

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"

Варіант 2

"РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ"

Виконав студент групи КІ-308:

Боряк В.В.

Перевірив:

Козак Н.Б.

Львів-2024

ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ

Варіант 2

Завдання на курсовий проект

1. Цільова мова транслятора – мова програмування C 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 C 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.

В моєму випадку це .b02

Опис вхідної мови програмування:

- Тип даних: LONGINT

- Блок тіла програми: PROGRAM <name>; VAR...; BEGIN END
- Оператор вводу: SCAN ()
- Оператор виводу: PRINT ()
- Оператори: IF ELSe (C)

GOTO (C)

FOR-TO-DO (Паскаль)

FOR-DOWNT0-DO (Паскаль)

WHILE (Бейсік)

REPEAT-UNTIL (Паскаль)

- Регістр ключових слів: Up
- Регістр ідентифікаторів: Low-Up6 перший символ Low
- Операції арифметичні: ADD, SUB, MUL, DIV, MOD
- Операції порівняння: EQ, NE, >, <
- Операції логічні: NOT, AND, OR
- 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).

Деталізація завдання на проектування:

1. В кожному завданні передбачається блок оголошення змінних; змінні зберігають значення цілих чисел i , 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.

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	7
1.ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ	8
2.ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ	12
2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура.....	12
Опис вхідної мови програмування у термінах розширеної форми Бекуса-Наура:	13
2.2. Опис термінальних символів та ключових слів.....	14
3.РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ	16
3.1. Вибір технології програмування	16
3.2. Проектування таблиць транслятора та вибір структур даних.....	17
3.3. Розробка лексичного аналізатора.....	19
3.3.1. Розробка алгоритму роботи лексичного аналізатора.....	20
3.3.2. Опис програми реалізації лексичного аналізатора.	22
3.4. Розробка синтаксичного та семантичного аналізатора.....	24
3.4.1 Розробка дерев граматичного розбору	25
3.4.2 Розробка алгоритму роботи синтаксичного та семантичного аналізатора	27
3.4.3 Опис програми реалізації синтаксичного та семантичного аналізатора.....	31
3.5 Розробка генератора коду.	32
3.5.2 Розробка алгоритму роботи генератора коду.	33
4.НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА	37
4.1 . Опис інтерфейсу та інструкції користувачу.....	42
4.2 . Виявлення лексичних і синтаксичних помилок.	43
4.3 . Перевірка роботи транслятора за допомогою тестових задач.	46
ВИСНОВКИ	53
СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ	54
ДОДАТКИ	55

ВСТУП

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

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

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

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

1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ

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 транслують 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 граматичній 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

Для задання синтаксису мов програмування використовують форму Бекуса- Наура або розширену форму Бекуса-Наура — це спосіб запису правил контекстно- вільної граматики, тобто форма опису формальної мови. Саме її типово використовують для запису правил мов програмування та протоколів комунікації.

БНФ визначає скінченну кількість символів (нетерміналів). Крім того, вона визначає правила заміни символу на якусь послідовність букв (терміналів) і символів. Процес отримання ланцюжка букв можна визначити поетапно: спочатку є один символ (символи зазвичай знаходяться у кутових дужках, а їх назва не несе жодної інформації). Потім цей символ замінюється на деяку послідовність букв і символів, відповідно до одного з правил. Потім процес повторюється (на кожному кроці один із символів замінюється на послідовність, згідно з правилом). Зрештою , виходить ланцюжок , що складається з букв і не містить символів. Це означає , що отриманий ланцюжок може бути виведений з початкового символу.

Нотація БНФ є набором «продукцій», кожна з яких відповідає зразку:

$$\langle \text{символ} \rangle = \langle \text{вираз, що містить символи} \rangle$$

де вираз, що містить символи це послідовність символів або послідовності символів, розділених вертикальною рисою |, що повністю перелічують можливий вибір символ з лівої частини формули.

У розширеній формі нотації Бекуса — Наура вирази, що можна пропускати або які можуть повторятись слід записувати у фігурних дужках { ... }:, а можлива поява може відображатися застосуванням квадратних дужок [...]:.

Опис вхідної мови програмування у термінах розширеної форми Бекуса- Наура:

```

program = "PROGRAM", identifier, ";", "VAR", variable_declaration, "BEGIN", {statement}, "END";
variable_declaration = "LONGINT", variable_list;
variable_list = identifier, {",", identifier};
identifier = low, {low_up};
low_up = low | up;
low = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" |
"v" | "w" | "x" | "y" | "z";
up = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" |
"T" | "U" | "V" | "W" | "X" | "Y" | "Z";
statement = input_statement | output_statement | assign_statement | if_else_statement | goto_statement |
for_statement | while_statement | repeat_until_statement | compound_statement;
input_statement = "SCAN", "(", identifier, ")";
output_statement = "PRINT", "(", arithmetic_expression, ")";
arithmetic_expression = low_priority_expression {low_priority_operator, low_priority_expression};
low_priority_operator = "ADD" | "SUB";
low_priority_expression      =      middle_priority_expression      {middle_priority_operator,
middle_priority_expression};
middle_priority_operator = "MUL" | "DIV" | "MOD";
middle_priority_expression = identifier | number | "(", arithmetic_expression, ")";
number = ["-"], (nonzero_digit, {digit} | "0");
nonzero_digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
assign_statement = identifier, "==>", arithmetic_expression;
if_else_statement = "IF", "(", logical_expression, ")", statement, ["ELSE", statement];
logical_expression = and_expression {logical_operator, and_expression};
logical_operator = "OR";
and_expression = comparison {logical_operator, comparison};
comparison = arithmetic_expression comparison_operator arithmetic_expression;
comparison_operator = "EQ" | "NE" | ">" | "<";
goto_statement = "GOTO", identifier;
for_statement = "FOR", assign_statement, ("TO" | "DOWNT0"), arithmetic_expression, "DO", statement;
while_statement = "WHILE", logical_expression, "DO", {statement}, "END";
repeat_until_statement = "REPEAT", {statement}, "UNTIL", "(", logical_expression, ")";
compound_statement = "BEGIN", {statement}, "END";
comment = "{" text "}";

```

2.2. Опис термінальних символів та ключових слів

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

Термінальний символ або ключове слово	Значення
PROGRAM	Початок програми
BEGIN	Початок тексту програми
VAR	Початок блоку опису змінних
END	Кінець розділу операторів
SCAN	Оператор вводу змінних
PRINT	Оператор виводу (змінних або рядкових констант)
==>	Оператор присвоєння
IF	Оператор умови
ELSE	Оператор умови
GOTO	Оператор переходу
LABEL	Мітка переходу
FOR	Оператор циклу
TO	Інкремент циклу
DOWNT0	Декремент циклу
DO	Початок тіла циклу
WHILE	Оператор циклу
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
<	Оператор перевірки чи менше
>	Оператор перевірки чи більше
NOT	Оператор логічного заперечення
AND	Оператор кон'юнкції
OR	Оператор диз'юнкції
LONGINT	32-ох розрядні знакові цілі
{...}	Коментар
,	Розділювач
;	Ознака кінця оператора
(Відкриваюча дужка
)	Закриваюча дужка

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

3. РОЗРОБКА ТРАНСЛЯТОРА 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	PROGRAM
Start	BEGIN
Vars	VAR
End	END
VarType	LONGINT
Read	SCAN
Write	PRINT
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	<
Greate	>
Not	NOT
And	AND
Or	OR
Plus	+
Minus	-
Identifier	
Number	
String	
Undefined	
Unknown	
Comma	,
Quotes	“
Semicolon	;
LBraket	(
RBraket)
LComment	{
RComment	}
Comment	

3.3. Розробка лексичного аналізатора.

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

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

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

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

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

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

3.3.1. Розробка алгоритму роботи лексичного аналізатора

Кроки для створення алгоритму лексичного аналізатора на основі цих даних:

1. Розбір вхідного потоку символів:

- Прочитати вхідний текст по символах.
- Ігнорувати пробіли, табуляцію та символи нових рядків (якщо вони не важливі для мови).

2. Розпізнавання tokenів:

- Використовувати регулярні вирази для розпізнавання ключових слів, ідентифікаторів, чисел, операторів, коментарів і т.д.
- Для ключових слів використовувати перевірку на наявність у списку зарезервованих слів (PROGRAM, SCAN, PRINT, IF, GOTO, і т.д.).
- Для ідентифікаторів: перевірка, чи перший символ — це мала літера, а решта можуть бути великими чи малими літерами чи цифрами.
- Для числових констант: перевірка на цілі числа (для типу LONGINT).

3. Обробка арифметичних і порівняльних операторів:

- Для розпізнавання операторів типу ADD, SUB, DIV, MOD — перевіряти відповідні ключові слова.
- Для порівняльних операторів (наприклад, EQ, NE, >, <) — розпізнавати спеціальні символи.

4. Обробка логічних операторів:

- Розпізнавати ключові слова NOT, AND, OR.

5. Обробка коментарів:

- Ігнорувати все між символами { i } для коментарів.

6. Створення tokenів:

- Після розпізнавання кожного елемента (токену) — створювати відповідний запис у вигляді пари (тип токена, значення токена).
- Для кожного токена можна зберігати його тип (наприклад, KEYWORD, IDENTIFIER, NUMBER, OPERATOR тощо).

7. Обробка помилок:

- Якщо зустрічається невідомий символ або некоректний токен — генерувати помилку.

8. Інтерфейс для взаємодії:

- Розробити функції для введення і виведення результатів аналізу (наприклад, виведення списку токенів після їх розпізнавання).

Можна на основі цього створити програму, яка буде ефективно обробляти текст програми і виділяти коректні токени для подальшої обробки компілятором.

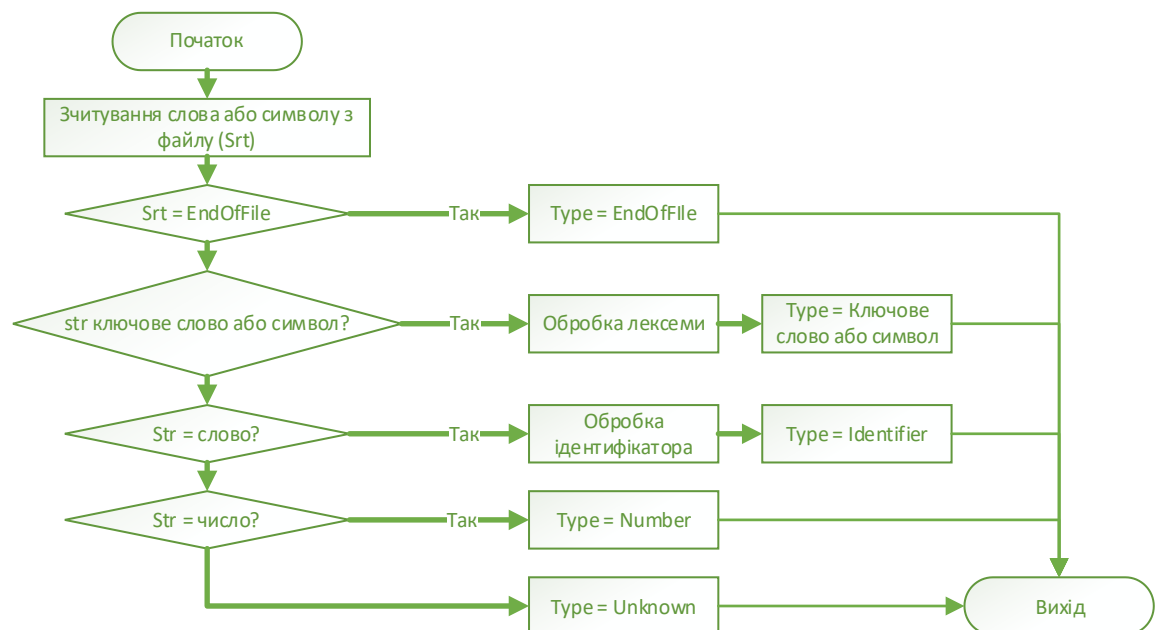


Рис. 3.1 Блок-схема роботи лексичного аналізатора

3.3.2. Опис програми реалізації лексичного аналізатора.

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

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

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

При лексичному аналізі виявляються і відзначаються лексичні помилки (наприклад, недопустимі символи і неправильні ідентифікатори). Лексична фаза відкидає також коментарі та символи лапок у конструкції `String`, оскільки вони

не мають ніякого впливу на виконання програми, отже й на синтаксичний розбір та генерацію коду.

