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 №3

Виконала: ст. гр. КІ-308
Бохонок О. П.
Перевірив:
Козак Н. Б.

Львів-2024

Анотація

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

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

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

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

Зміст

Завдання до курсового проекту

Варіант 3

Завдання на курсовий проект

1. Цільова мова транслятора – мова програмування С.
2. Для отримання виконавчого файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio або будь-яким іншим.
3. Мова розробки транслятора: С++.
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 С;*
   - *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 .b03

Опис вхідної мови програмування:

- Тип даних: Int16
- Блок тіла програми: Program <name>; Var…; Begin End
- Оператор вводу: Get ()
- Оператор виводу: Put ()
- Оператори: If Else (C)
  Goto (C)
  For-To-Do (Паскаль)
  For-DownTo-Do (Паскаль)
  While (Бейсік)
  Repeat-Until (Паскаль)
- Регістр ключових слів: Up-Low перший символ Up
- Регістр ідентифікаторів: Up6
- Операції арифметичні: +, -, Mul, Div, Mod
- Операції порівняння: Eg, Ne, >>, <<
- Операції логічні: !, And, Or
- Коментар: !!... !!
- Ідентифікатори змінних, числові константи
- Оператор присвоєння: ==>

Вступ

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

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

У відміну 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, такій як С. Наведене визначення застосовне до різноманітних транслюючих 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 С. 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 С. 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 = "Program", {"Var", variable_declaration, ";"}, "Begin", {statement, ";"}, "End";

variable_declaration = "Int16", variable_list;

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

identifier = up, up, up, up, up, up;

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 = "Get", identifier;

output_statement = "Put", arithmetic_expression;

arithmetic_expression = low_priority_expression {low_priority_operator,

low_priority_expression};

low_priority_operator = "+" | "-";

low_priority_expression = middle_priority_expression {middle_priority_operator,

middle_priority_expression};

middle_priority_operator = "Mul" | "Div" | "Mod";

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 = "Or";

and_expression = comparison {and_operator, and_expression};

and_operator = "And";

comparison = comparison_expression | [not_operator] "(", logical_expression, ")";

not_operator = "!";

comparison_expression = arithmetic_expression comparison_operator

arithmetic_expression;

comparison_operator = "Eg" | "Ne" | "<<" | ">>";

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}, "End",

"While”;

repeat_until_statement = "Repeat", {statement, ";"}, "Until", "(",logical_expression, ")";

compoundStatement = "Begin", {statement, ";"}, "End";

2.2. Опис термінальних символів та ключових слів

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

Термінальний символ або ключове слово	Значення
Program	Початок програми
Begin	Початок тексту програми
Var	Початок блоку опису змінних
End	Кінець розділу операторів
Get	Оператор вводу змінних
Put	Оператор виводу (змінних або рядкових констант)
==>	Оператор присвоєння
If	Оператор умови
Else	Оператор умови
Goto	Оператор переходу
Label	Мітка переходу
For	Оператор циклу
To	Інкремент циклу
DownTo	Декремент циклу
Do	Початок тіла циклу
While	Оператор циклу
Continue	Оператор циклу
Exit	Оператор циклу
Repeat	Початок тіла циклу
Until	Оператор циклу
+	Оператор додавання
-	Оператор віднімання
Mul	Оператор множення

Div	Оператор ділення
Mod	Оператор знаходження залишку від ділення
Eg	Оператор перевірки на рівність
Ne	Оператор перевірки на нерівність
<<	Оператор перевірки чи менше
>>	Оператор перевірки чи більше
!	Оператор логічного заперечення
And	Оператор кон’юнкції
Or	Оператор диз’юнкції
Int16	16-ти 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 символі та ключових слів

Токен	**Значення**
Program	Program
Start	Begin
Vars	Var
End	End
VarType	Int16
Read	Get
Write	Put
Assignment	==>
If	If
Else	Else
Goto	Goto
Colon	:
Label	
For	For
To	To
DownTo	Downto
Do	Do
While	While
Continue	Continue
Exit	Exit
Repeat	Repeat
Until	Until
Addition	+
Subtraction	-

Multiplication	Mul
Division	Div
Mod	Mod
Equal	Eg
NotEqual	Ne
Less	<<
Greate	>>
Not	!
And	And
Or	Or
Identifier	
Number	
Unknown	
Comma	,
Semicolon	;
LBraket	(
RBraket	)
LComment	!!
RComment	!!
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-Program, 1-Var, 2-Begin, 3-End, 4-Get, 5-Put, 6-Int16, 7-If, 8-Else, 9-Goto, 10-For, 11-To, 12-Do, 13-Downto, 14-While, 15-Repeat, 16-Until, 17-Exit)
- Ідентифікатори (18-максимум 6 великих літер)
- Числові константи (19-ціле число без знаку)
- Оператор присвоєння (20- ==>)
- Знаки операції (21- +, 22- -, 23- Mul, 24- Div, 25- Mod, 26- >>, 27- <<, 28- Eg, 29- Ne, 30- !, 31- And, 32- Or)
- 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.

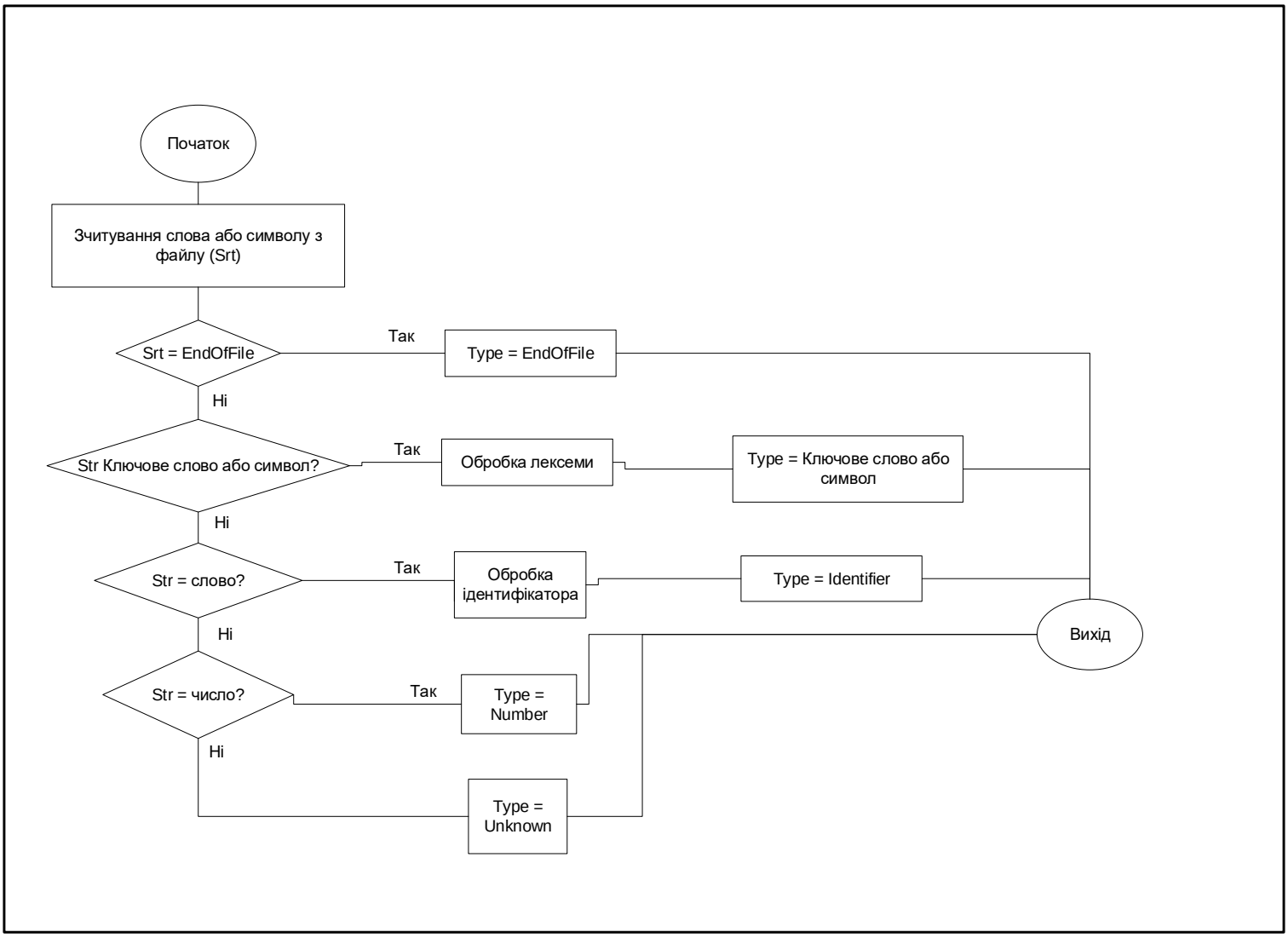


Рис. 3.1 Блок-схема роботи лексичного аналізатора

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

При виділенні лексеми вона розпізнається та записується у таблиця 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     // видалення коментаря
};
```

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-граматик, 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. Розробка алгоритму роботи синтаксичного і семантичного аналізатора

Одним з найбільш простих і найбільш популярних методів низхідного синтаксичного аналізу є метод рекурсивного спуску (recursive descent method).

Метод заснований на тому, що в склад синтаксичного аналізатора входить множина рекурсивних процедур граматичного розбору, по одній для кожного правила граматики.

Визначимо назви процедур, що відповідають нетерміналам граматики таким чином :

```
// набір функцій для рекурсивного спуску
// на кожне правило - окрема функція
void program(FILE* errFile); // topRule = "Program", identifier, ";",
varsBlok, ";", "Begin”, operators, "End"

void variable_declaration(FILE* errFile); // varsBlok = "Var", "Int16",
identifier, [{ commaAndIdentifier }]

void variable_list(FILE* errFile); // identifier = up_letter, up_letter,
up_letter, up_letter, up_letter, up_letter; commaAndIdentifier = ",",
identifier

void program_body(FILE* errFile); // codeBlok = "Begin", operators, "End"

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_not_zero
[{digit_not_zero}]

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

void output(FILE* errFile); // write = "Put", "(", 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); // while = "While", "(", equation,
")", "Begin", operators | whileContinue | whileExit, "End", "While"

void repeat_until(FILE* errFile); // repeatUntil = "Repeat", operators,
"Until", "(", equation, ")"

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

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

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

void compound_statement(FILE* errFile); // codeBlok = "Begin", operators,
"End"
```

3.4.2. Опис програми реалізації синтаксичного та семантичного аналізатора

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

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

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

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

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

Основні етапи роботи аналізатора:

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

Основні функції

- **program()**: Аналізує основну структуру програми.
- **variable_declaration()**: Аналізує оголошення змінних.
- **variable_list()**: Аналізує список змінних.
- **program_body()**: Аналізує тіло програми.
- **statement()**: Визначає тип оператора та викликає відповідну функцію для його аналізу.
- **assignment()**: Аналізує оператор присвоєння.
- **arithmetic_expression()**: Аналізує арифметичний вираз.
- **term()**: Аналізує доданок у виразі.
- **factor()**: Аналізує множник у виразі.
- **input()**: Аналізує оператор вводу.
- **output()**: Аналізує оператор виводу.
- **conditional()**: Аналізує умовний оператор.
- **goto_statement()**: Аналізує оператор переходу.
- **label_statement()**: Аналізує мітку.
- **for_to_do()**: Аналізує цикл for з інкрементом.
- **for_downto_do()**: Аналізує цикл for з декрементом.
- **while_statement()**: Аналізує цикл while.
- **repeat_until()**: Аналізує цикл repeat until.
- **logical_expression()**: Аналізує логічний вираз.
- **and_expression()**: Аналізує логічний вираз з операцією AND.
- **comparison()**: Аналізує операції порівняння.
- **compound_statement()**: Аналізує складений оператор.

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

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);
```

