*	Оператор множення

/	Оператор ділення
%	Оператор знаходження залишку від ділення
==	Оператор перевірки на рівність
<>	Оператор перевірки на нерівність
<<	Оператор перевірки чи менше
>>	Оператор перевірки чи більше
!	Оператор логічного заперечення
&	Оператор кон'юнкції
	Оператор диз'юнкції
longint	32-ох розрядні знакові цілі
@ @ ...	Коментар
,	Розділювач
;	Ознака кінця оператора
(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 наступне:

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

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

- 2) Таблиця лексичних класів

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

    If,
    Else,

    Goto,
    Label,

    For,
    To,
    DownTo,
    Do,

    While,
    Exit,
    Continue,
    End,

    Repeat,
    Until,

    Identifier,
    Number,
    Assign,
    Add,
    Sub,
    Mul,
    Div,
    Mod,
    Equality,
    NotEquality,
    Greate,
    Less,
    Not,
    And,
    Or,
    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	startprogram
Start	startblok
Vars	variable
End	endblok
VarType	longint
Read	read
Write	write
Assignment	==>
If	if
Else	else
Goto	goto
Colon	:
Label	
For	for
To	to
DownTo	downto
Do	do
While	while
Continue	continue
Exit	exit
Repeat	repeat
Until	until

Addition	add
Subtraction	sub
Multiplication	*
Division	/
Mod	%
Equal	==
NotEqual	<>
Less	<<
Greate	>>
Not	!
And	&
Or	
Identifier	
Number	
Unknown	
Comma	,
Semicolon	;
LBraket	(
RBraket)
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.

За варіантом розділено лексеми на типи або лексичні класи:

- Ключові слова (0-**startprogram**, 1-**startblok**, 2-**variable**, 3-**endblok**, 4-**read**, 5-**write**, 6-**longint**, 7-**if**, 8-**else**, 9-**for**, 10-**to**, 11-**downto**, 12-**goto**, 13-**while**, 14-**continue**, 15-**exit**, 16-**repeat**, 17-**until**)
- Ідентифікатори (18- максимум 12 маленьких літер)
- Числові константи (19-ціле число без знаку)
- Оператор присвоєння (20- **==>**)
- Знаки операції (21-**add**, 22-**sub**, 23-*****, 24-**/**, 25-**>>**, 26-**<<**, 27-**==**, 28-**<>**, 29-**!**, 30-**&**, 31-**|**, 32- **%**)
- Розділювачі(33-**;**, 34-**,**)
- Дужки (35-**(**, 36-**)**)
- Невідома лексема (37-символи і ланцюжки символів, які не підпадають під вище описані правила)

3.3.1. Розробка блок-схеми алгоритму

Лексичний аналізатор виконує обробку тексту з вхідного файлу та визначає типи лексем за наступними етапами. Процес роботи можна описати наступним чином:

- **Початок роботи:**
алгоритм починається зі зчитування чергового слова або символу з файлу (Str).
- **Перевірка на кінець файлу:**
якщо зчитане значення Str дорівнює EndOfFile, тип лексеми визначається як EndOfFile, після чого завершується обробка.

- **Ключове слово або символ:**
якщо зчитане значення є ключовим словом або символом, виконується обробка лексеми, і тип визначається як Ключове слово або символ.
- **Розпізнавання слова (ідентифікатора):**
якщо значення є словом, проводиться обробка ідентифікатора, а тип встановлюється як Identifier.
- **Розпізнавання числа:**
якщо значення є числом, тип лексеми визначається як Number.
- **Невідомий тип:**
якщо жодна з вищезазначених умов не виконана, тип лексеми визначається як Unknown.
- **Завершення роботи:**
після визначення типу лексеми алгоритм повертається до зчитування наступного слова або символу, доки не буде досягнуто стану EndOfFile.

Алгоритм працює на основі автомату з наступними станами:

- **Start** – початковий стан, зчитування символу.
- **EndOfFile** – завершення обробки файлу.
- **Ключове слово або символ** – обробка зарезервованих слів або символів.
- **Identifier** – обробка лексем, що є ідентифікаторами.
- **Number** – обробка числових значень.
- **Unknown** – визначення невідомих лексем.

Ця модель дозволяє покроково аналізувати вхідний файл, виділяючи ключові слова, ідентифікатори, числа та інші елементи, забезпечуючи коректну обробку кожної лексеми.

Алгоритм роботи лексичного аналізатора можна зобразити у вигляді блок-схеми.

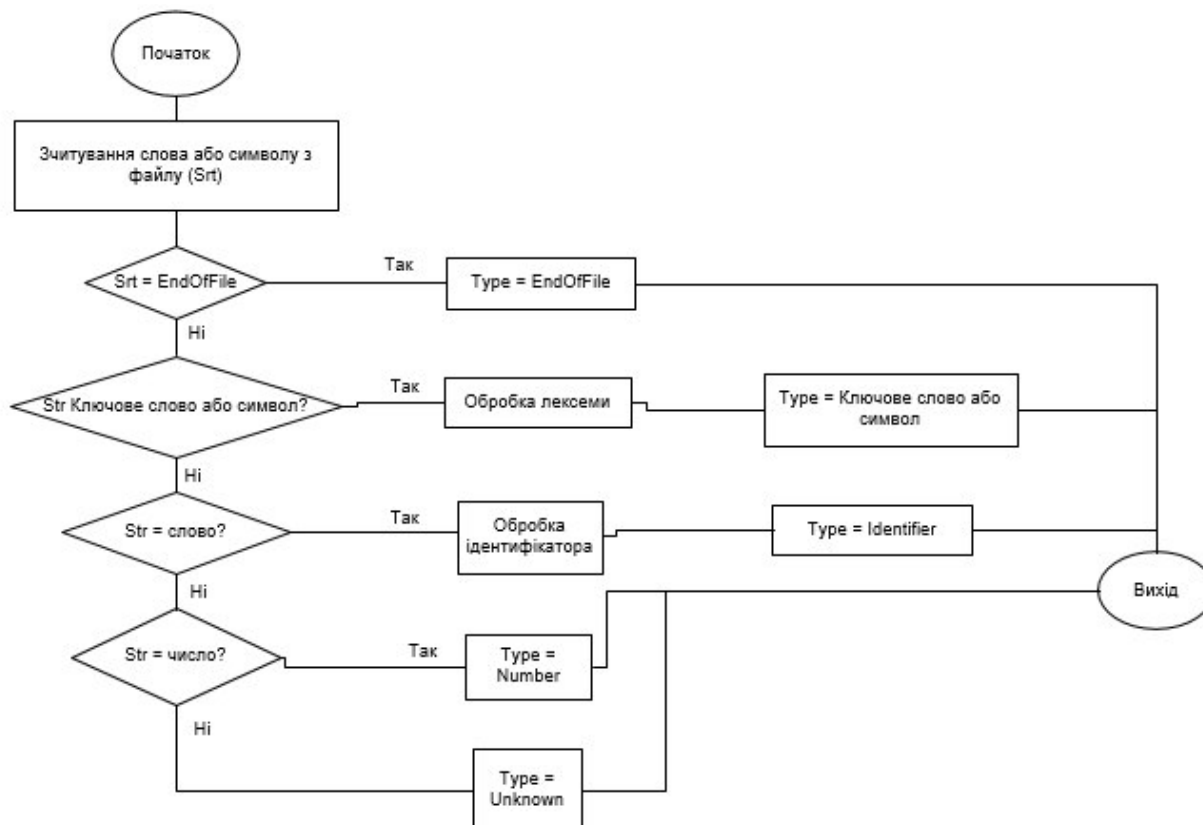


Рис. 3.1 Блок-схема роботи лексичного аналізатора

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 зарезервованими словами якщо є співпадіння то присвоює лексемі відповідний тип або значення, якщо це числова константа.

При виділенні лексеми вона розпізнається та записується у таблицю за допомогою відповідного типу лексеми, що є унікальним для кожної лексеми із

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

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

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

Створимо структуру даних для зберігання стану аналізатора:

```
// 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         //
```

Напишемо функцію, яка реалізує лексичний аналіз:

```
// 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);
І функції, які друкують список лексем:

// 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);
```

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.

Структура синтаксичного аналізатора буде такою:

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

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

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

Синтаксичний аналізатор працює за методом рекурсивного спуску, а отже функція parser() викликає функцію program(), яка в свою чергу викликає інші функції.

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

```
// функція записує оголошені ідентифікатори в таблицю
ідентифікаторів IdTable
// повертає кількість ідентифікаторів
// перевіряє чи усі використані ідентифікатори оголошені
unsigned int IdIdentification(Id IdTable[], Token
TokenTable[], unsigned int tokenCount, FILE* errFile);
```

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.

Визначимо назви процедур, що відповідають нетерміналам граматики таким чином :

```
// набір функцій для рекурсивного спуску
// на кожне правило - окрема функція
void program(FILE* errFile); // topRule = "startprogram", "startblok",
varsBlok, ";", operators, "endblok"

void variable_declaration(FILE* errFile); // varsBlok = "variable",
"longint", identifier, [{ commaAndIdentifier }]

void variable_list(FILE* errFile); //identifier = low_letter, {
low_letter | number } {11}; commaAndIdentifier = ",", identifier;

void program_body(FILE* errFile); // codeBlok = "startblok", operators,
"endblok"

void statement(FILE* errFile); // operators = write | read | assignment |
ifStatement | goto_statement | labelRule | forToOrDownToDoRule | while |
repeatUntil

void assignment(FILE* errFile); // assignment = identifier, "==>",
equation

void arithmetic_expression(FILE* errFile); // equation = signedNumber |
identifier | notRule [{ operationAndIdentOrNumber | equation }]

void term(FILE* errFile); // operationAndIdentOrNumber = mult |
arithmetic | logic | compare signedNumber | identifier | equation

void factor(FILE* errFile); // signedNumber = [ sign ] digit [{digit}]

void input(FILE* errFile); // read = "read", "(", identifier, ")"

void output(FILE* errFile); // write = "write", "(", equation |
stringRule, ")"

void conditional(FILE* errFile); // ifStatement = "if", "(", equation,
")", codeBlok, ["Else", codeBlok]

void goto_statement(FILE* errFile); // goto_statement = "goto", ident
```

```

void label_statement(FILE* errFile); // labelRule = identifier, ":"

void for_to_do(FILE* errFile); // forToOrDownToDoRule = "for",
cycle_counter, "==>", equation, "to", cycle_counter_last_value, "do",
codeBlok

void for_downto_do(FILE* errFile); // forToOrDownToDoRule = "for",
cycle_counter, "==>", equation, "downto", cycle_counter_last_value, "do",
codeBlok

void while_statement(FILE* errFile); //whileRule = "while", condition,
[statements], [ "continue while" ], [statements], [ "exit while" ],
[statements], "end while";
void repeat_until(FILE* errFile); // repeatUntil = "repeat", operators,
"until", "(", equation, ")"

void logical_expression(FILE* errFile); // logic = "&" | "|"; notRule =
notOperation, signedNumber | identifier | equation

void and_expression(FILE* errFile); // logic ("&")

void comparison(FILE* errFile); // compare = "==" | "<>" | "<<" | ">>"

void compound_statement(FILE* errFile); // codeBlok = "startblok",
operators, "endblok"

