Міністерство освіти і науки України
Національний університет "Львівська Політехніка"
Кафедра ЕОМ

Пояснювальна записка

до курсового проєкту "СИСТЕМНЕ ПРОГРАМУВАННЯ"

на тему: “РОЗРОБКА СИСТЕМНИХ ПРОГРАМНИХ МОДУЛІВ ТА КОМПОНЕНТ СИСТЕМ ПРОГРАМУВАННЯ”

Індивідуальне завдання

“РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ”

Варіант №20

Виконав:
ст. гр. КІ-307
Маринович Марко
Перевірив:
Козак Н. Б.

Львів-2024

ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ

1. Цільова мова транслятора – мова програмування С або асемблер для 32/64 розрядного процесора.
2. Для отримання виконуваного файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio або будь-яким іншим.
3. Мова розробки транслятора: C/C++.
4. Реалізувати графічну 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 або асемблера;

   об'єктний файл;

   виконуваний 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.

Деталізація завдання на проектування:

1. В кожному завданні передбачається блок оголошення змінних; змінні зберігають значення цілих чисел і, в залежності від варіанту, можуть бути 16/32 розрядними. За потребою можна реалізувати логічний тип даних.
2. Необхідно реалізувати арифметичні операції – додавання, віднімання, множення, ділення, залишок від ділення; операції порівняння – перевірка на рівність і нерівність, більше і менше; логічні операції – заперечення, "логічне І" і "логічне АБО".

Пріоритет операцій наступний – круглі дужки (), логічне заперечення, мультиплікативні (множення, ділення, залишок від ділення), адитивні (додавання, віднімання), відношення (більше, менше), перевірка на рівність і нерівність, логічне І, логічне АБО.

3. За допомогою оператора вводу можна зчитати з клавіатури значення змінної; за допомогою оператора виводу можна вивести на екран значення змінної, виразу чи цілої константи.
4. В кожному завданні обов'язковим є оператор присвоєння за допомогою якого можна реалізувати обчислення виразів з використанням заданих операцій і операції круглі дужки (); у якості операндів можуть бути цілі константи, змінні, а також інші вирази.
5. В кожному завданні обов'язковим є оператор типу "блок" (складений оператор), його вигляд має бути таким, як і блок тіла програми.
6. Необхідно реалізувати задані варіантом оператори, синтаксис операторів наведено у таблиці 1.1. Синтаксис вхідної мови має забезпечити реалізацію обчислень лінійних алгоритмів, алгоритмів з розгалуженням і циклічних алгоритмів. Опис формальної мови студент погоджує з викладачем.
7. Оператори можуть бути довільної вкладеності і в будь-якій послідовності.
8. Для перевірки роботи розробленого транслятора, необхідно написати три тестові програми на вхідній мові програмування.

Деталізований опис власної мови програмування:

Опис вхідної мови програмування:

- Тип даних: INT_4
- Блок тіла програми: STARTPROGRAM VARIABLE…; STARTBLOK ENDBLOK
- Оператор вводу: READ ()
- Оператор виводу: WRITE ()

- Оператори: IF ELSE (C)

GOTO (C)

FOR-TO-DO (Паскаль)

FOR-DOWNTO-DO (Паскаль)

WHILE (Бейсік)

REPEAT-UNTIL (Паскаль)

- Регістр ключових слів: Up
- Регістр ідентифікаторів: Low4 перший символ _
- Операції арифметичні: ADD, SUB, MUL, DIV, MOD
- Операції порівняння: EQ, NE, GT, LT
- Операції логічні: !, &, |
- Коментар: #*... *#
- Ідентифікатори змінних, числові константи

- Оператор присвоєння: <==

Для отримання виконавчого файлу на виході розробленого транслятора скористатися програмами ml.exe (компілятор мови асемблера) і link.exe (редактор зв’язків).

АНОТАЦІЯ

Цей курсовий проект приводить до розробки транслятора, який здатен конвертувати вхідну мову, визначену відповідно до варіанту, у мову асемблера. 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 асемблерного коду для кожного блоку лексем. Отриманий код записується у результуючий файл, готовий для виконання.

Отриманий після трансляції код можна скомпілювати за допомогою відповідних програм (наприклад, LINK, ML і т. д.).

ЗМІСТ

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 ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ

Термін "транслятор" визначає обслуговуючу програму, що проводить трансляцію вихідної програми, представленої на вхідній мові програмування, у робочу програму, яка відображена на об'єктній мові. Наведене визначення застосовне до різноманітних транслюють 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.

Асемблер - це системна обслуговуюча програма, яка перетворює символічні конструкції в команди машинної мови. Типовою особливістю асемблерів є дослівна трансляція однієї символічної команди в одну машинну.

Компілятор - обслуговуюча програма, яка виконує трансляцію програми, написаної мовою оригіналу програмування, в машинну мову. Схоже до асемблера, компілятор виконує перетворення програми з однієї мови в іншу, найчастіше - у мову конкретного комп'ютера.

Інтерпретатор - це програма чи пристрій, що виконує пооператорну трансляцію та виконання вихідної програми. Відмінно від компілятора, інтерпретатор не створює на виході програму на машинній мові. 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 оптимізацію та створення машинно-залежного коду відповідно.

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

Під час процесу виділення лексем лексичний аналізатор може виконувати дві основні функції: автоматично побудову таблиць об'єктів (таких як ідентифікатори, рядки, числа і т. д.) 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 во внутрішнє представлення. Це здійснюється з метою оптимізації та/або для полегшення генерації коду. Крім того, перетворення програми у внутрішнє представлення може бути використано для створення переносимого компілятора. У цьому випадку, тільки остання фаза (генерація коду) є залежною від конкретної архітектури. В якості внутрішнього представлення може використовуватися префіксний або постфіксний запис, 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-залежні оптимізації, як правило, проводяться на етапі генерації коду, і вони орієнтовані на конкретну архітектуру машини. Ці оптимізації можуть включати розподіл регістрів, вибір довгих або коротких переходів та оптимізацію вартості команд для конкретних послідовностей команд.

Глобальна оптимізація спрямована на поліпшення ефективності всієї програми і базується на глобальному потоковому аналізі, який виконується на графі програми.

Цей аналіз враховує властивості програми, такі як межпроцедурний аналіз, міжмодульний аналіз та аналіз галузей життя змінних.

Фінальна фаза трансляції - генерація коду, результатом якої є або асемблерний модуль, або об'єктний (або завантажувальний) модуль. На цьому етапі можуть застосовуватися деякі локальні оптимізації для полегшення генерації вартісного та ефективного коду.

Важливо відзначити, що фази транслятора можуть бути відсутніми або об'єднаними в залежності від конкретної реалізації. В простіших випадках, таких як у випадку однопроходових трансляторів, може відсутній окремий етап генерації проміжного представлення та оптимізації, а інші фази можуть бути об'єднані в одну, при цьому не створюється 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 (Backus/Naur Form - BNF).

```
topRule  = "STARTPROGRAM", {"VARIABLE", variable_declaration, ";"}, "STARTBLOK",
{statement, ";"}, "ENDBLOK";
varsBlok  = "INT_4", variable_list;
variable_list = identifier, {",", identifier};
identifier  = "_", low_letter, low_letter, low_letter;
commaAndIdentifier  = ",",  identifier;
statement  =  write  |  read  |  assignment  |  ifStatement  |  goto_statement  |
labelRule  |  forToOrDownToDoRule  |  while  |  repeatUntil;
statementForWhile = statement | continueWhile | exitWhile;
read  = "READ", identifier;
write  = "WRITE", arithmetic_expression;
arithmetic_expression = arithmetic { mult , arithmetic};
assignment  =  identifier,  "<-",  equation;
ifStatement  = "IF", "(",  equation,  ")",  { statementForWhile }, ";"  ["ELSE",  {
statementForWhile }, ";"];
goto_statement  =  "GOTO",  identifier;
labelRule  =  identifier, ":";
forToOrDownToDoRule  =  "FOR",  assignment, ("TO" | "DOWNTO"),  equation,  "DO",  {
statement }, ";";
while  = "WHILE", equation, { statementsForWhile }, "END", "WHILE";
whileContinue = "CONTINUE", "WHILE";
whileExit = "EXIT", "WHILE";
repeatUntil  = "REPEAT",  { statement }, "UNTIL", equation;
equation  =  signedNumber  |  identifier  |  notRule,  [  operationAndIdentOrNumber
|  equation  ];
notRule  =  notOperation,  (signedNumber  |  identifier  |  equation);
operationAndIdentOrNumber  =  mult  |  arithmetic  |  logic  |  compare
signedNumber  |  identifier  |  equation;
arithmetic  = "ADD" | "SUB";
mult  = "MUL" | "DIV" | "MOD";
logic  = "&" | "|";
notOperation  = "!";
compare  = "EQ" | "NE" | "LT" | "GT";
comment  =  "LComment" string "RComment";
LComment  =  "#*";
RComment  =  "*#";
string  = {  low_letter  |  up_letter  |  number };
signedNumber = [ sign ], (nonZeroDigit, digit | "0");
sign = "+" | "-";
low_letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" |
"m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z";
```

```
up_letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" |
"M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";
digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9", "0";
nonzero_digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

2.2 Опис термінальних символів та ключових слів.

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

Термінальний символ або ключове слово	Значення
STARTPROGRAM	Початок програми
VARIABLE	Початок блоку опису змінних
STARTBLOK	Початок розділу операторів
ENDBLOK	Кінець розділу операторів
READ	Оператор вводу змінних
WRITE	Оператор виводу (змінних або рядкових констант)
<-	Оператор присвоєння
IF	Оператор умови
ELSE	Оператор умови
GOTO	Оператор переходу
LABEL	Мітка переходу
FOR	Оператор циклу
TO	Інкремент циклу
DOWNTO	Декремент циклу
DO	Початок тіла циклу
WHILE	Оператор циклу
REPEAT	Початок тіла циклу
UNTIL	Оператор циклу
ADD	Оператор додавання
SUB	Оператор віднімання
MUL	Оператор множення

DIV	Оператор ділення
MOD	Оператор знаходження залишку від ділення
EQ	Оператор перевірки на рівність
NE	Оператор перевірки на нерівність
LT	Оператор перевірки чи менше
GT	Оператор перевірки чи більше
!	Оператор логічного заперечення
&	Оператор кон’юнкції
|	Оператор диз’юнкції
INT_4	32-ох розрядні знакові цілі
#*…*#	Коментар
,	Розділювач
(	Відкриваюча дужка
)	Закриваюча дужка

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

3. РОЗРОБКА ТРАНСЛЯТОРА З 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 та вибір структур даних.

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

1) Мульти мапа для лексеми, значення та рядка кожного токена.

```
std::multimap<int, std::shared_ptr<IToken>> m_priorityTokens;