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

3.4. Розробка синтаксичного та семантичного аналізатора.

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

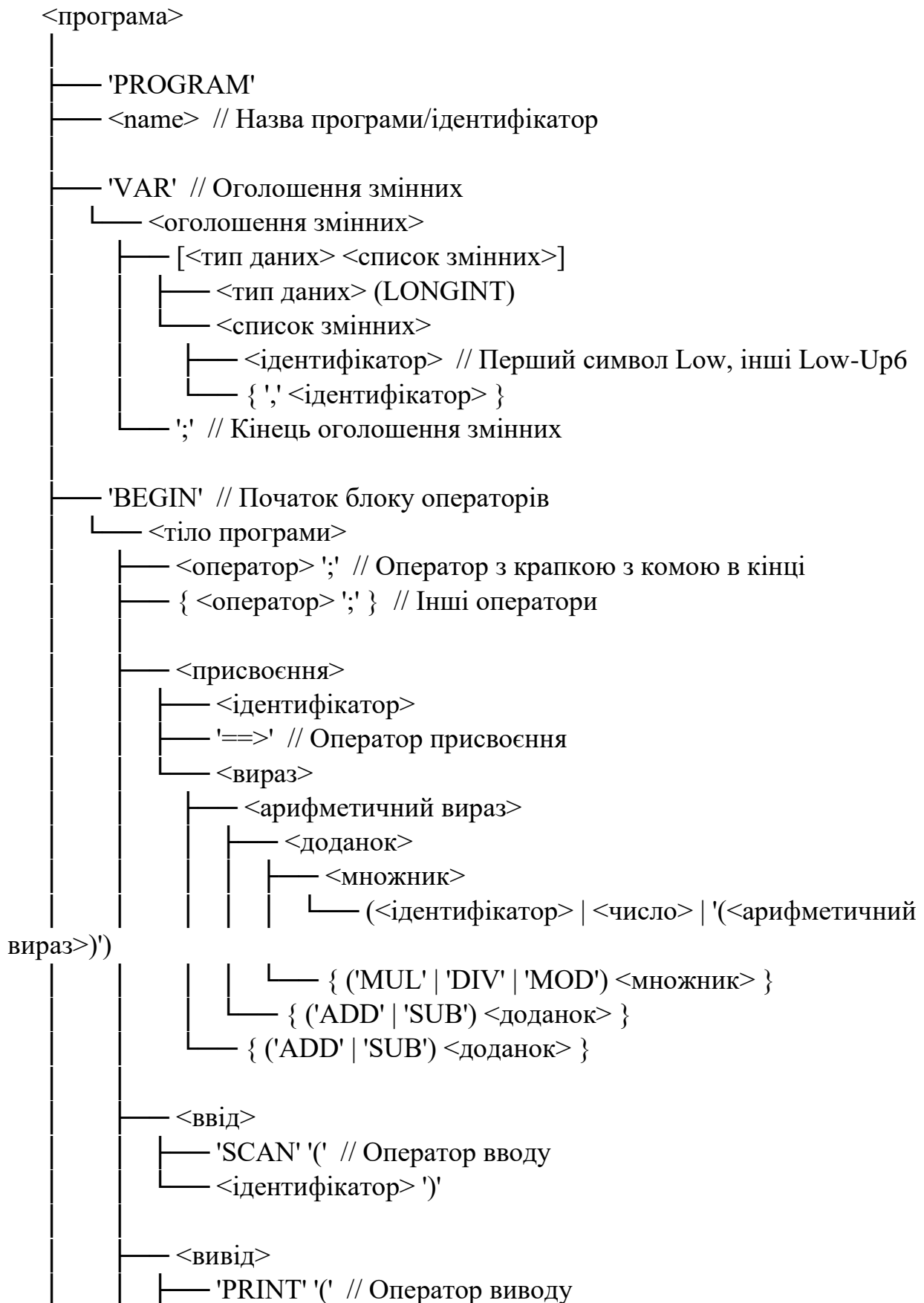
В основі синтаксичного аналізатора лежить Розпізнавач тексту вхідної програми на основі граматики вхідного мови. Як правило, синтаксичні конструкції мов програмування можуть бути описані за допомогою КС-грамматик, рідше зустрічаються мови, які можуть бути описані за допомогою регулярних грамматик. Найчастіше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

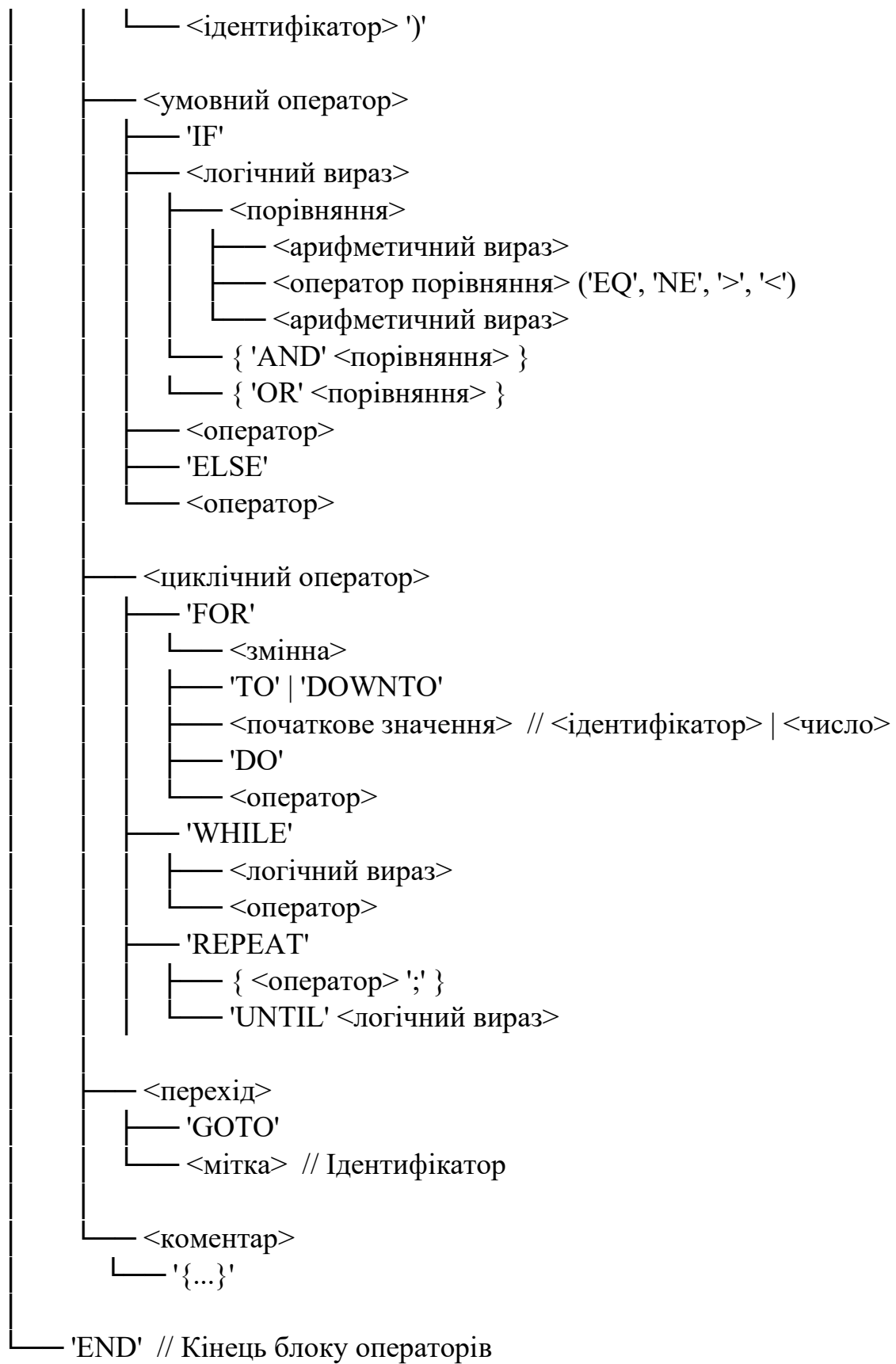
Синтаксичний розбір - це основна частина компіляції на етапі аналізу. Без виконання синтаксичного розбору робота компілятора безглузда, у той час як лексичний аналізатор є зовсім необов'язковим. Усі завдання з перевірки лексики вхідного мови можуть бути вирішені на етапі синтаксичного розбору. Сканер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

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

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

3.4.1 Розробка дерев граматичного розбору





3.4.2 Розробка алгоритму роботи синтаксичного та семантичного аналізатора

Алгоритм роботи:

1. Ініціалізація правил:

- Створюється правило через статичний метод `MakeRule`, що приймає ім'я правила та список елементів `BackusRuleItem`.
- Використовується об'єкт типу `EnableMakeShared`, похідний від `BackusRule`, для створення спільного покажчика через `std::make_shared`.

2. Реєстрація правила:

- Коли нове правило додається до сховища (`BackusRuleStorage`), перевіряється наявність вже зареєстрованих правил з таким самим типом.
- Якщо правило з таким типом вже існує, генерується помилка. Якщо ні, правило додається до сховища для подальшого використання.

3. Перевірка правила (метод `check`):

- Для кожного елементу правила (в `m_backusItem`) виконується перевірка з використанням встановлених прапорців (`Optional`, `PairStart`, `PairEnd`, `Several`).
- Алгоритм працює таким чином:
 - Якщо елемент є обов'язковим і не є кінцем пари (без прапорця `PairEnd`), то реєструється помилка.
 - Якщо елемент допускає кілька варіантів, викликаються методи `oneOrMoreCheck` або `checkItem` для перевірки кожного варіанту.

4. Перевірка одного або більше елементів (метод `oneOrMoreCheck`):

- Якщо прапорець `Several` встановлений, цей метод викликає перевірку елемента один або більше разів, поки не буде досягнуто кінця або елемент більше не відповідатиме умовам.
- Для кожного варіанту викликається метод `checkItem`, який перевіряє конкретний елемент.

5. Перевірка одного елементу (метод `checkItem`):

- Метод перевіряє, чи відповідає конкретний елемент встановленим правилам.
- Якщо елемент не відповідає, помилка додається до `errorsInfo`, і процес перевірки переходить до наступного елементу.

6. Перевірка наявності прапорця (метод `HasFlag`):

- Метод перевіряє, чи містить політика конкретний прапорець за допомогою побітової операції `AND`.
- Це дозволяє визначати, чи застосовуються специфічні умови до даного елементу (наприклад, `Optional`, `PairStart`).

7. Обробка помилок:

- Якщо під час перевірки виявляються помилки, вони реєструються в `errorsInfo`, що включає інформацію про рядок, значення правила, та типи порушених правил.
- Помилки можуть бути пов'язані з тим, що елемент не відповідає правилам або відсутні необхідні варіанти.

8. Обробка після перевірки (Post-handler):

- Метод `setPostHandler` дозволяє визначити функцію, що буде викликана після виконання перевірки.

- Це може бути додаткове оброблення результатів перевірки, корекція чи доповнення результатів перед завершенням процесу.

9. Обробка елементів правила:

- Для кожного елементу виконується перевірка на відповідність з реєстрованими правилами.
- Якщо правило допускає кілька варіантів (Several, OneOrMore), перевірка триває для кожного варіанту.
- Якщо елемент не відповідає правилам і він не є опціональним, перевірка припиняється, і додається помилка.

10. Обробка прапорців (Flag checks):

- Правила можуть мати спеціальні прапорці, такі як Optional, OnlyOne, Several, PairStart, PairEnd, що визначають поведінку перевірки.
- Наприклад, якщо встановлений прапорець PairStart, перевірка має на увазі, що наступний елемент має бути парою для цього елементу.

11. Завершення перевірки:

- Якщо всі елементи правил були успішно перевірені, алгоритм завершується без помилок.
- Якщо виникають помилки, вони додаються до errorsInfo, і процес перевірки продовжується до завершення або до досягнення кінця.