```

3.4.2. Розробка граф-схеми алгоритму

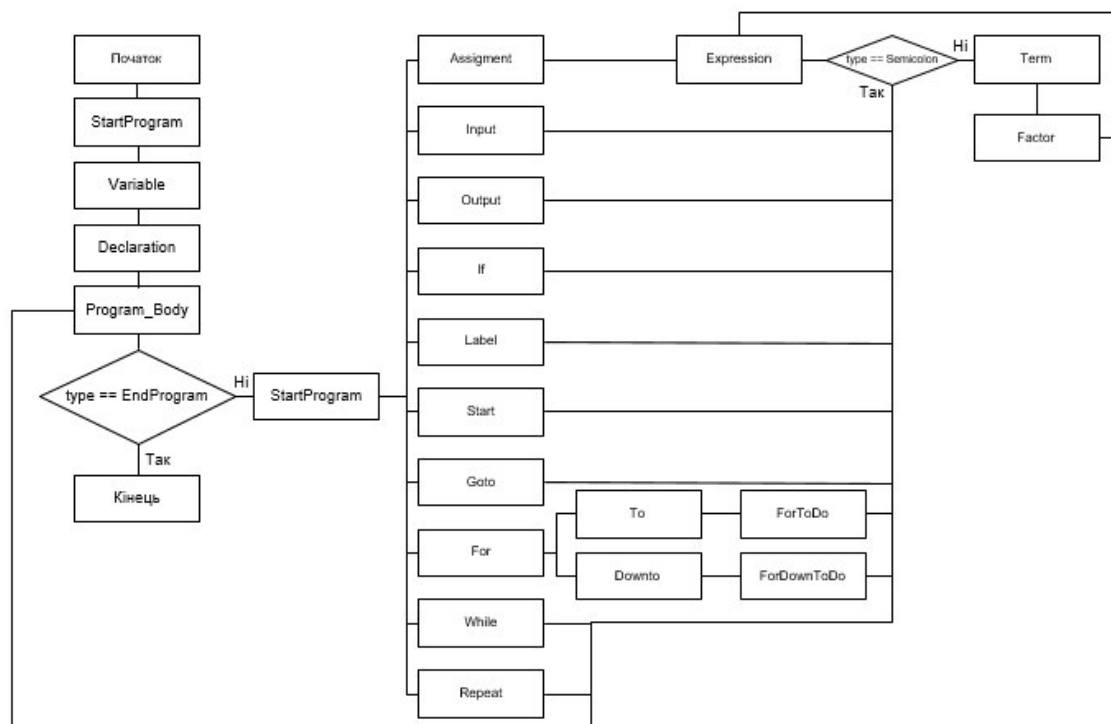


Рис. 3.2 Блок-схема роботи синтаксичного аналізатора

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 граф-схеми алгоритму



Рис. 3.3 Блок схема генератора коду

3.5.2. Опис програми реалізації генератора коду

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

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

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

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

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

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

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

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

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

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

4. Опис програми

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

```
// структура генератора коду
// 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;

void generateCCode(FILE* outFile)
{
    fprintf(outFile, "#include <stdio.h>\n");
    fprintf(outFile, "#include <stdlib.h>\n");
    fprintf(outFile, "#include <stdint.h>\n\n");
    fprintf(outFile, "int main() \n{\n");

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

    pos++;
    pos++;
    gen_program_body(outFile);
    fprintf(outFile, "    system(\"pause\");\n ");
    fprintf(outFile, "    return 0;\n");
    fprintf(outFile, "}\n");
}
```

enum TypeOfTokens

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

    If,
    Else,

    Goto,
    Label,

    For,
```

```

    To,
    DownTo,
    Do,

    While,
    Exit,
    Continue,
    End,

    Repeat,
    Until,

    Identifier,
    Number,
    Assign,
    Add,
    Sub,
    Mul,
    Div,
    Mod,
    Equality,
    NotEquality,
    Greate,
    Less,
    Not,
    And,
    Or,
    LBracket,
    RBracket,
    Semicolon,
    Colon,
    Comma,
    Unknown
};

// 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 p16. 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_p16.exe <ім'я програми>.p16"

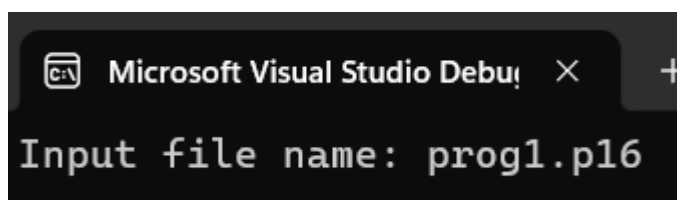
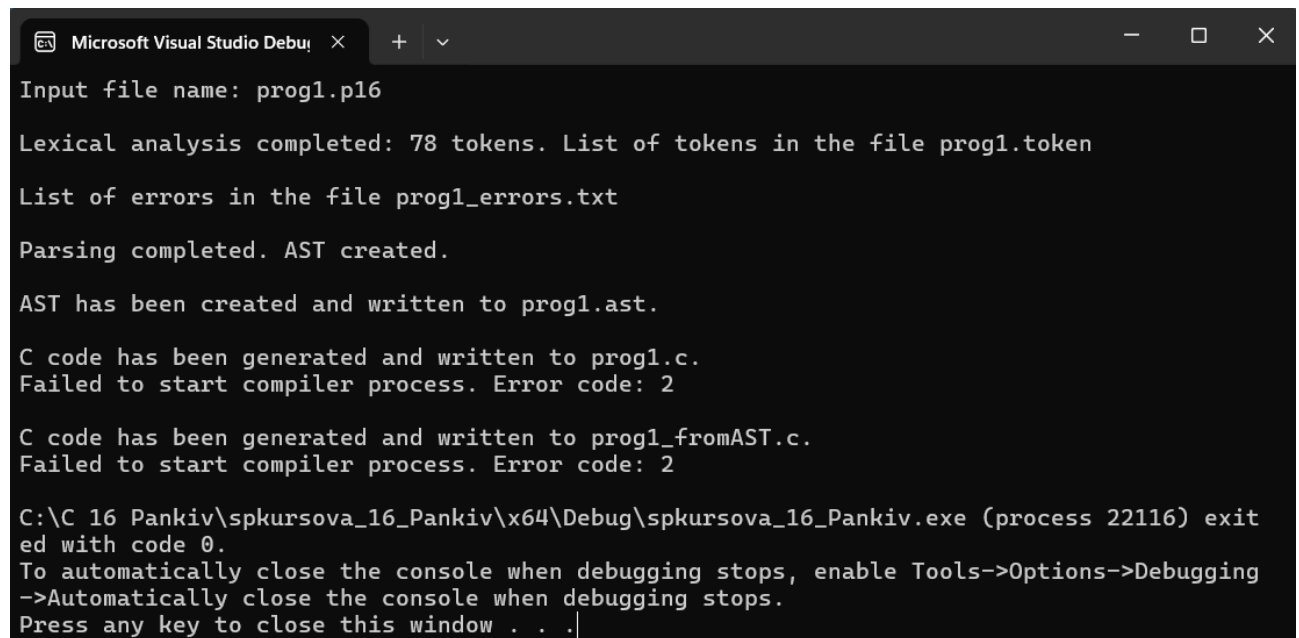


Рис. 4.1 Інтерфейс розробленого транслятора

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.



```
Microsoft Visual Studio Debug Console
Input file name: prog1.p16
Lexical analysis completed: 78 tokens. List of tokens in the file prog1.token
List of errors in the file prog1_errors.txt
Parsing completed. AST created.
AST has been created and written to prog1.ast.
C code has been generated and written to prog1.c.
Failed to start compiler process. Error code: 2
C code has been generated and written to prog1_fromAST.c.
Failed to start compiler process. Error code: 2
C:\C 16 Pankiv\spkursova_16_Pankiv\x64\Debug\spkursova_16_Pankiv.exe (process 22116) exit
ed with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging
->Automatically close the console when debugging stops.
Press any key to close this window . . .|
```

Рис. 4.2. Результати роботи розробленого транслятора.