3.4.3. Розробка граф-схеми алгоритму

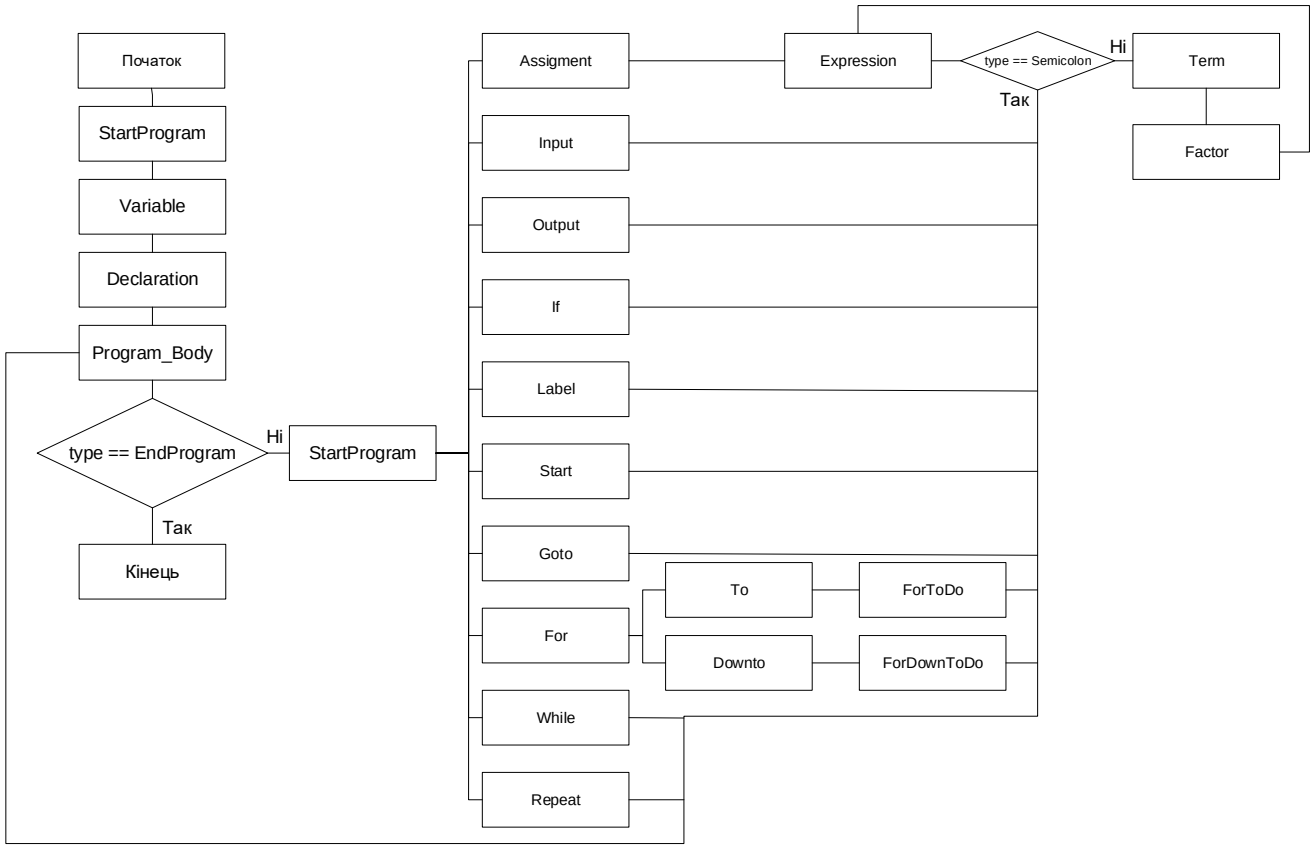


Рис. 3.2 Граф-схема роботи синтаксичного аналізатора

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

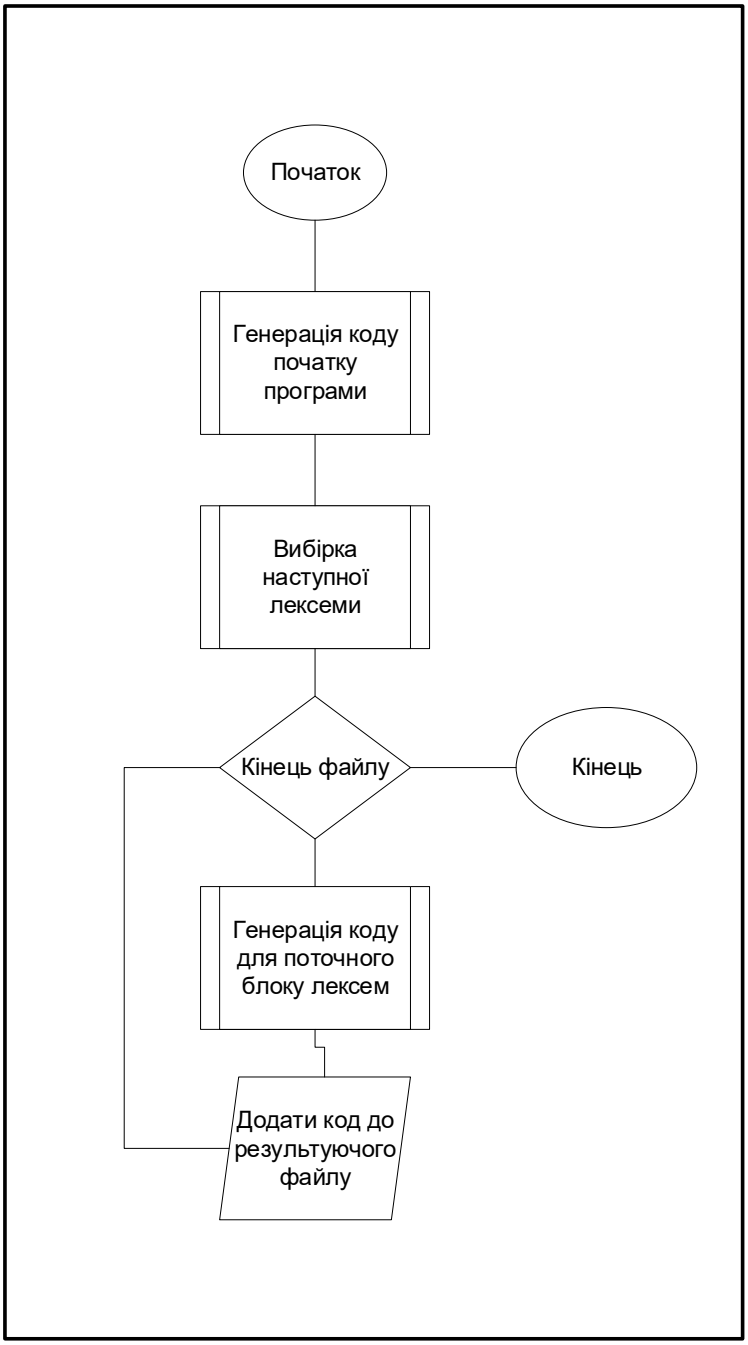


Рис. 3.3 Блок схема генератора коду

3.5.2. Опис програми реалізації генератора коду

У компілятора, реалізованого в даному курсовому проекті, вихідна мова - програма на мові С. 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 С, 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 С. Для цього з таблиці лексем вибирається ім’я змінної (типи змінних відповідають типам у С, 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 С, із врахуванням пріоритетів операцій.

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

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

4. Опис програми

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

```
// 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,
    LBraket,
    RBraket,
    Semicolon,
    Colon,
    Comma,
    Unknown
};


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

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 анадізувати програми великого обсягу.

4.1. Опис інтерфейсу та інструкція користувачеві

Вхідним файлом для даної програми є звичайний текстовий файл з розширенням b03. 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_b03.exe <ім’я програми>.b03"

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

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

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

5. Відлагодження та тестування програми

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

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

5.1. Виявлення лексичних та синтаксичних помилок

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

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

Приклад виявлення:

Текст програми з помилками

```
!!Prog1!!
Program prog1;
Var Int16 AAAAAA,BBB BBB,XXXXXX,YYYYYY;
Begin
Get AAAAAA
Get BBBBBB
Put AAAAAA + BBBBBB
Put AAAAAA - BBBBBB
Put AAAAAA Mul BBBBBB
Put AAAAAA Div BBBBBB
Put AAAAAA Mod BBBBBB

(AAAAAA - BBBBBB) Mul 10 + (AAAAAA + BBBBBB) Div 10 ==> XXXXXX
XXXXXX + (XXXXXX Mod 10) ==> YYYYYY
Put XXXXXX
Put YYYYYY
End
```

Текст файлу з повідомленнями про помилки

```
Lexical Error: line 3, lexem BBB is Unknown
Lexical Error: line 3, lexem BBB is Unknown

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

Syntax error: type Unknown
Expected Type: Identifier
```

5.2. Виявлення семантичних помилок

Суттю виявлення семантичних помилок є перевірка числових констант на відповідність типу Int16, тобто знаковому цілому числу з відповідним діапазоном значень і перевірку на коректність використання змінних Int16 у цілочисельних і логічних виразах.

5.3. Загальна перевірка коректності роботи транслятора

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

Текст коректної програми

```
!!Prog1!!
Program prog1;
Var Int16 AAAAAA,BBBBBB,XXXXXX,YYYYYY;
Begin
Get AAAAAA
Get BBBBBB
Put AAAAAA + BBBBBB
Put AAAAAA - BBBBBB
Put AAAAAA Mul BBBBBB
Put AAAAAA Div BBBBBB
Put AAAAAA Mod BBBBBB

(AAAAAA - BBBBBB) Mul 10 + (AAAAAA + BBBBBB) Div 10 ==> XXXXXX
XXXXXX + (XXXXXX Mod 10) ==> YYYYYY
Put XXXXXX
Put YYYYYY
End
```

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

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

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

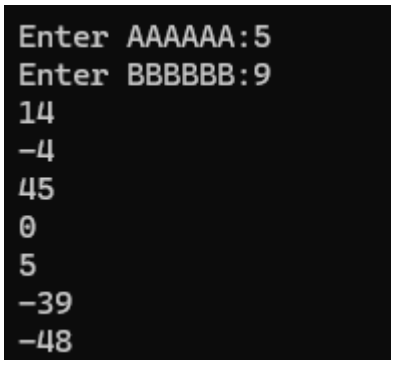


Рис. 5.1 Результат виконання коректної програми

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

5.4. Тестова програма №1

Текст програми

```
!!Prog1!!
Program prog1;
Var Int16 AAAAAA,BBBBBB,XXXXXX,YYYYYY;
Begin
Get AAAAAA
Get BBBBBB
Put AAAAAA + BBBBBB
Put AAAAAA - BBBBBB
Put AAAAAA Mul BBBBBB
Put AAAAAA Div BBBBBB
Put AAAAAA Mod BBBBBB

(AAAAAA - BBBBBB) Mul 10 + (AAAAAA + BBBBBB) Div 10 ==> XXXXXX
XXXXXX + (XXXXXX Mod 10) ==> YYYYYY
Put XXXXXX
Put YYYYYY
End
```

Результат виконання

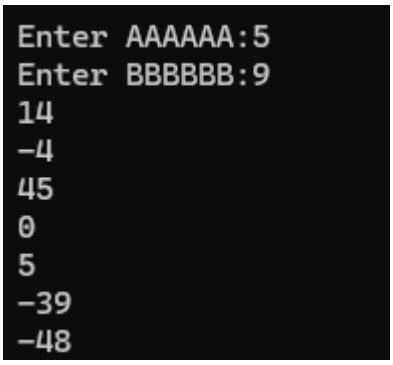


Рис. 5.2 Результат виконання тестової програми №1

5.5. Тестова програма №2

Текст програми

