

Міністерство освіти і науки України  
Національний університет “Львівська політехніка”



## **Курсовий проект**

З дисципліни «Системне програмування»  
на тему: "Розробка системних програмних модулів  
та компонент систем програмування."  
Розробка транслятора з вхідної мови програмування"  
**Варіант №5**

**Виконав:** ст. гр. КІ-307  
Ващишин Ігор  
**Перевірив:**  
Козак Н.Б.

Львів-2024

## Анотація

Цей курсовий проект приводить до розробки транслятора, який здатен конвертувати вхідну мову, визначену відповідно до варіанту, у мову 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

Анотація2

Завдання до курсового проекту4

Вступ6

1. Огляд методів та способів проектування трансляторів7
2. Формальний опис вхідної мови програмування10
  - 2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура10
  - 2.2. Опис термінальних символів та ключових слів12
3. Розробка транслятора вхідної мови програмування14
  - 3.1. Вибір технології програмування14
  - 3.2. Проектування таблиць транслятора15
  - 3.3. Розробка лексичного аналізатора17
    - 3.3.1. Розробка блок-схеми алгоритму18
    - 3.3.2. Опис програми реалізації лексичного аналізатора18
  - 3.4. Розробка синтаксичного та семантичного аналізатора20
    - 3.4.1. Опис програми реалізації синтаксичного та семантичного аналізатора21
    - 3.4.2. Розробка граф-схеми алгоритму22
  - 3.5. Розробка генератора коду23
    - 3.5.1. Розробка граф-схеми алгоритму24
    - 3.5.2. Опис програми реалізації генератора коду25
4. Опис програми26
  - 4.1. Опис інтерфейсу та інструкція користувачеві29
5. Відлагодження та тестування програми30
  - 5.1. Виявлення лексичних та синтаксичних помилок30
  - 5.2. Виявлення семантичних помилок31
  - 5.3. Загальна перевірка коректності роботи транслятора31
  - 5.4. Тестова програма №133
  - 5.5. Тестова програма №234
  - 5.6. Тестова програма №335

Висновки38

Список використаної літератури39

Додатки40

# Завдання до курсового проекту

## Варіант 5

Завдання на курсовий проект

1. Цільова мова транслятора – асемблер для 32-розрядного процесора.
2. Для отримання виконавчого файлу на виході розробленого транслятора скористатися програмами ml.exe і link.exe.
3. Мова розробки транслятора: C++.
4. Реалізувати оболонку або інтерфейс з командного рядка.
5. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.
6. На виході розробленого транслятора мають створюватись такі файли:
  - файл з лексемами;
  - файл з повідомленнями про помилки (або про їх відсутність);
  - файл на мові асемблера;
  - об'єктний файл;
  - виконавчий файл.
7. Назва вхідної мови програмування утворюється від першої букви у прізвищі студента та останніх двох цифр номера його варіанту. Саме таке розширення повинні мати текстові файли, написані на цій мові програмування.

В моєму випадку це .v05

Опис вхідної мови програмування:

- Тип даних: INT\_2
- Блок тіла програми: PROGRAM <name>; BEGIN VAR...; END
- Оператор вводу: INPUT ()
- Оператор виводу: OUTPUT ()
- Оператори: IF ELSE (C)  
GOTO (C)  
FOR-TO-DO (Паскаль)  
FOR-DOWNTO-DO (Паскаль)  
WHILE (Бейсік)  
REPEAT-UNTIL (Паскаль)
- Регістр ключових слів: Up
- Регістр ідентифікаторів: Up-Low4 перший символ Up
- Операції арифметичні: +, -, \*, DIV, MOD
- Операції порівняння: =, <>, GT, LT
- Операції логічні: !!, &&, ||
- Коментар: { \*... \* }
- Ідентифікатори змінних, числові константи

- Оператор присвоєння: <-

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

## Вступ

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

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

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

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

# 1.Огляд методів та способів проектування трансляторів

Термін "транслятор" визначає обслуговуючу програму, що проводить трансляцію вихідної програми, представленої на вхідній мові програмування, у робочу програму, яка відображена на 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 граматиці мови програмування. У сучасній практиці найчастіше використовуються методи аналізу, такі як LL (1) або LR (1) та їхні варіанти (рекурсивний спуск для LL (1) або LR (1), LR (0), SLR (1), LALR (1) та інші для LR (1)). Рекурсивний спуск застосовується частіше при ручному програмуванні синтаксичного аналізатора, тоді як LR (1) використовується при автоматичній генерації синтаксичних аналізаторів.

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

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

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

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

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

Машинно-залежні оптимізації, як правило, проводяться на етапі генерації коду, і вони орієнтовані на конкретну архітектуру машини. Ці оптимізації можуть включати розподіл регістрів, вибір довгих або коротких переходів та оптимізацію вартості команд для конкретних послідовностей команд.