 std::string m_lexeme; //Лексема

 std::string m_value;  //Значення

 int m_line = -1;      //Рядок
```

2) Таблиця лексичних класів

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	STARTPROGRAM
Start	STARTBLOK
Vars	VARIABLE
End	ENDBLOK
VarType	INT_4
Read	READ
Write	WRITE
Assignment	<-
If	IF
Else	ELSE
Goto	GOTO
Colon	:
Label	
For	FOR
To	TO
DownTo	DOWNTO
Do	DO
While	WHILE
Repeat	REPEAT
Until	UNTIL
Addition	ADD
Subtraction	SUB
Multiplication	MUL

Division	DIV
Mod	MOD
Equal	EQ
NotEqual	NE
Less	LT
Greate	GT
Not	!
And	&
Or	|
Plus	+
Minus	-
Identifier	
Number	
String	
Undefined	
Unknown	
Comma	,
Quotes	“
Semicolon	;
LBraket	(
RBraket	)
LComment	#*
RComment	*#
Comment	

3.3. Розробка лексичного аналізатора

Основна задача лексичного аналізу — розбити вхідний текст, що складається з послідовності символів, на послідовність лексем (слів), тобто виділити ці слова з безперервної послідовності символів. Всі символи вхідної послідовності можна поділити на дві категорії:

1. Символи, що належать яким-небудь лексемам (ключові слова, ідентифікатори, числові константи, оператори та інші).
2. Символи, що розділяють лексеми (пробіли, знаки операцій, нові рядки тощо).

Лексичний аналізатор працює в два основних режими:

- Як підпрограма, що викликається синтаксичним аналізатором для отримання чергової лексеми.
- Як повний прохід, результатом якого є файл лексем, що містить усі лексеми програми.

Ми обрали другий варіант, де спочатку виконується фаза лексичного аналізу, а результат цієї фази передається для подальшої обробки на етап синтаксичного аналізу.

3.3.1. Розробка алгоритму роботи лексичного аналізатора

Лексичний аналізатор працює за принципом скінченного автомату, що містить такі стани:

- Start — початок виділення чергової лексеми.
- Finish — кінець виділення чергової лексеми.
- EndOfFile — кінець файлу, завершення розпізнавання лексем.
- Letter — перший символ є літерою або символом _, розпізнаються ключові слова та ідентифікатори.
- Digit — перший символ є цифрою, розпізнаються числові константи.
- Separator — обробка роздільників (пробіли, табуляції, нові рядки).
- SComment — початок коментаря.
- Comment — ігнорування коментаря.
- Another — обробка інших символів, зокрема операторів.

Алгоритм лексичного аналізатора передбачає послідовне читання символів з вхідного файлу. Кожен символ порівнюється з набором правил для лексем. Якщо лексема відповідає певному правилу (наприклад, ключове слово або ідентифікатор), вона записується в таблицю лексем разом з її типом та додатковою інформацією.

Лексеми, які не відповідають жодному з правил, позначаються як невизначені (невідомі) лексеми.

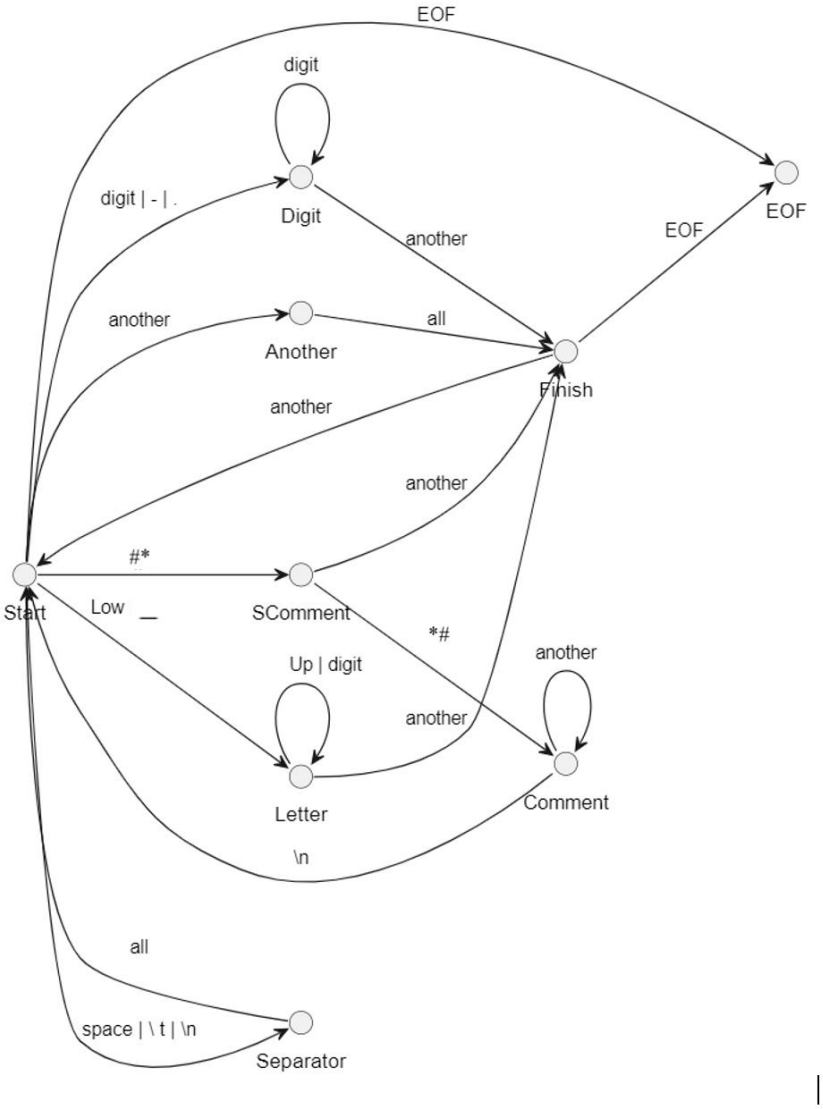


Рис. 3.1. Схема алгоритму роботи лексичного аналізатора

3.3.2. Опис програми реалізації лексичного аналізатора

У програмі для реалізації лексичного аналізу використовуються такі основні компоненти:

1. Типи лексем (Tokens):

- Ключові слова: STARTPROGRAM, STARTBLOCK, VARIABLE, ENDBLOCK, INT16, INPUT, OUTPUT, IF, ELSE, FOR, TO, DOWNTO, DO, WHILE, WEND, REPEAT, UNTIL, DIV, MOD.
- Ідентифікатори: рядок, що починається з символу _, за яким йдуть літери або цифри, максимум 6 символів.
- Числові константи: ціле число.
- Оператори: ==>, +, -, *, =, !=, >>, <<, !!, &&, ||.
- Розділювачі: “,” .
- Дужки: “(“ , ”)“.
- Невідома лексема: символи, що не підпадають під описані правила.

2. Алгоритм роботи лексичного аналізатора:
   - Читання файлу та виділення лексем через функцію tokenize().
   - Визначення типу лексеми за допомогою порівняння з ключовими словами та шаблонами.
   - Формування таблиці лексем m_tokens, де для кожної лексеми вказується її тип, значення та рядок, на якому вона була знайдена.
   - Виявлення лексичних помилок, таких як недопустимі символи, неправильні ідентифікатори або числові константи.
3. Структура даних для зберігання стану аналізатора:

```
enum States {
   Start,      // початковий стан
   Finish,     // кінцевий стан
   Letter,     // опрацювання слів (ключові слова та ідентифікатори)
   Digit,      // опрацювання цифр
   Separator,  // опрацювання роздільників
   Another,    // опрацювання інших символів
   EndOfFile,  // кінець файлу
   SComment,   // початок коментаря
   Comment     // ігнорування коментаря
};
```

4. Функції для лексичного аналізатора:

- unsigned int getTokens(FILE* F): основна функція для отримання лексем з файлу.
- void printTokens(void): друк лексем.
- void fprintTokens(FILE* F): запис лексем у файл.

3.4. Розробка генератора коду.

Синтаксичне дерево в чистому вигляді несе тільки інформацію про структуру програми. Насправді в процесі генерації коду потрібна також інформація про змінні (наприклад, їх адреси), процедури (також адреси, рівні), 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 процедури (адреси, рівні, модулі, в яких процедури описані, і т.д.). При вході в процедуру в таблиці рівнів процедур заводиться новий вхід - вказівник на таблицю описів. При виході вказівник поновлюється на старе значення. Якщо проміжне представлення - дерево, то інформація може зберігатися в вершинах самого дерева.

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 асемблерного 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 асемблерний код відповідно операторам які були в програмі, другий файл містить таблицю змінних. Інформація з

них зчитується в відповідному порядку, основні константні конструкції записуються в файл asm.

3.4.1. **Розробка дерева граматичного розбору.**

Схема дерева розбору виглядає наступним чином:

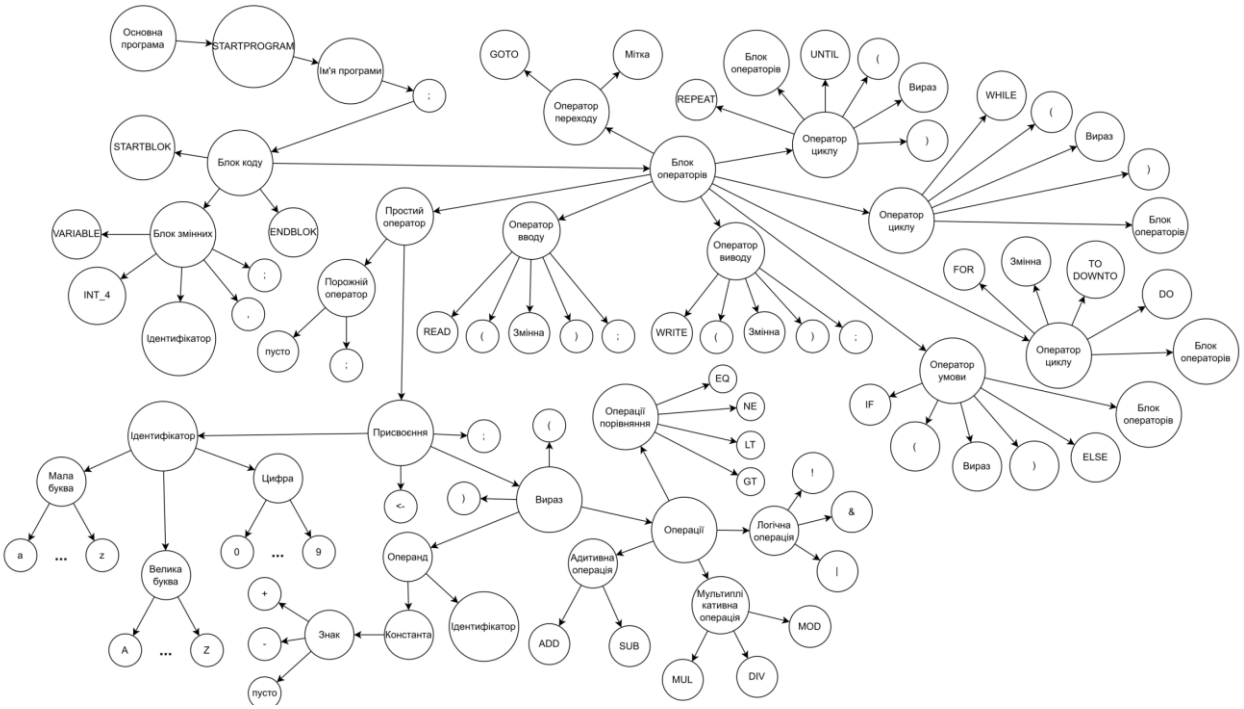


Рис. 3.2. Дерево граматичного розбору.

3.4.1. Розробка алгоритму роботи синтаксичного і 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.

Процес перевірки EBNF в проекті реалізований через систему правил Backus та складається з наступних компонентів:

Базова структура перевірки:

- Інтерфейс IBackusRule визначає базовий контракт для всіх правил.

Політики перевірки правил:

Enum RuleCountPolicy визначає можливі варіанти входження правил:

- NoPolicy – без політики
- Optional – необов'язкове правило
- OnlyOne – тільки один раз
- Several – декілька разів
- OneOrMore – один або більше разів
- PairStart/PairEnd – парні конструкції

Реєстрація та зберігання правил:

- Клас Controller відповідає за реєстрацію правил.
- BackusRuleStorage зберігає зареєстровані правила.

Визначення правил граматики:

- Правила визначаються через BackusRuleItem з вказанням політики.
- Підтримується ієрархічна структура правил.

Процес перевірки:

- Базовий клас BackusRuleBase реалізує базову перевірку типів.
- Клас BackusRule реалізує складну перевірку правил з урахуванням політик.

Обробка помилок:

- Помилки збираються в multimap з інформацією про тип помилки та контекст.
- Кожне правило може генерувати власні помилки.

Цей механізм дозволяє:

- Перевіряти відповідність коду заданій EBNF граматиці.
- Гнучко налаштовувати правила перевірки.
- Отримувати детальну інформацію про помилки.
- Розширювати граматику новими правилами.

Визначимо назви процедур, що відповідають нетерміналам граматики таким чином:

```
void program();  // розбір програми
void programBody();  // розбір тіла програми
void variableDeclaration();  // оголошення змінних
void variableList();  // список змінних
void statement();  // оператори
void inputStatement();  // оператор вводу
void outputStatement();  // оператор виводу
void arithmeticExpression();  // арифметичні вирази
void lowPriorityExpression();  // низький пріоритет виразів
void middlePriorityExpression();  // середній пріоритет виразів
void assignStatement();  // оператор присвоєння
void ifStatement();  // оператор if
void logicalExpression();  // логічні вирази
void andExpression();  // вирази з оператором AND
void comparison();  // порівняння
void comparisonExpression();  // порівняння двох арифметичних виразів
void gotoStatement();  // оператор GOTO
void labelPoint();  // мітка
void forStatement();  // оператор FOR
void whileStatement();  // оператор WHILE
void repeatStatement();  // оператор REPEAT
```

Блок-схема алгоритму роботи синтаксичного аналізатора виглядатиме наступним чином:

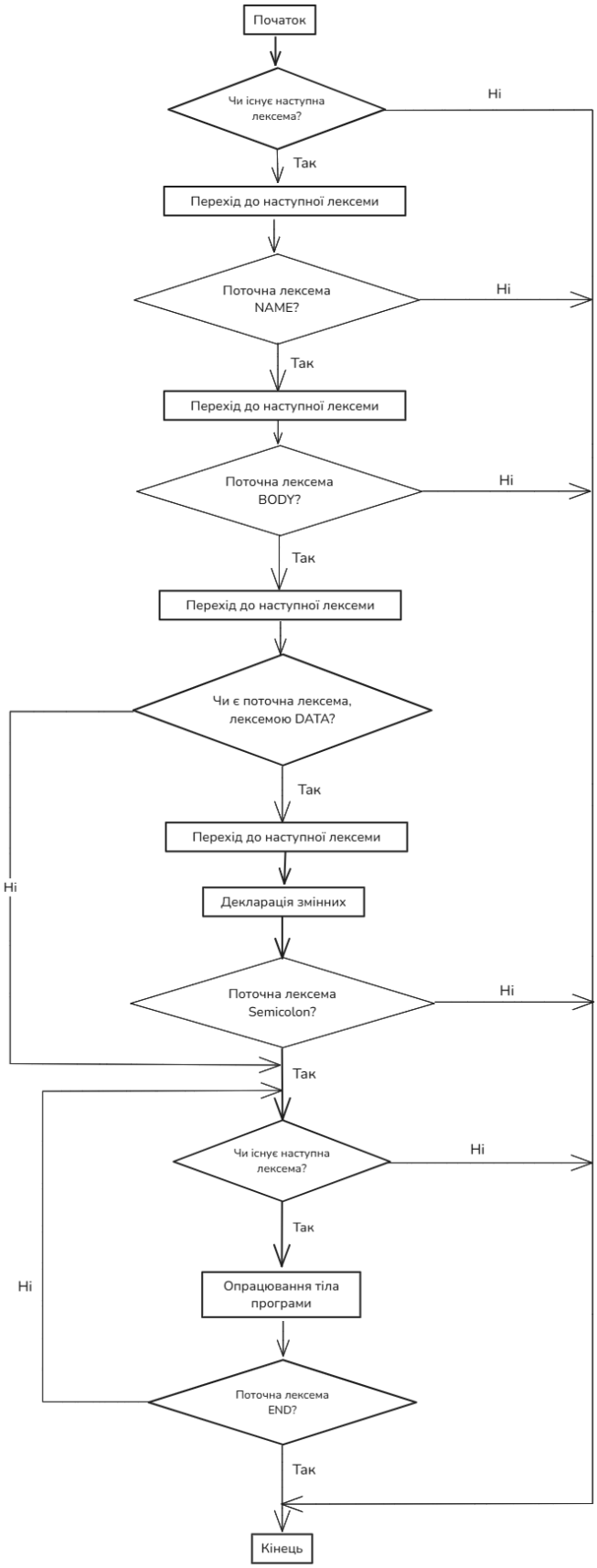


Рис. 3.3. Блок-сема алгоритму роботи синтаксичного аналізатора.

Верхньорівневий код, який описує блок схема 3.3