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
startprogram
startblok
variable longint aaaaaaa aaaaa,bbbbbbbbbbbb,xxxxxxxxxxxxx,yyyyyyyyyyyyy;
read aaaaaaaaaaaa;
read bbbbbbbbbbbb
write aaaaaaaaaaaa add bbbbbbbbbbbb;
write aaaaaaaaaaaa sub bbbbbbbbbbbb;
write aaaaaaaaaaaa * bbbbbbbbbbbb;
write aaaaaaaaaaaa / bbbbbbbbbbbb;
write aaaaaaaaaaaa % bbbbbbbbbbbb;

xxxxxxxxxxxx==>(aaaaaaaaaaaa sub bbbbbbbbbbbb) * 10 add (aaaaaaaaaaaa add bbbbbbbbbbbb) /
10;
yyyyyyyyyyyy==>xxxxxxxxxxxx add (xxxxxxxxxxxx % 10);
write xxxxxxxxxxxx;
```

```
write yyyyyyyyyyyyyy;  
endblok
```

Текст файлу з повідомленнями про помилки

Lexical Error: line 4, lexem aaaaaaa is Unknown
Lexical Error: line 4, lexem aaaaa is Unknown

Syntax error in line 4 : 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 longint, тобто знаковому цілому числу з відповідним діапазоном значень і перевірку на коректність використання змінних longint 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.

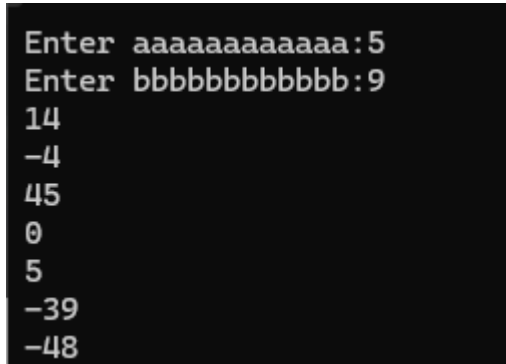
Текст коректної програми

```
@ @Prog1  
startprogram  
startblok  
variable longint aaaaaaaaaa,bbbbbbbbbbbbbb,xxxxxxxxxxxxx,yyyyyyyyyyyyy;  
read aaaaaaaaaa;  
read bbbbbbbbbbbbbbb;  
write aaaaaaaaaa add bbbbbbbbbbbbbbb;  
write aaaaaaaaaa sub bbbbbbbbbbbbbbb;  
write aaaaaaaaaa * bbbbbbbbbbbbbbb;  
write aaaaaaaaaa / bbbbbbbbbbbbbbb;  
write aaaaaaaaaa % bbbbbbbbbbbbbbb;  
  
xxxxxxxxxxxxx==>(aaaaaaaaaaaa sub bbbbbbbbbbbbbbb) * 10 add (aaaaaaaaaaaa add bbbbbbbbbbbbbbb) /  
10;  
yyyyyyyyyyyyy==>xxxxxxxxxxxxx add (xxxxxxxxxxxxx % 10);  
write xxxxxxxxxxxxxx;  
write yyyyyyyyyyyyyy;  
endblok
```

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

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

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



```
Enter aaaaaaaaaaaaaa:5
Enter bbbbbbbbbbbbbbb: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
startprogram
startblok
variable longint aaaaaaaaaaaaaa,bbbbbbbbbbbbbb,xxxxxxxxxxxxx,yyyyyyyyyyyyy;
read aaaaaaaaaaaaaa;
read bbbbbbbbbbbbbbb;
write aaaaaaaaaaaaaa add bbbbbbbbbbbbbbb;
write aaaaaaaaaaaaaa sub bbbbbbbbbbbbbbb;
write aaaaaaaaaaaaaa * bbbbbbbbbbbbbbb;
write aaaaaaaaaaaaaa / bbbbbbbbbbbbbbb;
write aaaaaaaaaaaaaa % bbbbbbbbbbbbbbb;

xxxxxxxxxxxxx==>(aaaaaaaaaaaaa sub bbbbbbbbbbbbbbb) * 10 add (aaaaaaaaaaaaa add bbbbbbbbbbbbbbb) /
10;
yyyyyyyyyyyyyy==>xxxxxxxxxxxxx add (xxxxxxxxxxxxx % 10);
write xxxxxxxxxxxxxx;
write yyyyyyyyyyyyyy;
endblok
```

Результат виконання

```

Enter aaaaaaaaaaaa:5
Enter bbbbbbbbbbbb:9
14
-4
45
0
5
-39
-48

```

Рис. 5.2 Результат виконання тестової програми №1

5.5. Тестова програма №2

Текст програми

```

@@Prog2
startprogram
startblok
variable longint aaaaaaaaaaaa,bbbbbbbbbbbbbb,cccccccccccc;
read aaaaaaaaaaaa;
read bbbbbbbbbbbb;
read ccccccccccc;
if(aaaaaaaaaaaa >> bbbbbbbbbbbb)
startblok
    if(aaaaaaaaaaaa >> ccccccccccc)
    startblok
        goto Abigger;
    endblok
    else
    startblok
        write ccccccccccc;
        goto Outofif;
    Abigger:
        write aaaaaaaaaaaa;
        goto Outofif;
    endblok
endblok
if(bbbbbbbbbbbb << ccccccccccc)
startblok
    write ccccccccccc;
endblok
else
startblok
    write bbbbbbbbbbbb;
endblok
Outofif:

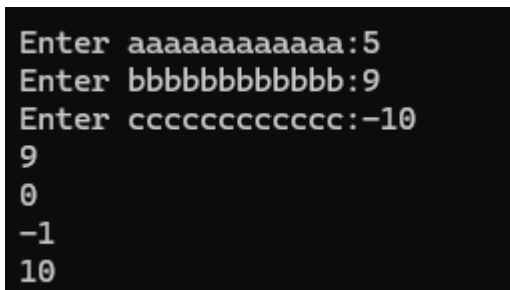
```

```

if((aaaaaaaaaaaa == bbbbbbbbbbbb) & (aaaaaaaaaaaa == cccccccccc) & (bbbbbbbbbbbb ==
ccccccccc))
startblok
    write 1;
endblok
else
startblok
    write 0;
endblok
if((aaaaaaaaaaaa << 0) | (bbbbbbbbbbbb << 0) | (ccccccccc << 0))
startblok
    write -1;
endblok
else
startblok
    write 0;
endblok
if(!(aaaaaaaaaaaa << (bbbbbbbbbbbb add cccccccccc)))
startblok
    write(10);
endblok
else
startblok
    write(0);
endblok
endblok

```

Результат виконання



```

Enter aaaaaaaaaa:5
Enter bbbbbbbbbbbb:9
Enter cccccccccc:-10
9
0
-1
10

```

Рис. 5.3 Результат виконання тестової програми №2

5.6. Тестова програма №3

Текст програми

```

@@Prog3
startprogram
startblok
variable longint
aaaaaaaaaaaa,aaaaaaaa2,bbbbbbbbbbbb,xxxxxxxxxxxx,ccccccccc1,ccccccccc2;
read aaaaaaaaaa;
read bbbbbbbbbbbb;
for aaaaaaaaaa2==>aaaaaaaa to bbbbbbbbbbbb do

```

```

write aaaaaaaaaa2 * aaaaaaaaaa2;

for aaaaaaaaaa2==>bbbbbbbbbbbb to aaaaaaaaaa do
    write aaaaaaaaaa2 * aaaaaaaaaa2;

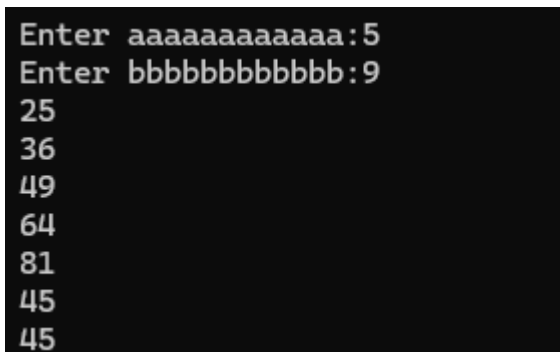
xxxxxxx==>0;
cccccccc1==>0;
while ccccccc1 << aaaaaaaaaa
startblok
    ccccccc2==>0;
    while ccccccc2 << bbbbbbbbbbb
    startblok
        xxxxxxx==>xxxxxxx add 1;
        ccccccc2==>ccccccc2 add 1;
    endblok
    end while
ccccccc1==>ccccccc1 add 1;
endblok
end while
write xxxxxxxxxx;

xxxxxxx==>0;
cccccccc1==>1;
repeat
startblok
    ccccccc2==>1;
    repeat
    startblok
        xxxxxxx==>xxxxxxx add 1;
        ccccccc2==>ccccccc2 add 1;
    endblok
    until !(ccccccc2 >> bbbbbbbbbbb)
    ccccccc1==>ccccccc1 add 1;
endblok
until !(ccccccc1 >> aaaaaaaaaa)
write xxxxxxxxxx;

endblok

```

Результат виконання



```

Enter aaaaaaaaaa:5
Enter bbbbbbbbbbb:9
25
36
49
64
81
45
45

```

Рис. 5.4 Результат виконання тестової програми №3

Висновки

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування p16, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створено компілятор мови програмування p16, 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 p16. 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 p16 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. 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.
2. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. –Харків: НТУ «ХПІ», 2021. – 133 с.
3. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії формальних мов: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 с.
4. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 с.
5. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. – 1038 с.
6. Системне програмування (курсовий проект) [Електронний ресурс] – Режим доступу до ресурсу: <https://vns.lpnu.ua/course/view.php?id=11685>.
7. MIT OpenCourseWare. Computer Language Engineering [Електронний ресурс] – Режим доступу до ресурсу: <https://ocw.mit.edu/courses/6-035-computer-language-engineering-spring2010>.
8. Рисований О.М. Системне програмування: підручник для студентів напрямку “Комп'ютерна інженерія” вищих навчальних закладів в 2-х томах. Том 1. – Видання четверте: виправлено та доповнено – Х.: “Слово”, 2015. – 576 с.
9. Рисований О.М. Системне програмування: підручник для студентів напрямку “Комп'ютерна інженерія” вищих навчальних закладів в 2-х томах. Том 2. – Видання четверте: виправлено та доповнено – Х.: “Слово”, 2015. – 378 с.

Додатки

Додаток А. (Код на мові С)

Prog1.c

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

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

```
int main()
```

```
{
```

```
    int aaaaaaaaaa, bbbbbbbbbbbb, xxxxxxxxxxxx, yyyyyyyyyyyy;
```

```
    printf("Enter aaaaaaaaaa:");
```

```
    scanf("%d", &aaaaaaaaaa);
```

```
    printf("Enter bbbbbbbbbbbb:");
```

```
    scanf("%d", &bbbbbbbbbb);
```

```
    printf("%d\n", aaaaaaaaaa + bbbbbbbbbbbb);
```

```
    printf("%d\n", aaaaaaaaaa - bbbbbbbbbbbb);
```

```
    printf("%d\n", aaaaaaaaaa * bbbbbbbbbbbb);
```

```
    printf("%d\n", aaaaaaaaaa / bbbbbbbbbbbb);
```

```
    printf("%d\n", aaaaaaaaaa % bbbbbbbbbbbb);
```

```
    xxxxxxxxxxxx = (aaaaaaaaaa - bbbbbbbbbbbb) * 10 + (aaaaaaaaaa + bbbbbbbbbbbb) /
```