```
!!Prog2!!
Program PROGRA;
Var Int16 AAAAAA,BBBBBB,CCCCCC;
Begin
Get AAAAAA
Get BBBBBB
Get CCCCCC
If(AAAAAA >> BBBBBB)
        If(AAAAAA >> CCCCCC)
                Goto ABIGER
        Else
                Put CCCCCC
                Goto OUTOFI
                ABIGER :
                Put AAAAAA
                Goto OUTOFI
If(BBBBBB << CCCCCC)
        Put CCCCCC
Else
        Put BBBBBB
OUTOFI :
If((AAAAAA Eg BBBBBB) And (AAAAAA Eg CCCCCC) And (BBBBBB Eg CCCCCC))
        Put 1
Else
        Put 0
If((AAAAAA << 0) Or (BBBBBB << 0) Or (CCCCCC << 0))
        Put -1
Else
        Put 0
If(!(AAAAAA << (BBBBBB + CCCCCC)))
        Put(10)
Else
        Put(0)
End
```

Результат виконання

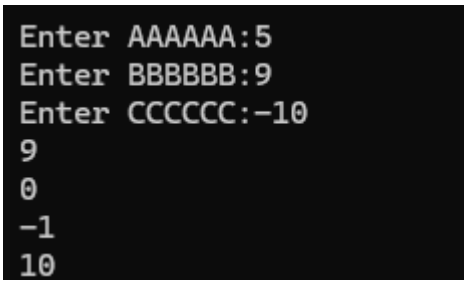


Рис. 5.3 Результат виконання тестової програми №2

5.6. Тестова програма №3

Текст програми

```
Program PROGRA;
Var Int16 AAAAAA, BBBBBB, XXXXXX, IIIIII, JJJJJJ;
Begin
  Get AAAAAA
  Get BBBBBB

For AAAAAA ==> XXXXXX To BBBBBB Do
    Put XXXXXX Mul XXXXXX
0 ==> XXXXXX
For 1 ==> IIIIII To AAAAAA Do
    For 1 ==> JJJJJJ To BBBBBB Do
       XXXXXX + 1 ==> XXXXXX

Put XXXXXX
End
```

Результат виконання

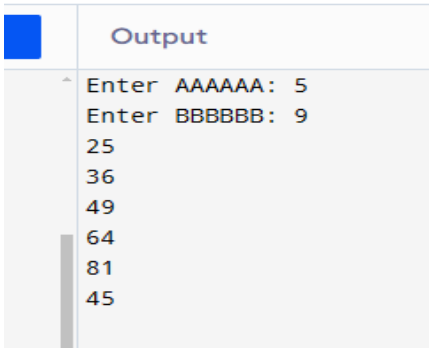


Рис. 5.4 Результат виконання тестової програми №3

Висновки

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування b03, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створено компілятор мови програмування b03, 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 b03. 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 b03 були успішно виявлені і відповідно оброблені.

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

Список використаної літератури

1. C Programming Language Tutorial - GeeksforGeeks
   URL: C Programming Language Tutorial - GeeksforGeeks

2. Error Handling in Compiler Design
   URL: Error Handling in Compiler Design - GeeksforGeeks

3. Symbol Table in Compiler
   URL: Symbol Table in Compiler - GeeksforGeeks

4. Вікіпедія
   URL: Wikipedia

5. Stack Overflow
   URL: Stack Overflow - Where Developers Learn, Share, & Build Careers

Додатки

Додаток А (Тестові програми)

Тестова програма «Лінійний алгоритм»

```
!!Prog1!!
Program PROGRA;
Var Int16 AAAAAA , BBBBBB , XXXXXX , YYYYYY ;
Begin
Get AAAAAA
Get BBBBBB
Put AAAAAA + BBBBBB
Put AAAAAA - BBBBBB
Put AAAAAA Mul BBBBBB
Put AAAAAA Div BBBBBB
Put AAAAAA Mod BBBBBB
(AAAAAA - BBBBBB) Mul 10 + (AAAAAA + BBBBBB) Div 10 ==> XXXXXX
XXXXXX + (XXXXXX Mod 10) ==> YYYYYY
Put XXXXXX
Put YYYYYY
End
```

Тестова програма «Алгоритм з розгалуженням»

```
!!Prog2!!
Program PROGRA;
Var Int16 AAAAAA,BBBBBB,CCCCCC;
Begin
Get AAAAAA
Get BBBBBB
Get CCCCCC
If(AAAAAA >> BBBBBB)
        If(AAAAAA >> CCCCCC)
                Goto ABIGER
        Else
                Put CCCCCC
                Goto OUTOFI
                ABIGER :
                Put AAAAAA
                Goto OUTOFI
If(BBBBBB << CCCCCC)
        Put CCCCCC
Else
        Put BBBBBB
OUTOFI :
If((AAAAAA Eg BBBBBB) And (AAAAAA Eg CCCCCC) And (BBBBBB Eg CCCCCC))
        Put 1
Else
        Put 0
If((AAAAAA << 0) Or (BBBBBB << 0) Or (CCCCCC << 0))
        Put -1
Else
        Put 0
If(!(AAAAAA << (BBBBBB + CCCCCC)))
        Put(10)
Else
```

```
        Put(0)
End
```

Тестова програма «Циклічний алгоритм»

```
Program PROGRA;
Var Int16 AAAAAA, BBBBBB, XXXXXX, IIIIII, JJJJJJ;
Begin
  Get AAAAAA
  Get BBBBBB

For AAAAAA ==> XXXXXX To BBBBBB Do
   Put XXXXXX Mul XXXXXX
0 ==> XXXXXX
For 1 ==> IIIIII To AAAAAA Do
   For 1 ==> JJJJJJ To BBBBBB Do
      XXXXXX + 1 ==> XXXXXX

Put XXXXXX
End
```

Додаток Б (Таблиці лексем для тесових програм)

Тестова програма «Лінійний алгоритм»

TOKEN TABLE				
line number	token	value	token code	type of token
2	Program	0	0	Program
2	PROGRA	0	20	Identifier
2	;	0	38	Semicolon
3	Begin	0	0	Program
4	Var	0	2	Variable
4	Int16	0	3	Type
4	AAAAAA	0	20	Identifier
4	,	0	40	Comma
4	BBBBBB	0	20	Identifier
4	,	0	40	Comma
4	XXXXXX	0	20	Identifier
4	,	0	40	Comma
4	YYYYYY	0	20	Identifier
4	;	0	38	Semicolon

5	Get	0	5	Input
5	AAAAAA	0	20	Identifier
6	Get	0	5	Input
6	BBBBBB	0	20	Identifier
7	Put	0	6	Output
7	AAAAAA	0	20	Identifier
7	+	0	24	Add
7	BBBBBB	0	20	Identifier
8	Put	0	6	Output
8	AAAAAA	0	20	Identifier
8	-	0	25	Sub
8	BBBBBB	0	20	Identifier
9	Put	0	6	Output
9	AAAAAA	0	20	Identifier
9	Mul	0	26	Mul
9	BBBBBB	0	20	Identifier
10	Put	0	6	Output
10	AAAAAA	0	20	Identifier
10	Div	0	27	Div
10	BBBBBB	0	20	Identifier
11	Put	0	6	Output
11	AAAAAA	0	20	Identifier
11	Mod	0	28	Mod
11	BBBBBB	0	20	Identifier
12	(	0	36	LBracket
12	AAAAAA	0	20	Identifier
12	-	0	25	Sub
12	BBBBBB	0	20	Identifier
12	)	0	37	RBracket

```
|          12 |             Mul |          0 |         26 | Mul           |
-------------------------------------------------------------------------
|          12 |              10 |         10 |         21 | Number        |
-------------------------------------------------------------------------
|          12 |               + |          0 |         24 | Add           |
-------------------------------------------------------------------------
|          12 |               ( |          0 |         36 | LBracket      |
-------------------------------------------------------------------------
|          12 |          AAAAAA |          0 |         20 | Identifier    |
-------------------------------------------------------------------------
|          12 |               + |          0 |         24 | Add           |
-------------------------------------------------------------------------
|          12 |          BBBBBB |          0 |         20 | Identifier    |
-------------------------------------------------------------------------
|          12 |               ) |          0 |         37 | RBracket      |
-------------------------------------------------------------------------
|          12 |             Div |          0 |         27 | Div           |
-------------------------------------------------------------------------
|          12 |              10 |         10 |         21 | Number        |
-------------------------------------------------------------------------
|          12 |             ==> |          0 |         23 | Assign        |
-------------------------------------------------------------------------
|          12 |          XXXXXX |          0 |         20 | Identifier    |
-------------------------------------------------------------------------
|          13 |          XXXXXX |          0 |         20 | Identifier    |
-------------------------------------------------------------------------
|          13 |               + |          0 |         24 | Add           |
-------------------------------------------------------------------------
|          13 |               ( |          0 |         36 | LBracket      |
-------------------------------------------------------------------------
|          13 |          XXXXXX |          0 |         20 | Identifier    |
-------------------------------------------------------------------------
|          13 |             Mod |          0 |         28 | Mod           |
-------------------------------------------------------------------------
|          13 |              10 |         10 |         21 | Number        |
-------------------------------------------------------------------------
|          13 |               ) |          0 |         37 | RBracket      |
-------------------------------------------------------------------------
|          13 |             ==> |          0 |         23 | Assign        |
-------------------------------------------------------------------------
|          13 |          YYYYYY |          0 |         20 | Identifier    |
-------------------------------------------------------------------------
|          14 |             Put |          0 |          6 | Output        |
-------------------------------------------------------------------------
|          14 |          XXXXXX |          0 |         20 | Identifier    |
-------------------------------------------------------------------------
|          15 |             Put |          0 |          6 | Output        |
-------------------------------------------------------------------------
|          15 |          YYYYYY |          0 |         20 | Identifier    |
-------------------------------------------------------------------------
|          16 |             End |          0 |          4 | EndBlock      |
-------------------------------------------------------------------------
```

Тестова програма «Алгоритм з розгалуженням»