Глобальна оптимізація спрямована на поліпшення ефективності всієї програми і базується на глобальному потоковому аналізі, який виконується на графі програми. Цей аналіз враховує властивості програми, такі як межпроцедурний аналіз, міжмодульний аналіз та аналіз галузей життя змінних.

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

Важливо відзначити, що фази транслятора можуть бути відсутніми або об'єднаними в залежності від конкретної реалізації. В простіших випадках, таких як у випадку однопроходових трансляторів, може відсутній окремий етап генерації проміжного представлення та оптимізації, а інші фази можуть бути об'єднані в одну, при цьому не створюється явно побудованого синтаксичного дерева.

## 2.Формальний опис вхідної мови програмування

### 2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура

Однією з перших задач, що виникають при побудові компілятора, є визначення вхідної мови програмування. Для цього використовують різні способи формального опису, серед яких я застосував розширену нотацію Бекуса-Наура (Backus/Naur Form - BNF).

```
topRule = "PROGRAM" identifier, ":", "BEGIN", [ varsBlok ], ":", write | read |  
assignment | ifStatement | goto_statement | labelRule | forToOrDownToDoRule |  
while | repeatUntil, "END"
```

```
varsBlok = "VAR INT_2", identifier, { ",", identifier }
```

```
identifier = up_letter, low_letter, low_letter, low_letter, low_letter
```

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

```
codeBlok = "BEGIN", write | read | assignment | ifStatement | goto_statement |  
labelRule | forToOrDownToDoRule | while | repeatUntil, "END"
```

```
read = "INPUT", identifier, ":"
```

```
write = "OUTPUT", ( equation | stringRule ), ":"
```

```
assignment = identifier, equation, ":"
```

```
ifStatement = "IF", equation, codeBlok, elseStatement
```

```
elseStatement = "ELSE", codeBlok
```

```
goto_statement = "GOTO", ident, ":"
```

```
labelRule = identifier
```

```
forToOrDownToDoRule = "FOR", assignment, "TO" | "DOWNT", equation,  
"DO", codeBlok
```

```
while = "WHILE", "equation ", "codeBlok "
```

```
repeatUntil = "REPEAT", codeBlok, "UNTIL", equation
```

```
equation = signedNumber | identifier | notRule , { (operationAndIdentOrNumber |  
equation) }
```

```
notRule = notOperation signedNumber | identifier | equation
```

```

operationAndIdentOrNumber = mult | arithmetic | logic | compare, signedNumber |
identifier | equation
arithmetic = "+" | "-"
mult = "*" | "DIV" | "MOD"
logic = "&&" | "||"
notOperation = "!!"
compare = "=" | "<>" | "LT" | "GT"
stringRule = "<string>"
comment = LComment, string, RComment
LComment = { *
RComment = *}
string = { low_letter | up_letter | signedNumber }
signedNumber = [sign] digit[{ digit}]
unsignedNumber = ((nz_digit, { digit})) | digit_0
sign = "+" | "-"
low_letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "n" | "m" |
"o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
up_letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "N" |
"M" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
digit = digit_0 | nz_digit
nz_digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
digit_0 = "0"

```

## 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
INPUT	Оператор вводу змінних
OUTPUT	Оператор виводу (змінних або рядкових констант)
<-	Оператор присвоєння
IF	Оператор умови
ELSE	Оператор умови
GOTO	Оператор переходу
LABEL	Мітка переходу
FOR	Оператор циклу
TO	Інкремент циклу
DOWNTO	Декремент циклу
DO	Початок тіла циклу
WHILE	Оператор циклу
REPEAT	Початок тіла циклу
UNTIL	Оператор циклу
+	Оператор додавання
-	Оператор віднімання

*	Оператор множення
DIV	Оператор ділення
MOD	Оператор знаходження залишку від ділення
=	Оператор перевірки на рівність
<>	Оператор перевірки на нерівність
LT	Оператор перевірки чи менше
GT	Оператор перевірки чи більше
!!	Оператор логічного заперечення
&&	Оператор кон'юнкції
	Оператор диз'юнкції
INT_2	16-ти розрядні знакові цілі
{*...*}	Коментар
,	Розділювач
;	Ознака кінця оператора
(	Відкриваюча дужка
)	Закриваюча дужка

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

## 3. Розробка транслятора вхідної мови програмування

### 3.1. Вибір технології програмування

Для ефективної роботи створюваної програми важливу роль відіграє попереднє складення алгоритму роботи програми, алгоритму написання програми і вибір технології програмування.

Тому при складанні транслятора треба брати до уваги швидкість компіляції, якість об'єктної програми. Проект повинен давати можливість просто вносити зміни.

В реалізації мов високого рівня часто використовується специфічний тільки для компіляції засіб “розкрутки”. З кожним транслятором завжди зв'язані три мови програмування:  $X$  – початкова,  $Y$  – об'єктна та  $Z$  – інструментальна. Транслятор перекладає програми мовою  $X$  в програми, складені мовою  $Y$ , при цьому сам транслятор є програмою написаною мовою  $Z$ .