```
auto topRule = controller->addRule("TopRule", {
        BackusRuleItem({    Program::Type()}, OnlyOne),
        BackusRuleItem({  identRule->type()}, OnlyOne),
        BackusRuleItem({ Symbols::Semicolon}, OnlyOne),
        BackusRuleItem({     Start::Type()}, OnlyOne),
        BackusRuleItem({       Vars::Type()}, OnlyOne),
        BackusRuleItem({   varsBlok->type()}, OnlyOne),
        BackusRuleItem({ operators->type(), operatorsWithSemicolon->type()}, Optional |
OneOrMore),
        BackusRuleItem({       End::Type()}, OnlyOne)
        });
```

Код що перевіряє валідність оголошених змінних

```
std::shared_ptr<IToken> tryCreateToken(std::string& lexeme) const override
   {
      if (lexeme.size() > (m_mask.size() + m_prefix.size()))
         return nullptr;

      bool res = true;
      if (!lexeme.starts_with(m_prefix))
      {
         return nullptr;
      }

      std::string_view ident{ lexeme.begin() + m_prefix.size(), lexeme.end() };
      for (size_t i = 0; i < ident.size(); i++)
      {
         if ((isupper(ident[i]) != isupper(m_mask[i])) && !isdigit(ident[i]))
         {
            res &= false;
            break;
         }
      }

      std::shared_ptr<IToken> token = nullptr;
      if (res)
      {
         token = clone();
         token->setValue(lexeme);
         lexeme.clear();
      }

      return token;
   };
```

І приватні поля що задають формат:

```
const std::string m_prefix = "_";
const std::string m_mask = "xXXXXXX";
```

3.4.2. Опис програми реалізації генератора коду.

У компілятора, реалізованого в даному курсовому проекті, вихідна мова - програма на мові Assembler. Ця програма записується у файл, що має таку ж саму назву, як і файл з вхідним текстом, але розширення “asm”. 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 ініціалізація сегменту даних. Далі виконується аналіз коду, та визначаються процедури, зміні, які використовуються.

Проаналізувавши змінні, які є у програмі, генератор формує код даних для асемблерної програми. Для цього з таблиці лексем вибирається ім’я змінної (типи змінних відповідають 4 байтам), та записується 0, в якості початкового значення.

Аналіз наявних процедур необхідний у зв’язку з тим, що процедури введення/виведення, виконання арифметичних та логічних операцій, виконано у вигляді окремих процедур 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 процедури виведення, попередньо записавши у співпроцесор значення, яке необхідно вивести. Якщо це арифметична операція, так само викликається дана процедура, але як і в попередньому випадку, спочатку у регістри співпроцесора записується інформація, яка вказує над якими значеннями виконувати дії.

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

В кінці своє роботи, генератор формує код завершення ассемблерної програми.

3.4.3. Розробка алгоритму роботи семантичного аналізатора

На етапі семантичного аналізу вирішується завдання ідентифікації ідентифікаторів. Алгоритм складається з двох частин:

- Обробка оголошень ідентифікаторів.
- Обробка використання ідентифікаторів.

Коли лексичний аналізатор виявляє чергову лексему, що є ідентифікатором, він формує структуру з атрибутами, такими як ім'я, тип і лексичний клас. Ця інформація передається семантичному аналізатору. Якщо обробляється оголошення ідентифікатора, основним завданням є запис інформації до таблиці ідентифікаторів.

При обробці використання ідентифікатора семантичний аналізатор використовує раніше створену таблицю ідентифікаторів. Для отримання даних про тип ідентифікатора необхідно прочитати відповідне поле цієї таблиці.

3.4.4. Опис програмної реалізації семантичного аналізатора

Семантичний аналізатор забезпечує перевірку правильності структури та логіки програми, аналізуючи лексеми та граматику. Реалізація включає кілька ключових функцій.

Основні аспекти реалізації:

1. **Лексеми та граматика**
   Семантичний аналізатор працює з таблицею лексем і граматикою, які є результатом лексичного та синтаксичного аналізу. Типи лексем визначаються полем type, а функція GetTypeName використовується для отримання назви типу.
2. **Перевірка конфліктів**
   Виявляються помилки в ідентифікаторах, щоб уникнути неоднозначностей і забезпечити коректність виконання програми.

3. **Обробка помилок**
   Усі знайдені помилки виводяться до консолі за допомогою механізму semErr.
4. **Рекурсивна перевірка правил**
   Аналізатор підтримує рекурсивну перевірку граматичних правил, обробку необов’язкових конструкцій, парних елементів і використання політик для визначення кількості правил (RuleCountPolicy).
5. **Виконання**
   Функція CheckSemantic відповідає за запуск семантичного аналізу, використовуючи об'єкт Context для зберігання поточного стану.

```
bool CheckSemantic(std::ostream& out, std::list<std::shared_ptr<T>>& tokens)
{
    auto endOfFileType = tokens.back()->type();
    std::list<std::shared_ptr<IBackusRule>> rules;
    for (auto token : tokens)
    {
        if (auto rule = std::dynamic_pointer_cast<IBackusRule>(token))
            rules.push_back(rule);
    }

    auto it = rules.begin();
    auto end = rules.end();
    std::multimap<int, std::pair<std::string, std::vector<std::string>>> errors;
    auto res = Controller::Instance()->topRule()->check(errors, it, end);

    rules.erase(++std::find_if(it, rules.end(), [&endOfFileType](const auto& rule) {
return rule->type() == endOfFileType; }), rules.end());
    end = --rules.end();

    std::multimap<int, std::string> errorsMsg;

    int lexErr = 0;
    int synErr = 0;
    int semErr = 0;

    tokens.clear();
    for (auto rule : rules)
    {
        tokens.push_back(std::dynamic_pointer_cast<T>(rule));
        if (rule->type() == Undefined::Type())
        {
            res = false;
            std::string err;
            if (auto erMsg = rule->customData("error"); !erMsg.empty())
            {
                semErr++;
                err = "Semantic error: " + erMsg;
            }
            else
            {
                semErr++;
                err = std::format("Semantic error: Undefined token: {}", rule-
>value());
            }
            errorsMsg.emplace(rule->line(), err);
        }
        else if (rule->type() == token::Unknown::Type())
        {
            lexErr++;
            res = false;
```

```
                errorsMsg.emplace(rule->line(), std::format("Lexical error: Unknown token:
{}", rule->value()));
            }
        }
```

Код який опрацьовує оголошення та використання ідентифікаторів, додає інформацію про ідентифікатор у таблицю ідентифікаторів

```
    identRule->setPostHandler([context](BackusRuleList::iterator&,
            BackusRuleList::iterator& it,
            BackusRuleList::iterator& end)
        {
            static bool isFirstIdentChecked = !context->IsFirstProgName();
            auto isVarBlockChecked = context->IsVarBlockChecked();
            auto& identTable = context->IdentTable();

            auto identIt = std::prev(it, 1);
            if (isVarBlockChecked)
            {
                if (!identTable.contains((*identIt)->value()))
                {
                    auto undef = std::make_shared<Undefined>();
                    undef->setValue((*identIt)->value());
                    undef->setLine((*identIt)->line());
                    undef->setCustomData((*identIt)->customData());
                    *identIt = undef;
                }
            }
            else
            {
                if (isFirstIdentChecked)
                {
                    identTable.insert((*identIt)->value());
                }
                else
                {
                    auto progName = std::make_shared<ProgramName>();
                    progName->setValue((*identIt)->value());
                    progName->setLine((*identIt)->line());
                    progName->setCustomData((*identIt)->customData());
                    *identIt = progName;
                    isFirstIdentChecked = true;
                }}
            (*identIt)->setCustomData((*identIt)->value() + "_");
        });
        return identRule;}
```

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 (наприклад, їх адреси), процедури (також адреси, рівні), 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 процедури (адреси, рівні, модулі, в яких процедури описані, і т.д.). При вході в процедуру в таблиці рівнів процедур заводиться новий вхід - вказівник на таблицю описів. При виході вказівник поновлюється на старе значення. Якщо проміжне представлення - дерево, то інформація може зберігатися в вершинах самого дерева.

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 асемблерного 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 асемблерний код відповідно операторам які були в програмі, другий файл містить таблицю змінних. Інформація з них зчитується в відповідному порядку, основні константні конструкції записуються в файл asm.

Розробка алгоритму роботи генератора коду

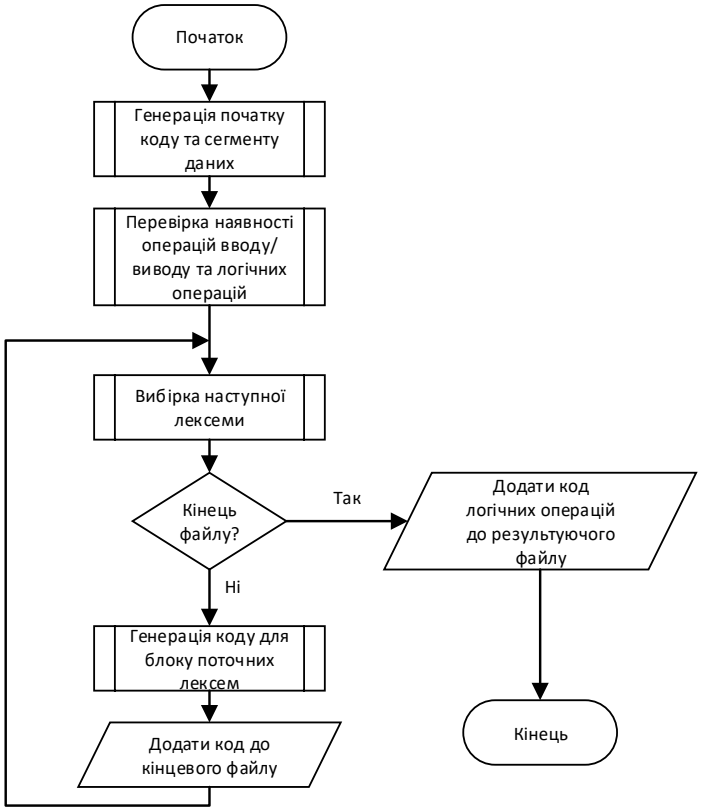


Рис. 3.5 Блок схема генератора коду

3.5.2. Опис програми реалізації генератора коду

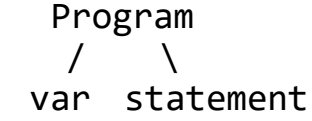
Основні особливості реалізації:

1. **Архітектура:**
   - Використовується патерн Singleton через клас `Generator`.
   - Базується на шаблоні Visitor: кожен токен або правило має метод `genCode()`.
   - Використовує `GeneratorDetails` для зберігання налаштувань і допоміжних даних.
2. **Етапи генерації:**
   - Генерація сегмента даних.
   - Генерація сегмента коду.
   - Створення процедур.
   - Завершення програми.
3. **Технічні особливості:**
   - Обчислення виконуються за стековою архітектурою.
   - Підтримується постфіксна форма виразів.
   - Для управління потоком виконання використовується система міток:
     - Унікальні мітки для циклів та умов.

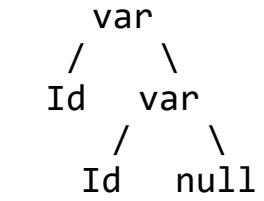
- Іменовані мітки для операторів `GOTO`.

4. **Оптимізації:**
   - Мінімізується використання регістрів через стекову модель.
   - Процедури перевикористовуються завдяки механізму реєстрації.
   - Генерація коду оптимізується для простих конструкцій.
5. **Обробка даних:**
   - Підтримуються числові й рядкові типи.
   - Для введення/виведення використовується Windows API.
   - Передбачена система форматування для різних типів даних.
6. **Розширюваність:**
   - Легке додавання нових операторів через систему токенів.
   - Реєстрація користувацьких процедур.
   - Гнучкі налаштування через `GeneratorDetails`.

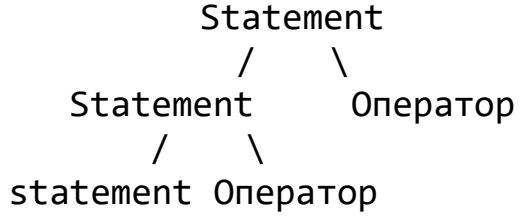
Програма має вигляд:



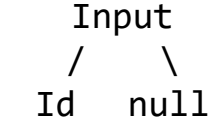
Оголошення змінних:



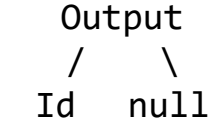
Тіло програми:



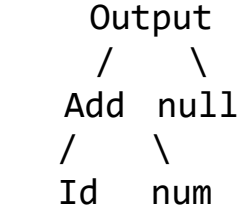
Оператор вводу:



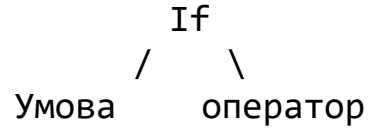
Оператор виводу:



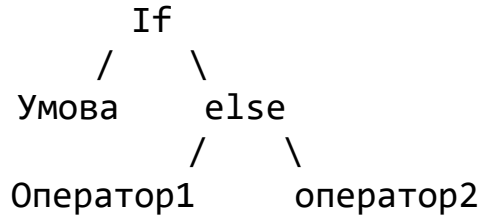
Також оператор виводу може мати за лівого нащадка різні арифметичні вирази, наприклад:



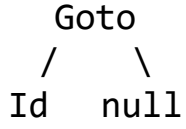
Умовний оператор (IF() оператор;):



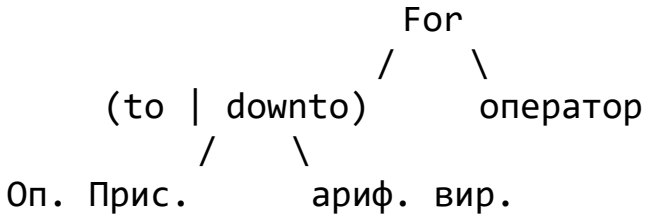
Умовний оператор (IF() оператор1; else оператор2;):



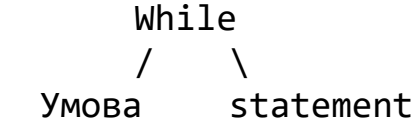
Оператор безумовного переходу:

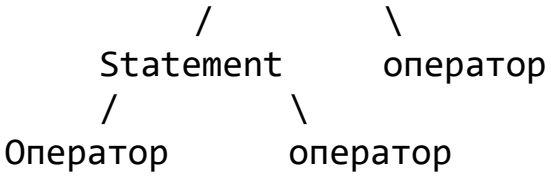


Оператор циклу for:

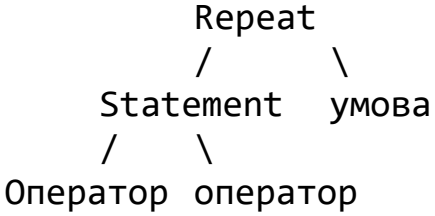


Оператор циклу while:

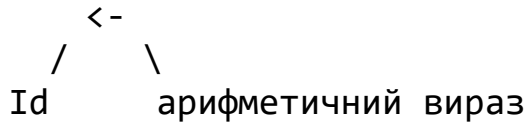




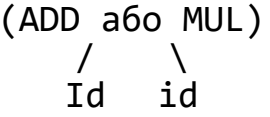
Оператор циклу repeat:



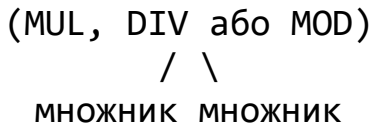
Оператор присвоєння:



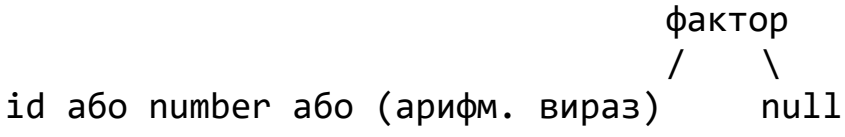
Арифметичний вираз:



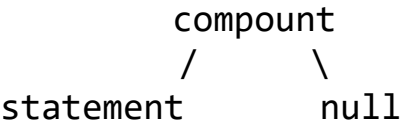
Доданок:



Множник:



Складений оператор:



Дана програма написана мовою C++ з при розробці якої було створено структури `BackusRule` та `BackusRuleItem` за допомогою яких можна чітко описати нотатки Бекуса-Наура, які використовуються для семантично-лексичного аналізу написаної програми для заданої мови програмування