```
-------------------------------------------------------------------------
| TOKEN TABLE                                                           |
-------------------------------------------------------------------------
```

line number	token	value	token code	type of token
2	Program	0	0	Program
2	PROGRA	0	20	Identifier
2	;	0	38	Semicolon
3	Var	0	2	Variable
3	Int16	0	3	Type
3	AAAAAA	0	20	Identifier
3	,	0	40	Comma
3	BBBBBB	0	20	Identifier
3	,	0	40	Comma
3	CCCCCC	0	20	Identifier
3	;	0	38	Semicolon
4	Begin	0	0	Program
5	Get	0	5	Input
5	AAAAAA	0	20	Identifier
6	Get	0	5	Input
6	BBBBBB	0	20	Identifier
7	Get	0	5	Input
7	CCCCCC	0	20	Identifier
8	If	0	7	If
8	(	0	36	LBracket
8	AAAAAA	0	20	Identifier
8	>>	0	32	Less
8	BBBBBB	0	20	Identifier
8	)	0	37	RBracket
9	If	0	7	If
9	(	0	36	LBracket
9	AAAAAA	0	20	Identifier
9	>>	0	32	Less

9	CCCCCC	0	20	Identifier
9	)	0	37	RBracket
10	Goto	0	9	Goto
10	ABIGER	0	20	Identifier
11	Else	0	8	Else
12	Put	0	6	Output
12	CCCCCC	0	20	Identifier
13	Goto	0	9	Goto
13	OUTOFI	0	20	Identifier
14	ABIGER	0	20	Identifier
14	:	0	39	Colon
15	Put	0	6	Output
15	AAAAAA	0	20	Identifier
16	Goto	0	9	Goto
16	OUTOFI	0	20	Identifier
17	If	0	7	If
17	(	0	36	LBracket
17	BBBBBB	0	20	Identifier
17	<<	0	31	Greate
17	CCCCCC	0	20	Identifier
17	)	0	37	RBracket
18	Put	0	6	Output
18	CCCCCC	0	20	Identifier
19	Else	0	8	Else
20	Put	0	6	Output
20	BBBBBB	0	20	Identifier
21	OUTOFI	0	20	Identifier
21	:	0	39	Colon
22	If	0	7	If

22	(	0	36	LBracket
22	(	0	36	LBracket
22	AAAAAA	0	20	Identifier
22	Eg	0	29	Equality
22	BBBBBB	0	20	Identifier
22	)	0	37	RBracket
22	And	0	34	And
22	(	0	36	LBracket
22	AAAAAA	0	20	Identifier
22	Eg	0	29	Equality
22	CCCCCC	0	20	Identifier
22	)	0	37	RBracket
22	And	0	34	And
22	(	0	36	LBracket
22	BBBBBB	0	20	Identifier
22	Eg	0	29	Equality
22	CCCCCC	0	20	Identifier
22	)	0	37	RBracket
22	)	0	37	RBracket
23	Put	0	6	Output
23	1	1	21	Number
24	Else	0	8	Else
25	Put	0	6	Output
25	0	0	21	Number
26	If	0	7	If
26	(	0	36	LBracket
26	(	0	36	LBracket
26	AAAAAA	0	20	Identifier
26	<<	0	31	Greate

26	0	0	21	Number
26	)	0	37	RBracket
26	Or	0	35	Or
26	(	0	36	LBracket
26	BBBBBB	0	20	Identifier
26	<<	0	31	Greate
26	0	0	21	Number
26	)	0	37	RBracket
26	Or	0	35	Or
26	(	0	36	LBracket
26	CCCCCC	0	20	Identifier
26	<<	0	31	Greate
26	0	0	21	Number
26	)	0	37	RBracket
26	)	0	37	RBracket
27	Put	0	6	Output
27	-1	-1	21	Number
28	Else	0	8	Else
29	Put	0	6	Output
29	0	0	21	Number
30	If	0	7	If
30	(	0	36	LBracket
30	(	0	36	LBracket
30	AAAAAA	0	20	Identifier
30	<<	0	31	Greate
30	(	0	36	LBracket
30	BBBBBB	0	20	Identifier
30	+	0	24	Add
30	CCCCCC	0	20	Identifier

30	)	0	37	RBracket
30	)	0	37	RBracket
30	)	0	37	RBracket
31	Put	0	6	Output
31	(	0	36	LBracket
31	10	10	21	Number
31	)	0	37	RBracket
32	Else	0	8	Else
33	Put	0	6	Output
33	(	0	36	LBracket
33	0	0	21	Number
33	)	0	37	RBracket
34	End	0	4	EndBlock

Тестова програма «Циклічний алгоритм»

TOKEN TABLE

line number	token	value	token code	type of token
1	Program	0	0	Program
1	PROGRA	0	20	Identifier
1	;	0	38	Semicolon
2	Var	0	2	Variable
2	Int16	0	3	Type
2	AAAAAA	0	20	Identifier
2	,	0	40	Comma
2	BBBBBB	0	20	Identifier
2	,	0	40	Comma
2	XXXXXX	0	20	Identifier

2	,	0	40	Comma
2	IIIIII	0	20	Identifier
2	,	0	40	Comma
2	JJJJJJ	0	20	Identifier
2	;	0	38	Semicolon
3	Begin	0	0	Program
4	Get	0	5	Input
4	AAAAAA	0	20	Identifier
5	Get	0	5	Input
5	BBBBBB	0	20	Identifier
7	For	0	10	For
7	AAAAAA	0	20	Identifier
7	==>	0	23	Assign
7	XXXXXX	0	20	Identifier
7	To	0	11	To
7	BBBBBB	0	20	Identifier
7	Do	0	13	Do
8	Put	0	6	Output
8	XXXXXX	0	20	Identifier
8	Mul	0	26	Mul
8	XXXXXX	0	20	Identifier
9	0	0	21	Number
9	==>	0	23	Assign
9	XXXXXX	0	20	Identifier
10	For	0	10	For
10	1	1	21	Number
10	==>	0	23	Assign
10	IIIIII	0	20	Identifier

10	To	0	11	To
10	AAAAAA	0	20	Identifier
10	Do	0	13	Do
11	For	0	10	For
11	1	1	21	Number
11	==>	0	23	Assign
11	JJJJJJ	0	20	Identifier
11	To	0	11	To
11	BBBBBB	0	20	Identifier
11	Do	0	13	Do
12	XXXXXX	0	20	Identifier
12	+	0	24	Add
12	1	1	21	Number
12	==>	0	23	Assign
12	XXXXXX	0	20	Identifier
14	Put	0	6	Output
14	XXXXXX	0	20	Identifier
15	End	0	4	EndBlock

Додаток В (Код на мові С)

Тестова програма «Лінійний алгоритм»

```
#include <stdio.h>
int main() {
int YYYYYY;
int XXXXXX;
int BBBBBB;
int AAAAAA;

printf("Enter AAAAAA: ");
scanf("%d", &AAAAAA);
printf("Enter BBBBBB: ");
scanf("%d", &BBBBBB);
printf("%d\n", (AAAAAA + BBBBBB));
printf("%d\n", (AAAAAA - BBBBBB));
printf("%d\n", (AAAAAA * BBBBBB));
printf("%d\n", (AAAAAA / BBBBBB));
printf("%d\n", (AAAAAA % BBBBBB));
XXXXXX = (((AAAAAA - BBBBBB) * 10) + ((AAAAAA + BBBBBB) / 10));
YYYYYY = (XXXXXX + (XXXXXX % 10));
printf("%d\n", XXXXXX);
printf("%d\n", YYYYYY);
return 0;
}
```

Тестова програма «Алгоритм з розгалуженням»

```
#include <stdio.h>
int main() {
int CCCCCC;
int BBBBBB;
int AAAAAA;

printf("Enter AAAAAA: ");
scanf("%d", &AAAAAA);
printf("Enter BBBBBB: ");
scanf("%d", &BBBBBB);
printf("Enter CCCCCC: ");
scanf("%d", &CCCCCC);
if ((AAAAAA < BBBBBB)) if ((AAAAAA < CCCCCC)) goto ABIGER;
else printf("%d\n", CCCCCC);
goto OUTOFI;
ABIGER:
printf("%d\n", AAAAAA);
goto OUTOFI;
if ((BBBBBB > CCCCCC)) printf("%d\n", CCCCCC);
else printf("%d\n", BBBBBB);
OUTOFI:
if (((AAAAAA == BBBBBB) && ((AAAAAA == CCCCCC) && (BBBBBB == CCCCCC))))
printf("%d\n", 1);
else printf("%d\n", 0);
if (((AAAAAA > 0) || ((BBBBBB > 0) || (CCCCCC > 0)))) printf("%d\n", -1);
```

```
else printf("%d\n", 0);
if ((AAAAAA > (BBBBBB + CCCCCC))) printf("%d\n", 10);
else printf("%d\n", 0);
return 0;
}
```

Тестова програма «Циклічний алгоритм»

```
#include <stdio.h>
int main() {
int JJJJJJ;
int IIIIII;
int XXXXXX;
int BBBBBB;
int AAAAAA;

printf("Enter AAAAAA: ");
scanf("%d", &AAAAAA);
printf("Enter BBBBBB: ");
scanf("%d", &BBBBBB);
for(
XXXXXX = AAAAAA;
XXXXXX <= BBBBBB;
++XXXXXX
) printf("%d\n", (XXXXXX * XXXXXX));
XXXXXX = 0;
for(
IIIIII = 1;
IIIIII <= AAAAAA;
++IIIIII
) for(
JJJJJJ = 1;
JJJJJJ <= BBBBBB;
++JJJJJJ
) XXXXXX = (XXXXXX + 1);
printf("%d\n", XXXXXX);
return 0;
}
```

Додаток Г (Абстрактне синтаксичне дерево для тестових прикладів)

Тестова програма «Лінійний алгоритм»

```
|-- Program(0)
|   |-- var(2)
|   |   |-- YYYYYY(1)
|   |   |-- var(2)
|   |   |   |-- XXXXXX(1)
|   |   |   |-- var(2)
|   |   |   |   |-- BBBBBB(1)
|   |   |   |   |-- var(2)
|   |   |   |   |   |-- AAAAAA(1)
|   |-- statement(3)
|   |   |-- statement(3)
|   |   |   |-- statement(3)
|   |   |   |   |-- statement(3)
|   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- Sub(7)
|   |   |   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |-- Mul(8)
|   |   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |-- Div(9)
|   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |-- Mod(10)
```

```
|   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |-- XXXXXX(1)
|   |   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |   |-- Mul(8)
|   |   |   |   |   |   |   |   |-- Sub(7)
|   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |   |-- 10(11)
|   |   |   |   |   |   |   |-- Div(9)
|   |   |   |   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |   |-- 10(11)
|   |   |   |   |-- assign(12)
|   |   |   |   |   |-- YYYYYY(1)
|   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |-- XXXXXX(1)
|   |   |   |   |   |   |-- Mod(10)
|   |   |   |   |   |   |   |-- XXXXXX(1)
|   |   |   |   |   |   |   |-- 10(11)
|   |   |   |-- output(5)
|   |   |   |   |-- XXXXXX(1)
|   |   |-- output(5)
|   |   |   |-- YYYYYY(1)
```

Тестова програма «Алгоритм з розгалуженням»

```
|-- Program(0)
|   |-- var(2)
|   |   |-- CCCCCC(1)
|   |   |-- var(2)
|   |   |   |-- BBBBBB(1)
|   |   |   |-- var(2)
|   |   |   |   |-- AAAAAA(1)
|   |-- statement(3)
|   |   |-- statement(3)
|   |   |   |-- statement(3)
|   |   |   |   |-- statement(3)
|   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |   |-- statement(3)
```

```
|   |   |   |   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |   |   |   |   |   |   |-- CCCCCC(1)
|   |   |   |   |   |   |   |   |   |   |   |-- if(13)
|   |   |   |   |   |   |   |   |   |   |   |   |-- Less(21)
|   |   |   |   |   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |   |   |   |   |   |-- if(13)
|   |   |   |   |   |   |   |   |   |   |   |   |   |-- Less(21)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |-- CCCCCC(1)
|   |   |   |   |   |   |   |   |   |   |   |   |   |-- else(14)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |-- goto(22)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |-- ABIGER(1)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |-- CCCCCC(1)
|   |   |   |   |   |   |   |   |   |   |   |-- goto(22)
|   |   |   |   |   |   |   |   |   |   |   |   |-- OUTOFI(1)
|   |   |   |   |   |   |   |   |   |   |-- label(23)
|   |   |   |   |   |   |   |   |   |   |   |-- ABIGER(1)
|   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |-- goto(22)
|   |   |   |   |   |   |   |   |   |-- OUTOFI(1)
|   |   |   |   |   |   |   |-- if(13)
|   |   |   |   |   |   |   |   |-- Greate(20)
|   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |   |   |-- CCCCCC(1)
|   |   |   |   |   |   |   |   |-- else(14)
|   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- CCCCCC(1)
|   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |-- label(23)
|   |   |   |   |   |   |-- OUTOFI(1)
|   |   |   |   |-- if(13)
|   |   |   |   |   |-- and(16)
|   |   |   |   |   |   |-- Equality(18)
|   |   |   |   |   |   |   |-- AAAAAA(1)
```

```
|   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |-- and(16)
|   |   |   |   |   |   |   |-- Equality(18)
|   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |   |-- CCCCCC(1)
|   |   |   |   |   |   |   |-- Equality(18)
|   |   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |   |-- CCCCCC(1)
|   |   |   |   |   |-- else(14)
|   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |-- 1(11)
|   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |-- 0(11)
|   |   |   |-- if(13)
|   |   |   |   |-- or(15)
|   |   |   |   |   |-- Greate(20)
|   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |-- 0(11)
|   |   |   |   |   |-- or(15)
|   |   |   |   |   |   |-- Greate(20)
|   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |   |-- 0(11)
|   |   |   |   |   |   |-- Greate(20)
|   |   |   |   |   |   |   |-- CCCCCC(1)
|   |   |   |   |   |   |   |-- 0(11)
|   |   |   |   |-- else(14)
|   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |-- -1(11)
|   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |-- 0(11)
|   |   |-- if(13)
|   |   |   |-- Greate(20)
|   |   |   |   |-- AAAAAA(1)
|   |   |   |   |-- Add(6)
|   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |-- CCCCCC(1)
|   |   |   |-- else(14)
|   |   |   |   |-- output(5)
|   |   |   |   |   |-- 10(11)
|   |   |   |   |-- output(5)
|   |   |   |   |   |-- 0(11)
```

Тестова програма «Циклічний алгоритм»