При розробці даного курсового проекту був використаний висхідний метод синтаксичного аналізу.

Також був обраний прямий метод лексичного аналізу. Характерною ознакою цього методу є те, що його реалізація відбувається без повернення назад. Його можна сприймати, як один спільний скінченний автомат. Такий автомат на кожному кроці читає один вхідний символ і переходить у наступний стан, що наближає його до розпізнавання поточної лексеми чи формування інформації про помилки. Для лексем, що мають однакові підланцюжки, автомат має спільні фрагменти, що реалізують єдину множину станів. Частини, що відрізняються, реалізуються своїми фрагментами

## 3.2. Проектування таблиць транслятора

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

- 1) 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 символів та ключових слів

Токен	Значення
Program	PROGRAM
Start	BEGIN
Vars	VAR
End	END
VarType	INT_2
Read	INPUT
Write	OUTPUT
Assignment	<-
If	IF
Else	ELSE
Goto	GOTO
Colon	:
Label	
For	FOR
To	TO
DownTo	DOWNT0

Do	DO
While	WHILE
Repeat	REPEAT
Until	UNTIL
Addition	+
Subtraction	-
Multiplication	*
Division	DIV
Mod	MOD
Equal	=
NotEqual	<>
Less	LT
Greater	GT
Not	!!
And	&&
Or	
Plus	+
Minus	-
Identifier	
Number	
String	
Undefined	
Unknown	
Comma	,
Quotes	“
Semicolon	;
LBracket	(
RBracket	)
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 блок-схеми алгоритму

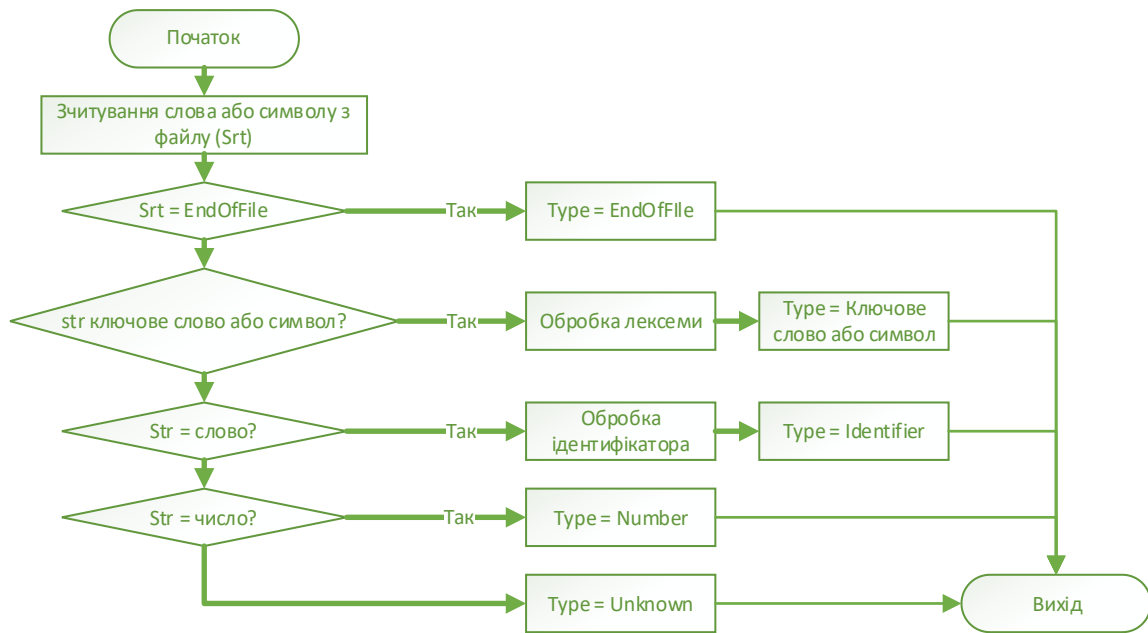


Рис. 3.1 Блок-схема роботи лексичного аналізатора

### 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, літери 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 зарезервованими 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 граматик. Найчастіше регулярні граматики застосовні до мов асемблера, а мови високого рівня побудовані на основі КС-мов.