```
auto assingmentRule = BackusRule::MakeRule("AssignmentRule", {
    BackusRuleItem({ identRule->type()}, OnlyOne),
    BackusRuleItem({Assignment::Type()}, OnlyOne),
```

```
    BackusRuleItem({  equation->type()}, OnlyOne)
    });

auto read = BackusRule::MakeRule("ReadRule", {
    BackusRuleItem({      Read::Type()}, OnlyOne),
    BackusRuleItem({   LBraket::Type()}, OnlyOne),
    BackusRuleItem({ identRule->type()}, OnlyOne),
    BackusRuleItem({   RBraket::Type()}, OnlyOne)
    });

auto write = BackusRule::MakeRule("WriteRule", {
    BackusRuleItem({     Write::Type()}, OnlyOne),
    BackusRuleItem({   LBraket::Type()}, OnlyOne | PairStart),
    BackusRuleItem({ stringRule->type(), equation->type() }, OnlyOne),
    BackusRuleItem({   RBraket::Type()}, OnlyOne | PairEnd)
    });

auto codeBlok = BackusRule::MakeRule("CodeBlok", {
    BackusRuleItem({     Start::Type()}, OnlyOne),
    BackusRuleItem({ operators->type(), operatorsWithSemicolon->type()}, Optional |
OneOrMore),
    BackusRuleItem({       End::Type()}, OnlyOne)
    });

auto topRule = BackusRule::MakeRule("TopRule", {
    BackusRuleItem({   Program::Type()}, OnlyOne),
    BackusRuleItem({ identRule->type()}, OnlyOne),
    BackusRuleItem({ Semicolon::Type()}, OnlyOne),
    BackusRuleItem({      Vars::Type()}, OnlyOne),
    BackusRuleItem({  varsBlok->type()}, OnlyOne),
    BackusRuleItem({  codeBlok->type()}, OnlyOne)
    });
```

Вище наведено приклад опису нотаток Бекуса-Наура за допомогою цих структур. Наприклад `topRule` це правило, що відповідає за правильну структуру написаної програми, тобто якими лексемами вона повинна починатись та які операції можуть бути використанні всередині виконавчого блоку програми.

Всередині структури `BackusRule` описаний порядок токенів для певного правила. А в структурі `BackusRuleItem` описані токени, які при перевірці трактуються програмою як «АБО», тобто повинен бути лише один з описаних токенів. Наприклад для `write` послідовно необхідний токен Write після якого йде ліва дужка, далі може бути або певний вираз або рядок тексту який необхідно вивести. І закінчується правило токеном правої дужки.

Основна частина програми складається з 3 компонентів: парсера лексем, правил Бекуса-Наура та генератора асемблерного коду. Кожен з цих компонентів працює зі власним інтерфейсом на певному етапі виконання програми.

Кожен токен це окремий клас що наслідує 3 інтерфейси:

- `IToken`
- `IBackusRule`
- `IGeneratorItem`

Наявність наслідування цих інтерфейсів кожним токеном дозволяє без проблем звертатись до кожного віддільного токена на усіх етапах виконання програми

Для процесу парсингу програми використовується інтерфейс `IToken`. Що дозволяє простіше з точки зору реалізації звертатись до токенів при аналізі вхідної програми.

Правила Бекуса-Наура для своєї роботи використовують інтерфейс `IBackusRule`. Це дозволяє викликати функцію перевірки `check` до кожного прописаного у коді правила запису як програми в цілому так і кожного віддільної операції, що спрощує подальший пошук ймовірних помилок у коді програми, яка буде транслюватись у асемблерний код.

Інтерфейс `IGeneratorItem` використовується генератором асемблерного коду при трансляції вхідної програми. Оскільки кожен токен є віддільним класом, то у ньому була реалізована функція `genCode` яка використовується генератором, що дозволяє записати необхідний асемблерний код який буде згенерований певним токеном. Наприклад:

Для класу та токену `Greate` що визначає при порівнянні який елемент більший, функція генерації відповідного коду виглядає наступним чином:

```
void genCode(std::ostream& out, GeneratorDetails& details,
    std::list<std::shared_ptr<IGeneratorItem>>::iterator& it,
    const std::list<std::shared_ptr<IGeneratorItem>>::iterator& end) const final
{
    RegPROC(details);
    out << "\tcall Greate_\n";
      };
```

За допомогою функції `RegPROC` токен за потреби реєструє процедуру у генераторі.

```
static void RegPROC(GeneratorDetails& details)
{
    if (!IsRegistered())
    {
        details.registerProc("Greate_", PrintGreate);
        SetRegistered();
    }
      }

static void PrintGreate(std::ostream& out, const GeneratorDetails::GeneratorArgs& args)
{
    out << ";===Procedure
Greate=============================================================================\n";
    out << "Greate_ PROC\n";
    out << "\tpushf\n";
    out << "\tpop cx\n\n";
    out << "\tmov " << args.regPrefix << "ax, [esp + " << args.posArg0 << "]\n";
    out << "\tcmp " << args.regPrefix << "ax, [esp + " << args.posArg1 << "]\n";
    out << "\tjle greate_false\n";
    out << "\tmov " << args.regPrefix << "ax, 1\n";
    out << "\tjmp greate_fin\n";
    out << "greate_false:\n";
    out << "\tmov " << args.regPrefix << "ax, 0\n";
    out << "greate_fin:\n";
    out << "\tpush cx\n";
```

```
    out << "\tpopf\n\n";
    GeneratorUtils::PrintResultToStack(out, args);
    out << "\tret\n";
    out << "Greate_ ENDP\n";
    out <<
";=============================================================================
===\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.

Генератор у свою чергу буде більш оптимізовано генерувати асемблерний код, створюючи код лише тих операцій, що буди використані у вхідній програмі.

4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА

Дана програма написана мовою C++ з при розробці якої було створено структури `BackusRule` та `BackusRuleItem` за допомогою яких можна чітко описати нотатки Бекуса-Наура, які використовуються для семантично-лексичного аналізу написаної програми для заданої мови програмування

```
auto assingmentRule = BackusRule::MakeRule("AssignmentRule", {

    BackusRuleItem({ identRule->type()}, OnlyOne),

    BackusRuleItem({Assignment::Type()}, OnlyOne),

    BackusRuleItem({  equation->type()}, OnlyOne)

    });


auto read = BackusRule::MakeRule("ReadRule", {

    BackusRuleItem({      Read::Type()}, OnlyOne),

    BackusRuleItem({   LBraket::Type()}, OnlyOne),

    BackusRuleItem({ identRule->type()}, OnlyOne),

    BackusRuleItem({   RBraket::Type()}, OnlyOne)
```

```
        });


    auto write = BackusRule::MakeRule("WriteRule", {
        BackusRuleItem({     Write::Type()}, OnlyOne),
        BackusRuleItem({   LBraket::Type()}, OnlyOne | PairStart),
        BackusRuleItem({ stringRule->type(), equation->type() }, OnlyOne),
        BackusRuleItem({   RBraket::Type()}, OnlyOne | PairEnd)
        });


    auto codeBlok = BackusRule::MakeRule("CodeBlok", {
        BackusRuleItem({     Start::Type()}, OnlyOne),
        BackusRuleItem({  operators->type(),  operatorsWithSemicolon->type()},  Optional  |
OneOrMore),
        BackusRuleItem({       End::Type()}, OnlyOne)
        });


    auto topRule = BackusRule::MakeRule("TopRule", {
        BackusRuleItem({   Program::Type()}, OnlyOne),
        BackusRuleItem({ identRule->type()}, OnlyOne),
        BackusRuleItem({ Semicolon::Type()}, OnlyOne),
        BackusRuleItem({      Vars::Type()}, OnlyOne),
        BackusRuleItem({  varsBlok->type()}, OnlyOne),
        BackusRuleItem({  codeBlok->type()}, OnlyOne)
        });
```

Вище наведено приклад опису нотаток Бекуса-Наура за допомогою цих структур. Наприклад `topRule` це правило, що відповідає за правильну структуру написаної програми, тобто якими лексемами вона повинна починатись та які операції можуть бути використанні всередині виконавчого блоку програми.

Всередині структури `BackusRule` описаний порядок токенів для певного правила. А в структурі `BackusRuleItem` описані токени, які при перевірці трактуються програмою як «АБО», тобто повинен бути лише один з описаних токенів. Наприклад для `write` послідовно необхідний токен Write після якого йде ліва дужка, далі може бути або певний вираз або рядок тексту який необхідно вивести. І закінчується правило токеном правої дужки.

Основна частина програми складається з 3 компонентів: парсера лексем, правил Бекуса-Наура та генератора асемблерного коду. Кожен з цих компонентів працює зі власним інтерфейсом на певному етапі виконання програми.

Кожен токен це окремий клас що наслідує 3 інтерфейси:

- `IToken`
- `IBackusRule`
- `IGeneratorItem`

Наявність наслідування цих інтерфейсів кожним токеном дозволяє без проблем звертатись до кожного віддільного токена на усіх етапах виконання програми

Для процесу парсингу програми використовується інтерфейс `IToken`. Що дозволяє простіше з точки зору реалізації звертатись до токенів при аналізі вхідної програми.

Правила Бекуса-Наура для своєї роботи використовують інтерфейс `IBackusRule`. Це дозволяє викликати функцію перевірки `check` до кожного прописаного у коді правила запису як програми в цілому так і кожного віддільної операції, що спрощує подальший пошук ймовірних помилок у коді програми, яка буде транслюватись у асемблерний код.