```
|-- Program(0)
|   |-- var(2)
|   |   |-- JJJJJ(1)
```

```
|   |   |-- var(2)
|   |   |   |-- IIIIII(1)
|   |   |   |-- var(2)
|   |   |   |   |-- XXXXXX(1)
|   |   |   |   |-- var(2)
|   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |-- var(2)
|   |   |   |   |   |   |-- AAAAAA(1)
|   |-- statement(3)
|   |   |-- statement(3)
|   |   |   |-- statement(3)
|   |   |   |   |-- statement(3)
|   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |-- for(24)
|   |   |   |   |   |   |-- to(25)
|   |   |   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |   |   |-- XXXXXX(1)
|   |   |   |   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |-- Mul(8)
|   |   |   |   |   |   |   |   |-- XXXXXX(1)
|   |   |   |   |   |   |   |   |-- XXXXXX(1)
|   |   |   |   |-- assign(12)
|   |   |   |   |   |-- XXXXXX(1)
|   |   |   |   |   |-- 0(11)
|   |   |   |-- for(24)
|   |   |   |   |-- to(25)
|   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |-- IIIIII(1)
|   |   |   |   |   |   |-- 1(11)
|   |   |   |   |   |-- AAAAAA(1)
|   |   |   |   |-- for(24)
|   |   |   |   |   |-- to(25)
|   |   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |   |-- JJJJJ(1)
|   |   |   |   |   |   |   |-- 1(11)
|   |   |   |   |   |   |-- BBBBBB(1)
|   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |-- XXXXXX(1)
|   |   |   |   |   |   |-- Add(6)
```

```
|   |   |   |   |   |   |   |-- XXXXXX(1)
|   |   |   |   |   |   |   |-- 1(11)
|   |   |-- output(5)
|   |   |   |-- XXXXXX(1)
```

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).

```
#include "Ast.hpp"

extern struct Token* TokenTable;
extern int pos;

namespace AST {
   void deleteNode(struct astNode* node) {
      if (node == nullptr) return;
      deleteNode(node->left);
      deleteNode(node->right);
      free(node);
   }

   struct astNode* createNode(enum TypeOfNode type, const char* name, struct astNode* left, struct astNode*
right) {
      struct astNode* node = (struct astNode*)malloc(sizeof(struct astNode));
      node->type = type;
      strcpy_s(node->name, name);
      node->left = left;
      node->right = right;
      return node;
   }

   void printAST(struct astNode* node, int level) {
      if (node == nullptr)
         return;

      for (int i = 0; i < level; i++)
         printf("|    ");

      printf("|-- %s(%d)", node->name, node->type);
      printf("\n");

      if (node->left || node->right)
      {
         printAST(node->left, level + 1);
         printAST(node->right, level + 1);
      }
   }
```

```
void fPrintAST(FILE* outFile, struct astNode* node, int level) {
    if (node == nullptr)
        return;

    for (int i = 0; i < level; i++)
        fprintf(outFile, "|    ");

    fprintf(outFile, "|-- %s(%d)", node->name, node->type);
    fprintf(outFile, "\n");

    if (node->left || node->right)
    {
        fPrintAST(outFile, node->left, level + 1);
        fPrintAST(outFile, node->right, level + 1);
    }
}

void match(enum TypeOfToken expectedType) {
    if (TokenTable[pos].type == expectedType)
        pos++;
    else {
        printf("\nSyntax error in line %d : another type of lexeme was expected (expected: %s | current: %s).\n",
TokenTable[pos].line, lexemeTypeName(expectedType), lexemeTypeName(TokenTable[pos].type));
        exit(1);
    }
}

struct astNode* astParser() {
    pos = 0;
    struct astNode* tree = program();

    printf("AST created.\n");

    return tree;
}

struct astNode* program() {
    match(StartProgram);
    match(Identifier);          // очікує ім'я програми, напр. AAAAAAAA
    match(Semicolon);           // очікує символ ';' після назви програми
    struct astNode* declaration = nullptr;
    if (TokenTable[pos].type == Variable) {
        ++pos;  //  for VARIABLE
        declaration = variableDeclaration();
        match(Semicolon);
    }
    match(StartProgram);
    struct astNode* body = programBody();
    match(EndBlock);
    return createNode(program_node, "Program", declaration, body);
}

struct astNode* variableDeclaration() {
    match(Type);
    return variableList();
}

struct astNode* variableList() {
    match(Identifier);
    struct astNode* id = createNode(id_node, TokenTable[pos - 1].name, nullptr, nullptr);
```

```
    struct astNode* list = list = createNode(var_node, "var", id, nullptr);
    while (TokenTable[pos].type == Comma)
    {
        match(Comma);
        match(Identifier);
        id = createNode(id_node, TokenTable[pos - 1].name, nullptr, nullptr);
        list = createNode(var_node, "var", id, list);
    }
    return list;
}

struct astNode* programBody() {
    if (TokenTable[pos].type != EndBlock) {
        struct astNode* stmt = statement();
        //match(Semicolon);
        struct astNode* body = stmt;
        while (TokenTable[pos].type != EndBlock)
        {
            struct astNode* nextStmt = statement();
            //match(Semicolon);
            body = createNode(statement_node, "statement", body, nextStmt);
        }
        return body;
    }
    return nullptr;
}

struct astNode* statement() {
    switch (TokenTable[pos].type) {
    case Input:     return  inputStatement();
    case Output:    return  outputStatement();
    case If:        return  ifStatement();
    case Goto:      return  gotoStatement();
    case For:       return  forStatement();
    case While:     return  whileStatement();
    case Repeat:    return  repeatStatement();
    case StartBlock:return  compoundStatement();
    default: {
        if (TokenTable[pos + 1].type == Colon)
            return  labelPoint();
        else
            return  assignStatement();
    }
    }
}

struct astNode* inputStatement() {
    match(Input);
    match(Identifier);
    return createNode(input_node, "input", createNode(id_node, TokenTable[pos - 1].name, nullptr, nullptr),
nullptr);
}

struct astNode* outputStatement() {
    match(Output);
    return createNode(output_node, "output", arithmeticExpression(), nullptr);
}

struct astNode* arithmeticExpression() {
    struct astNode* left = lowPriorityExpression();
    if (TokenTable[pos].type == Add || TokenTable[pos].type == Sub) {
```

```
            enum TypeOfToken op = TokenTable[pos].type;
            ++pos;  //  for add or sub
            struct astNode* right = arithmeticExpression();
            return createNode(op == Add ? add_node : sub_node, lexemeTypeName(op), left, right);
        }
        return left;
    }

    struct astNode* lowPriorityExpression() {
        struct astNode* left = middlePriorityExpression();
        if (TokenTable[pos].type == Mul || TokenTable[pos].type == Mod || TokenTable[pos].type == Div) {
            enum TypeOfToken op = TokenTable[pos].type;
            ++pos;  //  for mul or mod or div
            struct astNode* right = lowPriorityExpression();
            return createNode(op == Mul ? mul_node : op == Div ? div_node : mod_node, lexemeTypeName(op), left,
right);
        }
        return left;
    }

    struct astNode* middlePriorityExpression() {
        switch (TokenTable[pos].type) {
        case Identifier:    match(Identifier);  return  createNode(id_node, TokenTable[pos - 1].name, nullptr, nullptr);
        case Number:        match(Number);      return  createNode(number_node, TokenTable[pos - 1].name, nullptr,
nullptr);
        case LBracket: {
            ++pos;  //  for (
            struct astNode* expr = arithmeticExpression();
            match(RBracket);
            return expr;
        }
        default: {
            printf("\nSyntax error in line %d, token number: %d : middle priority operation was expected (current:
%s).\n", TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));
            exit(1);
        }
        }
    }

    struct astNode* assignStatement() {
        struct astNode* right = arithmeticExpression();
        match(Assign);
        match(Identifier);
        return createNode(assign_node, "assign", createNode(id_node, TokenTable[pos - 1].name, nullptr, nullptr),
right);
    }

    struct astNode* ifStatement() {
        match(If);
        match(LBracket);
        struct astNode* expr = logicalExpression();
        match(RBracket);
        struct astNode* stmt = statement();
        if (TokenTable[pos].type == Else) {
            //match(Semicolon);
            ++pos;  //  for ELSE
            struct astNode* elseStmt = statement();
            stmt = createNode(else_node, "else", stmt, elseStmt);
        }
        return createNode(if_node, "if", expr, stmt);
    }
```

```
struct astNode* logicalExpression() {
    struct astNode* left = andExpression();
    if (TokenTable[pos].type == Or) {
        ++pos;  //  for Or
        struct astNode* right = logicalExpression();
        return createNode(or_node, "or", left, right);
    }
    return left;
}

struct astNode* andExpression() {
    struct astNode* left = comparison();
    if (TokenTable[pos].type == And) {
        ++pos;  //  for And
        struct astNode* right = andExpression();
        return createNode(and_node, "and", left, right);
    }
    return left;
}

struct astNode* comparison() {
    struct astNode* comp;
    switch (TokenTable[pos].type) {
    case Not: {
        ++pos;  //  for Not
        match(LBracket);
        comp = createNode(not_node, "not", logicalExpression(), nullptr);
        match(RBracket);
        break;
    }
    case LBracket: {
        ++pos;  //  for (
        comp = logicalExpression();
        match(RBracket);
        break;
    }
    default: {
        comp = comparisonExpression();
    }
    }
    return comp;
}

struct astNode* comparisonExpression() {
    struct astNode* left = arithmeticExpression();
    if (TokenTable[pos].type == Equality ||
        TokenTable[pos].type == NotEquality ||
        TokenTable[pos].type == Greate ||
        TokenTable[pos].type == Less
        ) {
        enum TypeOfToken op = TokenTable[pos].type;
        ++pos;  //  for Equality or NotEquality or Greate or Less
        struct astNode* right = arithmeticExpression();
        return createNode(op == Equality ? eq_node : op == NotEquality ? neq_node : op == Greate ? gr_node :
ls_node, lexemeTypeName(op), left, right);
    }
    else {
        printf("\nSyntax error in line %d : Comparison operator was Expected, %s token gained.\n",
TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));
        exit(1);
```

```
        }
    }

    struct astNode* gotoStatement() {
        match(Goto);
        match(Identifier);
        return createNode(goto_node, "goto", createNode(id_node, TokenTable[pos - 1].name, nullptr, nullptr),
nullptr);
    }

    struct astNode* labelPoint() {
        match(Identifier);
        match(Colon);
        return createNode(label_node, "label", createNode(id_node, TokenTable[pos - 2].name, nullptr, nullptr),
nullptr);
    }

    struct astNode* forStatement() {
        match(For);
        struct astNode* assign = assignStatement();
        switch (TokenTable[pos].type) {
        case To: {
            ++pos;  //  for To
            struct astNode* expr = arithmeticExpression();
            match(Do);
            return createNode(for_node, "for", createNode(to_node, "to", assign, expr), statement());
        }
        case Downto: {
            ++pos;  // for Downto
            struct astNode* expr = arithmeticExpression();
            match(Do);
            return createNode(for_node, "for", createNode(to_node, "downto", assign, expr), statement());
        }
        default: {
            printf("\nSyntax error in line %d : TO | DOWNTO operator was xpected, %s token gained).\n",
TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));
            exit(1);
        }
        }
    }

    struct astNode* whileStatement() {
        match(While);
        struct astNode* expr = logicalExpression();
        struct astNode* body = whileBody();
        match(End);
        return createNode(while_node, "while", expr, body);
    }

    struct astNode* whileBody() {
        if (TokenTable[pos].type != End) {
            struct astNode* stmt = statementInWhile();
            //match(Semicolon);
            struct astNode* body = stmt;
            while (TokenTable[pos].type != End)
            {
                struct astNode* nextStmt = statementInWhile();
                //match(Semicolon);
                body = createNode(statement_node, "statement", body, nextStmt);
            }
            return body;
```