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

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

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

### **3.4.1. Опис програми реалізації синтаксичного та семантичного аналізатора**

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

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

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

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

### 3.4.2. Розробка граф-схеми алгоритму

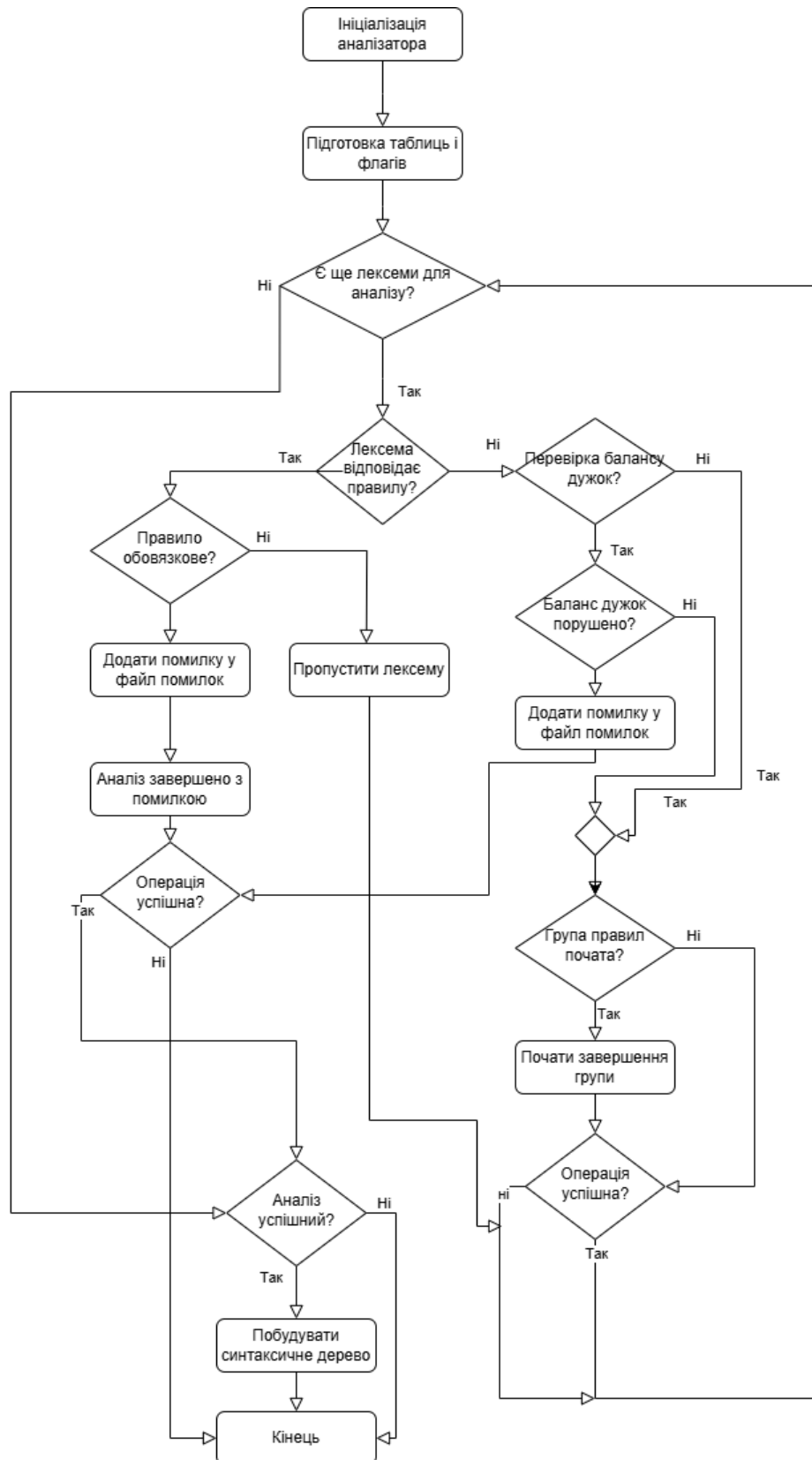


Рис. 3.2 Граф-схема роботи синтаксичного аналізатора

### 3.5. Розробка генератора коду

Синтаксичне дерево в чистому вигляді несе тільки інформацію про структуру програми. Насправді в процесі генерації коду потрібна також інформація про змінні (наприклад, їх адреси), процедури (також адреси, рівні), мітки і т.д. Для представлення цієї інформації можливі різні рішення. Найбільш поширені два:

- інформація зберігається у таблицях генератора коду;
- інформація зберігається у відповідних вершинах дерева.

Розглянемо, наприклад, структуру таблиць, які можуть бути використані в поєднанні з Лідер-представленням. Оскільки Лідер-представлення не містить інформації про адреси змінних, значить, цю інформацію потрібно формувати в процесі обробки оголошень і зберігати в таблицях. Це стосується і описів масивів, записів і т.д. Крім того, в таблицях також повинна міститися інформація про процедури (адреси, рівні, модулі, в яких процедури описані, і т.д.). При вході в процедуру в таблиці рівнів процедур заводиться новий вхід - вказівник на таблицю описів. При виході вказівник поновлюється на старе значення. Якщо проміжне представлення - дерево, то інформація може зберігатися в вершинах самого дерева.

Генерація коду – це машинно-залежний етап компіляції, під час якого відбувається побудова машинного еквівалента вхідної програми. Зазвичай входом для генератора коду служить проміжна форма представлення програми, а на виході може з'являтися об'єктний код або модуль завантаження.

Генератор асемблерного коду приймає масив лексем без помилок. Якщо на двох попередніх етапах виявлено помилки, то ця фаза не виконується.