Інтерфейс `IGeneratorItem` використовується генератором асемблерного коду при трансляції вхідної програми. Оскільки кожен токен є віддільним класом, то у ньому була реалізована функція `genCode` яка використовується генератором, що дозволяє записати необхідний асемблерний код який буде згенерований певним токеном. Наприклад:

Для класу та токену `Greate` що визначає при порівнянні який елемент більший, функція генерації відповідного коду виглядає наступним чином:

```
void genCode(std::ostream& out, GeneratorDetails& details,
    std::list<std::shared_ptr<IGeneratorItem>>::iterator& it,
    const std::list<std::shared_ptr<IGeneratorItem>>::iterator& end) const final
{
    RegPROC(details);
    out << "\tcall Greate_\n";
};
```

За допомогою функції `RegPROC` токен за потреби реєструє процедуру у генераторі.

```
static void RegPROC(GeneratorDetails& details)
{
    if (!IsRegistered())
    {
        details.registerProc("Greate_", PrintGreate);
        SetRegistered();
    }
}

static void PrintGreate(std::ostream& out, const GeneratorDetails::GeneratorArgs&
args)
{
    out << ";===Procedure
Greate=============================================================================\n";
    out << "Greate_ PROC\n";
    out << "\tpushf\n";
```

```
        out << "\tpop cx\n\n";

        out << "\tmov " << args.regPrefix << "ax, [esp + " << args.posArg0 << "]\n";

        out << "\tcmp " << args.regPrefix << "ax, [esp + " << args.posArg1 << "]\n";

        out << "\tjle greate_false\n";

        out << "\tmov " << args.regPrefix << "ax, 1\n";

        out << "\tjmp greate_fin\n";

        out << "greate_false:\n";

        out << "\tmov " << args.regPrefix << "ax, 0\n";

        out << "greate_fin:\n";

        out << "\tpush cx\n";

        out << "\tpopf\n\n";

        GeneratorUtils::PrintResultToStack(out, args);

        out << "\tret\n";

        out << "Greate_ ENDP\n";

        out                                                                                    <<
";=============================================================================================
===\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.

Генератор у свою чергу буде більш оптимізовано генерувати асемблерний код, створюючи код лише тих операцій, що буди використані у вхідній програмі.

4.1. Опис інтерфейсу та інструкції користувачу.

Вхідним файлом для даної програми є звичайний текстовий файл з розширенням m20. 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_m20.exe <ім’я програми>.m20"

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>.asm.

Для отримання виконавчого файлу необхідно скористатись програмою Masm32.exe

4.2. Виявлення лексичних і синтаксичних помилок.

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

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

Приклад виявлення:

Текст програми з помилками

```
#*Prog1*#

STARTPROGRAM
VARIABLE INT_4 _aa,_bbb,_xxx,_yyy;
STARTBLOCK

READ _aaa
READ _bbb

WRIE _aaa ADD _bbb
WRITE _aaa SUB _bbb
WRITE _aaa MUL _bbb
WRITE _aaa DIV _bbb
WRITE _aaa MOD _bbb

_xxx <- (_aaa SUB _bbb) MUL 10 ADD (_aaa ADD _bbb) DIV 10
_yyy <- _xxx ADD (_xxx MOD 10)

WRITE _xxx
WRITE _yyy

ENDBLOCK
```

Текст файлу з повідомленнями про помилки

Syntax error in line 8, token number: 17 : another type of lexeme was expected (expected: Assign | current: Identifier).

4.3. Перевірка роботи транслятора за допомогою тестових задач.

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

Текст коректної програми

```
STARTPROGRAM

VARIABLE INT_4 _aaa,_bbb,_xxx,_yyy;

STARTBLOCK


READ _aaa

READ _bbb

WRITE _aaa ADD _bbb

WRITE _aaa SUB _bbb

WRITE _aaa MUL _bbb

WRITE _aaa DIV _bbb

WRITE _aaa MOD _bbb


_xxx <- (_aaa SUB _bbb) MUL 10 ADD (_aaa ADD _bbb) DIV 10

_yyy <- _xxx ADD (_xxx MOD 10)


WRITE _xxx

WRITE _yyy

ENDBLOCK
```

Дерево виводу для тестової програми:

```
|-- Program(0)
|   |-- var(2)
|   |   |-- _yyy(1)
|   |   |-- var(2)
|   |   |   |-- _xxx(1)
|   |   |   |-- var(2)
|   |   |   |   |-- _bbb(1)
|   |   |   |   |-- var(2)
|   |   |   |   |   |-- _aaa(1)
|   |-- statement(3)
|   |   |-- statement(3)
|   |   |   |-- statement(3)
|   |   |   |   |-- statement(3)
|   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- Sub(7)
|   |   |   |   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |-- Mul(8)
|   |   |   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |-- Div(9)
|   |   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |-- Mod(10)
|   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |-- _xxx(1)
|   |   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |   |-- Mul(8)
```

```
|   |   |   |   |   |   |   |   |-- Sub(7)
|   |   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |   |   |-- 10(11)
|   |   |   |   |   |   |   |   |-- Div(9)
|   |   |   |   |   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |   |   |-- 10(11)
|   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |-- _yyy(1)
|   |   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |   |-- _xxx(1)
|   |   |   |   |   |   |   |-- Mod(10)
|   |   |   |   |   |   |   |   |-- _xxx(1)
|   |   |   |   |   |   |   |   |-- 10(11)
|   |   |   |   |-- output(5)
|   |   |   |   |   |-- _xxx(1)
|   |   |   |-- output(5)
|   |   |   |   |-- _yyy(1)
```

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

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

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

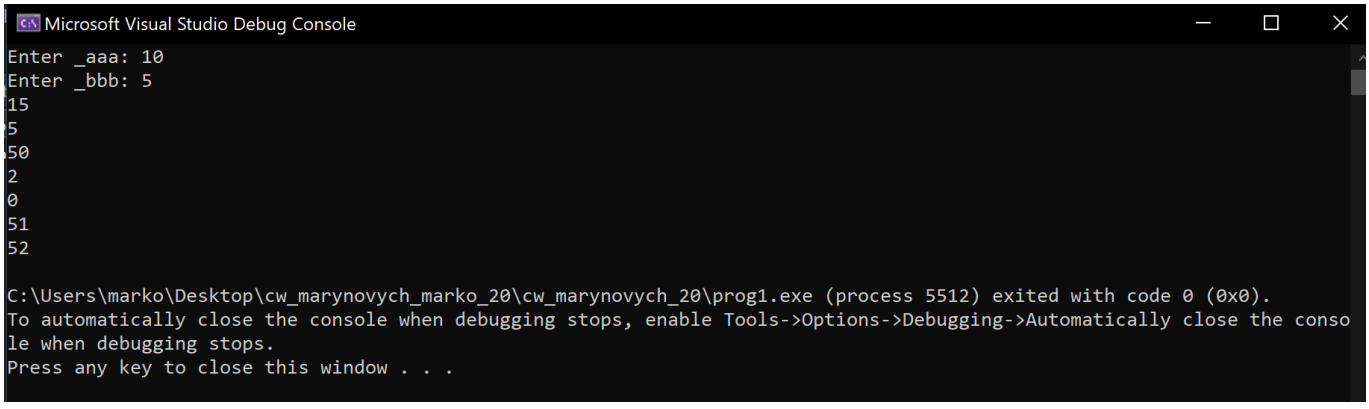


Рис. 4.1 Результат виконання коректної програми

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

Тестова програма №2

Текст програми

```
#*Prog2*#

STARTPROGRAM
VARIABLE INT_4 _aaa,_bbb,_ccc;
STARTBLOCK
READ _aaa
READ _bbb
READ _ccc
IF (_aaa LT _bbb)
   IF(_aaa LT _ccc)
      WRITE _aaa
   ELSE
      WRITE _ccc
ELSE
   IF (_bbb LT _ccc)
         WRITE _bbb
      ELSE
         WRITE _ccc
IF ((_aaa EQ _bbb) & (_aaa EQ _ccc) & (_bbb EQ _ccc))
   WRITE 1
ELSE
   WRITE 0
IF ((_aaa GT 0) | (_bbb GT 0) | (_ccc GT 0))
   WRITE -1
ELSE
   WRITE 0
IF (!(_aaa GT (_bbb ADD _ccc)))
   WRITE 10
ELSE
   WRITE 0
ENDBLOCK
```

Дерево виводу для тестової програми:

```
|-- Program(0)
|   |-- var(2)
|   |   |-- _ccc(1)
|   |   |-- var(2)
|   |   |   |-- _bbb(1)
|   |   |   |-- var(2)
|   |   |   |   |-- _aaa(1)
|   |-- statement(3)
|   |   |-- statement(3)
|   |   |   |-- statement(3)
|   |   |   |   |-- statement(3)
|   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |-- _ccc(1)
|   |   |   |   |   |-- if(13)
|   |   |   |   |   |   |-- Less(21)
|   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |-- else(14)
|   |   |   |   |   |   |   |-- if(13)
|   |   |   |   |   |   |   |   |-- Less(21)
|   |   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |   |-- _ccc(1)
|   |   |   |   |   |   |   |   |-- else(14)
|   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |-- _ccc(1)
|   |   |   |   |   |   |   |-- if(13)
|   |   |   |   |   |   |   |   |-- Less(21)
|   |   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |   |   |-- _ccc(1)
|   |   |   |   |   |   |   |   |-- else(14)
|   |   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |   |   |   |-- _ccc(1)
|   |   |   |   |-- if(13)
|   |   |   |   |   |-- and(16)
|   |   |   |   |   |   |-- Equality(18)
|   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |-- and(16)
|   |   |   |   |   |   |   |-- Equality(18)
|   |   |   |   |   |   |   |   |-- _aaa(1)
```

```
|   |   |   |   |   |   |   |   |-- _ccc(1)
|   |   |   |   |   |   |   |-- Equality(18)
|   |   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |   |-- _ccc(1)
|   |   |   |   |   |-- else(14)
|   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |-- 1(11)
|   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |-- 0(11)
|   |   |   |-- if(13)
|   |   |   |   |-- or(15)
|   |   |   |   |   |-- Greate(20)
|   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |-- 0(11)
|   |   |   |   |   |-- or(15)
|   |   |   |   |   |   |-- Greate(20)
|   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |   |-- 0(11)
|   |   |   |   |   |   |-- Greate(20)
|   |   |   |   |   |   |   |-- _ccc(1)
|   |   |   |   |   |   |   |-- 0(11)
|   |   |   |   |-- else(14)
|   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |-- -1(11)
|   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |-- 0(11)
|   |   |-- if(13)
|   |   |   |-- not(17)
|   |   |   |   |-- Greate(20)
|   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |-- _ccc(1)
|   |   |   |-- else(14)
|   |   |   |   |-- output(5)
|   |   |   |   |   |-- 10(11)
|   |   |   |   |-- output(5)
|   |   |   |   |   |-- 0(11)
```

Результат виконання

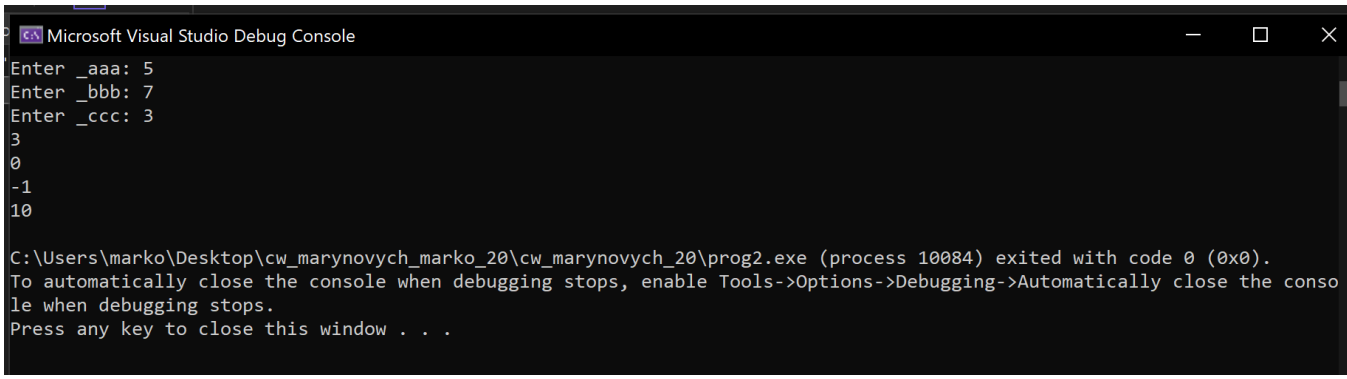


Рис. 4.3 Результат виконання тестової програми №2

Тестова програма №3

Текст програми

```
#*Prog3*#

STARTPROGRAM
VARIABLE INT_4 _aaa,_bbb,_xxx,_iii,_jjj;
STARTBLOCK

READ _aaa
READ _bbb
FOR _xxx <- _aaa TO _bbb DO
   WRITE _xxx MUL _xxx

_xxx <- 0
FOR _iii <- 1 TO _aaa DO
   FOR _jjj <- 1 TO _bbb DO
      _xxx <- _xxx ADD 1

WRITE _xxx

ENDBLOCK
```

Дерево виводу для тестової програми:

```
|-- Program(0)
|   |-- var(2)
|   |   |-- _jjj(1)
|   |   |-- var(2)
|   |   |   |-- _iii(1)
|   |   |   |-- var(2)
|   |   |   |   |-- _xxx(1)
|   |   |   |   |-- var(2)
|   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |-- var(2)
|   |   |   |   |   |   |-- _aaa(1)
|   |-- statement(3)
|   |   |-- statement(3)
|   |   |   |-- statement(3)
|   |   |   |   |-- statement(3)
|   |   |   |   |   |-- statement(3)
|   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |-- input(4)
|   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |-- for(24)
|   |   |   |   |   |   |-- to(25)
|   |   |   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |   |   |-- _xxx(1)
|   |   |   |   |   |   |   |   |-- _aaa(1)
|   |   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |   |-- output(5)
|   |   |   |   |   |   |   |-- Mul(8)
|   |   |   |   |   |   |   |   |-- _xxx(1)
|   |   |   |   |   |   |   |   |-- _xxx(1)
|   |   |   |   |-- assign(12)
|   |   |   |   |   |-- _xxx(1)
|   |   |   |   |   |-- 0(11)
|   |   |   |-- for(24)
|   |   |   |   |-- to(25)
|   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |-- _iii(1)
|   |   |   |   |   |   |-- 1(11)
|   |   |   |   |   |-- _aaa(1)
|   |   |   |   |-- for(24)
|   |   |   |   |   |-- to(25)
|   |   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |   |-- _jjj(1)
|   |   |   |   |   |   |   |-- 1(11)
|   |   |   |   |   |   |-- _bbb(1)
|   |   |   |   |   |-- assign(12)
|   |   |   |   |   |   |-- _xxx(1)
```