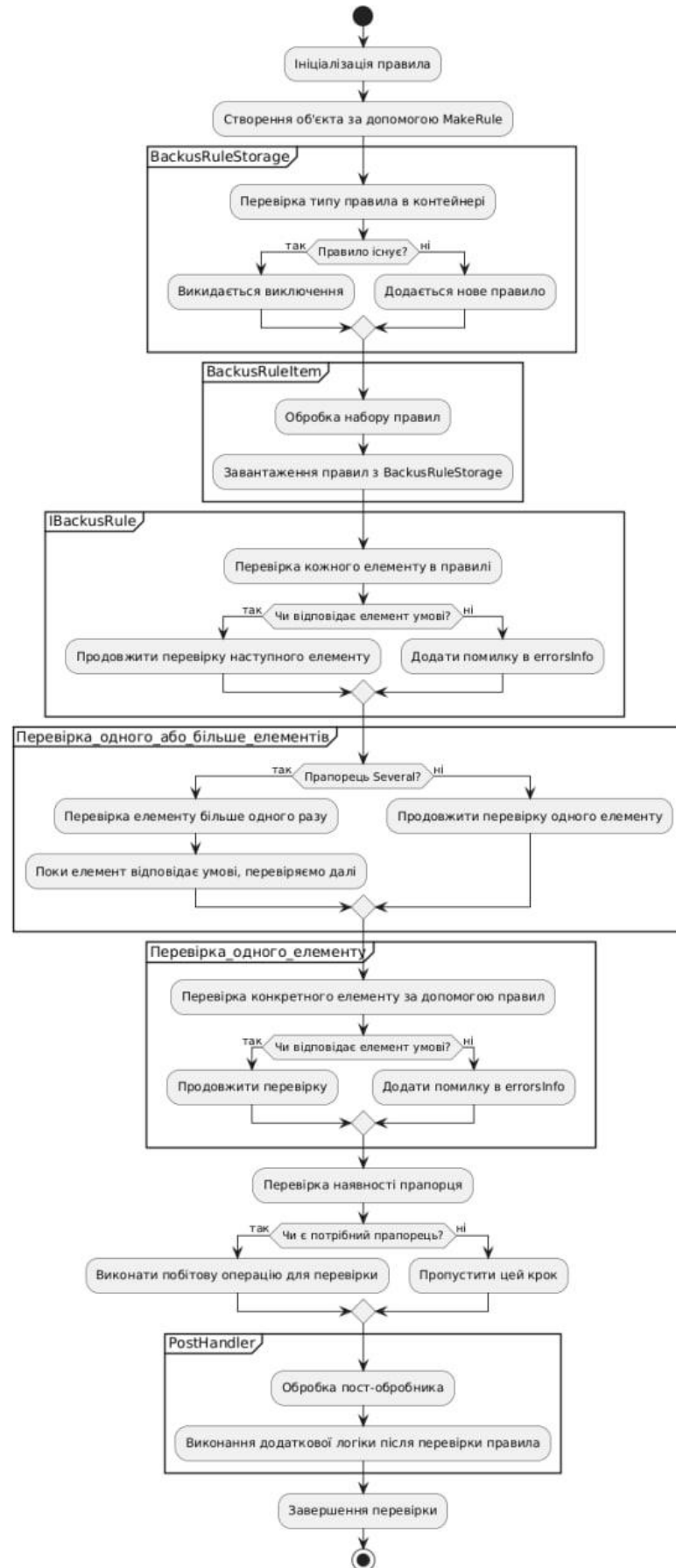


Рис. 3.2 Граф-схема роботи синтаксичного та семантичного аналізатора

3.4.3 Опис програми реалізації синтаксичного та семантичного аналізатора.

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

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

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

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

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 проєкті 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.

3.5.2 Розробка алгоритму роботи 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, зміни, 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 своєї роботи, генератор формує код завершення асемблерної програми.

1. Ініціалізація генератора:

- `setDetails` — встановлюються параметри генерації коду (наприклад, типи даних, префікси для регістрів тощо).
- Створюється список елементів для генерації, який складається з користувацьких елементів коду (числа, рядки, функції).

2. Генерація коду:

- `generateCode` — викликається для створення коду. Використовуються наступні кроки:
- Генерація частин коду (заголовки, дані, код).
- Перетворення введених виразів у постфіксну нотацію для подальшого виконання.
- Генерація кожної частини коду (наприклад, ініціалізація змінних, виклики функцій тощо).

3. Генерація секцій коду:

- `PrintBegin` — додається початкова частина асемблерного коду (директиви, імпорт бібліотек).

- `PrintData` — генерується секція даних (змінні, строки, числа).
- `PrintBeginCodeSegment` — додається кодова секція для ініціалізації та викликів функцій.
- `PrintEnding` — генерується кінець програми (виведення результату, завершення процесу).

4. Обробка операцій:

- `GeneratorUtils` — утилітний клас для перетворення виразів у постфіксну нотацію та обробки операцій.
- Перевіряється пріоритет операцій та порядок їх виконання.

5. Обробка даних:

- Кожен тип даних (число, рядок, сирі дані) реєструється через методи `registerNumberData`, `registerStringData`, `registerRawData`.
- Генератор зберігає ці дані для подальшого використання в коді.

6. Фінальна генерація:

- Виводяться всі частини коду в кінцевий потік (`std::ostream`), що дозволяє створити готовий асемблерний файл.

7. Реєстрація операцій та операндів:

- Операції та операнди реєструються через методи `RegisterOperation` та `RegisterOperand` в класі `GeneratorUtils`.

8. Обробка результатів:

- Генерація результатів виведення на консоль, обробка викликів функцій через стандартні бібліотеки Windows.

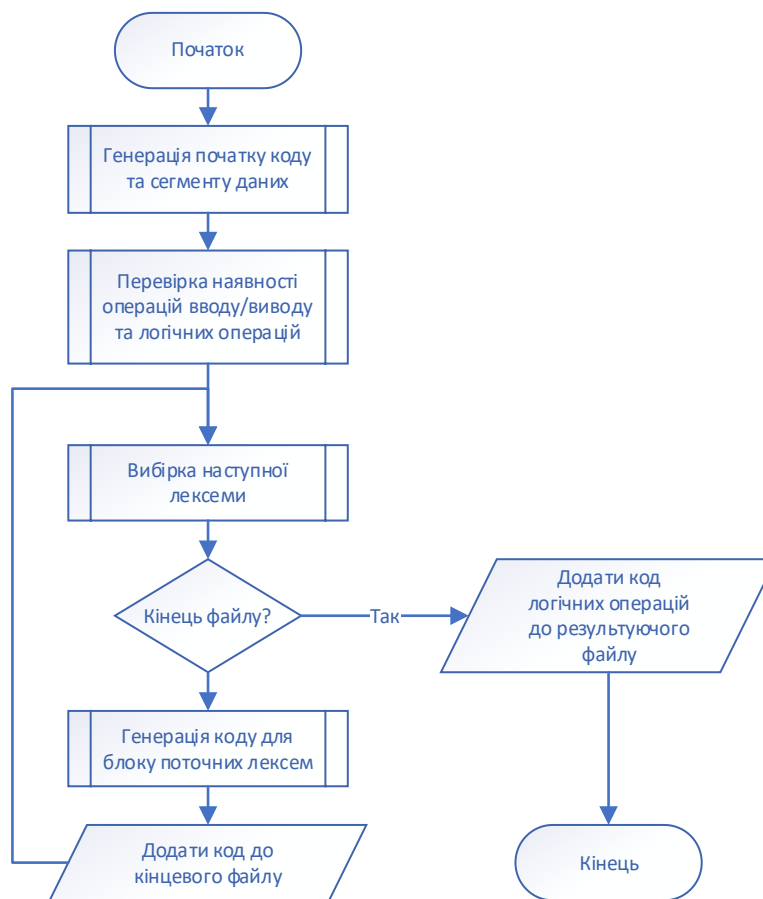


Рис. 3.3 Блок схема генератора коду

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 tokenів для певного правила. А в структурі `BackusRuleItem` описані токени, які при перевірці трактуються програмою як «АБО», тобто повинен бути лише один з описаних tokenів. Наприклад для `write` послідовно необхідний token `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 token 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 токена `Create` 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 << "Greate=====";
    out << "=====Procedure\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 аналізувати 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 b02. 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_b02.exe <ім'я програми>.b02"

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.

Текст програми з помилками

```
{Prog1}
```

```
PROGRAM pROGRA1;
```

```
VAR LONGINT aAAAAAA,bBBBBBB,xXXXXXX,yYYYYYY;
```

```
BEGIN
```

```
PR INT("Input A: ");
```

```
SCAN(aAAeAAAA);
```

```
PRINT("Input B: ");
```

```
SCAN(bBBBBBB);
```

```
PRINT("A + B: ");
```

```
PRINT(aAAAAAA ADD bBBBBBB);
```

```
PRINT("\nA - B: ");
```

```

PRINT(aAAAAAA SUB bBBBBBB);

PRINT("\nA * B: ");

PRINT(aAAAAAA MUL bBBBBBB);

PRINT("\nA / B: ");

PRINT(aAAAAAA DIV bBBBBBB);

PRINT("\nA % B: ");

PRINT(aAAAAAA MOD bBBBBBB);

xXXXXXXX==>(aAAAAAA SUB bBBBBBB) MUL 10 ADD (aAAAAAA ADD bBBBBBB) DIV
10;

yYYYYYYY==>xXXXXXXX ADD (xXXXXXXX MOD 10);

PRINT("\nX = (A - B) * 10 + (A + B) / 10\n");

PRINT(xXXXXXXX);

PRINT("\nY = X + (X MOD 10)\n");

PRINT(yYYYYYYY);

END

```

Текст файлу з повідомленнями про помилки

List of errors

=====

There are 3 lexical errors.

There are 3 syntax errors.

There are 0 semantic errors.

Line 4: Syntax error: Expected: CodeBlok before PR

Line 5: Lexical error: Unknown token: PR

Line 5: Lexical error: Unknown token: INT

Line 5: Syntax error: Expected: End before PR

Line 5: Syntax error: Expected: Operators or OperatorsWithSemicolon before PR

Line 6: Lexical error: Unknown token: aAAeAAAA

4.3. Перевірка роботи транслятора за допомогою тестових задач.

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

Тестова програма №1

```
{Prog1}

PROGRAM pROGRA1;

VAR LONGINT aAAAAAA,bBBBBBB,xXXXXXX,yYYYYYY;

BEGIN

PRINT("Input A: ");

SCAN(aAAAAAA);

PRINT("Input B: ");

SCAN(bBBBBBB);

PRINT("A + B: ");

PRINT(aAAAAAA ADD bBBBBBB);

PRINT("\nA - B: ");

PRINT(aAAAAAA SUB bBBBBBB);

PRINT("\nA * B: ");

PRINT(aAAAAAA MUL bBBBBBB);

PRINT("\nA / B: ");

PRINT(aAAAAAA DIV bBBBBBB);

PRINT("\nA % B: ");

PRINT(aAAAAAA MOD bBBBBBB);

xxxxxxx==>(aAAAAAA SUB bBBBBBB) MUL 10 ADD (aAAAAAA ADD bBBBBBB) DIV
10;
```

```

yYYYYYYY==>xXXXXXXX ADD (xXXXXXXX MOD 10);

PRINT("\nX = (A - B) * 10 + (A + B) / 10\n");

PRINT(xXXXXXXX);

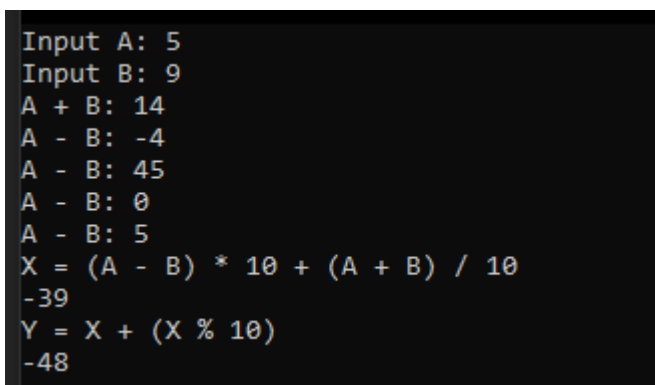
PRINT("\nY = X + (X MOD 10)\n");

PRINT(yYYYYYYY);

END

```

Результат виконання



```

Input A: 5
Input B: 9
A + B: 14
A - B: -4
A - B: 45
A - B: 0
A - B: 5
X = (A - B) * 10 + (A + B) / 10
-39
Y = X + (X % 10)
-48