В даному курсовому проекті генерація коду реалізується як окремий етап. Можливість його виконання є лише за умови, що попередньо успішно виконався етап синтаксичного аналізу. І використовує результат виконання попереднього аналізу, тобто два файли: перший містить згенерований асемблерний код відповідно операторам які були в програмі, другий файл містить таблицю змінних. Інформація з них зчитується в відповідному порядку, основні константні конструкції записуються в файл `asm`.

### 3.5.1. Розробка граф-схеми алгоритму

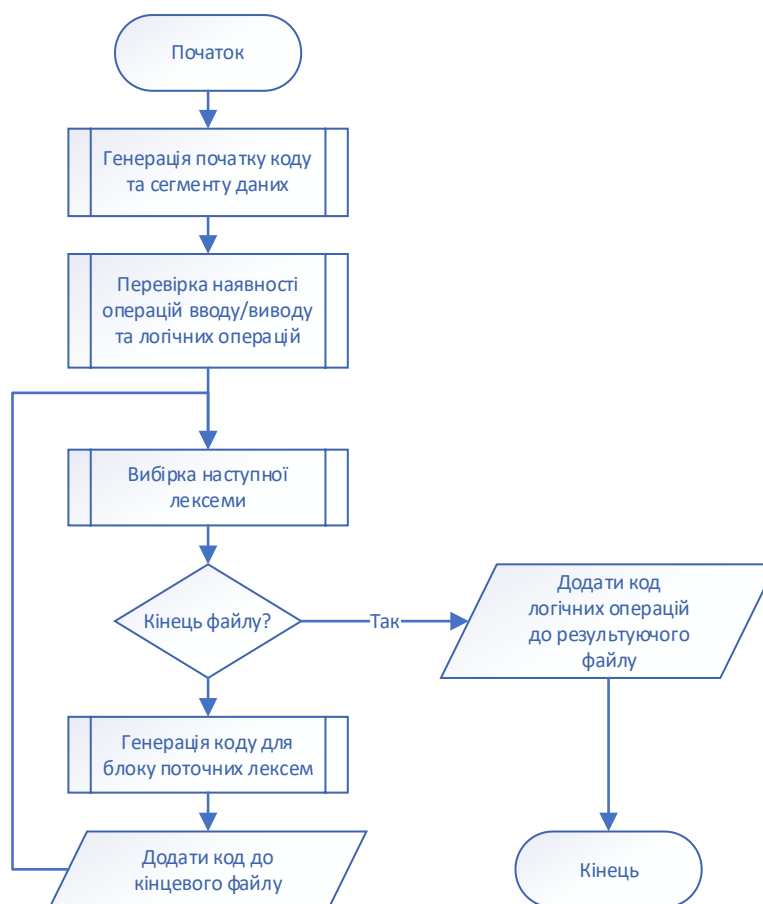


Рис. 3.3 Блок схема генератора коду



### 3.5.2. Опис програми реалізації генератора коду

У компілятора, реалізованого в даному курсовому проєкті, вихідна мова - програма на мові 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.

## 4. Опис програми

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 assignmentRule = 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({ LBracket::Type()}, OnlyOne),
    BackusRuleItem({ identRule->type()}, OnlyOne),
    BackusRuleItem({ RBracket::Type()}, OnlyOne)
});

auto write = BackusRule::MakeRule("WriteRule", {
    BackusRuleItem({ Write::Type()}, OnlyOne),
    BackusRuleItem({ LBracket::Type()}, OnlyOne | PairStart),
    BackusRuleItem({ stringRule->type(), equation->type() }, OnlyOne),
    BackusRuleItem({ RBracket::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 трактується 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 токен це окремий клас що наслідує 3 інтерфейси:

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

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

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

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

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

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

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

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

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

static void PrintGreate(std::ostream& out, const GeneratorDetails::GeneratorArgs&
args)
```

```

{
    out << " ;==Procedure
Greate=====\\n";
    out << "Greate_ PROC\\n";
    out << "\\tpushf\\n";
    out << "\\tpop cx\\n\\n";
    out << "\\tmov " << args.regPrefix << "ax, [esp + " << args.posArg0 << "]"\\n";
    out << "\\tcmp " << args.regPrefix << "ax, [esp + " << args.posArg1 << "]"\\n";
    out << "\\tjle greate_false\\n";
    out << "\\tmov " << args.regPrefix << "ax, 1\\n";
    out << "\\tjmp greate_fin\\n";
    out << "greate_false:\\n";
    out << "\\tmov " << args.regPrefix << "ax, 0\\n";
    out << "greate_fin:\\n";
    out << "\\tpush cx\\n";
    out << "\\tpopf\\n\\n";
    GeneratorUtils::PrintResultToStack(out, args);
    out << "\\tret\\n";
    out << "Greate_ ENDP\\n";
    out <<
";=====\\n";
=====\\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 інструкція користувачеві

Вхідним файлом для даної програми є звичайний текстовий файл з розширенням v05. 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\_v05.exe <ім'я програми>.v05"