```
10;
```

```
    yyyyyyyyyyyy = xxxxxxxxxxxx + (xxxxxxxxxxx % 10);
```

```
    printf("%d\n", xxxxxxxxxxxx);
```

```
    printf("%d\n", yyyyyyyyyyyy);
```

```
    system("pause");
```

```
    return 0;
```

```
}
```

Prog2.c

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

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

```
int main()
```

```
{
```

```
    int aaaaaaaaaa, bbbbbbbbbbbb, cccccccccc;
```

```
    printf("Enter aaaaaaaaaa:");
```

```
    scanf("%d", &aaaaaaaaaa);
```

```
    printf("Enter bbbbbbbbbbbb:");
```

```
    scanf("%d", &bbbbbbbbbb);
```

```
    printf("Enter cccccccccc:");
```

```
    scanf("%d", &ccccccccc);
```

```
    if ((aaaaaaaaaa > bbbbbbbbbbbb))
```

```
    {
```

```
        if ((aaaaaaaaaa > cccccccccc))
```

```
        {
```

```
            goto Abigger;
```

```
        }
```

```
    else
```

```
    {
```

```
        printf("%d\n", cccccccccc);
```

```
        goto Outofif;
```

Abigger:

```
printf("%d\n", aaaaaaaaaa);
goto Outofif;
}
}
if ((bbbbbbbbbbbbbb < cccccccccccc))
{
printf("%d\n", cccccccccccc);
}
else
{
printf("%d\n", bbbbbbbbbbbb);
}
```

Outofif:

```
if (((aaaaaaaaaaaa == bbbbbbbbbbbb) && (aaaaaaaaaaaa == cccccccccccc) &&
(bbbbbbbbbbbb == cccccccccccc)))
{
printf("%d\n", 1);
}
else
{
printf("%d\n", 0);
}
if (((aaaaaaaaaaaa < 0) || (bbbbbbbbbbbbbb < 0) || (cccccccccccc < 0)))
{
printf("%d\n", -1);
}
else
{
printf("%d\n", 0);
}
if (!(aaaaaaaaaaaa < (bbbbbbbbbbbbbb + cccccccccccc)))
{
printf("%d\n", (10));
}
else
{
printf("%d\n", (0));
}
system("pause");
return 0;
}
```

Prog3.c

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

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

```
int main()
```

```
{
    int aaaaaaaaaa, aaaaaaaaaa2, bbbbbbbbbbbb, xxxxxxxxxxxx, ccccccccc1, ccccccccc2;
    printf("Enter aaaaaaaaaa:");
```

```

scanf("%d", &aaaaaaaaaaaa);
printf("Enter bbbbbbbbbbbb:");
scanf("%d", &bbbbbbbbbbbb);
for (int aaaaaaaaaa2 = aaaaaaaaaa; aaaaaaaaaa2 <= bbbbbbbbbbbb; aaaaaaaaaa2++)
printf("%d\n", aaaaaaaaaa2 * aaaaaaaaaa2);
for (int aaaaaaaaaa2 = bbbbbbbbbbbb; aaaaaaaaaa2 <= aaaaaaaaaa; aaaaaaaaaa2++)
printf("%d\n", aaaaaaaaaa2 * aaaaaaaaaa2);
xxxxxxxxxxxx = 0;
cccccccccc1 = 0;
while (cccccccccc1 < aaaaaaaaaa)
{
{
cccccccccc2 = 0;
while (cccccccccc2 < bbbbbbbbbbbb)
{
{
xxxxxxxxxxxx = xxxxxxxxxxxx + 1;
cccccccccc2 = ccccccccc2 + 1;
}
}
cccccccccc1 = ccccccccc1 + 1;
}
}
printf("%d\n", xxxxxxxxxxxx);
xxxxxxxxxxxx = 0;
cccccccccc1 = 1;
do
{
cccccccccc2 = 1;
do
{
xxxxxxxxxxxx = xxxxxxxxxxxx + 1;
cccccccccc2 = ccccccccc2 + 1;
}
while (!(cccccccccc2 > bbbbbbbbbbbb));
cccccccccc1 = ccccccccc1 + 1;
}
while (!(cccccccccc1 > aaaaaaaaaa));
printf("%d\n", xxxxxxxxxxxx);
system("pause");
return 0;
}

```

Додаток Б. (Тестові програми)

Тестова програма «Лінійний алгоритм»

```
@ @Prog1
startprogram
startblok
variable longint aaaaaaaaaa,bbbbbbbbbbbb,xxxxxxxxxxxx,yyyyyyyyyyyy;
read aaaaaaaaaa;
read bbbbbbbbbbbb;
write aaaaaaaaaa add bbbbbbbbbbbb;
write aaaaaaaaaa sub bbbbbbbbbbbb;
write aaaaaaaaaa * bbbbbbbbbbbb;
write aaaaaaaaaa / bbbbbbbbbbbb;
write aaaaaaaaaa % bbbbbbbbbbbb;

xxxxxxxxxxxx==>(aaaaaaaaaaaa sub bbbbbbbbbbbb) * 10 add (aaaaaaaaaaaa add bbbbbbbbbbbb) /
10;
yyyyyyyyyyyy==>xxxxxxxxxxxx add (xxxxxxxxxxxx % 10);
write xxxxxxxxxxxx;
write yyyyyyyyyy;
endblok
```

Тестова програма «Алгоритм з розгалуженням»

```
@ @Prog2
startprogram
startblok
variable longint aaaaaaaaaa,bbbbbbbbbbbb,cccccccccc;
read aaaaaaaaaa;
read bbbbbbbbbbbb;
read cccccccccc;
if(aaaaaaaaaa >> bbbbbbbbbbbb)
startblok
    if(aaaaaaaaaa >> cccccccccc)
    startblok
        goto Abigger;
    endblok
    else
    startblok
        write cccccccccc;
        goto Outofif;
        Abigger:
        write aaaaaaaaaa;
        goto Outofif;
    endblok
endblok
if(bbbbbbbbbbbb << cccccccccc)
startblok
```

```

        write ccccccccccc;
endblok
else
startblok
    write bbbbbbbbbbb;
endblok
Outofif:

if((aaaaaaaaaaaa == bbbbbbbbbbbb) & (aaaaaaaaaaaa == ccccccccccc) & (bbbbbbbbbbbb ==
cccccccccc))
startblok
    write 1;
endblok
else
startblok
    write 0;
endblok
if((aaaaaaaaaaaa << 0) | (bbbbbbbbbbbb << 0) | (cccccccccc << 0))
startblok
    write -1;
endblok
else
startblok
    write 0;
endblok
if(!(aaaaaaaaaaaa << (bbbbbbbbbbbb add ccccccccccc)))
startblok
    write(10);
endblok
else
startblok
    write(0);
endblok
endblok

```

Тестова програма «Циклічний алгоритм»

```

@@Prog3
startprogram
startblok
variable longint
aaaaaaaaaaaa,aaaaaaaaaa2,bbbbbbbbbbbb,xxxxxxxxxxx,cccccccccc1,cccccccccc2;
read aaaaaaaaaa;
read bbbbbbbbbbb;
for aaaaaaaaaaa2==>aaaaaaaaaaaa to bbbbbbbbbbbb do
    write aaaaaaaaaaa2 * aaaaaaaaaaa2;

for aaaaaaaaaaa2==>bbbbbbbbbbbb to aaaaaaaaaaaaa do
    write aaaaaaaaaaa2 * aaaaaaaaaaa2;

```

```

xxxxxxxxxxxx==>0;
cccccccccc1==>0;
while ccccccccc1 << aaaaaaaaaa
startblok
    ccccccccc2==>0;
    while ccccccccc2 << bbbbbbbbbbbb
    startblok
        xxxxxxxxxxx==>xxxxxxxxxxx add 1;
        ccccccccc2==>ccccccccc2 add 1;
    endblok
    end while
cccccccccc1==>cccccccccc1 add 1;
endblok
end while
write xxxxxxxxxxxxxx;

xxxxxxxxxxxx==>0;
cccccccccc1==>1;
repeat
startblok
    ccccccccc2==>1;
    repeat
    startblok
        xxxxxxxxxxx==>xxxxxxxxxxx add 1;
        ccccccccc2==>ccccccccc2 add 1;
    endblok
    until !(ccccccccc2 >> bbbbbbbbbbbb)
    ccccccccc1==>ccccccccc1 add 1;
    endblok
until !(ccccccccc1 >> aaaaaaaaaa)
write xxxxxxxxxxxxxx;