```

Рис. 5.2 Результат виконання тестової програми №1

Тестова програма №2

```

{Prog2}

PROGRAM pROGRA2;

VAR LONGINT aAAAAAA,bBBBBBB,cCCCCC;

BEGIN

PRINT("Input A: ");

SCAN(aAAAAAA);

PRINT("Input B: ");

SCAN(bBBBBBB);

PRINT("Input C: ");

```

```

SCAN(cCCCCCC);

IF(aAAAAAA > bBBBBBB)

BEGIN

    IF(aAAAAAA > cCCCCCC)

    BEGIN

        GOTO tEMPORA;

    END

    ELSE

    BEGIN

        PRINT(cCCCCCC);

        GOTO oUTCHEK;

        tEMPORA:

        PRINT(aAAAAAA);

        GOTO oUTCHEK;

    END

END

IF(bBBBBBB < cCCCCCC)

BEGIN

    PRINT(cCCCCCC);

END

ELSE

BEGIN

    PRINT(bBBBBBB);

END

oUTCHEK:

```



```
PRINT("\n");
```

```
IF((aAAAAAA EQ bBBBBBB) AND (aAAAAAA EQ cCCCCC) AND (bBBBBBB EQ cCCCCC))
```

```
BEGIN
```

```
    PRINT(1);
```

```
END
```

```
ELSE
```

```
BEGIN
```

```
    PRINT(0);
```

```
END
```

```
PRINT("\n");
```

```
IF((aAAAAAA < 0) OR (bBBBBBB < 0) OR (cCCCCC < 0))
```

```
BEGIN
```

```
    PRINT(- 1);
```

```
END
```

```
ELSE
```

```
BEGIN
```

```
    PRINT(0);
```

```
END
```

```
PRINT("\n");
```

```
IF(NOT(aAAAAAA < (bBBBBBB ADD cCCCCC)))
```

```
BEGIN
```

```
    PRINT(10);
```

```
END
```

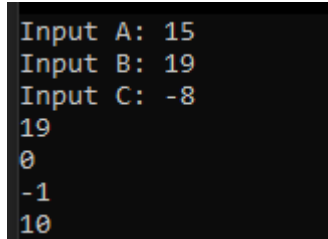
```
ELSE
```

```

BEGIN
    PRINT(0);
END
END

```

Результат виконання



```

Input A: 15
Input B: 19
Input C: -8
19
0
-1
10

```

Рис. 5.3 Результат виконання тестової програми №2

Тестова програма №3

```

{Prog3}
PROGRAM pROGRA3;
VAR LONGINT aAAAAAA,aAAAAA2,bBBBBBBB,xXXXXXXX,cCCCCC1,cCCCCC2;
BEGIN
    PRINT("Input A: ");
    SCAN(aAAAAAA);
    PRINT("Input B: ");
    SCAN(bBBBBBBB);
    PRINT("FOR TO DO");
    FOR aAAAAA2==>aAAAAAA TO bBBBBBBB DO
    BEGIN
        PRINT("\n");
        PRINT(aAAAAA2 MUL aAAAAA2);
    END
END

```

```

PRINT("\nFOR DOWNT0 DO");

FOR aAAAAA2==>bBBBBBB DOWNT0 aAAAAA DO

BEGIN

    PRINT("\n");

    PRINT(aAAAAA2 MUL aAAAAA2);

END

```

```

PRINT("\nWHILE A MUL B: ");

xXXXXXX==>0;

cCCCCC1==>0;

WHILE(cCCCCC1 < aAAAAA)

BEGIN

    cCCCCC2==>0;

    WHILE (cCCCCC2 < bBBBBBB)

        BEGIN

            xXXXXXX==>xXXXXXX ADD 1;

            cCCCCC2==>cCCCCC2 ADD 1;

        END

    cCCCCC1==>cCCCCC1 ADD 1;

END

PRINT(xXXXXXX);

```

```

PRINT("\nREPEAT UNTIL A MUL B: ");

xXXXXXX==>0;

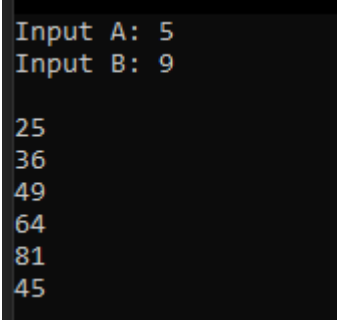
cCCCCC1==>1;

REPEAT

```

```
cCCCCC2==>1;  
  
REPEAT  
  xXXXXXX==>xXXXXXX ADD 1;  
  cCCCCC2==>cCCCCC2 ADD 1;  
UNTIL(NOT(cCCCCC2 > bBBBBBB))  
  
cCCCCC1==>cCCCCC1 ADD 1;  
UNTIL(NOT(cCCCCC1 > aAAAAAA))  
  
PRINT(xXXXXXX);  
  
END
```

Результат виконання

A screenshot of a terminal window with a black background and white text. The text shows the input values for variables A and B, followed by a list of six numbers.

```
Input A: 5  
Input B: 9  
  
25  
36  
49  
64  
81  
45
```

Рис. 5.4 Результат виконання тестової програми №3

ВИСНОВКИ

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування b02, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створено компілятор мови програмування b02, 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 b02. 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 b02 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. Language Processors: Assembler, Compiler and Interpreter

URL: [Language Processors: Assembler, Compiler and Interpreter - GeeksforGeeks](#)

2. Error Handling in Compiler Design

URL: [Error Handling in Compiler Design - GeeksforGeeks](#)

3. Symbol Table in Compiler

URL: [Symbol Table in Compiler - GeeksforGeeks](#)

4. Вікіпедія

URL: [Wikipedia](#)

5. Stack Overflow

URL: [Stack Overflow - Where Developers Learn, Share, & Build Careers](#)

ДОДАТКИ

Додаток А

Таблиці лексем для тестових прикладів

Prog1

#	SYMBOL	TYPE	VALUE	LINE
1	{	LComment	{	1
2		Comment	Prog1	1
3	}	RComment	}	1
4	PROGRAM	Program	PROGRAM	2
5		Identifier	pROGRA1	2
6	;	Semicolon	;	2
7	VAR	Vars	VAR	3
8	LONGINT	VarType	LONGINT	3
9		Identifier	aAAAAAA	3
10	,	Comma	,	3
11		Identifier	bBBBBBB	3
12	,	Comma	,	3
13		Identifier	xXXXXXX	3
14	,	Comma	,	3
15		Identifier	yYYYYYY	3
16	;	Semicolon	;	3
17	BEGIN	Start	BEGIN	4
18	PRINT	Write	PRINT	5
19	(LBracket	(5
20	"	Quotes	"	5
21		String	Input A:	5
22	"	Quotes	"	5
23)	RBracket)	5
24	;	Semicolon	;	5
25	SCAN	Read	SCAN	6
26	(LBracket	(6
27		Identifier	aAAAAAA	6
28)	RBracket)	6
29	;	Semicolon	;	6
30	PRINT	Write	PRINT	7
31	(LBracket	(7
32	"	Quotes	"	7
33		String	Input B:	7
34	"	Quotes	"	7
35)	RBracket)	7
36	;	Semicolon	;	7
37	SCAN	Read	SCAN	8
38	(LBracket	(8
39		Identifier	bBBBBBB	8
40)	RBracket)	8
41	;	Semicolon	;	8
42	PRINT	Write	PRINT	9
43	(LBracket	(9
44	"	Quotes	"	9
45		String	A + B:	9
46	"	Quotes	"	9
47)	RBracket)	9
48	;	Semicolon	;	9
49	PRINT	Write	PRINT	10

50	(LBracket	(10
51		Identifier	aAAAAAA	10
52	ADD	Addition	ADD	10
53		Identifier	bBBBBBB	10
54)	RBracket)	10
55	;	Semicolon	;	10
56	PRINT	Write	PRINT	11
57	(LBracket	(11
58	"	Quotes	"	11
59		String	\nA - B:	11
60	"	Quotes	"	11
61)	RBracket)	11
62	;	Semicolon	;	11
63	PRINT	Write	PRINT	12
64	(LBracket	(12
65		Identifier	aAAAAAA	12
66	SUB	Subtraction	SUB	12
67		Identifier	bBBBBBB	12
68)	RBracket)	12
69	;	Semicolon	;	12
70	PRINT	Write	PRINT	13
71	(LBracket	(13
72	"	Quotes	"	13
73		String	\nA * B:	13
74	"	Quotes	"	13
75)	RBracket)	13
76	;	Semicolon	;	13
77	PRINT	Write	PRINT	14
78	(LBracket	(14
79		Identifier	aAAAAAA	14
80	MUL	Multiplication	MUL	14
81		Identifier	bBBBBBB	14
82)	RBracket)	14
83	;	Semicolon	;	14
84	PRINT	Write	PRINT	15
85	(LBracket	(15
86	"	Quotes	"	15
87		String	\nA / B:	15
88	"	Quotes	"	15
89)	RBracket)	15
90	;	Semicolon	;	15
91	PRINT	Write	PRINT	16
92	(LBracket	(16
93		Identifier	aAAAAAA	16
94	DIV	Division	DIV	16
95		Identifier	bBBBBBB	16
96)	RBracket)	16
97	;	Semicolon	;	16
98	PRINT	Write	PRINT	17
99	(LBracket	(17
100	"	Quotes	"	17
101		String	\nA % B:	17
102	"	Quotes	"	17
103)	RBracket)	17
104	;	Semicolon	;	17
105	PRINT	Write	PRINT	18
106	(LBracket	(18
107		Identifier	aAAAAAA	18
108	MOD	Mod	MOD	18
109		Identifier	bBBBBBB	18
110)	RBracket)	18
111	;	Semicolon	;	18
112		Identifier	xxxxxxx	19

113	==>	Assignment	==>	19
114	(LBracket	(19
115		Identifier	aAAAAAA	19
116	SUB	Subtraction	SUB	19
117		Identifier	bBBBBBB	19
118)	RBracket)	19
119	MUL	Multiplication	MUL	19
120		Number	10	19
121	ADD	Addition	ADD	19
122	(LBracket	(19
123		Identifier	aAAAAAA	19
124	ADD	Addition	ADD	19
125		Identifier	bBBBBBB	19
126)	RBracket)	19
127	DIV	Division	DIV	19
128		Number	10	19
129	;	Semicolon	;	19
130		Identifier	yYYYYYY	20
131	==>	Assignment	==>	20
132		Identifier	xXXXXXX	20
133	ADD	Addition	ADD	20
134	(LBracket	(20
135		Identifier	xXXXXXX	20
136	MOD	Mod	MOD	20
137		Number	10	20
138)	RBracket)	20
139	;	Semicolon	;	20
140	PRINT	Write	PRINT	21
141	(LBracket	(21
142	"	Quotes	"	21
143		String	\nX = (A - B) * 10 + (A + B) / 10\n	21
144	"	Quotes	"	21
145)	RBracket)	21
146	;	Semicolon	;	21
147	PRINT	Write	PRINT	22
148	(LBracket	(22
149		Identifier	xXXXXXX	22
150)	RBracket)	22
151	;	Semicolon	;	22
152	PRINT	Write	PRINT	23
153	(LBracket	(23
154	"	Quotes	"	23
155		String	\nY = X + (X MOD 10)\n	23
156	"	Quotes	"	23
157)	RBracket)	23
158	;	Semicolon	;	23
159	PRINT	Write	PRINT	24
160	(LBracket	(24
161		Identifier	yYYYYYY	24
162)	RBracket)	24
163	;	Semicolon	;	24
164	END	End	END	25
165		EndOfFile		-1