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

## 5. Відлагодження та тестування програми

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

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

### 5.1. Виявлення лексичних та синтаксичних помилок

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

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

Приклад виявлення:

#### *Текст програми з помилками*

```
{*Prog1*}  
PROGRAM Prog1;  
BEGIN  
V AR INT_2 Aa,Bbb bb,Xxxxx,Yyyyy;  
OUTPUT("INPUT  Aaaaa: ");  
INPUT(Aaaaa);  
OUTPUT("INPUT  Bbbbbb: ");  
INP UT(Bbbbbb);  
OUTPUT("Aaaaa + Bbbbbb: ");  
OUTPUT(Aaaaa + Bbbbbb);  
OUTPUT("\n_AAAAAAAAAAAAAAAAAA - Bbbbbb: ");  
OUTPUT(Aaaaa - Bbbbbb);  
OUTPUT("\n_AAAAAAAAAAAAAAAAAA * Bbbbbb: ");  
OUTPUT(Aaaaa * Bbbbbb);  
OUTPUT("\n_AAAAAAAAAAAAAAAAAA / Bbbbbb: ");  
OUTPUT(Aaaaa DIV Bbbbbb);
```

```

OUTPUT("\n_AAAAAAAAAAAAAAAAAA % Bbbbb: ");
OUTPUT(Aaaaa MOD Bbbbb);
Xxxxx<-(Aaaaa - Bbbbb) * 10 + (Aaaaa + Bbbbb) DIV 10;
Yyyyy<-Xxxxx + (Xxxxx MOD 10);
OUTPUT("\n_XXXXXXXXXXXXXXXXXX = (Aaaaa - Bbbbb) * 10 + (Aaaaa + Bbbbb) / 10\n");
OUTPUT(Xxxxx);
OUTPUT("\n_YYYYYYYYYYYYYYYYYY = Xxxxx + (Xxxxx % 10)\n");
OUTPUT(Yyyyy);
END

```

### ***Текст файлу з повідомленнями про помилки***

List of errors

```

=====
There are 7 lexical errors.
There are 1 syntax errors.
There are 0 semantic errors.

```

```

Line 4: Lexical error: Unknown token: V
Line 4: Lexical error: Unknown token: AR
Line 4: Lexical error: Unknown token: Aa
Line 4: Lexical error: Unknown token: Bbb
Line 4: Lexical error: Unknown token: bb
Line 4: Syntax error: Expected: Vars before V
Line 8: Lexical error: Unknown token: INP
Line 8: Lexical error: Unknown token: UT

```

## **5.2. Виявлення семантичних помилок**

Суттю виявлення семантичних помилок є перевірка числових констант на відповідність типу INT\_2, тобто знаковому цілому числу з відповідним діапазоном значень і перевірку на коректність використання змінних INT\_2 у цілочисельних і логічних виразах.

## **5.3. Загальна перевірка коректності роботи транслятора**

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

### ***Текст коректної програми***

```

{*Prog1*}
PROGRAM Prog1;
BEGIN
VAR INT_2 Aaaaa,Bbbbb,Xxxxx,Yyyyy;
OUTPUT("INPUT Aaaaa: ");
INPUT(Aaaaa);
OUTPUT("INPUT Bbbbb: ");
INPUT(Bbbbb);
OUTPUT("Aaaaa + Bbbbb: ");

```

```

OUTPUT(Aaaaa + Bbbbb);
OUTPUT("\n_AAAAAAAAAAAAAAAAAA - Bbbbb: ");
OUTPUT(Aaaaa - Bbbbb);
OUTPUT("\n_AAAAAAAAAAAAAAAAAA * Bbbbb: ");
OUTPUT(Aaaaa * Bbbbb);
OUTPUT("\n_AAAAAAAAAAAAAAAAAA / Bbbbb: ");
OUTPUT(Aaaaa DIV Bbbbb);
OUTPUT("\n_AAAAAAAAAAAAAAAAAA % Bbbbb: ");
OUTPUT(Aaaaa MOD Bbbbb);
Xxxx<-(Aaaaa - Bbbbb) * 10 + (Aaaaa + Bbbbb) DIV 10;
Yyyy<-Xxxx + (Xxxx MOD 10);
OUTPUT("\n_XXXXXXXXXXXXXXXXXX = (Aaaaa - Bbbbb) * 10 + (Aaaaa + Bbbbb) / 10\n");
OUTPUT(Xxxx);
OUTPUT("\n_YYYYYYYYYYYYYYYYYY = Xxxx + (Xxxx % 10)\n");
OUTPUT(Yyyy);
END

```

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

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

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

```

INPUT Aaaaa: 5
INPUT Bbbbb: 9
Aaaaa + Bbbbb: 14
_AAAAAAAAAAAAAAAAAA - Bbbbb: -4
_AAAAAAAAAAAAAAAAAA * Bbbbb: 45
_AAAAAAAAAAAAAAAAAA / Bbbbb: 0
_AAAAAAAAAAAAAAAAAA % Bbbbb: 5
_XXXXXXXXXXXXXXXXXX = (Aaaaa - Bbbbb) * 10 + (Aaaaa + Bbbbb) / 10
-39
_YYYYYYYYYYYYYYYYYY = Xxxx + (Xxxx % 10)
-48