endblok

```


Додаток В. (Таблиці лексем для тестових прикладів)

Тестова програма «Лінійний алгоритм»

TOKEN TABLE					
line number	token	value	token code	type of token	
2	startprogram	0	0	MainProgram	
3	startblok	0	1	StartProgram	
4	variable	0	2	Variable	
4	longint	0	3	Integer	
4	aaaaaaaaaaaa	0	22	Identifier	
4	,	0	41	Comma	
4	bbbbbbbbbbbb	0	22	Identifier	
4	,	0	41	Comma	
4	xxxxxxxxxxxx	0	22	Identifier	
4	,	0	41	Comma	
4	yyyyyyyyyyyy	0	22	Identifier	
4	;	0	39	Semicolon	
5	read	0	5	Input	
5	aaaaaaaaaaaa	0	22	Identifier	
5	;	0	39	Semicolon	
6	read	0	5	Input	
6	bbbbbbbbbbbb	0	22	Identifier	
6	;	0	39	Semicolon	
7	write	0	6	Output	
7	aaaaaaaaaaaa	0	22	Identifier	
7	add	0	25	Add	
7	bbbbbbbbbbbb	0	22	Identifier	
7	;	0	39	Semicolon	
8	write	0	6	Output	

8	aaaaaaaaaaaa	0	22	Identifier
8	sub	0	26	Sub
8	bbbbbbbbbbbb	0	22	Identifier
8	;	0	39	Semicolon
9	write	0	6	Output
9	aaaaaaaaaaaa	0	22	Identifier
9	*	0	27	Mul
9	bbbbbbbbbbbb	0	22	Identifier
9	;	0	39	Semicolon
10	write	0	6	Output
10	aaaaaaaaaaaa	0	22	Identifier
10	/	0	28	Div
10	bbbbbbbbbbbb	0	22	Identifier
10	;	0	39	Semicolon
11	write	0	6	Output
11	aaaaaaaaaaaa	0	22	Identifier
11	%	0	29	Mod
11	bbbbbbbbbbbb	0	22	Identifier
11	;	0	39	Semicolon
13	xxxxxxxxxxxx	0	22	Identifier
13	=>	0	24	Assign
13	(0	37	LBraket
13	aaaaaaaaaaaa	0	22	Identifier
13	sub	0	26	Sub
13	bbbbbbbbbbbb	0	22	Identifier
13)	0	38	RBraket
13	*	0	27	Mul
13	10	10	23	Number

	13		add		0		25		Add	
	13		(0		37		LBracket	
	13		aaaaaaaaaaaa		0		22		Identifier	
	13		add		0		25		Add	
	13		bbbbbbbbbbbbbb		0		22		Identifier	
	13)		0		38		RBracket	
	13		/		0		28		Div	
	13		10		10		23		Number	
	13		;		0		39		Semicolon	
	14		yyyyyyyyyyyyyy		0		22		Identifier	
	14		==>		0		24		Assign	
	14		xxxxxxxxxxxxxx		0		22		Identifier	
	14		add		0		25		Add	
	14		(0		37		LBracket	
	14		xxxxxxxxxxxxxx		0		22		Identifier	
	14		%		0		29		Mod	
	14		10		10		23		Number	
	14)		0		38		RBracket	
	14		;		0		39		Semicolon	
	15		write		0		6		Output	
	15		xxxxxxxxxxxxxx		0		22		Identifier	
	15		;		0		39		Semicolon	
	16		write		0		6		Output	
	16		yyyyyyyyyyyyyy		0		22		Identifier	
	16		;		0		39		Semicolon	
	17		endblok		0		4		EndProgram	

Тестова програма «Алгоритм з розгалуженням»

TOKEN TABLE					
line number	token	value	token code	type of token	
2	startprogram	0	0	MainProgram	
3	startblok	0	1	StartProgram	
4	variable	0	2	Variable	
4	longint	0	3	Integer	
4	aaaaaaaaaaaa	0	22	Identifier	
4	,	0	41	Comma	
4	bbbbbbbbbbbbbb	0	22	Identifier	
4	,	0	41	Comma	
4	cccccccccccccc	0	22	Identifier	
4	;	0	39	Semicolon	
5	read	0	5	Input	
5	aaaaaaaaaaaa	0	22	Identifier	
5	;	0	39	Semicolon	
6	read	0	5	Input	
6	bbbbbbbbbbbbbb	0	22	Identifier	
6	;	0	39	Semicolon	
7	read	0	5	Input	
7	cccccccccccccc	0	22	Identifier	
7	;	0	39	Semicolon	
8	if	0	7	If	
8	(0	37	LBraket	
8	aaaaaaaaaaaa	0	22	Identifier	
8	>>	0	32	Greate	
8	bbbbbbbbbbbbbb	0	22	Identifier	
8)	0	38	RBraket	

9	startblok	0	1	StartProgram
10	if	0	7	If
10	(0	37	LBraket
10	aaaaaaaaaaaa	0	22	Identifier
10	>>	0	32	Greate
10	cccccccccccc	0	22	Identifier
10)	0	38	RBraket
11	startblok	0	1	StartProgram
12	goto	0	10	Goto
12	Abigger	0	22	Identifier
12	;	0	39	Semicolon
13	endblok	0	4	EndProgram
14	else	0	9	Else
15	startblok	0	1	StartProgram
16	write	0	6	Output
16	cccccccccccc	0	22	Identifier
16	;	0	39	Semicolon
17	goto	0	10	Goto
17	Outofif	0	22	Identifier
17	;	0	39	Semicolon
18	Abigger	0	11	Label
19	write	0	6	Output
19	aaaaaaaaaaaa	0	22	Identifier
19	;	0	39	Semicolon
20	goto	0	10	Goto
20	Outofif	0	22	Identifier
20	;	0	39	Semicolon
21	endblok	0	4	EndProgram

	22		endblok		0		4		EndProgram	
	23		if		0		7		If	
	23		(0		37		LBraket	
	23		bbbbbbbbbbbb		0		22		Identifier	
	23		<<		0		33		Less	
	23		cccccccccccc		0		22		Identifier	
	23)		0		38		RBraket	
	24		startblok		0		1		StartProgram	
	25		write		0		6		Output	
	25		cccccccccccc		0		22		Identifier	
	25		;		0		39		Semicolon	
	26		endblok		0		4		EndProgram	
	27		else		0		9		Else	
	28		startblok		0		1		StartProgram	
	29		write		0		6		Output	
	29		bbbbbbbbbbbb		0		22		Identifier	
	29		;		0		39		Semicolon	
	30		endblok		0		4		EndProgram	
	31		Outofif		0		11		Label	
	33		if		0		7		If	
	33		(0		37		LBraket	
	33		(0		37		LBraket	
	33		aaaaaaaaaaaa		0		22		Identifier	
	33		==		0		30		Equality	
	33		bbbbbbbbbbbb		0		22		Identifier	
	33)		0		38		RBraket	
	33		&		0		35		And	
	33		(0		37		LBraket	
	33		aaaaaaaaaaaa		0		22		Identifier	

33	==	0	30	Equality
33	cccccccccccc	0	22	Identifier
33)	0	38	RBracket
33	&	0	35	And
33	(0	37	LBracket
33	bbbbbbbbbbbb	0	22	Identifier
33	==	0	30	Equality
33	cccccccccccc	0	22	Identifier
33)	0	38	RBracket
33)	0	38	RBracket
34	startblok	0	1	StartProgram
35	write	0	6	Output
35	1	1	23	Number
35	;	0	39	Semicolon
36	endblok	0	4	EndProgram
37	else	0	9	Else
38	startblok	0	1	StartProgram
39	write	0	6	Output
39	0	0	23	Number
39	;	0	39	Semicolon
40	endblok	0	4	EndProgram
41	if	0	7	If
41	(0	37	LBracket
41	(0	37	LBracket
41	aaaaaaaaaaaa	0	22	Identifier
41	<<	0	33	Less
41	0	0	23	Number
41)	0	38	RBracket

	41			0		36		Or	
	41		(0		37		LBraket
	41		bbbbbbbbbbbb		0		22		Identifier
	41		<<		0		33		Less
	41		0		0		23		Number
	41)		0		38		RBraket
	41				0		36		Or
	41		(0		37		LBraket
	41		cccccccccccc		0		22		Identifier
	41		<<		0		33		Less
	41		0		0		23		Number
	41)		0		38		RBraket
	41)		0		38		RBraket
	42		startblok		0		1		StartProgram
	43		write		0		6		Output
	43		-		0		42		Minus
	43		1		1		23		Number
	43		;		0		39		Semicolon
	44		endblok		0		4		EndProgram
	45		else		0		9		Else
	46		startblok		0		1		StartProgram
	47		write		0		6		Output
	47		0		0		23		Number
	47		;		0		39		Semicolon
	48		endblok		0		4		EndProgram
	49		if		0		7		If
	49		(0		37		LBraket
	49		!!		0		34		Not
	49		(0		37		LBraket

49	aaaaaaaaaaaa	0	22	Identifier
49	<<	0	33	Less
49	(0	37	LBracket
49	bbbbbbbbbbbb	0	22	Identifier
49	add	0	25	Add
49	cccccccccccc	0	22	Identifier
49)	0	38	RBracket
49)	0	38	RBracket
49)	0	38	RBracket
50	startblok	0	1	StartProgram
51	write	0	6	Output
51	(0	37	LBracket
51	10	10	23	Number
51)	0	38	RBracket
51	;	0	39	Semicolon
52	endblok	0	4	EndProgram
53	else	0	9	Else
54	startblok	0	1	StartProgram
55	write	0	6	Output
55	(0	37	LBracket
55	0	0	23	Number
55)	0	38	RBracket
55	;	0	39	Semicolon
56	endblok	0	4	EndProgram
57	endblok	0	4	EndProgram

Тестова програма «Циклічний алгоритм»

TOKEN TABLE					
line number	token	value	token code	type of token	
2	startprogram	0	0	MainProgram	
3	startblok	0	1	StartProgram	
4	variable	0	2	Variable	
4	longint	0	3	Integer	
4	aaaaaaaaaaaa	0	22	Identifier	
4	,	0	41	Comma	
4	aaaaaaaaaaaa2	0	22	Identifier	
4	,	0	41	Comma	
4	bbbbbbbbbbbbbb	0	22	Identifier	
4	,	0	41	Comma	
4	xxxxxxxxxxxxxx	0	22	Identifier	
4	,	0	41	Comma	
4	cccccccccccc1	0	22	Identifier	
4	,	0	41	Comma	
4	cccccccccccc2	0	22	Identifier	
4	;	0	39	Semicolon	
5	read	0	5	Input	
5	aaaaaaaaaaaa	0	22	Identifier	
5	;	0	39	Semicolon	
6	read	0	5	Input	
6	bbbbbbbbbbbbbb	0	22	Identifier	
6	;	0	39	Semicolon	
7	for	0	12	For	
7	aaaaaaaaaaaa2	0	22	Identifier	

	7		==>		0		24		Assign	
	7		aaaaaaaaaaaa		0		22		Identifier	
	7		to		0		13		To	
	7		bbbbbbbbbbbb		0		22		Identifier	
	7		do		0		15		Do	
	8		write		0		6		Output	
	8		aaaaaaaaaaaa2		0		22		Identifier	
	8		*		0		27		Mul	
	8		aaaaaaaaaaaa2		0		22		Identifier	
	8		;		0		39		Semicolon	
	10		for		0		12		For	
	10		aaaaaaaaaaaa2		0		22		Identifier	
	10		==>		0		24		Assign	
	10		bbbbbbbbbbbb		0		22		Identifier	
	10		to		0		13		To	
	10		aaaaaaaaaaaa		0		22		Identifier	
	10		do		0		15		Do	
	11		write		0		6		Output	
	11		aaaaaaaaaaaa2		0		22		Identifier	
	11		*		0		27		Mul	
	11		aaaaaaaaaaaa2		0		22		Identifier	
	11		;		0		39		Semicolon	
	13		xxxxxxxxxxxx		0		22		Identifier	
	13		==>		0		24		Assign	
	13		0		0		23		Number	
	13		;		0		39		Semicolon	
	14		cccccccccccc1		0		22		Identifier	
	14		==>		0		24		Assign	
	14		0		0		23		Number	

14	;	0	39	Semicolon
15	while	0	16	While
15	cccccccccccc1	0	22	Identifier
15	<<	0	33	Less
15	aaaaaaaaaaaa	0	22	Identifier
16	startblok	0	1	StartProgram
17	cccccccccccc2	0	22	Identifier
17	=>	0	24	Assign
17	0	0	23	Number
17	;	0	39	Semicolon
18	while	0	16	While
18	cccccccccccc2	0	22	Identifier
18	<<	0	33	Less
18	bbbbbbbbbbbbbb	0	22	Identifier
19	startblok	0	1	StartProgram
20	xxxxxxxxxxxxxx	0	22	Identifier
20	=>	0	24	Assign
20	xxxxxxxxxxxxxx	0	22	Identifier
20	add	0	25	Add
20	1	1	23	Number
20	;	0	39	Semicolon
21	cccccccccccc2	0	22	Identifier
21	=>	0	24	Assign
21	cccccccccccc2	0	22	Identifier
21	add	0	25	Add
21	1	1	23	Number
21	;	0	39	Semicolon
22	endblok	0	4	EndProgram

	23		end		0		19		End	
	23		while		0		16		While	
	24		cccccccccccc1		0		22		Identifier	
	24		==>		0		24		Assign	
	24		cccccccccccc1		0		22		Identifier	
	24		add		0		25		Add	
	24		1		1		23		Number	
	24		;		0		39		Semicolon	
	25		endblok		0		4		EndProgram	
	26		end		0		19		End	
	26		while		0		16		While	
	27		write		0		6		Output	
	27		xxxxxxxxxxxxx		0		22		Identifier	
	27		;		0		39		Semicolon	
	29		xxxxxxxxxxxxx		0		22		Identifier	
	29		==>		0		24		Assign	
	29		0		0		23		Number	
	29		;		0		39		Semicolon	
	30		cccccccccccc1		0		22		Identifier	
	30		==>		0		24		Assign	
	30		1		1		23		Number	
	30		;		0		39		Semicolon	
	31		repeat		0		20		Repeat	
	32		startblok		0		1		StartProgram	
	33		cccccccccccc2		0		22		Identifier	
	33		==>		0		24		Assign	
	33		1		1		23		Number	
	33		;		0		39		Semicolon	
	34		repeat		0		20		Repeat	

35	startblok	0	1	StartProgram
36	xxxxxxxxxxxxx	0	22	Identifier
36	==>	0	24	Assign
36	xxxxxxxxxxxxx	0	22	Identifier
36	add	0	25	Add
36	1	1	23	Number
36	;	0	39	Semicolon
37	cccccccccccc2	0	22	Identifier
37	==>	0	24	Assign
37	cccccccccccc2	0	22	Identifier
37	add	0	25	Add
37	1	1	23	Number
37	;	0	39	Semicolon
38	endblok	0	4	EndProgram
39	until	0	21	Until
39	!!	0	34	Not
39	(0	37	LBraket
39	cccccccccccc2	0	22	Identifier
39	>>	0	32	Greate
39	bbbbbbbbbbbbbb	0	22	Identifier
39)	0	38	RBraket
40	cccccccccccc1	0	22	Identifier
40	==>	0	24	Assign
40	cccccccccccc1	0	22	Identifier
40	add	0	25	Add
40	1	1	23	Number
40	;	0	39	Semicolon
41	endblok	0	4	EndProgram

	42		until		0		21		Until	
	42		!!		0		34		Not	
	42		(0		37		LBracket	
	42		cccccccccccc1		0		22		Identifier	
	42		>>		0		32		Greate	
	42		aaaaaaaaaaaa		0		22		Identifier	
	42)		0		38		RBracket	
	43		write		0		6		Output	
	43		xxxxxxxxxxxxxx		0		22		Identifier	
	43		;		0		39		Semicolon	
	45		endblok		0		4		EndProgram	

Додаток Г (Абстрактне синтаксичне дерево для тестових прикладів)

Тестова програма «Лінійний алгоритм»