```
|   |   |   |   |   |   |-- Add(6)
|   |   |   |   |   |   |   |-- _xxx(1)
|   |   |   |   |   |   |   |-- 1(11)
|   |   |-- output(5)
|   |   |   |-- _xxx(1)
```

Результат виконання

Рис. 4.4 Результат виконання тестової програми №3

ВИСНОВКИ

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування m20, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створено компілятор мови програмування m20, 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 m20. Вихідним кодом генератора є програма на мові Assembler(x86).

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 m20 були успішно виявлені і відповідно оброблені.

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

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 - «Комп'ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. - Київ: КПІ ім. Ігоря Сікорського, 2021. - 108 с.
2. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. - Харків: НТУ «ХПІ», 2021. - 133 с.
3. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії формальних мов: Навчальний посібник у двох частинах. - Чернівці: ЧНУ, 2008. - 84 с.
4. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. - Чернівці: ЧНУ, 2008. - 84 с.
5. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. - 1038 с.
6. Системне програмування (курсовий проект) [Електронний ресурс] - Режим доступу до ресурсу: https://vns.lpnu.ua/course/view.php?id=11685.

MIT OpenCourseWare. Computer Language Engineering [Електронний ресурс] - Режим доступу до ресурсу: https://ocw.mit.edu/courses/6-035-computer- language-engineering-spring-2010.

ДОДАТКИ

Додаток А (Таблиці лексем)

Програма 1

```
-------------------------------------------------------------------
| TOKEN TABLE                                                     |
-------------------------------------------------------------------
| line number | token          | value      | token code | type of token |
-------------------------------------------------------------------
|           1 |    STARTPROGRAM |          0 |          0 | StartProgram  |
-------------------------------------------------------------------
|           2 |        VARIABLE |          0 |          2 | Variable      |
-------------------------------------------------------------------
|           2 |          INT_4 |          0 |          3 | Type          |
-------------------------------------------------------------------
|           2 |           _aaa |          0 |         20 | Identifier    |
-------------------------------------------------------------------
|           2 |              , |          0 |         40 | Comma         |
-------------------------------------------------------------------
|           2 |           _bbb |          0 |         20 | Identifier    |
-------------------------------------------------------------------
|           2 |              , |          0 |         40 | Comma         |
-------------------------------------------------------------------
|           2 |           _xxx |          0 |         20 | Identifier    |
-------------------------------------------------------------------
|           2 |              , |          0 |         40 | Comma         |
-------------------------------------------------------------------
|           2 |           _yyy |          0 |         20 | Identifier    |
-------------------------------------------------------------------
|           2 |              ; |          0 |         38 | Semicolon     |
-------------------------------------------------------------------
|           3 |     STARTBLOCK |          0 |          1 | StartBlock    |
-------------------------------------------------------------------
|           5 |           READ |          0 |          5 | Input         |
-------------------------------------------------------------------
```

5	_aaa	0	20	Identifier
6	READ	0	5	Input
6	_bbb	0	20	Identifier
8	WRITE	0	6	Output
8	_aaa	0	20	Identifier
8	ADD	0	24	Add
8	_bbb	0	20	Identifier
9	WRITE	0	6	Output
9	_aaa	0	20	Identifier
9	SUB	0	25	Sub
9	_bbb	0	20	Identifier
10	WRITE	0	6	Output
10	_aaa	0	20	Identifier
10	MUL	0	26	Mul
10	_bbb	0	20	Identifier
11	WRITE	0	6	Output
11	_aaa	0	20	Identifier
11	DIV	0	27	Div

11	_bbb	0	20	Identifier
12	WRITE	0	6	Output
12	_aaa	0	20	Identifier
12	MOD	0	28	Mod
12	_bbb	0	20	Identifier
14	_xxx	0	20	Identifier
14	<-	0	23	Assign
14	(	0	36	LBracket
14	_aaa	0	20	Identifier
14	SUB	0	25	Sub
14	_bbb	0	20	Identifier
14	)	0	37	RBracket
14	MUL	0	26	Mul
14	10	10	21	Number
14	ADD	0	24	Add
14	(	0	36	LBracket
14	_aaa	0	20	Identifier
14	ADD	0	24	Add

14	_bbb	0	20	Identifier
14	)	0	37	RBracket
14	DIV	0	27	Div
14	10	10	21	Number
15	_yyy	0	20	Identifier
15	<-	0	23	Assign
15	_xxx	0	20	Identifier
15	ADD	0	24	Add
15	(	0	36	LBracket
15	_xxx	0	20	Identifier
15	MOD	0	28	Mod
15	10	10	21	Number
15	)	0	37	RBracket
17	WRITE	0	6	Output
17	_xxx	0	20	Identifier
18	WRITE	0	6	Output
18	_yyy	0	20	Identifier
20	ENDBLOCK	0	4	EndBlock

Програма 2

TOKEN TABLE				
line number	token	value	token code	type of token
1	STARTPROGRAM	0	0	StartProgram
2	VARIABLE	0	2	Variable
2	INT_4	0	3	Type
2	_aaa	0	20	Identifier
2	,	0	40	Comma
2	_bbb	0	20	Identifier
2	,	0	40	Comma
2	_ccc	0	20	Identifier
2	;	0	38	Semicolon
3	STARTBLOCK	0	1	StartBlock
4	READ	0	5	Input
4	_aaa	0	20	Identifier
5	READ	0	5	Input
5	_bbb	0	20	Identifier
6	READ	0	5	Input
6	_ccc	0	20	Identifier
7	IF	0	7	If

7	(	0	36	LBracket
7	_aaa	0	20	Identifier
7	LT	0	32	Less
7	_bbb	0	20	Identifier
7	)	0	37	RBracket
8	IF	0	7	If
8	(	0	36	LBracket
8	_aaa	0	20	Identifier
8	LT	0	32	Less
8	_ccc	0	20	Identifier
8	)	0	37	RBracket
9	WRITE	0	6	Output
9	_aaa	0	20	Identifier
10	ELSE	0	8	Else
11	WRITE	0	6	Output
11	_ccc	0	20	Identifier
12	ELSE	0	8	Else
13	IF	0	7	If
13	(	0	36	LBracket
13	_bbb	0	20	Identifier
13	LT	0	32	Less

13	_ccc	0	20	Identifier
13	)	0	37	RBracket
14	WRITE	0	6	Output
14	_bbb	0	20	Identifier
15	ELSE	0	8	Else
16	WRITE	0	6	Output
16	_ccc	0	20	Identifier
17	IF	0	7	If
17	(	0	36	LBracket
17	(	0	36	LBracket
17	_aaa	0	20	Identifier
17	EQ	0	29	Equality
17	_bbb	0	20	Identifier
17	)	0	37	RBracket
17	&	0	34	And
17	(	0	36	LBracket
17	_aaa	0	20	Identifier
17	EQ	0	29	Equality
17	_ccc	0	20	Identifier
17	)	0	37	RBracket

17	&	0	34	And
17	(	0	36	LBracket
17	_bbb	0	20	Identifier
17	EQ	0	29	Equality
17	_ccc	0	20	Identifier
17	)	0	37	RBracket
17	)	0	37	RBracket
18	WRITE	0	6	Output
18	1	1	21	Number
19	ELSE	0	8	Else
20	WRITE	0	6	Output
20	0	0	21	Number
21	IF	0	7	If
21	(	0	36	LBracket
21	(	0	36	LBracket
21	_aaa	0	20	Identifier
21	GT	0	31	Greate
21	0	0	21	Number
21	)	0	37	RBracket
21	| |	0	35	Or
21	(	0	36	LBracket

21	_bbb	0	20	Identifier
21	GT	0	31	Greate
21	0	0	21	Number
21	)	0	37	RBracket
21	||	0	35	Or
21	(	0	36	LBracket
21	_ccc	0	20	Identifier
21	GT	0	31	Greate
21	0	0	21	Number
21	)	0	37	RBracket
21	)	0	37	RBracket
22	WRITE	0	6	Output
22	-1	-1	21	Number
23	ELSE	0	8	Else
24	WRITE	0	6	Output
24	0	0	21	Number
25	IF	0	7	If
25	(	0	36	LBracket
25	!	0	33	Not
25	(	0	36	LBracket

25	_aaa	0	20	Identifier
25	GT	0	31	Greate
25	(	0	36	LBracket
25	_bbb	0	20	Identifier
25	ADD	0	24	Add
25	_ccc	0	20	Identifier
25	)	0	37	RBracket
25	)	0	37	RBracket
25	)	0	37	RBracket
26	WRITE	0	6	Output
26	10	10	21	Number
27	ELSE	0	8	Else
28	WRITE	0	6	Output
28	0	0	21	Number
29	ENDBLOCK	0	4	EndBlock

Програма 3

TOKEN TABLE				
line number	token	value	token code	type of token
1	STARTPROGRAM	0	0	StartProgram
2	VARIABLE	0	2	Variable
2	INT_4	0	3	Type
2	_aaa	0	20	Identifier
2	,	0	40	Comma
2	_bbb	0	20	Identifier
2	,	0	40	Comma
2	_xxx	0	20	Identifier
2	,	0	40	Comma
2	_iii	0	20	Identifier
2	,	0	40	Comma
2	_jjj	0	20	Identifier
2	;	0	38	Semicolon
3	STARTBLOCK	0	1	StartBlock
5	READ	0	5	Input
5	_aaa	0	20	Identifier
6	READ	0	5	Input

6	_bbb	0	20	Identifier
8	FOR	0	10	For
8	_xxx	0	20	Identifier
8	<-	0	23	Assign
8	_aaa	0	20	Identifier
8	TO	0	11	To
8	_bbb	0	20	Identifier
8	DO	0	13	Do
9	WRITE	0	6	Output
9	_xxx	0	20	Identifier
9	MUL	0	26	Mul
9	_xxx	0	20	Identifier
11	_xxx	0	20	Identifier
11	<-	0	23	Assign
11	0	0	21	Number
12	FOR	0	10	For
12	_iii	0	20	Identifier
12	<-	0	23	Assign
12	1	1	21	Number
12	TO	0	11	To
12	_aaa	0	20	Identifier

```
-------------------------------------------------------------------
|        12 |           DO |         0 |        13 | Do          |
-------------------------------------------------------------------
|        13 |          FOR |         0 |        10 | For         |
-------------------------------------------------------------------
|        13 |         _jjj |         0 |        20 | Identifier  |
-------------------------------------------------------------------
|        13 |           <- |         0 |        23 | Assign      |
-------------------------------------------------------------------
|        13 |            1 |         1 |        21 | Number      |
-------------------------------------------------------------------
|        13 |           TO |         0 |        11 | To          |
-------------------------------------------------------------------
|        13 |         _bbb |         0 |        20 | Identifier  |
-------------------------------------------------------------------
|        13 |           DO |         0 |        13 | Do          |
-------------------------------------------------------------------
|        14 |         _xxx |         0 |        20 | Identifier  |
-------------------------------------------------------------------
|        14 |           <- |         0 |        23 | Assign      |
-------------------------------------------------------------------
|        14 |         _xxx |         0 |        20 | Identifier  |
-------------------------------------------------------------------
|        14 |          ADD |         0 |        24 | Add         |
-------------------------------------------------------------------
|        14 |            1 |         1 |        21 | Number      |
-------------------------------------------------------------------
|        16 |        WRITE |         0 |         6 | Output      |
-------------------------------------------------------------------
|        16 |         _xxx |         0 |        20 | Identifier  |
-------------------------------------------------------------------
|        18 |     ENDBLOCK |         0 |         4 | EndBlock    |
-------------------------------------------------------------------
```

Додаток Б (С код (або код на асемблері), отриманий на виході транслятора для тестових прикладів)

Prog1.c