Prog2

#	SYMBOL	TYPE	VALUE	LINE
1	{	LComment	{	1
2		Comment	Prog2	1

3	}	RComment	}	1
4	PROGRAM	Program	PROGRAM	2
5		Identifier	pROGRA2	2
6	;	Semicolon	;	2
7	VAR	Vars	VAR	3
8	LONGINT	VarType	LONGINT	3
9		Identifier	aAAAAAA	3
10	,	Comma	,	3
11		Identifier	bBBBBBB	3
12	,	Comma	,	3
13		Identifier	cCCCCCC	3
14	;	Semicolon	;	3
15	BEGIN	Start	BEGIN	4
16	PRINT	Write	PRINT	5
17	(LBracket	(5
18	"	Quotes	"	5
19		String	Input A:	5
20	"	Quotes	"	5
21)	RBracket)	5
22	;	Semicolon	;	5
23	SCAN	Read	SCAN	6
24	(LBracket	(6
25		Identifier	aAAAAAA	6
26)	RBracket)	6
27	;	Semicolon	;	6
28	PRINT	Write	PRINT	7
29	(LBracket	(7
30	"	Quotes	"	7
31		String	Input B:	7
32	"	Quotes	"	7
33)	RBracket)	7
34	;	Semicolon	;	7
35	SCAN	Read	SCAN	8
36	(LBracket	(8
37		Identifier	bBBBBBB	8
38)	RBracket)	8
39	;	Semicolon	;	8
40	PRINT	Write	PRINT	9
41	(LBracket	(9
42	"	Quotes	"	9
43		String	Input C:	9
44	"	Quotes	"	9
45)	RBracket)	9
46	;	Semicolon	;	9
47	SCAN	Read	SCAN	10
48	(LBracket	(10
49		Identifier	cCCCCCC	10
50)	RBracket)	10
51	;	Semicolon	;	10
52	IF	If	IF	11
53	(LBracket	(11
54		Identifier	aAAAAAA	11
55	>	Greate	>	11
56		Identifier	bBBBBBB	11
57)	RBracket)	11
58	BEGIN	Start	BEGIN	12
59	IF	If	IF	13
60	(LBracket	(13
61		Identifier	aAAAAAA	13
62	>	Greate	>	13
63		Identifier	cCCCCCC	13
64)	RBracket)	13
65	BEGIN	Start	BEGIN	14

66	GOTO	Goto	GOTO	15
67		Identifier	tEMPORA	15
68	;	Semicolon	;	15
69	END	End	END	16
70	ELSE	Else	ELSE	17
71	BEGIN	Start	BEGIN	18
72	PRINT	Write	PRINT	19
73	(LBracket	(19
74		Identifier	cCCCCCC	19
75)	RBracket)	19
76	;	Semicolon	;	19
77	GOTO	Goto	GOTO	20
78		Identifier	oUTCHEK	20
79	;	Semicolon	;	20
80		Identifier	tEMPORA	21
81	:	Colon	:	21
82	PRINT	Write	PRINT	22
83	(LBracket	(22
84		Identifier	aAAAAAA	22
85)	RBracket)	22
86	;	Semicolon	;	22
87	GOTO	Goto	GOTO	23
88		Identifier	oUTCHEK	23
89	;	Semicolon	;	23
90	END	End	END	24
91	END	End	END	25
92	IF	If	IF	26
93	(LBracket	(26
94		Identifier	bBBBBBB	26
95	<	Less	<	26
96		Identifier	cCCCCCC	26
97)	RBracket)	26
98	BEGIN	Start	BEGIN	27
99	PRINT	Write	PRINT	28
100	(LBracket	(28
101		Identifier	cCCCCCC	28
102)	RBracket)	28
103	;	Semicolon	;	28
104	END	End	END	29
105	ELSE	Else	ELSE	30
106	BEGIN	Start	BEGIN	31
107	PRINT	Write	PRINT	32
108	(LBracket	(32
109		Identifier	bBBBBBB	32
110)	RBracket)	32
111	;	Semicolon	;	32
112	END	End	END	33
113		Identifier	oUTCHEK	34
114	:	Colon	:	34
115	PRINT	Write	PRINT	35
116	(LBracket	(35
117	"	Quotes	"	35
118		String	\n	35
119	"	Quotes	"	35
120)	RBracket)	35
121	;	Semicolon	;	35
122	IF	If	IF	36
123	(LBracket	(36
124	(LBracket	(36
125		Identifier	aAAAAAA	36
126	EQ	Equal	EQ	36
127		Identifier	bBBBBBB	36
128)	RBracket)	36

129	AND	And	AND	36
130	(LBracket	(36
131		Identifier	aAAAAAA	36
132	EQ	Equal	EQ	36
133		Identifier	cCCCCCC	36
134)	RBracket)	36
135	AND	And	AND	36
136	(LBracket	(36
137		Identifier	bBBBBBB	36
138	EQ	Equal	EQ	36
139		Identifier	cCCCCCC	36
140)	RBracket)	36
141)	RBracket)	36
142	BEGIN	Start	BEGIN	37
143	PRINT	Write	PRINT	38
144	(LBracket	(38
145		Number	1	38
146)	RBracket)	38
147	;	Semicolon	;	38
148	END	End	END	39
149	ELSE	Else	ELSE	40
150	BEGIN	Start	BEGIN	41
151	PRINT	Write	PRINT	42
152	(LBracket	(42
153		Number	0	42
154)	RBracket)	42
155	;	Semicolon	;	42
156	END	End	END	43
157	PRINT	Write	PRINT	44
158	(LBracket	(44
159	"	Quotes	"	44
160		String	\n	44
161	"	Quotes	"	44
162)	RBracket)	44
163	;	Semicolon	;	44
164	IF	If	IF	45
165	(LBracket	(45
166	(LBracket	(45
167		Identifier	aAAAAAA	45
168	<	Less	<	45
169		Number	0	45
170)	RBracket)	45
171	OR	Or	OR	45
172	(LBracket	(45
173		Identifier	bBBBBBB	45
174	<	Less	<	45
175		Number	0	45
176)	RBracket)	45
177	OR	Or	OR	45
178	(LBracket	(45
179		Identifier	cCCCCCC	45
180	<	Less	<	45
181		Number	0	45
182)	RBracket)	45
183)	RBracket)	45
184	BEGIN	Start	BEGIN	46
185	PRINT	Write	PRINT	47
186	(LBracket	(47
187	-	Minus	-	47
188		Number	1	47
189)	RBracket)	47
190	;	Semicolon	;	47
191	END	End	END	48

192	ELSE	Else	ELSE	49
193	BEGIN	Start	BEGIN	50
194	PRINT	Write	PRINT	51
195	(LBracket	(51
196		Number	0	51
197)	RBracket)	51
198	;	Semicolon	;	51
199	END	End	END	52
200	PRINT	Write	PRINT	53
201	(LBracket	(53
202	"	Quotes	"	53
203		String	\n	53
204	"	Quotes	"	53
205)	RBracket)	53
206	;	Semicolon	;	53
207	IF	If	IF	54
208	(LBracket	(54
209	NOT	Not	NOT	54
210	(LBracket	(54
211		Identifier	aAAAAAA	54
212	<	Less	<	54
213	(LBracket	(54
214		Identifier	bBBBBBB	54
215	ADD	Addition	ADD	54
216		Identifier	cCCCCCC	54
217)	RBracket)	54
218)	RBracket)	54
219)	RBracket)	54
220	BEGIN	Start	BEGIN	55
221	PRINT	Write	PRINT	56
222	(LBracket	(56
223		Number	10	56
224)	RBracket)	56
225	;	Semicolon	;	56
226	END	End	END	57
227	ELSE	Else	ELSE	58
228	BEGIN	Start	BEGIN	59
229	PRINT	Write	PRINT	60
230	(LBracket	(60
231		Number	0	60
232)	RBracket)	60
233	;	Semicolon	;	60
234	END	End	END	61
235	END	End	END	62
236		EndOfFile		-1

Prog3

#	SYMBOL	TYPE	VALUE	LINE
1	{	LComment	{	1
2		Comment	Prog3	1
3	}	RComment	}	1
4	PROGRAM	Program	PROGRAM	2
5		Identifier	pROGRA3	2
6	;	Semicolon	;	2
7	VAR	Vars	VAR	3
8	LONGINT	VarType	LONGINT	3
9		Identifier	aAAAAAA	3
10	,	Comma	,	3
11		Identifier	aAAAAA2	3
12	,	Comma	,	3

13		Identifier	bBBBBBBB	3
14	,	Comma	,	3
15		Identifier	xxxxxxx	3
16	,	Comma	,	3
17		Identifier	cCCCCC1	3
18	,	Comma	,	3
19		Identifier	cCCCCC2	3
20	;	Semicolon	;	3
21	BEGIN	Start	BEGIN	4
22	PRINT	Write	PRINT	5
23	(LBracket	(5
24	"	Quotes	"	5
25		String	Input A:	5
26	"	Quotes	"	5
27)	RBracket)	5
28	;	Semicolon	;	5
29	SCAN	Read	SCAN	6
30	(LBracket	(6
31		Identifier	aAAAAAA	6
32)	RBracket)	6
33	;	Semicolon	;	6
34	PRINT	Write	PRINT	7
35	(LBracket	(7
36	"	Quotes	"	7
37		String	Input B:	7
38	"	Quotes	"	7
39)	RBracket)	7
40	;	Semicolon	;	7
41	SCAN	Read	SCAN	8
42	(LBracket	(8
43		Identifier	bBBBBBBB	8
44)	RBracket)	8
45	;	Semicolon	;	8
46	PRINT	Write	PRINT	9
47	(LBracket	(9
48	"	Quotes	"	9
49		String	FOR TO DO	9
50	"	Quotes	"	9
51)	RBracket)	9
52	;	Semicolon	;	9
53	FOR	For	FOR	10
54		Identifier	aAAAAA2	10
55	==>	Assignment	==>	10
56		Identifier	aAAAAAA	10
57	TO	To	TO	10
58		Identifier	bBBBBBBB	10
59	DO	Do	DO	10
60	BEGIN	Start	BEGIN	11
61	PRINT	Write	PRINT	12
62	(LBracket	(12
63	"	Quotes	"	12
64		String	\n	12
65	"	Quotes	"	12
66)	RBracket)	12
67	;	Semicolon	;	12
68	PRINT	Write	PRINT	13
69	(LBracket	(13
70		Identifier	aAAAAA2	13
71	MUL	Multiplication	MUL	13
72		Identifier	aAAAAA2	13
73)	RBracket)	13
74	;	Semicolon	;	13
75	END	End	END	14

76	PRINT	Write	PRINT	15
77	(LBracket	(15
78	"	Quotes	"	15
79		String	\nFOR DOWNT0 DO	15
80	"	Quotes	"	15
81)	RBracket)	15
82	;	Semicolon	;	15
83	FOR	For	FOR	16
84		Identifier	aAAAAA2	16
85	==>	Assignment	==>	16
86		Identifier	bBBBBBBB	16
87	DOWNT0	DownTo	DOWNT0	16
88		Identifier	aAAAAAA	16
89	DO	Do	DO	16
90	BEGIN	Start	BEGIN	17
91	PRINT	Write	PRINT	18
92	(LBracket	(18
93	"	Quotes	"	18
94		String	\n	18
95	"	Quotes	"	18
96)	RBracket)	18
97	;	Semicolon	;	18
98	PRINT	Write	PRINT	19
99	(LBracket	(19
100		Identifier	aAAAAA2	19
101	MUL	Multiplication	MUL	19
102		Identifier	aAAAAA2	19
103)	RBracket)	19
104	;	Semicolon	;	19
105	END	End	END	20
106	PRINT	Write	PRINT	22
107	(LBracket	(22
108	"	Quotes	"	22
109		String	\nWHILE A MUL B:	22
110	"	Quotes	"	22
111)	RBracket)	22
112	;	Semicolon	;	22
113		Identifier	xxxxxxx	23
114	==>	Assignment	==>	23
115		Number	0	23
116	;	Semicolon	;	23
117		Identifier	cCCCCC1	24
118	==>	Assignment	==>	24
119		Number	0	24
120	;	Semicolon	;	24
121	WHILE	While	WHILE	25
122	(LBracket	(25
123		Identifier	cCCCCC1	25
124	<	Less	<	25
125		Identifier	aAAAAAA	25
126)	RBracket)	25
127	BEGIN	Start	BEGIN	26
128		Identifier	cCCCCC2	27
129	==>	Assignment	==>	27
130		Number	0	27
131	;	Semicolon	;	27
132	WHILE	While	WHILE	28
133	(LBracket	(28
134		Identifier	cCCCCC2	28
135	<	Less	<	28
136		Identifier	bBBBBBBB	28
137)	RBracket)	28
138	BEGIN	Start	BEGIN	29