```
-- program
|  |-- var
|  |  |-- yyyyyyyyyyyy
|  |  |-- var
|  |  |  |-- xxxxxxxxxxxxxx
|  |  |  |-- var
|  |  |  |  |-- bbbbbbbbbbbb
|  |  |  |  |-- var
|  |  |  |  |  |-- aaaaaaaaaaaa
|  |-- statement
|  |  |-- statement
|  |  |  |-- statement
|  |  |  |  |-- statement
|  |  |  |  |  |-- statement
|  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |  |  |-- statement
|  |  |  |  |  |  |  |  |  |-- input
|  |  |  |  |  |  |  |  |  |  |-- aaaaaaaaaaaa
|  |  |  |  |  |  |  |  |  |  |-- input
|  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbbbbbb
|  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |-- +
|  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbbbbbb
|  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |-- -
|  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbbbbbb
|  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |-- *
|  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbbbbbb
|  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |-- /
|  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbbbbbb
|  |  |  |  |  |  |  |  |  |  |-- output
|  |  |  |  |  |  |  |  |  |  |-- %
|  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbbbbbb
|  |  |  |  |  |  |  |  |  |  |-- ==>
|  |  |  |  |  |  |  |  |  |  |  |-- xxxxxxxxxxxxxx
|  |  |  |  |  |  |  |  |  |  |  |-- +
|  |  |  |  |  |  |  |  |  |  |  |-- *
|  |  |  |  |  |  |  |  |  |  |  |-- -
|  |  |  |  |  |  |  |  |  |  |  |-- aaaaaaaaaaaa
|  |  |  |  |  |  |  |  |  |  |  |-- bbbbbbbbbbbb
|  |  |  |  |  |  |  |  |  |  |-- 10
```



```
| | | | | | | |-- /  
| | | | | | | |-- +  
| | | | | | | |-- aaaaaaaaaaaaaa  
| | | | | | | |-- bbbbbbbbbbbbbb  
| | | | | | | |-- 10  
  
| | | | | |-- ==>  
| | | | | |-- yyyyyyyyyyyyyy  
| | | | | |-- +  
| | | | | |-- xxxxxxxxxxxxxx  
| | | | | |-- %  
| | | | | |-- xxxxxxxxxxxxxx  
| | | | | |-- 10  
  
| | | |-- output  
| | | |-- xxxxxxxxxxxxxx  
  
| | |-- output  
| | |-- yyyyyyyyyyyyyy
```

Тестова програма «Алгоритм з розгалуженням»

```

|-- program
|-- var
|   |-- cccccccccccc
|   |-- var
|   |   |-- bbbbbbbbbbbbbb
|   |   |-- var
|   |   |   |-- aaaaaaaaaaaaaa
|-- statement
|   |-- statement
|   |   |-- statement
|   |   |   |-- statement
|   |   |   |   |-- statement
|   |   |   |   |   |-- statement
|   |   |   |   |   |-- input
|   |   |   |   |   |   |-- aaaaaaaaaaaaaa
|   |   |   |   |   |   |-- input
|   |   |   |   |   |   |-- bbbbbbbbbbbbbb
|   |   |   |   |-- input
|   |   |   |-- cccccccccccc
|   |-- if
|   |   |-- >>
|   |   |   |-- aaaaaaaaaaaaaa
|   |   |   |-- bbbbbbbbbbbbbb
|   |   |-- branches
|   |   |   |-- compound
|   |   |   |   |-- if
|   |   |   |   |   |-- >>
|   |   |   |   |   |   |-- aaaaaaaaaaaaaa
|   |   |   |   |   |   |-- cccccccccccc
|   |   |   |   |-- branches
|   |   |   |   |   |-- compound
|   |   |   |   |   |-- goto
|   |   |   |   |   |   |-- Abigger
|   |   |-- compound
|   |   |   |-- statement

```

```

|-- statement
|-- statement
|-- statement
|-- output
|-- cccccccccc
|-- goto
|-- Outofif
|-- Abigger
|-- output
|-- aaaaaaaaaaaa
|-- goto
|-- Outofif

|-- if
|-- <<
|-- bbbbbbbbbbbb
|-- cccccccccc
|-- branches
|-- compound
|-- output
|-- cccccccccc
|-- compound
|-- output
|-- bbbbbbbbbbbb

|-- Outofif
|-- if
|-- &
|-- &
|-- ==
|-- aaaaaaaaaaaa
|-- bbbbbbbbbbbb
|-- ==
|-- aaaaaaaaaaaa
|-- cccccccccc
|-- ==
|-- bbbbbbbbbbbb
|-- cccccccccc

|-- branches
|-- compound
|-- output
|-- 1
|-- compound
|-- output
|-- 0

|-- if
|-- |
|-- |
|-- <<
|-- aaaaaaaaaaaa
|-- 0
|-- <<
|-- bbbbbbbbbbbb
|-- 0
|-- <<
|-- cccccccccc
|-- 0

|-- branches
|-- compound
|-- output

```



```

-- body
-- bbbbbbbbbbbbbbb
-- output
-- *
-- aaaaaaaaaaaa2
-- aaaaaaaaaaaa2

-- for-to
-- ==>
-- aaaaaaaaaaaa2
-- bbbbbbbbbbbbbbb
-- body
-- aaaaaaaaaaaa
-- output
-- *
-- aaaaaaaaaaaa2
-- aaaaaaaaaaaa2

-- ==>
-- xxxxxxxxxxxxxx
-- 0

-- ==>
-- cccccccccc1
-- 0

-- while
-- <<
-- cccccccccc1
-- aaaaaaaaaaaa
-- statement
-- compound
-- statement
-- statement
-- ==>
-- cccccccccc2
-- 0
-- while
-- <<
-- cccccccccc2
-- bbbbbbbbbbbbbbb
-- statement
-- compound
-- statement
-- ==>
-- xxxxxxxxxxxxxx
-- +
-- xxxxxxxxxxxxxx
-- 1