```
        }
        return nullptr;
    }

    struct astNode* statementInWhile() {
        switch (TokenTable[pos].type) {
        case Continue: {
            ++pos;  //  for CONTINUE
            match(While);
            return createNode(continue_node, "continue", nullptr, nullptr);
        }
        case Exit: {
            ++pos;  //  for EXIT
            match(While);
            return createNode(exit_node, "exit", nullptr, nullptr);
        }
        default:    return statement();
        }
    }

    struct astNode* repeatStatement() {
        match(Repeat);
        struct astNode* body = repeatBody();
        match(Until);
        match(LBracket);
        struct astNode* expr = logicalExpression();
        match(RBracket);
        return createNode(repeat_node, "repeat", body, expr);
    }

    struct astNode* repeatBody() {
        if (TokenTable[pos].type != Until) {
            struct astNode* stmt = statement();
            //match(Semicolon);
            struct astNode* body = stmt;
            while (TokenTable[pos].type != Until)
            {
                struct astNode* nextStmt = statement();
                //match(Semicolon);
                body = createNode(statement_node, "statement", body, nextStmt);
            }
            return body;
        }
        return nullptr;
    }

    struct astNode* compoundStatement() {
        match(StartBlock);
        struct astNode* body = programBody();
        match(EndBlock);
        return createNode(compound_node, "compound", body, nullptr);
    }
}
```

Codegen.cpp

Ця програма генерує C-код із абстрактного синтаксичного дерева (AST).

```
#include "Codegen.hpp"

namespace Codegen {
   void codegen(FILE* outFile, struct astNode* node) {
      if (node == 0) {
         return;
      }

      switch (node->type) {
      case program_node: {
         fprintf(outFile, "#include <stdio.h>\n");
         fprintf(outFile, "int main() {\n");
         codegen(outFile, node->left);   //  for declaration
         fprintf(outFile, "\n");
         codegen(outFile, node->right);  //  for statements
                              fprintf(outFile, "return 0;\n}\n");
         break;
      }

      case var_node: {
         fprintf(outFile, "int ");
         codegen(outFile, node->left);
         fprintf(outFile, ";\n");
         codegen(outFile, node->right);
         break;
      }

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

      case statement_node: {
         codegen(outFile, node->left);
         codegen(outFile, node->right);
                              break;
      }
      case input_node: {
         fprintf(outFile, "printf(\"Enter ");
         codegen(outFile, node->left);
         fprintf(outFile, ": \");\n");
         fprintf(outFile, "scanf(\"%%d\", &");
         codegen(outFile, node->left);
         fprintf(outFile, ");\n");
         break;
      }

      case output_node: {
         fprintf(outFile, "printf(\"%%d\\n\", ");
         codegen(outFile, node->left);
                              fprintf(outFile, ");\n");
         break;
      }

      case add_node: {
         fprintf(outFile, "(");
         codegen(outFile, node->left);
```

```
    fprintf(outFile, " + ");
    codegen(outFile, node->right);
    fprintf(outFile, ")");
    break;
}

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

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

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

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

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

case if_node: {
    fprintf(outFile, "if (");
    codegen(outFile, node->left);
    fprintf(outFile, ") ");
    codegen (outFile, node->right);
    break;
}

case else_node: {
    codegen(outFile, node->left);
    fprintf(outFile, "else ");
                codegen(outFile, node->right);
                break;
}
```

```
case or_node: {
   fprintf(outFile, "(");
   codegen(outFile, node->left);
   fprintf(outFile, " || ");
   codegen(outFile, node->right);
   fprintf(outFile, ")");
   break;
}

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

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

case eq_node: {
   fprintf(outFile, "(");
   codegen(outFile, node->left);
   fprintf(outFile, " == ");
   codegen(outFile, node->right);
   fprintf(outFile, ")");
   break;
}

case neq_node: {
   fprintf(outFile, "(");
   codegen(outFile, node->left);
   fprintf(outFile, " != ");
   codegen(outFile, node->right);
   fprintf(outFile, ")");
   break;
}

case gr_node: {
   fprintf(outFile, "(");
   codegen(outFile, node->left);
   fprintf(outFile, " > ");
   codegen(outFile, node->right);
   fprintf(outFile, ")");
   break;
}

case ls_node: {
   fprintf(outFile, "(");
   codegen(outFile, node->left);
   fprintf(outFile, " < ");
   codegen(outFile, node->right);
   fprintf(outFile, ")");
   break;
}

case goto_node: {
   fprintf(outFile, "goto ");
```

```
                        codegen(outFile, node->left);
    fprintf(outFile, ";\n");
    break;
}

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

case for_node: {
    fprintf(outFile, "for(\n");
    codegen(outFile, node->left);
    fprintf(outFile, "\n) ");
                        codegen(outFile, node->right);
    break;
}

case to_node: {
    codegen(outFile, node->left);
    codegen(outFile, node->left->left);
    fprintf(outFile, " <= ");
    codegen(outFile, node->right);
    fprintf(outFile, ";\n++");
    codegen(outFile, node->left->left);
    break;
}

case downto_node: {
    codegen(outFile, node->left);
    codegen(outFile, node->left->left);
    fprintf(outFile, " >= ");
    codegen(outFile, node->right);
    fprintf(outFile, ";\n--");
    codegen(outFile, node->left->left);
    break;
}

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

case continue_node: {
    fprintf(outFile, "continue;\n");
    break;
}

case exit_node: {
    fprintf(outFile, "break;\n");
    break;
}

case repeat_node: {
    fprintf(outFile, "do {\n");
    codegen(outFile, node->left);
                        fprintf(outFile, "} while(");
                        codegen(outFile, node->right);
                        fprintf(outFile, ");\n");
```

```
            break;
        }

        case compound_node: {
            fprintf(outFile, "{\n");
            codegen(outFile, node->left);
            codegen(outFile, node->right);
            fprintf(outFile, "}\n");
            break;
        }

        default: {
            exit(1);
            printf("Undescribed node type: %d\n", node->type);
            break;
        }
        }
    }
}
```

header.cpp

Ця програма визначає функцію lexemeTypeName, яка повертає рядкове ім'я типу лексеми (токена) на основі її перерахованого значення (enum TypeOfToken).

```
#include "header.hpp"

const char* lexemeTypeName(enum TypeOfToken type) {
    switch (type) {
    case StartProgram:  return "Program";
    case StartBlock:    return "Start";
    case Variable:      return "Variable";
    case Type:          return "Type";
    case EndBlock:      return "EndBlock";
    case Input:         return "Input";
    case Output:        return "Output";
    case If:            return "If";
    case Else:          return "Else";
    case Goto:          return "Goto";
    case For:           return "For";
    case To:            return "To";
    case Downto:        return "Downto";
    case Do:            return "Do";
    case While:         return "While";
    case End:           return "WEnd";
    case Repeat:        return "Repeat";
    case Until:         return "Until";
    case Identifier:    return "Identifier";
    case Number:        return "Number";
    case Float:         return "Float";
    case Assign:        return "Assign";
    case Add:           return "Add";
    case Sub:           return "Sub";
```

```
    case Mod:          return "Mod";
    case Mul:          return "Mul";
    case Div:          return "Div";
    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 LBracket:     return "LBracket";
    case RBracket:     return "RBracket";
    case Semicolon:    return "Semicolon";
    case Colon:        return "Colon";
    case Comma:        return "Comma";
    case Unknown_:     return "Unknown_";
    }

    return "Forgotten";
}
```

LexicAnalyzer.cpp

Ця програма є реалізацією лексичного аналізатора для обробки вхідного файлу з кодом, розділення його на токени та їх класифікацію відповідно до типів токенів.

```
#include "LexicAnalyzer.hpp"


#include <stdbool.h>

#include <string.h>

#include <stdlib.h>


extern struct Token* TokenTable;   // Таблиця лексем

extern unsigned int TokensNum;     // Кількість лексем


unsigned int LexicAnalyzer::getTokens(FILE* F) {

    enum States state = Start;

    struct Token tempToken;


    char ch, buf[16];

    unsigned int tokenCount = 0;
```

```
int line = 1;
int tokenLength = 0;

ch = getc(F);

while (true) {
    switch (state) {
    case Start: {
        if (ch == EOF) {
            state = EndOFile;
        }
        else if (('0' <= ch && ch <= '9') || ch == '-') {
            state = Digit;
        }
        else if (ch == '!' && (ch = getc(F)) == '!') {
            state = Comment;
        }
        else if (('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z') || ch == '_') {
            state = Letter;
        }
        else if (ch == ' ' || ch == '\t' || ch == '\n') {
            state = Separator;
        }
        else {
            state = Another;
        }
        break;
    }

    case Digit: {
        buf[0] = ch;
        int j = 1;
        ch = getc(F);
        while (((ch <= '9' && ch >= '0') || ch == '_' || ch == '.') && j < 10) {
            buf[j++] = ch;
            ch = getc(F);
        }
```

```
buf[j] = '\0';

if (!strcmp(buf, "-")) {
    strcpy_s(tempToken.name, "-");
    tempToken.type = Sub;
    tempToken.value = 0;
    tempToken.line = line;
    state = Finish;
    break;
}

if (!strcmp(buf, ".")) {
    strcpy_s(tempToken.name, ".");
    tempToken.type = Unknown_;
    tempToken.value = 0;
    tempToken.line = line;
    state = Finish;
    break;
}

short dotCounter = 0, currCharIndex = 0;
while (buf[currCharIndex] != '\0') {
    if (buf[currCharIndex] == '.') {
        ++dotCounter;
    }
    ++currCharIndex;
}

if (dotCounter > 1) {
    strcpy_s(tempToken.name, buf);
    tempToken.type = Unknown_;
    tempToken.value = 0;
    tempToken.line = line;
    state = Finish;
    break;
}
```

```
    if (dotCounter == 1) {
        strcpy_s(tempToken.name, buf);
        tempToken.type = Float;
        tempToken.value = atof(buf);
        tempToken.line = line;
        state = Finish;
        break;
    }

    if ((buf[0] == '0' && buf[1] != '\0') || (buf[0] == '-' && buf[1] == '0')) {
        strcpy_s(tempToken.name, buf);
        tempToken.type = Unknown_;
        tempToken.value = 0;
        tempToken.line = line;
        state = Finish;
        break;
    }
    strcpy_s(tempToken.name, buf);
    tempToken.type = Number;
    tempToken.value = atoi(buf);
    tempToken.line = line;
    state = Finish;
    break;
}

case Separator: {
    if (ch == '\n') {
        line++;
    }
    ch = getc(F);

    state = Start;
    break;
}

case SComment: {
```

```
    ch = getc(F);
    if (ch == '!') {
        state = Comment;
    }
    else {
        strcpy_s(tempToken.name, "!");
        tempToken.type = Unknown_;
        tempToken.value = 0;
        tempToken.line = line;
        state = Finish;
    }
    break;
}

case Comment: {
    ch = getc(F);

    if (ch == '!')
    {
        ch = getc(F);
        if (ch == '!')
        {
            state = Start;
            ch = getc(F);
            break;
        }
    }
    if (ch == EOF)
    {
        printf("Error: Comment not closed!\n");
        state = EndOFile;
        break;
    }
    break;
}

case Finish: {
```

```
    if (tokenCount < MAX_TOKENS) {
        TokenTable[tokenCount++] = tempToken;
        if (ch != EOF) {
            state = Start;
        }
        else {
            state = EndOFile;
        }
    }
    else {
        printf("\n\t\t\ttoo many tokens !!!\n");
        return TokensNum = tokenCount - 1;
    }
    break;
}

case EndOFile: {
    return TokensNum = tokenCount;
}

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


    enum TypeOfToken tempType = Unknown_;
```

```
if (!strcmp(buf, "Program")) {
   tempType = StartProgram;
}
else if (!strcmp(buf, "Begin")) {
   tempType = StartProgram;
}
else if (!strcmp(buf, "Var")) {
   tempType = Variable;
}
else if (!strcmp(buf, "Int16")) {
   tempType = Type;
}
else if (!strcmp(buf, "End")) {
   tempType = EndBlock;
}
else if (!strcmp(buf, "Get")) {
   tempType = Input;
}
else if (!strcmp(buf, "Put")) {
   tempType = Output;
}
else if (!strcmp(buf, "If")) {
   tempType = If;
}
else if (!strcmp(buf, "Else")) {
   tempType = Else;
}
else if (!strcmp(buf, "Goto")) {
   tempType = Goto;
}
else if (!strcmp(buf, "For")) {
   tempType = For;
}
else if (!strcmp(buf, "To")) {
   tempType = To;
}
else if (!strcmp(buf, "Downto")) {
```