139		Identifier	xxxxxxx	30
140	==>	Assignment	==>	30
141		Identifier	xxxxxxx	30
142	ADD	Addition	ADD	30
143		Number	1	30
144	;	Semicolon	;	30
145		Identifier	cCCCCC2	31
146	==>	Assignment	==>	31
147		Identifier	cCCCCC2	31
148	ADD	Addition	ADD	31
149		Number	1	31
150	;	Semicolon	;	31
151	END	End	END	32
152		Identifier	cCCCCC1	33
153	==>	Assignment	==>	33
154		Identifier	cCCCCC1	33
155	ADD	Addition	ADD	33
156		Number	1	33
157	;	Semicolon	;	33
158	END	End	END	34
159	PRINT	Write	PRINT	35
160	(LBracket	(35
161		Identifier	xxxxxxx	35
162)	RBracket)	35
163	;	Semicolon	;	35
164	PRINT	Write	PRINT	37
165	(LBracket	(37
166	"	Quotes	"	37
167		String	\nREPEAT UNTIL A MUL B:	37
168	"	Quotes	"	37
169)	RBracket)	37
170	;	Semicolon	;	37
171		Identifier	xxxxxxx	38
172	==>	Assignment	==>	38
173		Number	0	38
174	;	Semicolon	;	38
175		Identifier	cCCCCC1	39
176	==>	Assignment	==>	39
177		Number	1	39
178	;	Semicolon	;	39
179	REPEAT	Repeat	REPEAT	40
180		Identifier	cCCCCC2	41
181	==>	Assignment	==>	41
182		Number	1	41
183	;	Semicolon	;	41
184	REPEAT	Repeat	REPEAT	42
185		Identifier	xxxxxxx	43
186	==>	Assignment	==>	43
187		Identifier	xxxxxxx	43
188	ADD	Addition	ADD	43
189		Number	1	43
190	;	Semicolon	;	43
191		Identifier	cCCCCC2	44
192	==>	Assignment	==>	44
193		Identifier	cCCCCC2	44
194	ADD	Addition	ADD	44
195		Number	1	44
196	;	Semicolon	;	44
197	UNTIL	Until	UNTIL	45
198	(LBracket	(45
199	NOT	Not	NOT	45
200	(LBracket	(45
201		Identifier	cCCCCC2	45

202	>	Greate	>	45
203		Identifier	bBBBBBBB	45
204)	RBracket)	45
205)	RBracket)	45
206		Identifier	cCCCCC1	46
207	==>	Assignment	==>	46
208		Identifier	cCCCCC1	46
209	ADD	Addition	ADD	46
210		Number	1	46
211	;	Semicolon	;	46
212	UNTIL	Until	UNTIL	47
213	(LBracket	(47
214	NOT	Not	NOT	47
215	(LBracket	(47
216		Identifier	cCCCCC1	47
217	>	Greate	>	47
218		Identifier	aAAAAAA	47
219)	RBracket)	47
220)	RBracket)	47
221	PRINT	Write	PRINT	48
222	(LBracket	(48
223		Identifier	xXXXXXXX	48
224)	RBracket)	48
225	;	Semicolon	;	48
226	END	End	END	50
227		EndOfFile		-1

Додаток Б

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

Prog1.asm

.386

.model flat, stdcall

option casemap :none

include masm32\include\windows.inc

include masm32\include\kernel32.inc

include masm32\include\masm32.inc

include masm32\include\user32.inc

include masm32\include\msvcrt.inc

includelib masm32\lib\kernel32.lib

includelib masm32\lib\masm32.lib

includelib masm32\lib\user32.lib

includelib masm32\lib\msvcrt.lib

.DATA

;===User

Data=====

=====

aAAAAAA_ dd 0

bBBBBBB_ dd 0

xXXXXXX_ dd 0

yYYYYYY_ dd 0

DivErrMsg db 13, 10, "Division: Error: division by zero", 0

ModErrMsg db 13, 10, "Mod: Error: division by zero", 0

String_0 db "Input A: ", 0

String_1 db "Input B: ", 0

String_2 db "A + B: ", 0

String_3 db 13, 10, "A - B: ", 0

String_4 db 13, 10, "A * B: ", 0

String_5 db 13, 10, "A / B: ", 0

String_6 db 13, 10, "A % B: ", 0

String_7 db 13, 10, "X = (A - B) * 10 + (A + B) / 10", 13, 10, 0

String_8 db 13, 10, "Y = X + (X MOD 10)", 13, 10, 0

;===Addition

Data=====

=====

hConsoleInputdd ?

hConsoleOutput dd ?

endBuff db 5 dup (?)

msg1310 db 13, 10, 0

CharsReadNum dd ?

InputBuf db 15 dup (?)

OutMessage db "%d", 0

ResMessage db 20 dup (?)

.CODE

```

start:
invoke AllocConsole
invoke GetStdHandle, STD_INPUT_HANDLE
mov hConsoleInput, eax
invoke GetStdHandle, STD_OUTPUT_HANDLE
mov hConsoleOutput, eax

    invoke WriteConsoleA, hConsoleOutput, ADDR String_0, SIZEOF String_0 - 1, 0, 0
    call Input_
    mov aAAAAAA_, eax
    invoke WriteConsoleA, hConsoleOutput, ADDR String_1, SIZEOF String_1 - 1, 0, 0
    call Input_
    mov bBBBBBB_, eax
    invoke WriteConsoleA, hConsoleOutput, ADDR String_2, SIZEOF String_2 - 1, 0, 0
    push aAAAAAA_
    push bBBBBBB_
    call Add_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3 - 1, 0, 0
    push aAAAAAA_
    push bBBBBBB_
    call Sub_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_4, SIZEOF String_4 - 1, 0, 0
    push aAAAAAA_
    push bBBBBBB_
    call Mul_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_5, SIZEOF String_5 - 1, 0, 0
    push aAAAAAA_
    push bBBBBBB_
    call Div_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_6, SIZEOF String_6 - 1, 0, 0
    push aAAAAAA_
    push bBBBBBB_

```

```

call Mod_
call Output_
push aAAAAAA_
push bBBBBBB_
call Sub_
push dword ptr 10
call Mul_
push aAAAAAA_
push bBBBBBB_
call Add_
push dword ptr 10
call Div_
call Add_
pop xXXXXXX_
push xXXXXXX_
push xXXXXXX_
push dword ptr 10
call Mod_
call Add_
pop yYYYYYY_
invoke WriteConsoleA, hConsoleOutput, ADDR String_7, SIZEOF String_7 - 1, 0, 0
push xXXXXXX_
call Output_
invoke WriteConsoleA, hConsoleOutput, ADDR String_8, SIZEOF String_8 - 1, 0, 0
push yYYYYYY_
call Output_
exit_label:
invoke WriteConsoleA, hConsoleOutput, ADDR msg1310, SIZEOF msg1310 - 1, 0, 0
invoke ReadConsoleA, hConsoleInput, ADDR endBuff, 5, 0, 0
invoke ExitProcess, 0

```

```

;===Procedure

```

```

Add=====

```

```

=====

```

Add_ PROC

```
    mov eax, [esp + 8]
    add eax, [esp + 4]
    mov [esp + 8], eax
    pop ecx
    pop eax
    push ecx
    ret
```

Add_ ENDP

;=====

=====

;===Procedure

Div=====

=====

Div_ PROC

```
    pushf
    pop cx
```

```
    mov eax, [esp + 4]
    cmp eax, 0
    jne end_check
    invoke WriteConsoleA, hConsoleOutput, ADDR DivErrMsg, SIZEOF DivErrMsg - 1, 0, 0
    jmp exit_label
```

end_check:

```
    mov eax, [esp + 8]
    cmp eax, 0
    jge gr
```

lo:

```
    mov edx, -1
    jmp less_fin
```

gr:

```
    mov edx, 0
```

less_fin:

```

mov eax, [esp + 8]
idiv dword ptr [esp + 4]
push cx
popf

```

```

mov [esp + 8], eax
pop ecx
pop eax
push ecx
ret

```

Div_ ENDP

;=====

=====

;===Procedure

Input=====

===

Input_ PROC

```

invoke ReadConsoleA, hConsoleInput, ADDR InputBuf, 13, ADDR CharsReadNum, 0

```

```

invoke crt_atoi, ADDR InputBuf

```

```

ret

```

Input_ ENDP

;=====

=====

;===Procedure

Mod=====

=====

Mod_ PROC

```

pushf

```

```

pop cx

```

```

mov eax, [esp + 4]

```

```

    cmp eax, 0
    jne end_check
    invoke WriteConsoleA, hConsoleOutput, ADDR ModErrMsg, SIZEOF ModErrMsg - 1, 0, 0
    jmp exit_label

```

```

end_check:

```

```

    mov eax, [esp + 8]
    cmp eax, 0
    jge gr

```

```

lo:

```

```

    mov edx, -1
    jmp less_fin

```

```

gr:

```

```

    mov edx, 0

```

```

less_fin:

```

```

    mov eax, [esp + 8]
    idiv dword ptr [esp + 4]
    mov eax, edx
    push cx
    popf

```

```

    mov [esp + 8], eax
    pop ecx
    pop eax
    push ecx
    ret

```

```

Mod_ ENDP

```

```

;=====

```

```

=====

```

```

;===Procedure

```

```

Mul=====

```

```

=====

```

```

Mul_ PROC

```

```

    mov eax, [esp + 8]

```

```

    imul dword ptr [esp + 4]
    mov [esp + 8], eax
    pop ecx
    pop eax
    push ecx
    ret

```

Mul_ ENDP

```

;=====

```

```

=====

```

```

;===Procedure

```

```

Output=====

```

```

=====

```

Output_ PROC value: dword

```

    invoke wsprintf, ADDR ResMessage, ADDR OutMessage, value
    invoke WriteConsoleA, hConsoleOutput, ADDR ResMessage, eax, 0, 0
    ret 4

```

Output_ ENDP

```

;=====

```

```

=====

```

```

;===Procedure

```

```

Sub=====

```

```

=====

```

Sub_ PROC

```

    mov eax, [esp + 8]
    sub eax, [esp + 4]
    mov [esp + 8], eax
    pop ecx
    pop eax
    push ecx
    ret

```

Sub_ ENDP


```
;=====
```

```
=====
```

```
end start
```

```
Prog2.asm
```

```
.386
```

```
.model flat, stdcall
```

```
option casemap :none
```

```
include masm32\include\windows.inc
```

```
include masm32\include\kernel32.inc
```

```
include masm32\include\masm32.inc
```

```
include masm32\include\user32.inc
```

```
include masm32\include\msvcrt.inc
```

```
includelib masm32\lib\kernel32.lib
```

```
includelib masm32\lib\masm32.lib
```

```
includelib masm32\lib\user32.lib
```

```
includelib masm32\lib\msvcrt.lib
```

```
.DATA
```

```
;===User
```

```
Data=====
```

```
=====
```

```
aAAAAAA_ dd 0
```

```
bBBBBBB_ dd 0
```

```
cCCCCC_ dd 0
```

```
String_0 db "Input A: ", 0
```

```
String_1 db "Input B: ", 0
```

```
String_2 db "Input C: ", 0
```

```
String_3 db 13, 10, 0
```

```
String_4 db 13, 10, 0
```

```
String_5 db 13, 10, 0
```

```
;===Addition
```

```
Data=====
```

```
hConsoleInputdd    ?
hConsoleOutput    dd    ?
endBuff           db     5 dup (?)
msg1310           db     13, 10, 0
```

```
CharsReadNum      dd    ?
InputBuf           db     15 dup (?)
OutMessage         db     "%d", 0
ResMessage         db     20 dup (?)
```

```
.CODE
```

```
start:
```

```
invoke AllocConsole
```

```
invoke GetStdHandle, STD_INPUT_HANDLE
```

```
mov hConsoleInput, eax
```

```
invoke GetStdHandle, STD_OUTPUT_HANDLE
```

```
mov hConsoleOutput, eax
```

```
    invoke WriteConsoleA, hConsoleOutput, ADDR String_0, SIZEOF String_0 - 1, 0, 0
```

```
    call Input_
```

```
    mov aAAAAAA_, eax
```

```
    invoke WriteConsoleA, hConsoleOutput, ADDR String_1, SIZEOF String_1 - 1, 0, 0
```

```
    call Input_
```

```
    mov bBBBBBB_, eax
```

```
    invoke WriteConsoleA, hConsoleOutput, ADDR String_2, SIZEOF String_2 - 1, 0, 0
```

```
    call Input_
```

```
    mov cCCCCC_, eax
```

```
    push aAAAAAA_
```

```
    push bBBBBBB_
```

```
    call Greate_
```

```
    pop eax
```

```
    cmp eax, 0
```

```
    je endIf2
```

```

    push aAAAAAAA_
    push cCCCCCCC_
    call Greate_
    pop eax
    cmp eax, 0
    je elseLabel1
    jmp tEMPORA_
    jmp endIf1
elseLabel1:
    push cCCCCCCC_
    call Output_
    jmp oUTCHEK_
tEMPORA_:
    push aAAAAAAA_
    call Output_
    jmp oUTCHEK_
endIf1:
endIf2:
    push bBBBBBBB_
    push cCCCCCCC_
    call Less_
    pop eax
    cmp eax, 0
    je elseLabel3
    push cCCCCCCC_
    call Output_
    jmp endIf3
elseLabel3:
    push bBBBBBBB_
    call Output_
endIf3:
oUTCHEK_:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3 - 1, 0, 0
    push aAAAAAAA_
    push bBBBBBBB_