```c
#include <stdio.h>
int main() {
int _yyy;
int _xxx;
```

```
int _bbb;
int _aaa;

printf("Enter _aaa: ");
scanf("%d", &_aaa);
printf("Enter _bbb: ");
scanf("%d", &_bbb);
printf("%d\n", (_aaa + _bbb));
printf("%d\n", (_aaa - _bbb));
printf("%d\n", (_aaa * _bbb));
printf("%d\n", (_aaa / _bbb));
printf("%d\n", (_aaa % _bbb));
_xxx = (((_aaa - _bbb) * 10) + ((_aaa + _bbb) / 10));
_yyy = (_xxx + (_xxx % 10));
printf("%d\n", _xxx);
printf("%d\n", _yyy);
return 0;
}
```

Prog2.c

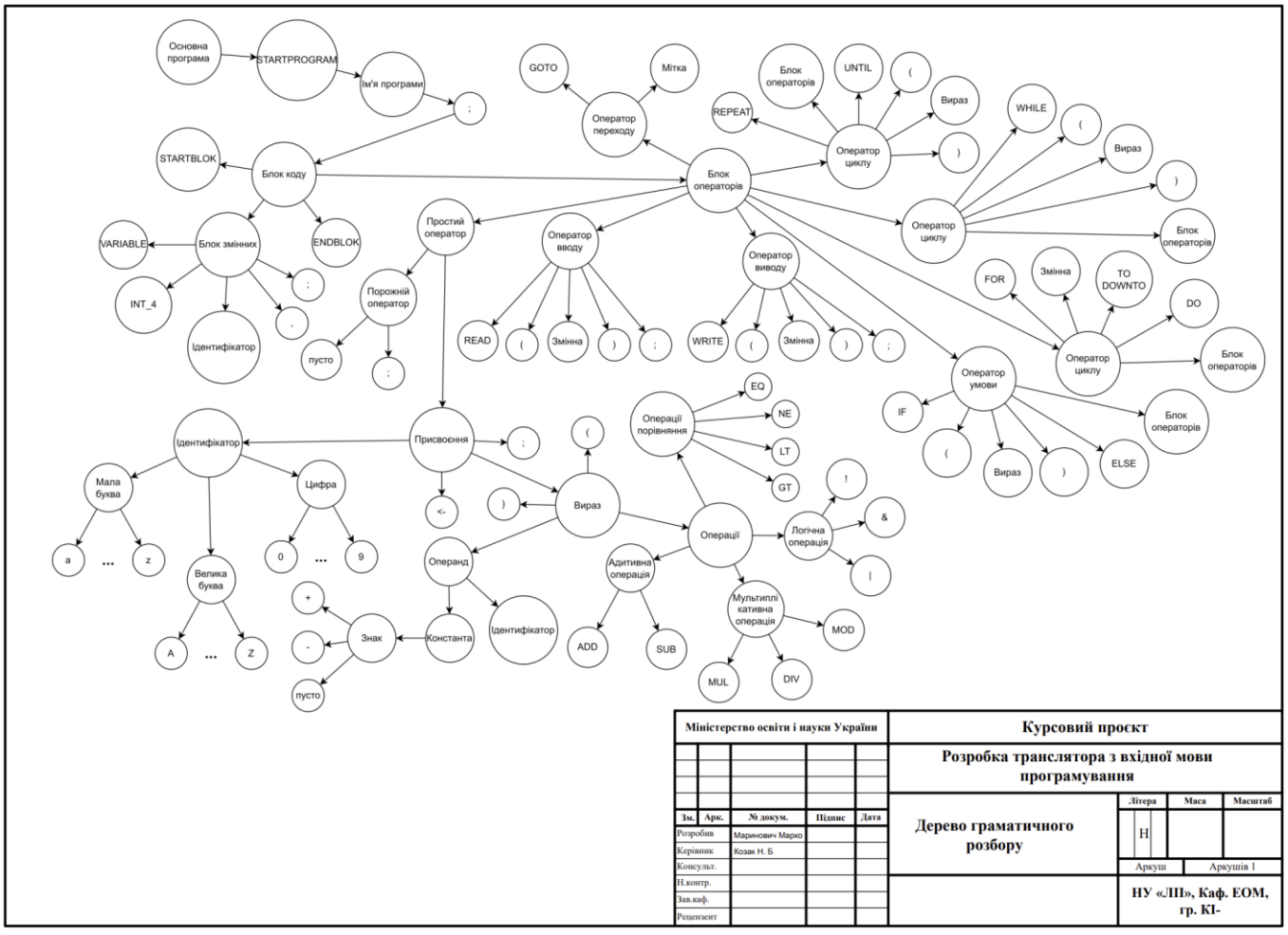
```
#include <stdio.h>
int main() {
int _ccc;
int _bbb;
int _aaa;

printf("Enter _aaa: ");
scanf("%d", &_aaa);
printf("Enter _bbb: ");
scanf("%d", &_bbb);
printf("Enter _ccc: ");
scanf("%d", &_ccc);
if ((_aaa < _bbb)) if ((_aaa < _ccc)) printf("%d\n", _aaa);
 else printf("%d\n", _ccc);

 else if ((_bbb < _ccc)) printf("%d\n", _bbb);
 else printf("%d\n", _ccc);


if (((_aaa == _bbb) && ((_aaa == _ccc) && (_bbb == _ccc)))) printf("%d\n", 1);
 else printf("%d\n", 0);

if (((_aaa > 0) || ((_bbb > 0) || (_ccc > 0)))) printf("%d\n", -1);
 else printf("%d\n", 0);

if (!((_aaa > (_bbb + _ccc)))) printf("%d\n", 10);
 else printf("%d\n", 0);

return 0;
}
```

Prog3.c

```
#include <stdio.h>
int main() {
int _jjj;
int _iii;
int _xxx;
int _bbb;
int _aaa;

printf("Enter _aaa: ");
scanf("%d", &_aaa);
printf("Enter _bbb: ");
scanf("%d", &_bbb);
```

```
for(
_xxx = _aaa;
_xxx <= _bbb;
++_xxx
) printf("%d\n", (_xxx * _xxx));
_xxx = 0;
for(
_iii = 1;
_iii <= _aaa;
++_iii
) for(
_jjj = 1;
_jjj <= _bbb;
++_jjj
) _xxx = (_xxx + 1);
printf("%d\n", _xxx);
return 0;
}
```

Додаток В (Абстрактні синтаксичні дерева для тестових прикладів (за наявності))



Додаток Г (Документований текст програмних модулів (лістинги))

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

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

#define LANGUAGE      ".m20"

struct Token* TokenTable;   // □□□□□□□ □□□□□□
unsigned int TokensNum;     // ‘□□□□□□ □□□□□□

struct id* idTable;        // □□□□□□□ □□□□□□□□□□□□□□□
unsigned int idNum;        // □□□□□□□ □□□□□□□□□□□□□□□

struct id* labelTable;     // □□□□□□□ □□□□
unsigned int labelNum;     // □□□□□□□ □□□□

FILE* errorFile;

int main(int argc, char* argv[])
{
        // □□□□□□□□ □□□'□□ □□ □□□□□□□ □□□□□
        TokenTable = (struct Token*)malloc(MAX_TOKENS * sizeof(struct Token));

        // □□□□□□□□ □□□'□□ □□ □□□□□□□ □□□□□□□□□□□□□□□
        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("Input 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 \".m20\"");
                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");

        // □□□□□□□□□ □□□□□       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);

        // □□□□□□□□□□□□ □□□□□
        Parser::Parser();
        Parser::Semantic();

        // □□□□□□□□□ □□□□□□□□□□□□ □□□□□□□□□□□□□ □□□□□□
        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);

		// □□□□□□□□□ □□□□□□□□□ □ □□□□
		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;
	}
	#include "Parser.h"

	#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);
	    }
	  }

	  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; // □□□□□□□ □□□□□□□ □□□□□□□□□□□□□□□
}

void program() {
    match(StartProgram);
    //match(Variable);
    if (TokenTable[pos].type == Variable) {
        ++pos;  //  for VARIABLE
        variableDeclaration();
        match(Semicolon);
    }
    match(StartBlock);
    programBody();
    match(EndBlock);
}

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 Continue: {
        // For "CONTINUE WHILE"
        ++pos;  // consume CONTINUE
        match(While);
        break;
    }
    case Exit: {
        // For "EXIT WHILE"
        ++pos;  // consume EXIT
```

```
        match(While);
        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
    while (TokenTable[pos].type == Mul || TokenTable[pos].type == Mod || TokenTable[pos].type == Div) {
        ++pos;  // for * or Mod or Div
        middlePriorityExpression();
    }
}

void middlePriorityExpression() {
    if (TokenTable[pos].type == Identifier || TokenTable[pos].type == Number) {
        ++pos;
    }
    else if (TokenTable[pos].type == LBracket) {
        ++pos;  // Consume '('
        arithmeticExpression();  // Parse full arithmetic expression
        match(RBracket);  // Ensure closing ')'
    }
    else {
        // If it's not a valid operand, print an error and exit
        printf("\nSyntax error in line %d, token number: %d : expected identifier, number, or '(', but found: %s.\n",
            TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));
        exit(1);
    }
}

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

void ifStatement() {
    match(If);
    match(LBracket);
    logicalExpression();
    match(RBracket);

    // Перевіряємо, чи після умови стоїть ";"
    // Якщо так — це порожня гілка if
    if (TokenTable[pos].type == Semicolon) {
        ++pos; // "з'їдаємо" крапку з комою
    }
    else {
        // Якщо це не ";", то парсимо звичайний statement
        statement();
    }

    // Перевіряємо, чи є else
    if (TokenTable[pos].type == Else) {
        ++pos;  // споживаємо 'else'

        // Аналогічно перевіряємо, чи else-гілка не порожня
        if (TokenTable[pos].type == Semicolon) {
            ++pos; // споживаємо порожню гілку
        }
        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();

    // парсимо тіло while до 'END'
    while (TokenTable[pos].type != EndBlock) {
        statement();
    }

    // коли зустріли 'END', треба з'їсти цей токен...
    match(EndBlock);  // match(EndBlock) = "END"
    // ...а потім обов'язково наступний 'WHILE'
    match(While);     // завершення конструкції "END WHILE"
}

void statementInWhile() {
    printf("STATEMENT IN WHILE");
    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
    logicalExpression();
}

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;
  }
}

#include "LexicAnalyzer.h"

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

extern struct Token* TokenTable;   // □□□□□□□ □□□□□□
extern unsigned int TokensNum;     // ┌□□□□□□ □□□□□□

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 == '-' || ch == '.') {
                state = Digit;
            }
            else if (ch == '#') {
                state = SComment;
            }
            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 == '.') && j < 10) {
                buf[j++] = ch;
                ch = getc(F);
            }

            buf[j] = '\0';


            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: {
    while (ch != EOF) {
        if (ch == '*' && (ch = getc(F)) == '#') {
            break;
        }
        ch = getc(F);
    }
    ch = getc(F);
    state = Start;
    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 ((('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z') || ('0' <= ch && ch <= '9') || ch == '_') && j < 15) {
        buf[j++] = ch;
        ch = getc(F);
    }

    buf[j] = '\0';

    enum TypeOfToken tempType = Unknown_;

    if (!strcmp(buf, "STARTPROGRAM")) {
        tempType = StartProgram;
    }
    else if (!strcmp(buf, "VARIABLE")) {
        tempType = Variable;
    }
    else if (!strcmp(buf, "STARTBLOCK")) {
        tempType = StartBlock;
    }
    else if (!strcmp(buf, "INT_4")) {
        tempType = Type;
    }
    else if (!strcmp(buf, "ENDBLOCK")) {
        tempType = EndBlock;
    }
    else if (!strcmp(buf, "READ")) {
```

```
            tempType = Input;
        }
        else if (!strcmp(buf, "WRITE")) {
            tempType = Output;
        }
        else if (!strcmp(buf, "IF")) {
            tempType = If;
        }
        else if (!strcmp(buf, "MUL")) {
            tempType = Mul;
        }
        else if (!strcmp(buf, "DIV")) {
            tempType = Div;
        }
        else if (!strcmp(buf, "ADD")) {
            tempType = Add;
        }
        else if (!strcmp(buf, "MOD")) {
            tempType = Mod;
        }
        else if (!strcmp(buf, "SUB")) {
            tempType = Sub;
        }
        else if (!strcmp(buf, "|")) {
            tempType = Or;
        }
        else if (!strcmp(buf, "&")) {
            tempType = And;
        }
        else if (!strcmp(buf, "!")) {
            tempType = Not;
        }
        else if (!strcmp(buf, "EQ")) {
            tempType = Equality;
        }
        else if (!strcmp(buf, "NE")) {
            tempType = NotEquality;
        }
        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, "WHILE")) {
            tempType = While;
        }
        else if (!strcmp(buf, "END")) {
            tempType = End;
        }
        else if (!strcmp(buf, "CONTINUE")) {
            tempType = Continue;
        }
        else if (!strcmp(buf, "EXIT")) {
            tempType = Exit;
        }
        else if (!strcmp(buf, "LT")) {
            tempType = Less;
        }
        else if (!strcmp(buf, "GT")) {
            tempType = Greate;
        }
        else if (!strcmp(buf, "REPEAT")) {
            tempType = Repeat;
        }
        else if (!strcmp(buf, "UNTIL")) {
            tempType = Until;
        }
        else if (strlen(buf) <= 4 && buf[0] == '_'
            && ('a' <= buf[1] && buf[1] <= 'z')
            && ('a' <= buf[2] && buf[2] <= 'Z')
            && ('a' <= buf[3] && buf[3] <= 'Z')
            ) {
            tempType = Identifier;
            }
        else {
            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 = And;
            tempToken.value = 0;
            tempToken.line = line;
            state = Finish;
            ch = getc(F);
            break;
        }

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

        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 '<': {
            ch = getc(F);
            if (ch == '-') {
                strcpy_s(tempToken.name, "<-");
                tempToken.type = Assign;
                tempToken.value = 0;
                tempToken.line = line;
                state = Finish;
                ch = getc(F);
            }
            else {
                strcpy_s(tempToken.name, "<");
                tempToken.type = Unknown_;
                tempToken.value = 0;
                tempToken.line = line;
                state = Finish;
            }
            break;
        }

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

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

  return TokensNum = tokenCount;
}
```

```
// □□□□□□□ □□□□□ □□□□□□□ □□□□□□ □□ □□□□□
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");
    }
}
#include "header.h"

const char* lexemeTypeName(enum TypeOfToken type) {
    switch (type) {
    case StartProgram:  return "StartProgram";
    case Variable:      return "Variable";
    case Type:          return "Type";
    case StartBlock:    return "StartBlock";
    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";
}

#include "Codegen.h"

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

        switch (node->type) {
        case empty_node: {
            break;
        }
        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: {
    // if( <□□□□□> )
    fprintf(outFile, "if (");
    codegen(outFile, node->left); // □□□□
    fprintf(outFile, ") ");

    // x□□ if □□□□□□□□□□ □ node->right.
    // □□□□□□□, □□ □□□□□□□□□□□□□□ statement_node / empty_node / else_node.
    if (node->right->type == else_node) {
        // □□□□ □□ else_node, □□□
        //   node->right->left = then-□□□□□□□
        //   node->right->right = else-□□□□□□□

        struct astNode* thenPart = node->right->left;
        struct astNode* elsePart = node->right->right;

        // □□□□□□□□ thenPart
        if (thenPart->type == empty_node) {
            fprintf(outFile, ";");
        }
        else {
            codegen(outFile, thenPart);
        }

        fprintf(outFile, " else ");
        if (elsePart->type == empty_node) {
            fprintf(outFile, ";");
        }
        else {
            codegen(outFile, elsePart);
        }
    }
    else {
        // □□□□ □□ □□□□□□ then-□□□□□□□ (□ □□□□ else)
        if (node->right->type == empty_node) {
```

```
            fprintf(outFile, ";");
        }
        else {
            codegen(outFile, node->right);
        }
    }

    fprintf(outFile, "\n");
    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;
        }
        }
    }
}
#include "Ast.h"

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);
    //match(Semicolon);

    struct astNode* declaration = nullptr;
    if (TokenTable[pos].type == Variable) {
        ++pos;  //  for VARIABLE
        declaration = variableDeclaration();
        match(Semicolon);
    }
    match(StartBlock);
    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() {
    // Match and store the identifier
    match(Identifier);
    char identifierName[16];
    // Safely copy with truncation if needed
    strncpy_s(identifierName, sizeof(identifierName), TokenTable[pos - 1].name, _TRUNCATE);

    // Match the assignment operator
    match(Assign);

    // Parse the right side expression
    struct astNode* right = arithmeticExpression();

    // Create the left (ID) node
    struct astNode* left = createNode(id_node, identifierName, nullptr, nullptr);

    // Return the assign node
    return createNode(assign_node, "assign", left, right);
}


struct astNode* ifStatement() {
    match(If);
    match(LBracket);
    struct astNode* expr = logicalExpression();
    match(RBracket);

    // THEN-□□□□□□□
    struct astNode* thenStmt = nullptr;
    if (TokenTable[pos].type == Semicolon) {
        // □□□□□□□ □□□□ IF (...)
        thenStmt = createNode(empty_node, "empty", nullptr, nullptr);
        ++pos; // consume ";"
    }
    else {
        // □□□□□□□ □□□□□□□□□□ statement
        thenStmt = statement();
        // □□□□ □□□□□□□□□□ □□□□□□ □□□□□□ □ □□□□□ □□□□ statement
        if (TokenTable[pos].type == Semicolon) {
            ++pos; // consume ";"
        }
    }

    // □□□□□□□□□□, □□ □ ELSE
    if (TokenTable[pos].type == Else) {
        ++pos;  // consume 'else'

        // ELSE-□□□□□□□
        struct astNode* elseStmt = nullptr;
        if (TokenTable[pos].type == Semicolon) {
            // □□□□□□□ else
            elseStmt = createNode(empty_node, "empty", nullptr, nullptr);
            ++pos; // consume ";"
        }
        else {
            // □□□□□□ statement
            elseStmt = statement();
            if (TokenTable[pos].type == Semicolon) {
                ++pos; // consume ";"
            }
        }

        // □□'□□□□□□ then- □ else-□□□□□□□ □ □□□□ □□□□□ (□□□□ □□□□ □□□□□□□ □□□ else_node)
        thenStmt = createNode(else_node, "else", thenStmt, elseStmt);
    }

    // □□□□□□□□□ □□□□□ if_node, □□
    //   node->left  = □□□□□□ □□□□□
    //   node->right = □□□ □□□□□□□□□□□□□ thenStmt,
    //              □□□ "else"-□□□□□, □□□□ □□□□□ then/else
    return createNode(if_node, "if", expr, thenStmt);
}

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);                       // "WHILE"
    struct astNode* expr = logicalExpression();
    struct astNode* body = whileBody();

    // □□□□□ □□□□□□□ "END" (□□□□ □ □□□□□□ □□ □□□□□ EndBlock)
    match(EndBlock);                    // "END"
    // j□□□ □□□□□ □□□□□□ □□□□□□□ "WHILE"
    match(While);                       // "WHILE"

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

struct astNode* whileBody() {
    // □□□□ □□□□□□ "END" □ □□□ □□□□□□
    if (TokenTable[pos].type == EndBlock) {
        return nullptr;
    }
    // □□□□□□ □□□□□□ □□□□□ □□□□□□□□□□
    struct astNode* stmt = statementInWhile();
    struct astNode* body = stmt;

    // □□□□, □□□□ □□ □□□□□□□ "END", □□□□□□□ □□□□□□□ □□□□□□□□□□
    while (TokenTable[pos].type != EndBlock) {
        struct astNode* nextStmt = statementInWhile();
        body = createNode(statement_node, "statement", body, nextStmt);
    }
    return body;
}

struct astNode* statementInWhile() {
    switch (TokenTable[pos].type) {
    case Continue:
        ++pos;  // consume CONTINUE
        match(While); // "CONTINUE WHILE"
        return createNode(continue_node, "continue", nullptr, nullptr);

    case Exit:
        ++pos;  // consume EXIT
        match(While); // "EXIT WHILE"
        return createNode(exit_node, "exit", nullptr, nullptr);

    default:
        // □□□□ □□ □□ Continue/Exit, □□□□□□□ □□ □□□□□□□□□ statement
        return statement();
    }
}


struct astNode* repeatStatement() {
    match(Repeat);
    struct astNode* body = repeatBody();
    match(Until);
    struct astNode* expr = logicalExpression();
    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);
}
}
```

```
#pragma once

#include "header.h"

#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;

namespace Parser {
    void Parser();
    void Semantic();
    void match(enum TypeOfToken expectedType);
    unsigned int IdIdentification(struct id idTable[], struct Token TokenTable[], unsigned int tokenCount);
    unsigned int LabelIdentification(struct id labelTable[], struct Token TokenTable[], unsigned int tokenCount);
    void printIdentifiers(int num, struct id* table);
    void fprintIdentifiers(FILE* F, int num, struct id* table);

    void program();
    void programBody();
    void variableDeclaration();
    void variableList();
    void statement();
    void inputStatement();
    void outputStatement();
    void arithmeticExpression();
    void lowPriorityExpression();
    void middlePriorityExpression();
    void assignStatement();
    void ifStatement();
    void logicalExpression();
    void andExpression();
    void comparison();
    void comparisonExpression();
    void gotoStatement();
    void labelPoint();
    void forStatement();
    void whileStatement();
    void statementInWhile();
    void repeatStatement();
    void compoundStatement();
}
```