```
    tempType = Downto;
}
else if (!strcmp(buf, "Do")) {
    tempType = Do;
}
else if (!strcmp(buf, "End")) {
    tempType = End;
}
else if (!strcmp(buf, "While")) {
    tempType = While;
}
else if (!strcmp(buf, "Continue")) {
    tempType = Continue;
}
else if (!strcmp(buf, "Exit")) {
    tempType = Exit;
}
else if (!strcmp(buf, "Repeat")) {
    tempType = Repeat;
}
else if (!strcmp(buf, "Until")) {
    tempType = Until;
}
else if (!strcmp(buf, "Mod")) {
    tempType = Mod;
}
else if (!strcmp(buf, "And")) {
    tempType = And;
}
else if (!strcmp(buf, "Or")) {
    tempType = Or;
}
else if (!strcmp(buf, "Eg")) {
    tempType = Equality;
}
else if (!strcmp(buf, "Ne")) {
    tempType = NotEquality;
```

```
    }
    else if (!strcmp(buf, "Mul")) {
        tempType = Mul;
    }
    else if (!strcmp(buf, "Div")) {
        tempType = Div;
    }
    else if (!strcmp(buf, "Mod")) {
        tempType = Mod;
    }
    else if (strlen(buf) == 6) {
        bool isValidIdentifier = true;
        for (int i = 0; i < 6; i++) {
            if (!('A' <= buf[i] && buf[i] <= 'Z')) {
                isValidIdentifier = false;
                break;
            }
        }
        if (isValidIdentifier) {
            tempType = Identifier;
        }
    }


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

case Another: {
    switch (ch) {
    case '(': {
        strcpy_s(tempToken.name, "(");
        tempToken.type = LBracket;
```

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

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

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

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

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

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

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

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

   return TokensNum = tokenCount;
}

// Функція друкує таблицю лексем на екран
void LexicAnalyzer::printTokens(void) {
   char type_tokens[16];
   printf("-------------------------------------------------------------------------\n");
   printf("| TOKEN TABLE                                                           |\n");
   printf("-------------------------------------------------------------------------\n");
   printf("| line number | token          | value      | token code | type of token |\n");
   printf("-------------------------------------------------------------------------\n");

   for (unsigned int i = 0; i < TokensNum; i++) {
      strcpy_s(type_tokens, lexemeTypeName(TokenTable[i].type));

      printf("|%12d |%16s |%11d |%11d | %-13s |\n",
```

```
            TokenTable[i].line,
            TokenTable[i].name,
            TokenTable[i].value,
            TokenTable[i].type,
            type_tokens);
        printf("------------------------------------------------------------------------\n");
    }
}

void LexicAnalyzer::fprintTokens(FILE* F) {

    char type_tokens[16];
    fprintf(F, "\n\n-------------------------------------------------------------------------\n");
    fprintf(F, "| TOKEN TABLE                                                           |\n");
    fprintf(F, "-------------------------------------------------------------------------\n");
    fprintf(F, "| line number | token           | value      | token code | type of token |\n");
    fprintf(F, "-------------------------------------------------------------------------\n");

    for (unsigned int i = 0; i < TokensNum; i++) {
        strcpy_s(type_tokens, lexemeTypeName(TokenTable[i].type));

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

main.cpp

Ця програма здійснює компіляцію програми, написаної на власній мові (файли з розширенням .b03).

```
#include <iostream>
#include <windows.h>

#include "LexicAnalyzer.hpp"
#include "Parser.hpp"
#include "Ast.hpp"
#include "Codegen.hpp"

#define LANGUAGE	".b03"

struct Token* TokenTable;	// Таблиця лексем
unsigned int TokensNum;		// Кількість лексем

struct id* idTable;		// Таблиця ідентифікаторів
unsigned int idNum;		// кількість ідентифікаторів

struct id* labelTable;		// Таблиця міток
unsigned int labelNum;		// кількість міток

FILE* errorFile;

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

	// виділення пам'яті під таблицю ідентифікаторів
	idTable = (struct id*)malloc(MAX_IDENTIFIER * sizeof(struct id));
	labelTable = (struct id*)malloc(MAX_IDENTIFIER * sizeof(struct id));

	char InputFile[32] = "";

	FILE* InFile, *TokenFileP;

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

	if (!strstr(InputFile, LANGUAGE)) {
		free(TokenTable);
		free(idTable);
		free(labelTable);
		_fcloseall();
		printf("Program file name should contain \"%s\"", LANGUAGE);
		system("pause");
		return 1;
	}

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

```
		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], errorFileName[32];
strcpy_s(TokenFile, NameFile);
strcat_s(TokenFile, ".token");
strcpy_s(errorFileName, NameFile);
strcat_s(errorFileName, ".errorlist");

// лексичний аналіз
if ((fopen_s(&TokenFileP, TokenFile, "wt")) != 0)
{
		printf("Error: Can not create file: %s\n", TokenFile);
		free(TokenTable);
		free(idTable);
		free(labelTable);
		_fcloseall();
		system("pause");
		return 1;
}

if ((fopen_s(&errorFile, errorFileName, "wt")) != 0)
{
		printf("Error: Can not create file: %s\n", errorFileName);
		free(TokenTable);
		free(idTable);
		free(labelTable);
		_fcloseall();
		system("pause");
		return 1;
}

TokensNum = LexicAnalyzer::getTokens(InFile);

LexicAnalyzer::fprintTokens(TokenFileP);
fclose(InFile);
fclose(TokenFileP);

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

// синтаксичний аналіз
Parser::Parser();
Parser::Semantic();

// створення абстрактного синтаксичного дерева
struct astNode* ast = AST::astParser();

//printf("\nAbstract Syntax Tree:\n");
//PrintAST(ASTree, 0);

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");
		free(TokenTable);
		free(idTable);
		free(labelTable);
		AST::deleteNode(ast);
		exit(1);
		system("pause");
	}
	AST::fPrintAST(ASTFile, ast, 0);
	printf("\nAST has been created and written to %s.\n", AST);

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

	// Open output file
	FILE* outFile;
	fopen_s(&outFile, OutputFile, "w");
	if (!outFile)
	{
		printf("Failed to open output file.\n");
		free(TokenTable);
		free(idTable);
		free(labelTable);
		AST::deleteNode(ast);
		exit(1);
		system("pause");
	}
	// Generate C code from AST
	Codegen::codegen(outFile, ast);

	// генерація вихідного С коду
	printf("\nC code has been generated and written to %s.\n", OutputFile);

	// Close the file
	_fcloseall();

	free(TokenTable);
	free(idTable);
	free(labelTable);

	char setVar[256] = "\"C:\\Program Files\\Microsoft Visual
Studio\\2022\\Community\\VC\\Auxiliary\\Build\\vcvars64.bat\"";

	char createExe[128];
	sprintf_s(createExe, "cl %s", OutputFile);
	strcat_s(setVar, " && ");
	strcat_s(setVar, createExe);
	system(setVar);
//*/
	system("pause");
	return 0;
}
```

Parser.cpp

Цей код — частина парсера для компілятора, який виконує синтаксичний та семантичний аналіз програми.

```
#include "Parser.hpp"

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

extern struct Token* TokenTable;
extern unsigned int TokensNum;

extern struct id* idTable;
extern unsigned int idNum;

extern struct id* labelTable;
extern unsigned int labelNum;

extern FILE* errorFile;

int pos = 0;

namespace Parser {
   void Parser() {
      program();
      printf("\nThe program is syntax correct.\n");
      fprintf(errorFile, "\nThe program is syntax correct.\n");
   }

   void Semantic() {
      idNum = IdIdentification(idTable, TokenTable, TokensNum);
      labelNum = LabelIdentification(labelTable, TokenTable, TokensNum);
      printf("\nThe program is semantic correct.\n");
      printf("\n%d labels found\n", labelNum);
      fprintf(errorFile, "\nThe program is semantic correct.\n");
      fprintf(errorFile, "\n%d labels found\n", labelNum);
      printIdentifiers(labelNum, labelTable);
      fprintIdentifiers(errorFile, labelNum, labelTable);
      printf("\n%d identifiers found\n", idNum);
      fprintf(errorFile, "\n%d identifiers found\n", idNum);
      printIdentifiers(idNum, idTable);
      fprintIdentifiers(errorFile, idNum, idTable);
   }

   void match(enum TypeOfToken expectedType) {
      if (TokenTable[pos].type == expectedType)
         pos++;
      else {
         printf("\nSyntax error in line %d, token number: %d : another type of lexeme was expected (expected: %s |
current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(expectedType),
lexemeTypeName(TokenTable[pos].type));
         fprintf(errorFile, "\nSyntax error in line %d, token number: %d : another type of lexeme was expected
(expected: %s | current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(expectedType),
lexemeTypeName(TokenTable[pos].type));
         exit(1);
      }
   }
```

```
// ôóíêö³ÿ çàïèñóº îãîëîøåí³ ³äåíòèô³êàòîðè â òàáëèöþ ³äåíòèô³êàòîð³â idTable
// ïîâåðòàº ê³ëüê³ñòü ³äåíòèô³êàòîð³â
// ïåðåâ³ðÿº ÷è óñ³ âèêîðèñòàí³ ³äåíòèô³êàòîðè îãîëîøåí³
unsigned int IdIdentification(struct id idTable[], struct Token TokenTable[], unsigned int tokenCount) {
    unsigned int idCount = 0;
    unsigned int i = 0;

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

    while (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);
                    fprintf(errorFile, "\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");
                    fprintf(errorFile, "\nToo many identifiers !\n");
                    return idCount;
                }
            }
            else
                ++i;
        }
        ++i;
    }

    for (; i < tokenCount; ++i) {
        if (
            TokenTable[i].type == Identifier
            && TokenTable[i - 1].type != Goto
            && TokenTable[i + 1].type != Colon
            ) {
            bool yes = 0;
            for (unsigned int j = 0; j < idCount; ++j) {
                if (!strcmp(TokenTable[i].name, idTable[j].name)) {
                    yes = 1;
                    break;
                }
            }
            if (!yes) {
                printf("\nSemantic Error In line %d, an undeclared identifier \"%s\"!", TokenTable[i].line,
TokenTable[i].name);
                fprintf(errorFile, "\nSemantic Error In line %d, an undeclared identifier \"%s\"!", TokenTable[i].line,
TokenTable[i].name);
                exit(1);
            }
```

```
        }

    }

    return idCount; // Ïîâåðòàº ê³ëüê³ñòü  ³äåíòèô³êàòîð³â
}


void program() {
    match(StartProgram);        // очікує токен "program"
    match(Identifier);          // очікує ім'я програми, напр. AAAAAAAA
    match(Semicolon);           // очікує символ ';' після назви програми

    // обробка оголошення змінних (якщо є)
    if (TokenTable[pos].type == Variable) {  // припускаємо, що 'var' має тип Data
        ++pos;                      // пропускаємо 'var'
        variableDeclaration();      // обробка оголошення: тип (Type) та список ідентифікаторів
        match(Semicolon);           // очікуємо ';' в кінці оголошення змінних
    }

    match(StartProgram);          // очікує токен "start"
    programBody();              // обробляємо тіло програми (оператори між start та finish)
    match(EndBlock);            // очікує токен "finish"
}

void variableDeclaration() {
    match(Type);
    variableList();
}

void variableList() {
    match(Identifier);
    while (TokenTable[pos].type == Comma) {
        ++pos;  // for Comma
        match(Identifier);
    }

}

void statement() {
    switch (TokenTable[pos].type) {
    case Input:     inputStatement();   break;
    case Output:    outputStatement();  break;
    case If:        ifStatement();      break;
    case Goto:      gotoStatement();    break;
    case For:       forStatement();     break;
    case While:     whileStatement();   break;
    case Repeat:    repeatStatement();  break;
    case StartBlock:compoundStatement(); break;
    default: {
        if (TokenTable[pos + 1].type == Colon)
            labelPoint();
        else
            assignStatement();
    }
    }
}

void inputStatement() {
    ++pos;  // for Input
    match(Identifier);
```