-- ==>
-- cccccccccc2
-- +
-- cccccccccc2
-- 1

-- ==>
-- cccccccccc1
-- +
-- cccccccccc1
-- 1

-- output
-- xxxxxxxxxxxxxx

-- ==>

```


Додаток Д (Документований текст програмних модулів (лістинги))

ast.cpp

Ця програма реалізує синтаксичний аналізатор для спрощеної мови програмування. Вона перетворює вхідний код на абстрактне синтаксичне дерево (AST).

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

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

static int pos = 0;

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

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

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

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

// набір функцій для рекурсивного спуску
// на кожне правило - окрема функція
ASTNode* program();
ASTNode* variable_declaration();
ASTNode* variable_list();
ASTNode* program_body();
ASTNode* statement();
ASTNode* assignment();
ASTNode* arithmetic_expression();
ASTNode* term();
```

```

ASTNode* factor();
ASTNode* input();
ASTNode* output();
ASTNode* conditional();

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

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

// функція синтаксичного аналізу і створення абстрактного синтаксичного дерева
ASTNode* ParserAST()
{
    ASTNode* tree = program();

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

    return tree;
}

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

// <програма> = 'start' 'var' <оголошення змінних> ';' <тіло програми> 'stop'
ASTNode* program()
{
    match(Mainprogram);
    match(StartProgram);
    match(Variable);
    ASTNode* declarations = variable_declaration();
    match(Semicolon);
    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();
// Повертаємо вузол циклу for-to
return createNode(for_to_node, "for-to",
    createNode(assign_node, "==>", var, start),
    createNode(statement_node, "body", end, body));
}

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

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

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

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

```

```

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

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

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

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

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

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

// <логічний вираз> = <вираз I> { '|' <вираз I> }
ASTNode* logical_expression()
{
    ASTNode* left = and_expression();
    while (TokenTable[pos].type == Or)
    {
        match(Or);
        ASTNode* right = and_expression();
        left = createNode(or_node, "|", left, right);
    }
    return left;
}

// <вираз I> = <порівняння> { '&' <порівняння> }
ASTNode* and_expression()
{
    ASTNode* left = comparison();

```

```

while (TokenTable[pos].type == And)
{
    match(And);
    ASTNode* right = comparison();
    left = createNode(and_node, "&", left, right);
}
return left;
}

// <порівняння> = <операція порівняння> | '!' ('<логічний вираз>') | ('<логічний вираз>')
// <операція порівняння> = <арифметичний вираз> <менше-більше> <арифметичний вираз>
// <менше-більше> = '>' | '<' | '=' | '<>'
ASTNode* comparison()
{
    if (TokenTable[pos].type == Not)
    {
        // Варіант: ! (<логічний вираз>)
        match(Not);
        match(LBracket);
        ASTNode* expr = logical_expression();
        match(RBracket);
        return createNode(not_node, "!", expr, NULL);
    }
    else
    if (TokenTable[pos].type == LBracket)
    {
        // Варіант: (<логічний вираз>)
        match(LBracket);
        ASTNode* expr = logical_expression();
        match(RBracket);
        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 піддерева
    if (node->left || node->right)
    {
        PrintASTToFile(node->left, level + 1, outFile);
        PrintASTToFile(node->right, level + 1, outFile);
    }
}

```

codegen.cpp

Ця програма генерує код на мові C із заданого списку лексем.

```

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

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

```

```

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

static int pos = 2;

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

void gen_variable_declaration(FILE* outFile);
void gen_variable_list(FILE* outFile);
void gen_program_body(FILE* outFile);
void gen_statement(FILE* outFile);
void gen_assignment(FILE* outFile);
void gen_arithmetic_expression(FILE* outFile);
void gen_term(FILE* outFile);
void gen_factor(FILE* outFile);
void gen_input(FILE* outFile);
void gen_output(FILE* outFile);
void gen_conditional(FILE* outFile);

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

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

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

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

```



```

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

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

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

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

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

// <вив>д = 'output' <идентификатор>
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  {\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> = <порядкованное выражение> { '&' <порядкованное выражение> }

```

void gen_and_expression(FILE* outFile)

```

```

{
    gen_comparison(outFile);
    while (TokenTable[pos].type == And)
    {
        fprintf(outFile, " && ");
        pos++;
        gen_comparison(outFile);
    }
}

```

// <порядкованное выражение> = <операция порядка> | C!C C(C <логический выражение> C)C | C(C <логический выражение> C)C

```

// <операцїя порівняння> = <арифметичний вираз> <менше-більше> <арифметичний вираз>
// <менше-більше> = C>C | C<C | C=C | C<>C
void gen_comparison(FILE* outFile)
{
    if (TokenTable[pos].type == Not)
    {
        // ¬арґант: ! (<логічний вираз>)
        fprintf(outFile, "!(");
        pos++;
        pos++;
        gen_logical_expression(outFile);
        fprintf(outFile, ")");
        pos++;
    }
    else
    {
        if (TokenTable[pos].type == LBraket)
        {
            // ¬арґант: ( <логічний вираз> )
            fprintf(outFile, "(");
            pos++;
            gen_logical_expression(outFile);
            fprintf(outFile, ")");
            pos++;
        }
        else
        {
            // ¬арґант: <арифметичний вираз> <менше-більше> <арифметичний вираз>
            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);
            }
        }
    }
}

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

```

codegenfromast.cpp

Цей код реалізує генератор коду для перекладу тексту на мові програмування високого рівня у C-подібний код.

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

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

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

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

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

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

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

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



```

        break;

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

case or_node:

```

```

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

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

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

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

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

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

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

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

```

compile.cpp

Цей код включає функції для компіляції вихідного файлу в виконуваний (.exe) файл за допомогою GCC і виконує перевірку доступності файлу.

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

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

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

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

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

    private:
        A _action;
    };

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

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

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

    wchar_t current_dir[MAX_PATH];
    if (!GetCurrentDirectoryW(MAX_PATH, current_dir))
    {

```

```

        printf("Error retrieving current directory. Error code: %lu\n",
GetLastError());
        return;
    }

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

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

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

    STARTUPINFO si = { 0 };
    PROCESS_INFORMATION pi = { 0 };
    si.cb = sizeof(si);

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

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

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

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

```

lexer.cpp

Ця програма виконує лексичний аналіз тексту з вхідного файлу, знаходячи та зберігаючи лексеми в таблиці. Програма працює через стани, використовуючи автомат для визначення типів лексем, таких як ключові слова, ідентифікатори, числа, оператори та роздільники.

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

// функція отримує лексеми з вхідного файлу F і записує їх у таблицю лексем TokenTable
// результат функції - кількість лексем
unsigned int GetTokens(FILE* F, Token TokenTable[], FILE* errFile)
{
    States state = Start;
    Token TempToken;
    // кількість лексем
    unsigned int NumberOfTokens = 0;
    char ch, buf[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\ttoo many tokens !!!\n");
        return NumberOfTokens - 1;
    }
    break;
}

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

```

ідентифікатор // стан Letter - поточний символ - маленька літера, поточна лексема - ключове слово або

```

case Letter:
{
    buf[0] = ch;
    int j = 1;

    ch = getc(F);

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

    TypeOfTokens temp_type = Unknown;

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

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

        while (((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) && next_j < 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;
    }
}

if (!strcmp(buf, "startprogram"))    temp_type = Mainprogram;
else if (!strcmp(buf, "startblok"))    temp_type = StartProgram;
else if (!strcmp(buf, "variable"))    temp_type = Variable;
else if (!strcmp(buf, "longint"))    temp_type = Type;
else if (!strcmp(buf, "endblok"))    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, "if"))    temp_type = If;
else if (!strcmp(buf, "else"))    temp_type = Else;
else if (!strcmp(buf, "goto"))    temp_type = Goto;
else if (!strcmp(buf, "for"))    temp_type = For;
else if (!strcmp(buf, "to"))    temp_type = To;
else if (!strcmp(buf, "downto"))    temp_type = DownTo;
else if (!strcmp(buf, "do"))    temp_type = Do;
else if (!strcmp(buf, "exit"))    temp_type = Exit;
else if (!strcmp(buf, "while"))    temp_type = While;
else if (!strcmp(buf, "continue"))    temp_type = Continue;
else if (!strcmp(buf, "repeat"))    temp_type = Repeat;
else if (!strcmp(buf, "until"))    temp_type = Until;
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) == 12))
{
    bool valid = true;
    for (int i = 1; i < 12; i++)
    {
        if (! (buf[i] >= 'a' && buf[i] <= 'z') && ! (buf[i] >= '0' && buf[i] <= '9'))
        {
            valid = false;

```



```

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

case Digit:
{
    buf[0] = ch;
    int j = 1;

    ch = getc(F);

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

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

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

    ch = getc(F);

    state = Start;
    break;
}

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

    break;
}

```

```

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

case Another:
{
    switch (ch)
    {

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

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

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

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

```

```

        break;
    }

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

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

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

    case '-':
    {
        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 = Mul;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
    }

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

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

case '>':

```

```

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

    case '<':
    {
        ch = getc(F);
        if (ch == '<')
        {
            strcpy_s(TempToken.name, "<<");
            TempToken.type = Less;
            TempToken.value = 0;
            TempToken.line = line;
            ch = getc(F);
            state = Finish;
        }
        if (ch == '>')
        {
            strcpy_s(TempToken.name, "<>");
            TempToken.type = NotEquality;
            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");
    printf("|          TOKEN TABLE          |\n");
    printf("-----\n");

```

```

printf("| line number |   token   |   value   | token code | type of token |\n");
printf("-----");
for (unsigned int i = 0; i < TokensNum; i++)
{
    switch (TokenTable[i].type)
    {
    case 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 Create:

```

```

        strcpy_s(type_tokens, "Greate");
        break;
case Less:
        strcpy_s(type_tokens, "Less");
        break;
case Not:
        strcpy_s(type_tokens, "Not");
        break;
case And:
        strcpy_s(type_tokens, "And");
        break;
case Or:
        strcpy_s(type_tokens, "Or");
        break;
case LBraket:
        strcpy_s(type_tokens, "LBraket");
        break;
case RBraket:
        strcpy_s(type_tokens, "RBraket");
        break;
case Number:
        strcpy_s(type_tokens, "Number");
        break;
case Semicolon:
        strcpy_s(type_tokens, "Semicolon");
        break;
case Comma:
        strcpy_s(type_tokens, "Comma");
        break;
case Goto:
        strcpy_s(type_tokens, "Goto");
        break;
case For:
        strcpy_s(type_tokens, "For");
        break;
case To:
        strcpy_s(type_tokens, "To");
        break;
case DownTo:
        strcpy_s(type_tokens, "DownTo");
        break;
case Do:
        strcpy_s(type_tokens, "Do");
        break;
case While:
        strcpy_s(type_tokens, "While");
        break;
case Exit:
        strcpy_s(type_tokens, "Exit");
        break;
case Continue:
        strcpy_s(type_tokens, "Continue");
        break;
case End:
        strcpy_s(type_tokens, "End");
        break;
case Repeat:
        strcpy_s(type_tokens, "Repeat");
        break;
case Until:
        strcpy_s(type_tokens, "Until");

```