```

Рис. 5.1 Результат виконання коректної програми

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

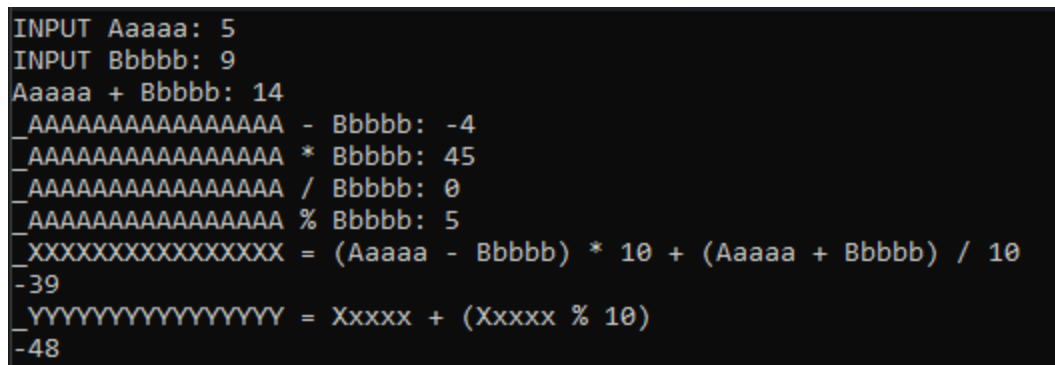


## 5.4. Тестова програма №1

### *Текст програми*

```
{*Prog1*}  
PROGRAM Prog1;  
BEGIN  
VAR INT_2 Aaaaa,Bbbbb,Xxxxx,Yyyyy;  
OUTPUT("INPUT Aaaaa: ");  
INPUT(Aaaaa);  
OUTPUT("INPUT Bbbbb: ");  
INPUT(Bbbbb);  
OUTPUT("Aaaaa + Bbbbb: ");  
OUTPUT(Aaaaa + Bbbbb);  
OUTPUT("\n_AAAAAAAAAAAAAAAAAA - Bbbbb: ");  
OUTPUT(Aaaaa - Bbbbb);  
OUTPUT("\n_AAAAAAAAAAAAAAAAAA * Bbbbb: ");  
OUTPUT(Aaaaa * Bbbbb);  
OUTPUT("\n_AAAAAAAAAAAAAAAAAA / Bbbbb: ");  
OUTPUT(Aaaaa DIV Bbbbb);  
OUTPUT("\n_AAAAAAAAAAAAAAAAAA % Bbbbb: ");  
OUTPUT(Aaaaa MOD Bbbbb);  
Xxxxx<-(Aaaaa - Bbbbb) * 10 + (Aaaaa + Bbbbb) DIV 10;  
Yyyyy<-Xxxxx + (Xxxxx MOD 10);  
OUTPUT("\n_XXXXXXXXXXXXXXXXXX = (Aaaaa - Bbbbb) * 10 + (Aaaaa + Bbbbb) / 10\n");  
OUTPUT(Xxxxx);  
OUTPUT("\n_YYYYYYYYYYYYYYYYYYY = XXXXXX + (XXXXXX % 10)\n");  
OUTPUT(Yyyyy);  
END
```

### *Результат виконання*



```
INPUT Aaaaa: 5  
INPUT Bbbbb: 9  
Aaaaa + Bbbbb: 14  
_AAAAAAAAAAAAAAAAA - Bbbbb: -4  
_AAAAAAAAAAAAAAAAA * Bbbbb: 45  
_AAAAAAAAAAAAAAAAA / Bbbbb: 0  
_AAAAAAAAAAAAAAAAA % Bbbbb: 5  
_XXXXXXXXXXXXXXXXXX = (Aaaaa - Bbbbb) * 10 + (Aaaaa + Bbbbb) / 10  
-39  
_YYYYYYYYYYYYYYYYYYY = XXXXXX + (XXXXXX % 10)  
-48
```