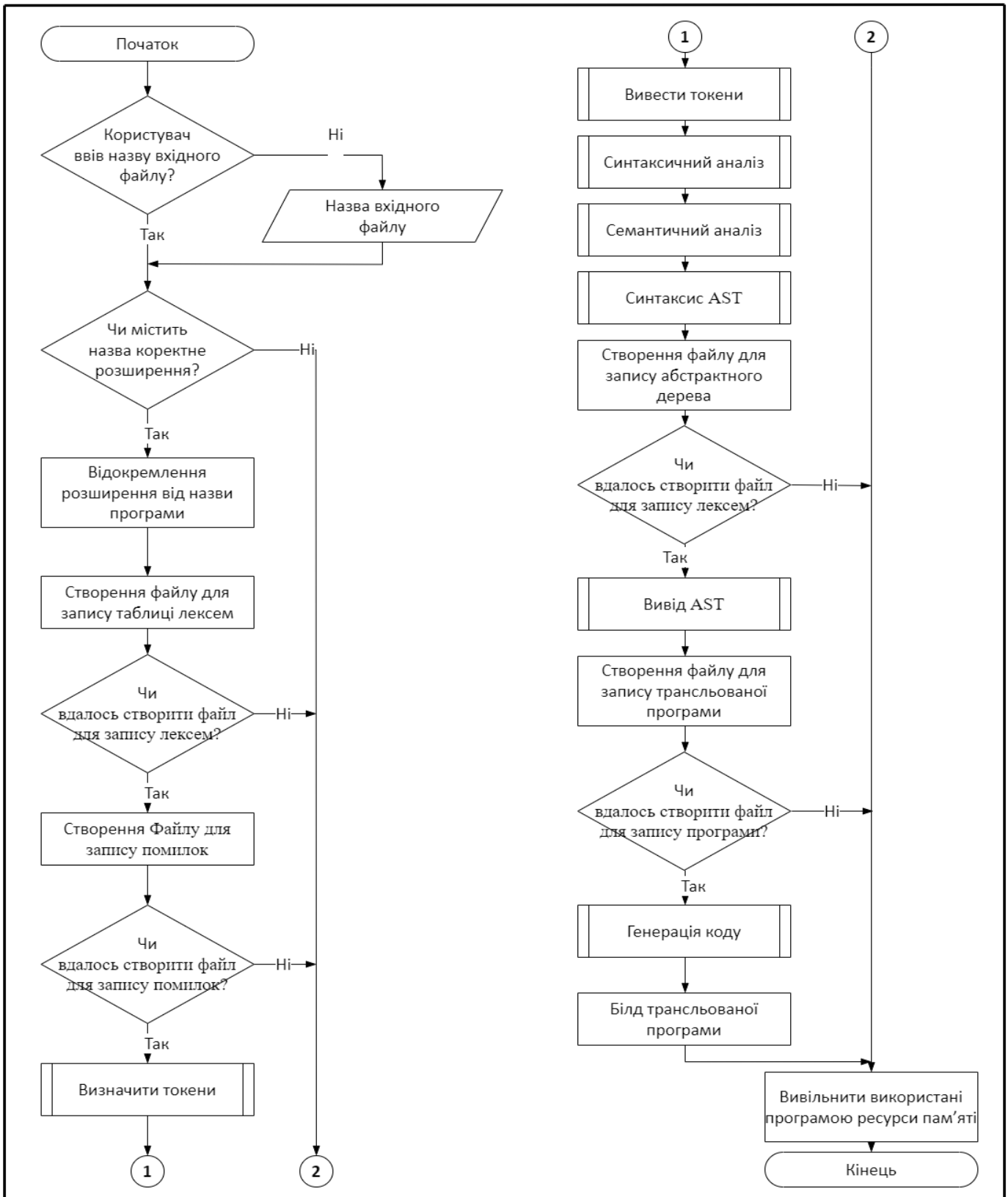
```
}

void outputStatement() {
   ++pos;  // for Output
   arithmeticExpression();
}

void arithmeticExpression() {
   lowPriorityExpression();
   //lowPriorityOperator
   while (TokenTable[pos].type == Add || TokenTable[pos].type == Sub) {
      ++pos;  // for + or -
      lowPriorityExpression();
   }
}

void lowPriorityExpression() {
   middlePriorityExpression();
   //  middlePriorityOperator
   if (TokenTable[pos].type == Mul || TokenTable[pos].type == Mod || TokenTable[pos].type == Div) {
      ++pos;  // for * or Mod or Div
      middlePriorityExpression();
   }
}

void middlePriorityExpression() {
   switch (TokenTable[pos].type) {
   case Identifier:    ++pos;  break;
   case Number:        ++pos;  break;
   case LBracket: {
      ++pos;  //  for (
      arithmeticExpression();
      match(RBracket);
      break;
   }
   default: {
      printf("\nSyntax error in line %d, token number: %d : middle priority operation was expected (current: %s).\n",
TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));
      fprintf(errorFile, "\nSyntax error in line %d, token number: %d : middle priority operation was expected
(current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));
      exit(1);
   }
   }
}

void assignStatement() {
   arithmeticExpression();
   match(Assign);
   match(Identifier);
}

void ifStatement() {
   match(If);
   match(LBracket);
   logicalExpression();
   match(RBracket);
   statement();
   if (TokenTable[pos].type == Else) {
      //match(Semicolon);
      ++pos;  //  for else
      statement();
```

```
    }
  }

  void logicalExpression() {
    andExpression();
    while (TokenTable[pos].type == Or) {
      ++pos;  //  for ||
      andExpression();
    }
  }

  void andExpression() {
    comparison();
    while (TokenTable[pos].type == And) {
      ++pos;  //  for &&
      andExpression();
    }
  }

  void comparison() {
    if (TokenTable[pos].type == Not) {
      ++pos;  //  for !
      match(LBracket);
      logicalExpression();
      match(RBracket);
    }
    else if (TokenTable[pos].type == LBracket) {
      ++pos;  //  for (
      logicalExpression();
      match(RBracket);
    }
    else {
      comparisonExpression();
    }
  }

  void comparisonExpression() {
    arithmeticExpression();
    if (TokenTable[pos].type == Equality ||
      TokenTable[pos].type == NotEquality ||
      TokenTable[pos].type == Greate ||
      TokenTable[pos].type == Less
      ) {
      ++pos;  //  for "=" | "!=" | "<<" | ">>"

    }
    else {
      printf("\nSyntax error in line %d : Comparison operator was Expected, %s token gained).\n",
TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));
      fprintf(errorFile, "\nSyntax error in line %d : Comparison operator was Expected, %s token gained).\n",
TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));
      exit(1);
    }
    arithmeticExpression();
  }

  void gotoStatement() {
    match(Goto);
    match(Identifier);
  }
```

```
   void labelPoint() {
      match(Identifier);
      match(Colon);
   }

   void forStatement() {
      match(For);
      assignStatement();
      if (TokenTable[pos].type == To || TokenTable[pos].type == Downto) {
         ++pos;  //  for to or downto
      }
      else {
         printf("\nSyntax error in line %d : TO or DOWNTO was expected, %s token gained).\n", TokenTable[pos].line,
lexemeTypeName(TokenTable[pos].type));
         fprintf(errorFile, "\nSyntax error in line %d : TO or DOWNTO was expected, %s token gained).\n",
TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));
         exit(1);
      }
      arithmeticExpression();
      match(Do);
      statement();
   }

   void whileStatement() {
      match(While);
      logicalExpression();
      while (TokenTable[pos].type != End) {
         statementInWhile();
         //match(Semicolon);
      }
      //match(WEnd);
      ++pos;  //  for WEND
   }

   void statementInWhile() {
      switch (TokenTable[pos].type) {
      case Continue: {
         ++pos;  //  for CONTINUE
         match(While);
         break;
      }
      case Exit: {
         ++pos;  //  for EXIT
         match(While);
         break;
      }
      default: {
         statement();
         break;
      }
      }
   }

   void repeatStatement() {
      match(Repeat);
      while (TokenTable[pos].type != Until) {
         statement();
         //match(Semicolon);
      }
      //match(Until);
      ++pos;  //  for UNTIL
```

```
    match(LBracket);
    logicalExpression();
    match(RBracket);
  }

  void compoundStatement() {
    match(StartBlock);
    programBody();
    match(EndBlock);
  }

  void programBody() {
    while (TokenTable[pos].type != EndBlock) {
      statement();
      //match(Semicolon);
    }
  }

  void printIdentifiers(int num, struct id* table) {
    for (int i = 0; i < num; i++) {
      printf("%s\n", table[i].name);
    }
  }
  void fprintIdentifiers(FILE* F, int num, struct id* table) {
    for (int i = 0; i < num; i++) {
      fprintf(F, "%s\n", table[i].name);
    }
  }

  unsigned int LabelIdentification(struct id labelTable[], struct Token TokenTable[], unsigned int tokenCount) {
    unsigned int labelNum = 0;
    unsigned int i = 0;
    unsigned int start = 0;

    while (TokenTable[start++].type != StartProgram);

    i = start;
    while (TokenTable[i].type != EndBlock) {
      if (TokenTable[i].type == Identifier && TokenTable[i + 1].type == Colon) {
        strcpy_s(labelTable[labelNum++].name, TokenTable[i].name);
      }
      ++i;
    }

    i = start;
    while (TokenTable[i].type != EndBlock) {
      if (TokenTable[i].type == Identifier && TokenTable[i - 1].type == Goto) {
        bool found = false;
        for (unsigned int j = 0; j < labelNum; j++) {
          if (strcmp(labelTable[j].name, TokenTable[i].name) == 0) {
            found = true;
            break;
          }
        }
        if (!found) {
          printf("\n Semantic error: In line %d label %s is not defined\n", TokenTable[i].line, TokenTable[i].name);
          fprintf(errorFile, "\n Semantic error: In line %d label %s is not defined\n", TokenTable[i].line,
TokenTable[i].name);
          exit(1);
        }
      }     ++i;}        return labelNum;    }}
```

Додаток Е (Алгоритм транслятора B03)



Міністерство освіти і науки України					КУРСОВИЙ ПРОЄКТ	
					Розробка системних програмних модулів та компонент систем програмування	
Зм.	Арк.	№ докум.	Підпис	Дата	Алгоритм транслятора B03	Літера / Маса / Масштаб
Виконав		Бохонок О.П.				
Керівник		Козак Н.Б.				Аркуш / Аркушів 1
Консульт.						
Консульт.						
Зав. каф.						НУ «ЛП», ІКТА, каф. ЕОМ, гр КІ-308
Реценз.						