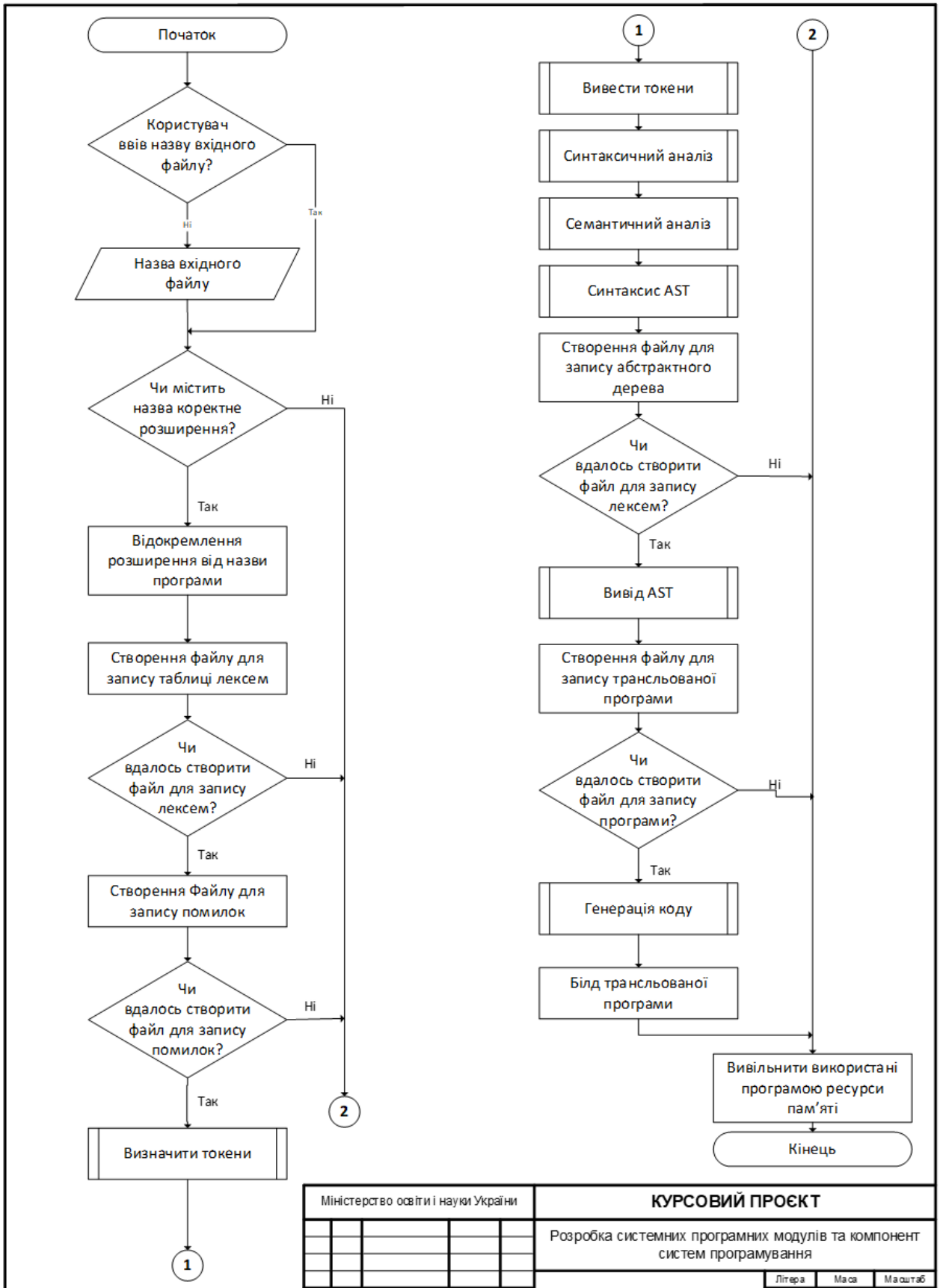
```
#pragma once

#include "header.h"
#include <stdio.h>

namespace LexicAnalyzer {
        unsigned int getTokens(FILE* F);
        void printTokens(void);
        void fprintTokens(FILE* F);
}
```

```
#pragma once

#include "header.h"

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

namespace Codegen {
    void codegen(FILE* outFile, struct astNode* node);
}
```

```
#pragma once

#include "header.h"

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

extern struct Token* TokenTable;
extern int pos;

namespace AST {
    struct astNode* program();
    struct astNode* programBody();
    struct astNode* variableDeclaration();
    struct astNode* variableList();
    struct astNode* statement();
    struct astNode* inputStatement();
    struct astNode* outputStatement();
    struct astNode* arithmeticExpression();
    struct astNode* lowPriorityExpression();

    struct astNode* middlePriorityExpression();

    struct astNode* assignStatement();
    struct astNode* ifStatement();
    struct astNode* logicalExpression();
```

```
    struct astNode* andExpression();

    struct astNode* comparison();

    struct astNode* comparisonExpression();

    struct astNode* gotoStatement();
    struct astNode* labelPoint();
    struct astNode* forStatement();
    struct astNode* whileStatement();
    struct astNode* statementInWhile();
    struct astNode* whileBody();
    struct astNode* repeatStatement();
    struct astNode* repeatBody();
    struct astNode* compoundStatement();

    void deleteNode(struct astNode* node);
    struct astNode* createNode(enum TypeOfNode type, const char* name, struct astNode* left, struct astNode* right);
    void printAST(struct astNode* node, int level);
    void fPrintAST(FILE* outFile, struct astNode* node, int level);
    void match(enum TypeOfToken expectedType);
    struct astNode* astParser();
}
```

Додаток Д (Загально блок-схема роботи транслятора)


Початок
Користувач ввів назву вхідного файлу?
Так
Ні
Назва вхідного файлу
Чи містить назва коректне розширення?
Ні
Так
Відокремлення розширення від назви програми
Створення файлу для запису таблиці лексем
Чи вдалось створити файл для запису лексем?
Ні
Так
Створення Файлу для запису помилок
Чи вдалось створити файл для запису помилок?
Ні
Так
Визначити токени
1
2
1
Вивести токени
Синтаксичний аналіз
Семантичний аналіз
Синтаксис AST
Створення файлу для запису абстрактного дерева
Чи вдалось створити файл для запису лексем?
Ні
Так
Вивід AST
Створення файлу для запису трансльованої програми
Чи вдалось створити файл для запису програми?
Ні
Так
Генерація коду
Білд трансльованої програми
2
Вивільнити використані програмою ресурси пам'яті
Кінець
Міністерство освіти і науки України
КУРСОВИЙ ПРОЄКТ
Розробка системних програмних модулів та компонент систем програмування
Зм.
Арк.
№ докум.
Підпис
Дата
Виконав
Маринович М
Керівник
Козак Н.Б.
Консульт.
Консульт.
Зав. каф.
Думець Р. Б.
Реценз.
Алгоритм транслятора
Літера
Маса
Масштаб
У
Аркуш
Аркушів 1
Додаток В
НУ «ЛП», ІКТА, каф. ЕОМ, гр КІ-307