```

```

call Equal_
push aAAAAAA_
push cCCCCC_
call Equal_
call And_
push bBBBBBB_
push cCCCCC_
call Equal_
call And_
pop eax
cmp eax, 0
je elseLabel4
push dword ptr 1
call Output_
jmp endIf4
elseLabel4:
    push dword ptr 0
    call Output_
endIf4:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_4, SIZEOF String_4 - 1, 0, 0
    push aAAAAAA_
    push dword ptr 0
    call Less_
    push bBBBBBB_
    push dword ptr 0
    call Less_
    call Or_
    push cCCCCC_
    push dword ptr 0
    call Less_
    call Or_
    pop eax
    cmp eax, 0
    je elseLabel5
    push dword ptr -1

```

```

    call Output_
    jmp endIf5
elseLabel5:
    push dword ptr 0
    call Output_
endIf5:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_5, SIZEOF String_5 - 1, 0, 0
    push aAAAAAA_
    push bBBBBBB_
    push cCCCCC_
    call Add_
    call Less_
    call Not_
    pop eax
    cmp eax, 0
    je elseLabel6
    push dword ptr 10
    call Output_
    jmp endIf6
elseLabel6:
    push dword ptr 0
    call Output_
endIf6:
exit_label:
    invoke WriteConsoleA, hConsoleOutput, ADDR msg1310, SIZEOF msg1310 - 1, 0, 0
    invoke ReadConsoleA, hConsoleInput, ADDR endBuff, 5, 0, 0
    invoke ExitProcess, 0

```

```

;===Procedure

```

```

Add=====
=====

```

```

Add_ PROC

```

```

    mov eax, [esp + 8]
    add eax, [esp + 4]

```

```

    mov [esp + 8], eax
    pop ecx
    pop eax
    push ecx
    ret

```

```

Add_ ENDP

```

```

;=====

```

```

=====

```

```

;===Procedure

```

```

And=====

```

```

=====

```

```

And_ PROC

```

```

    pushf
    pop cx

```

```

    mov eax, [esp + 8]
    cmp eax, 0
    jnz and_t1
    jz and_false

```

```

and_t1:

```

```

    mov eax, [esp + 4]
    cmp eax, 0
    jnz and_true

```

```

and_false:

```

```

    mov eax, 0
    jmp and_fin

```

```

and_true:

```

```

    mov eax, 1

```

```

and_fin:

```

```

    push cx
    popf

```

```

    mov [esp + 8], eax

```

```

    pop ecx
    pop eax
    push ecx
    ret

```

```
And_ ENDP
```

```
;=====
```

```
=====
```

```
;===Procedure
```

```
Equal=====
```

```
=====
```

```
Equal_ PROC
```

```
    pushf
    pop cx

```

```

    mov eax, [esp + 8]
    cmp eax, [esp + 4]
    jne equal_false
    mov eax, 1
    jmp equal_fin

```

```
equal_false:
```

```
    mov eax, 0
```

```
equal_fin:
```

```
    push cx
    popf

```

```

    mov [esp + 8], eax
    pop ecx
    pop eax
    push ecx
    ret

```

```
Equal_ ENDP
```

```
;=====
```

```
=====
```

;===Procedure

Greate=====

====

Greate_ PROC

pushf

pop cx

mov eax, [esp + 8]

cmp eax, [esp + 4]

jle greate_false

mov eax, 1

jmp greate_fin

greate_false:

mov eax, 0

greate_fin:

push cx

popf

mov [esp + 8], eax

pop ecx

pop eax

push ecx

ret

Greate_ ENDP

;=====

=====

;===Procedure

Input=====

====

Input_ PROC

invoke ReadConsoleA, hConsoleInput, ADDR InputBuf, 13, ADDR CharsReadNum, 0


```
    invoke crt_atoi, ADDR InputBuf
```

```
    ret
```

```
Input_ ENDP
```

```
;=====
```

```
=====
```

```
;===Procedure
```

```
Less=====
```

```
===
```

```
Less_ PROC
```

```
    pushf
```

```
    pop cx
```

```
    mov eax, [esp + 8]
```

```
    cmp eax, [esp + 4]
```

```
    jge less_false
```

```
    mov eax, 1
```

```
    jmp less_fin
```

```
less_false:
```

```
    mov eax, 0
```

```
less_fin:
```

```
    push cx
```

```
    popf
```

```
    mov [esp + 8], eax
```

```
    pop ecx
```

```
    pop eax
```

```
    push ecx
```

```
    ret
```

```
Less_ ENDP
```

```
;=====
```

```
=====
```

;===Procedure

Not=====

Not_ PROC

pushf

pop cx

mov eax, [esp + 4]

cmp eax, 0

jnz not_false

not_t1:

mov eax, 1

jmp not_fin

not_false:

mov eax, 0

not_fin:

push cx

popf

mov [esp + 4], eax

ret

Not_ ENDP

=====

;===Procedure

Or=====

Or_ PROC

pushf

pop cx

mov eax, [esp + 8]

cmp eax, 0

```

    jnz or_true
    jz or_t1
or_t1:
    mov eax, [esp + 4]
    cmp eax, 0
    jnz or_true
or_false:
    mov eax, 0
    jmp or_fin
or_true:
    mov eax, 1
or_fin:
    push cx
    popf

    mov [esp + 8], eax
    pop ecx
    pop eax
    push ecx
    ret

```

Or_ ENDP

```

;=====

```

```

;===Procedure

```

```

Output=====

```

```

====
Output_ PROC value: dword

```

```

    invoke wsprintf, ADDR ResMessage, ADDR OutMessage, value
    invoke WriteConsoleA, hConsoleOutput, ADDR ResMessage, eax, 0, 0
    ret 4

```

Output_ ENDP

```

;=====

```

end start

Prog3.asm

.386

.model flat, stdcall

option casemap :none

include masm32\include\windows.inc

include masm32\include\kernel32.inc

include masm32\include\masm32.inc

include masm32\include\user32.inc

include masm32\include\msvcrt.inc

includelib masm32\lib\kernel32.lib

includelib masm32\lib\masm32.lib

includelib masm32\lib\user32.lib

includelib masm32\lib\msvcrt.lib

.DATA

;===User

Data=====

=====

aAAAAA2_ dd 0

aAAAAAA_ dd 0

bBBBBBB_ dd 0

cCCCCC1_ dd 0

cCCCCC2_ dd 0

xXXXXXX_ dd 0

String_0 db "Input A: ", 0

String_1 db "Input B: ", 0

String_2 db "FOR TO DO", 0

String_3 db 13, 10, 0

String_4 db 13, 10, "FOR DOWNT0 DO", 0

String_5 db 13, 10, 0

String_6 db 13, 10, "WHILE A MUL B: ", 0

```
String_7      db      13, 10, "REPEAT UNTIL A MUL B: ", 0
```

```
;===Addition
```

```
Data=====
```

```
=====
```

```
hConsoleInputdd      ?
hConsoleOutput      dd      ?
endBuff              db      5 dup (?)
msg1310              db      13, 10, 0
```

```
CharsReadNum        dd      ?
InputBuf             db      15 dup (?)
OutMessage           db      "%d", 0
ResMessage           db      20 dup (?)
```

```
.CODE
```

```
start:
```

```
invoke AllocConsole
```

```
invoke GetStdHandle, STD_INPUT_HANDLE
```

```
mov hConsoleInput, eax
```

```
invoke GetStdHandle, STD_OUTPUT_HANDLE
```

```
mov hConsoleOutput, eax
```

```
invoke WriteConsoleA, hConsoleOutput, ADDR String_0, SIZEOF String_0 - 1, 0, 0
```

```
call Input_
```

```
mov aAAAAAA_, eax
```

```
invoke WriteConsoleA, hConsoleOutput, ADDR String_1, SIZEOF String_1 - 1, 0, 0
```

```
call Input_
```

```
mov bBBBBBBB_, eax
```

```
invoke WriteConsoleA, hConsoleOutput, ADDR String_2, SIZEOF String_2 - 1, 0, 0
```

```
push aAAAAAA_
```

```
pop aAAAAA2_
```

```
forPasStart1:
```

```
push bBBBBBBB_
```

```
push aAAAAA2_
```

```
call Less_
```

```

call Not_
pop eax
cmp eax, 0
je forPasEnd1
invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3 - 1, 0, 0
push aAAAAA2_
push aAAAAA2_
call Mul_
call Output_
push aAAAAA2_
push dword ptr 1
call Add_
pop aAAAAA2_
jmp forPasStart1
forPasEnd1:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_4, SIZEOF String_4 - 1, 0, 0
    push bBBBBBBB_
    pop aAAAAA2_
forPasStart2:
    push aAAAAAA_
    push aAAAAA2_
    call Greate_
    call Not_
    pop eax
    cmp eax, 0
    je forPasEnd2
    invoke WriteConsoleA, hConsoleOutput, ADDR String_5, SIZEOF String_5 - 1, 0, 0
    push aAAAAA2_
    push aAAAAA2_
    call Mul_
    call Output_
    push aAAAAA2_
    push dword ptr 1
    call Sub_
    pop aAAAAA2_

```

```

    jmp forPasStart2
forPasEnd2:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_6, SIZEOF String_6 - 1, 0, 0
    push dword ptr 0
    pop xXXXXXXX_
    push dword ptr 0
    pop cCCCCC1_
whileStart2:
    push cCCCCC1_
    push aAAAAAA_
    call Less_
    pop eax
    cmp eax, 0
    je whileEnd2
    push dword ptr 0
    pop cCCCCC2_
whileStart1:
    push cCCCCC2_
    push bBBBBBB_
    call Less_
    pop eax
    cmp eax, 0
    je whileEnd1
    push xXXXXXXX_
    push dword ptr 1
    call Add_
    pop xXXXXXXX_
    push cCCCCC2_
    push dword ptr 1
    call Add_
    pop cCCCCC2_
    jmp whileStart1
whileEnd1:
    push cCCCCC1_
    push dword ptr 1

```

```

    call Add_
    pop cCCCCC1_
    jmp whileStart2
whileEnd2:
    push xXXXXXXX_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_7, SIZEOF String_7 - 1, 0, 0
    push dword ptr 0
    pop xXXXXXXX_
    push dword ptr 1
    pop cCCCCC1_
repeatStart2:
    push dword ptr 1
    pop cCCCCC2_
repeatStart1:
    push xXXXXXXX_
    push dword ptr 1
    call Add_
    pop xXXXXXXX_
    push cCCCCC2_
    push dword ptr 1
    call Add_
    pop cCCCCC2_
    push cCCCCC2_
    push bBBBBBBB_
    call Greate_
    call Not_
    pop eax
    cmp eax, 0
    je repeatEnd1
    jmp repeatStart1
repeatEnd1:
    push cCCCCC1_
    push dword ptr 1
    call Add_

```



```

    pop cCCCCC1_
    push cCCCCC1_
    push aAAAAAA_
    call Greate_
    call Not_
    pop eax
    cmp eax, 0
    je repeatEnd2
    jmp repeatStart2
repeatEnd2:
    push xXXXXXXX_
    call Output_
exit_label:
invoke WriteConsoleA, hConsoleOutput, ADDR msg1310, SIZEOF msg1310 - 1, 0, 0
invoke ReadConsoleA, hConsoleInput, ADDR endBuff, 5, 0, 0
invoke ExitProcess, 0