```

        break;
    case Label:
        strcpy_s(type_tokens, "Label");
        break;
    case Minus:
        strcpy_s(type_tokens, "Minus");
        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 Greate:
        strcpy_s(type_tokens, "Greate");
        break;
case Less:
        strcpy_s(type_tokens, "Less");
        break;
case Not:
        strcpy_s(type_tokens, "Not");
        break;
case And:
        strcpy_s(type_tokens, "And");
        break;
case Or:
        strcpy_s(type_tokens, "Or");
        break;
case LBraket:
        strcpy_s(type_tokens, "LBraket");
        break;
case RBraket:
        strcpy_s(type_tokens, "RBraket");
        break;
case Number:
        strcpy_s(type_tokens, "Number");
        break;
case Semicolon:
        strcpy_s(type_tokens, "Semicolon");

```

```

        break;
case Comma:
    strcpy_s(type_tokens, "Comma");
    break;
case Goto:
    strcpy_s(type_tokens, "Goto");
    break;
case For:
    strcpy_s(type_tokens, "For");
    break;
case To:
    strcpy_s(type_tokens, "To");
    break;
case DownTo:
    strcpy_s(type_tokens, "DownTo");
    break;
case Do:
    strcpy_s(type_tokens, "Do");
    break;
case While:
    strcpy_s(type_tokens, "While");
    break;
case Exit:
    strcpy_s(type_tokens, "Exit");
    break;
case Continue:
    strcpy_s(type_tokens, "Continue");
    break;
case End:
    strcpy_s(type_tokens, "End");
    break;
case Repeat:
    strcpy_s(type_tokens, "Repeat");
    break;
case Until:
    strcpy_s(type_tokens, "Until");
    break;
case Label:
    strcpy_s(type_tokens, "Label");
    break;
case Minus:
    strcpy_s(type_tokens, "Minus");
    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

Цей код є частиною програми, що виконує лексичний та синтаксичний аналіз вхідного файлу, генерує вихідний C код на основі аналізу та створює виконуваний файл.

```
#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 = strrchr(fileName, '.');
    if (!dot || dot == fileName) return 0; // No extension found
    return strcmp(dot, extension) == 0;
}

int main(int argc, char* argv[])
{
    // виділення пам'яті під таблицю лексем
    TokenTable = new Token[MAX_TOKENS];

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

    char InputFile[32] = "";

    FILE* InFile;

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

    // Check if the input file has the correct extension
    if (!hasValidExtension(InputFile, ".p16"))
    {
        printf("Error: Input file has invalid extension.\n");
        return 1;
    }

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

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

```

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

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

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

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

TokensNum = GetTokens(InFile, TokenTable, errFile);

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

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

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

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

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

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

fclose(outFile);

```

```

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

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

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

fclose(outFileFromAST);

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

// Close the file
_fcloseall();

destroyTree(ASTree);

delete[] TokenTable;
delete[] IdTable;

return 0;
}

```

parser.cpp

Цей код є частиною синтаксичного аналізатора (парсера) для мови програмування, яка може включати різні конструкції типу змінних, умовних операторів, циклів, виразів та інших елементів.

```
#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(TypeOfTokens type);

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

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

```

}

void match(TypeOfTokens 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(StartProgram, errFile);
    match(Variable, errFile);
    variable_declaration(errFile);
    match(Semicolon, 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 == LBracket)
            {
                match(LBracket, errFile);
                arithmetic_expression(errFile);
                match(RBracket, 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(LBracket, errFile);
        logical_expression(errFile);
        match(RBracket, errFile);
    }
    else
        if (TokenTable[pos].type == LBracket)
        {
            pos++;
            logical_expression(errFile);
            match(RBracket, 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
    {
        printf("\nSyntax error: A comparison operation is expected.\n");
        exit(12);
    }
}
}

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

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;
}

```

```

std::string TokenTypeToString(TypeOfTokens 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 Greate: return "Greate";
case Less: return "Less";
case Not: return "Not";
case And: return "And";
case Or: return "Or";
case LBraket: return "LBraket";
case RBraket: return "RBraket";
case Semicolon: return "Semicolon";
case Colon: return "Colon";
case Comma: return "Comma";
case Unknown: return "Unknown";
default: return "InvalidType";
}
}
```

translator.h

Цей код визначає структури даних та функції, які є основою для компілятора / інтерпретатора.
#pragma once

```
#pragma once
```

```
#define MAX_TOKENS 1000  
#define MAX_IDENTIFIER 10
```

```
// перерахування, яке описує всі можливі типи лексем  
enum TypeOfTokens
```

```
{  
    Mainprogram,  
    StartProgram,  
    Variable,  
    Type,  
    EndProgram,  
    Input,  
    Output,  
  
    If,  
    Then,  
    Else,  
  
    Goto,  
    Label,  
  
    For,  
    To,  
    DownTo,  
    Do,  
  
    While,  
    Exit,  
    Continue,  
    End,  
  
    Repeat,  
    Until,  
  
    Identifier,  
    Number,  
    Assign,  
    Add,  
    Sub,  
    Mul,  
    Div,  
    Mod,  
    Equality,  
    NotEquality,  
    Greate,  
    Less,  
    Not,  
    And,  
    Or,  
    LBraket,  
    RBraket,  
    Semicolon,  
    Colon,  
    Comma,  
    Minus,  
    Unknown  
};
```

```
// структура для зберігання інформації про лексему
```

```

struct Token
{
    char name[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,
    minus_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);

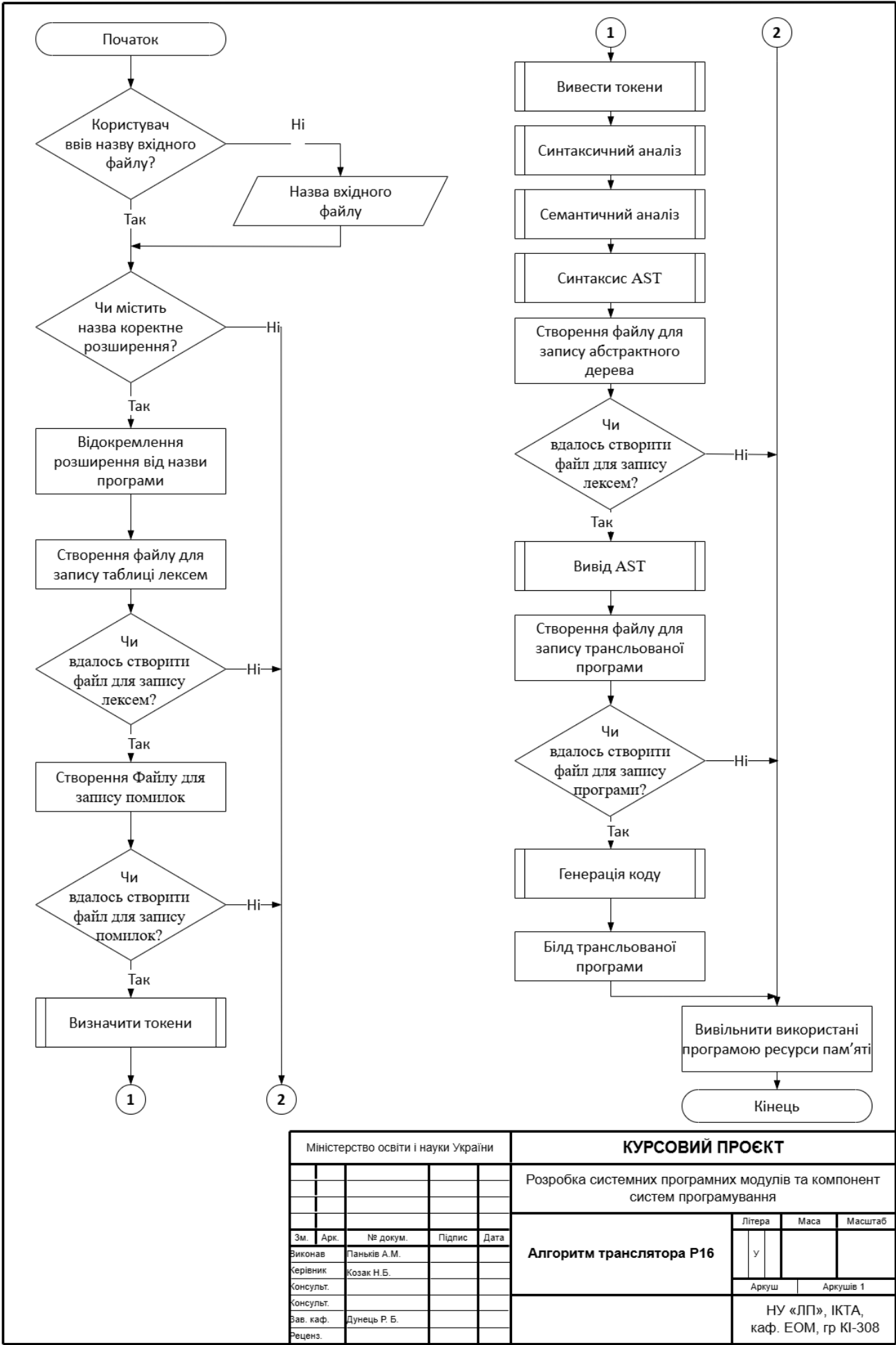
// Рекурсивна функція для генерації коду з AST
void generateCodefromAST(ASTNode* node, FILE* output);

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

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

```

Додаток Е.(Алгоритм транслятора П16)



Міністерство освіти і науки України					КУРСОВИЙ ПРОЄКТ			
					Розробка системних програмних модулів та компонент систем програмування			
					Алгоритм транслятора Р16			
Зм.	Арк.	№ докум.	Підпис	Дата	Літера	Маса	Масштаб	
Виконав		Паньків А. М.			у			
Керівник		Козак Н.Б.						
Консульт.					Аркуш		Аркушів 1	
Консульт.					НУ «ЛП», ІКТА, каф. ЕОМ, гр КІ-308			
Зав. каф.		Дунець Р. Б.						
Реценз.								