```

```

;===Procedure

```

```

Add=====

```

```

=====

```

```

Add_ PROC

```

```

    mov eax, [esp + 8]
    add eax, [esp + 4]
    mov [esp + 8], eax
    pop ecx
    pop eax
    push ecx
    ret

```

```

Add_ ENDP

```

```

;=====

```

```

=====

```

```
;===Procedure
```

```
Greate=====
```

```
=====
```

```
Greate_ PROC
```

```
    pushf
```

```
    pop cx
```

```
    mov eax, [esp + 8]
```

```
    cmp eax, [esp + 4]
```

```
    jle greate_false
```

```
    mov eax, 1
```

```
    jmp greate_fin
```

```
greate_false:
```

```
    mov eax, 0
```

```
greate_fin:
```

```
    push cx
```

```
    popf
```

```
    mov [esp + 8], eax
```

```
    pop ecx
```

```
    pop eax
```

```
    push ecx
```

```
    ret
```

```
Greate_ ENDP
```

```
;=====
```

```
=====
```

```
;===Procedure
```

```
Input=====
```

```
=====
```

```
Input_ PROC
```

```
    invoke ReadConsoleA, hConsoleInput, ADDR InputBuf, 13, ADDR CharsReadNum, 0
```

```
    invoke crt_atoi, ADDR InputBuf
```

```
    ret
```

Input_ ENDP

```
;=====
=====
```

```
;===Procedure
```

```
Less=====
===
```

Less_ PROC

pushf

pop cx

mov eax, [esp + 8]

cmp eax, [esp + 4]

jge less_false

mov eax, 1

jmp less_fin

less_false:

mov eax, 0

less_fin:

push cx

popf

mov [esp + 8], eax

pop ecx

pop eax

push ecx

ret

Less_ ENDP

```
;=====
=====
```

====Procedure

Mul=====

Mul_ PROC

```
    mov eax, [esp + 8]
    imul dword ptr [esp + 4]
    mov [esp + 8], eax
    pop ecx
    pop eax
    push ecx
    ret
```

Mul_ ENDP

=====

====Procedure

Not=====

Not_ PROC

```
    pushf
    pop cx
```

```
    mov eax, [esp + 4]
    cmp eax, 0
    jnz not_false
```

not_t1:

```
    mov eax, 1
    jmp not_fin
```

not_false:

```
    mov eax, 0
```

not_fin:

```
    push cx
    popf
```

```
    mov [esp + 4], eax
```

```
    ret
```

```
Not_ ENDP
```

```
;=====
```

```
=====
```

```
;===Procedure
```

```
Output=====
```

```
=====
```

```
Output_ PROC value: dword
```

```
    invoke wsprintf, ADDR ResMessage, ADDR OutMessage, value
```

```
    invoke WriteConsoleA, hConsoleOutput, ADDR ResMessage, eax, 0, 0
```

```
    ret 4
```

```
Output_ ENDP
```

```
;=====
```

```
=====
```

```
;===Procedure
```

```
Sub=====
```

```
=====
```

```
Sub_ PROC
```

```
    mov eax, [esp + 8]
```

```
    sub eax, [esp + 4]
```

```
    mov [esp + 8], eax
```

```
    pop ecx
```

```
    pop eax
```

```
    push ecx
```

```
    ret
```

```
Sub_ ENDP
```

```
;=====
```

```
=====
```

```
end start
```

В. Документований текст програмних модулів (лістинги)

Main.cpp

```
#include "stdafx.h"
#include "Controller.h"
#include "Core/Parser/TokenRegister.h"
#include "Core/Parser/TokenParser.h"
#include "Core/Generator/Generator.h"

int main(int argc, std::string* argv)
{
    try
    {
        std::filesystem::path file;

        const std::string extension = ".b02";

        const std::string longLine =
            "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~";

        std::cout << longLine << std::endl;
        std::cout << "TRANSLATOR (" << extension << "->ASSEMBLER)" << std::endl;
        std::cout << longLine << std::endl;

        if (argc != 2)
        {
            printf("Input file name\n");
            std::cin >> file;
        }
        else
        {
            file = argv->c_str();
        }

        Init();

        if (file.extension() != extension)
        {
            std::cout << longLine << std::endl;
            std::cout << "Wrong file extension" << std::endl;
            system("pause");
            return 0;
        }

        std::string fileName = file.replace_extension("").string();
        std::string errorFileName = fileName + "_errors.txt";
        std::string lexemsFileName = fileName + "_lexems.txt";
        std::string tokensFileName = fileName + "_tokens.txt";
        std::string asmFileName = fileName + ".asm";

        std::cout << longLine << std::endl;
        std::cout << "Breaking into lexems are starting..." << std::endl;
        std::fstream inputFile{ fileName + extension, std::ios::in };
        auto tokens = TokenParser::Instance()->tokenize(inputFile);
        inputFile.close();
        std::cout << "Breaking into lexems completed. There are " << tokens.size() << "
lexems" << std::endl;

        std::fstream lexemsFile(lexemsFileName, std::ios::out);
        TokenParser::PrintTokens(lexemsFile, tokens);
        lexemsFile.close();
        std::cout << "Report file: " << lexemsFileName << std::endl;
    }
}
```

```

std::cout << longLine << std::endl;

std::cout << "Error checking are starting... " << std::endl;
std::fstream errorFile(errorFileName, std::ios::out);
auto semanticCheckRes = CheckSemantic(errorFile, tokens);
errorFile.close();
if (semanticCheckRes)
{
    std::cout << "There are no errors in the file" << std::endl;
    std::cout << longLine << std::endl;
}
else
{
    std::cout << "There are errors in the file. Check " << errorFileName << " for
more information" << std::endl;
    std::cout << longLine << std::endl;
}

std::fstream tokensFile(tokensFileName, std::ios::out);
TokenParser::PrintTokens(tokensFile, tokens);
tokensFile.close();
std::cout << "There are " << tokens.size() << " tokens." << std::endl;
std::cout << "Report file: " << tokensFileName << std::endl;

if (semanticCheckRes)
{
    std::cout << longLine << std::endl;
    std::cout << "Code generation is starting..." << std::endl;
    std::fstream asmFile(asmFileName, std::ios::out);
    Generator::Instance()->generateCode(asmFile, tokens);
    asmFile.close();

    if (std::filesystem::is_directory("masm32"))
    {
        std::cout << "Code generation is completed" << std::endl;
        std::cout << longLine << std::endl;
        system(std::string("masm32\\bin\\ml /c /coff " + fileName +
".asm").c_str());
        system(std::string("masm32\\bin\\Link /SUBSYSTEM:WINDOWS " + fileName +
".obj").c_str());
    }
    else
    {
        std::cout << "WARNING!" << std::endl;
        std::cout << "Can't compile asm file, because masm32 doesn't exist" <<
std::endl;
    }
}

}
catch (const std::exception& ex)
{
    std::cout << "Error: " << ex.what() << std::endl;
}
catch (...)
{
    std::cout << "Unknown internal error. Better call Saul" << std::endl;
}

system("pause");
return 0;

}

```

Parser.cpp

```

#include "stdafx.h"
#include "Core/Parser/TokenParser.h"
#include "Utils/StringUtils.h"
#include "Tokens/Common/EndOfFile.h"

std::list<std::shared_ptr<IToken>> TokenParser::tokenize(std::istream& input)
{
    m_tokens.clear();

    int curLine = 1;
    std::string token;
    for (char ch; input.get(ch);)
    {
        if (!token.empty() && ((IsAllowedSymbol(token.front()) != IsAllowedSymbol(ch)) ||
IsTabulation(ch)))
            recognizeToken(token, curLine);

        if (IsNewLine(ch))
            ++curLine;

        if (isUnchangedTextTokenLast())
        {
            std::string unchangedTextTokenValue{ token };
            token.clear();
            int unchangedTextTokenLine{ curLine };

            const auto& [target, left, right] = m_unchangedTextTokens[m_tokens.back()-
>lexeme()];
            auto rBorderLex = right ? right->lexeme() : "\n";

            do
            {
                if (IsNewLine(ch))
                    ++curLine;

                unchangedTextTokenValue += ch;
            }
            while (!StringUtils::Compare(unchangedTextTokenValue, rBorderLex,
StringUtils::EndWith) && input.get(ch));

            unchangedTextTokenValue = unchangedTextTokenValue.substr(0,
unchangedTextTokenValue.size() - rBorderLex.size());
            m_tokens.push_back(target->tryCreateToken(unchangedTextTokenValue));
            m_tokens.back()->setLine(unchangedTextTokenLine);

            if (right)
            {
                m_tokens.push_back(right->tryCreateToken(rBorderLex));
                m_tokens.back()->setLine(curLine);
            }

            continue;
        }

        if (!IsTabulation(ch))
            token += ch;
    }

    if (!token.empty())
        recognizeToken(token, curLine);
}

```



```

    m_tokens.push_back(std::make_shared<EndOfFile>());
    return m_tokens;
}

void TokenParser::regToken(std::shared_ptr<IToken> token, int priority)
{
    throwIfTokenRegistered(token);

    if (priority == NoPriority)
        priority = static_cast<int>(token->lexeme().size());

    m_priorityTokens.insert(std::make_pair(priority, token));
}

void TokenParser::regUnchangedTextToken(std::shared_ptr<IToken> target,
std::shared_ptr<IToken> lBorder, std::shared_ptr<IToken> rBorder)
{
    if(rBorder)
        throwIfTokenRegistered(rBorder);

    regToken(lBorder);
    throwIfTokenRegistered(target);

    m_unchangedTextTokens.try_emplace(lBorder->lexeme(), target, lBorder, rBorder);
}

void TokenParser::throwIfTokenRegistered(std::shared_ptr<IToken> token)
{
    auto start = m_priorityTokens.lower_bound(static_cast<int>(token->lexeme().size()));

    auto priorToken = std::find_if(start, m_priorityTokens.end(),
        [&token](const auto& pair) {
            return token->type() == pair.second->type();
        });

    auto unchTextToken = std::ranges::find_if(m_unchangedTextTokens,
        [&token](const auto& pair) {
            auto type = token->type();
            const auto& [main, left, right] = pair.second;
            return type == main->type() ||
                type == left->type() ||
                right && type == right->type();
        });

    if(priorToken != m_priorityTokens.end() || unchTextToken !=
m_unchangedTextTokens.end())
        throw std::runtime_error("TokenParser: Token with type " + token->type() + "
already registered");
}

void TokenParser::recognizeToken(std::string& token, int curLine)
{
    if(m_priorityTokens.empty())
        throw std::runtime_error("TokenParser: No tokens registered");

    auto start = m_priorityTokens.lower_bound(static_cast<int>(token.size()));

    for (auto it = start; it != m_priorityTokens.end(); ++it)
    {
        auto curRegToken = it->second;
        if (auto newToken = curRegToken->tryCreateToken(token); newToken)
        {
            m_tokens.push_back(newToken);
            m_tokens.back()->setLine(curLine);
            break;
        }
    }
}

```

```

    }
}

if (!token.empty() && !isUnchangedTextTokenLast())
    recognizeToken(token, curLine);
}

bool TokenParser::isUnchangedTextTokenLast()
{
    if (!m_tokens.empty() && m_unchangedTextTokens.contains(m_tokens.back()->lexeme()))
    {
        auto const& [target, left, right] = m_unchangedTextTokens[m_tokens.back()-
>lexeme()];
        if (m_tokens.size() >= 2)
        {
            if (target->type() != (*(++m_tokens.rbegin()))->type())
                return true;
        }
        else
            return true;
    }

    return false;
}

bool TokenParser::IsNewLine(const char& ch)
{
    return ch == '\n';
}

bool TokenParser::IsTabulation(const char& ch)
{
    return ch == ' ' || ch == '\t' || IsNewLine(ch);
}

bool TokenParser::IsAllowedSymbol(const char& ch)
{
    return !isalpha(ch) || !isdigit(ch) || IsAllowedSpecialSymbol(ch);
}

bool TokenParser::IsAllowedSpecialSymbol(const char& ch)
{
    std::set<char> allowedSymblos{ '_', '-' };
    return allowedSymblos.contains(ch);
}

```