Рис. 5.2 Результат виконання тестової програми №1

## 5.5. Тестова програма №2

### *Текст програми*

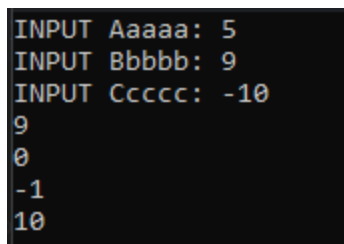
```
{*Prog2*}  
PROGRAM Prog2;  
BEGIN  
  VAR INT_2 Aaaaa,Bbbbb,Ccccc;  
  OUTPUT("INPUT  Aaaaa: ");  
  INPUT(Aaaaa);  
  OUTPUT("INPUT  Bbbbb: ");  
  INPUT(Bbbbb);  
  OUTPUT("INPUT  Ccccc: ");  
  INPUT(Ccccc);  
  IF(Aaaaa GT Bbbbb)  
  BEGIN  
    IF(Aaaaa GT Ccccc)  
    BEGIN  
      GOTO Aibig;  
    END  
    ELSE  
    BEGIN  
      OUTPUT(Ccccc);  
      GOTO Outif;  
    Aibig:  
      OUTPUT(Aaaaa);  
      GOTO Outif;  
    END  
  END  
END  
  IF(Bbbbb LT Ccccc)  
  BEGIN  
    OUTPUT(Ccccc);  
  END  
  ELSE  
  BEGIN  
    OUTPUT(Bbbbb);  
  END  
Outif:  
OUTPUT("\n");  
IF((Aaaaa = Bbbbb) && (Aaaaa = Ccccc) && (Bbbbb = Ccccc))  
BEGIN  
  OUTPUT(1);  
END  
ELSE  
BEGIN  
  OUTPUT(0);  
END  
OUTPUT("\n");
```

```

IF((Aaaaa LT 0) || (Bbbbb LT 0) || (Ccccc LT 0))
BEGIN
    OUTPUT(-1);
END
ELSE
BEGIN
    OUTPUT(0);
END
OUTPUT("\n");
IF(!(Aaaaa LT (Bbbbb + Ccccc)))
BEGIN
    OUTPUT(10);
END
ELSE
BEGIN
    OUTPUT(0);
END
END

```

### *Результат виконання*



```

INPUT Aaaaa: 5
INPUT Bbbbb: 9
INPUT Ccccc: -10
9
0
-1
10

```

Рис. 5.3 Результат виконання тестової програми №2

## **5.6. Тестова програма №3**

### *Текст програми*

```

{*Prog3*}
PROGRAM Prog3;
BEGIN
VAR INT_2 Aaaaa,Aaaa2,Bbbbb,Xxxxx,Cccc1,Cccc2;
OUTPUT("INPUT Aaaaa: ");
INPUT(Aaaaa);
OUTPUT("INPUT Bbbbb: ");
INPUT(Bbbbb);
OUTPUT("FOR TO do");
FOR Aaaa2<-Aaaaa TO Bbbbb DO
BEGIN
    OUTPUT("\n");
    OUTPUT(Aaaa2 * Aaaa2);
END
OUTPUT("\nFor DOWNT0 do");
FOR Aaaa2<-Bbbbb DOWNT0 Aaaaa DO

```

```

BEGIN
    OUTPUT("\n");
    OUTPUT(Aaaa2 * Aaaa2);
END

OUTPUT("\nWhile Aaaaa * Bbbbb: ");
Xxxxx<-0;
Cccc1<-0;
WHILE(Cccc1 LT Aaaaa)
BEGIN
    Cccc2<-0;
    WHILE (Cccc2 LT Bbbbb)
    BEGIN
        Xxxxx<-Xxxxx + 1;
        Cccc2<-Cccc2 + 1;
    END
    Cccc1<-Cccc1 + 1;
END
OUTPUT(Xxxxx);

OUTPUT("\nRepeat UNTIL Aaaaa * Bbbbb: ");
Xxxxx<-0;
Cccc1<-1;
REPEAT
    Cccc2<-1;
    REPEAT
        Xxxxx<-Xxxxx+1;
        Cccc2<-Cccc2+1;
    UNTIL(!(Cccc2 GT Bbbbb))
    Cccc1<-Cccc1+1;
UNTIL(!(Cccc1 GT Aaaaa))
OUTPUT(Xxxxx);

END

```

### *Результат виконання*

```
INPUT Aaaaa: 5
INPUT Bbbbb: 9
FOR TO do
25
36
49
64
81
For DOWNT0 do
81
64
49
36
25
While Aaaaa * Bbbbb: 45
Repeat UNTIL Aaaaa * Bbbbb: 45
```

Рис. 5.4 Результат виконання тестової програми №3

## Висновки

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування v05, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створено компілятор мови програмування v05, 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 v05. 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 v05 були успішно виявлені і відповідно оброблені.

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

# Список використаної літератури

1. 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 (Код на мові Асемблер)

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

include lib masm32\lib\kernel32.lib

include lib masm32\lib\masm32.lib

include lib masm32\lib\user32.lib

include lib masm32\lib\msvcrt.lib

.DATA

;===User

Data=====

Aaaaa\_ dd 0  
Bbbbb\_ dd 0  
Xxxx\_ dd 0  
Yyyy\_ 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 Aaaaa: ", 0  
String\_1 db "INPUT Bbbbb: ", 0  
String\_2 db "Aaaaa + Bbbbb: ", 0  
String\_3 db 13, 10, "\_AAAAAAAAAAAAAAAAAAAA - Bbbbb: ", 0  
String\_4 db 13, 10, "\_AAAAAAAAAAAAAAAAAAAA \* Bbbbb: ", 0  
String\_5 db 13, 10, "\_AAAAAAAAAAAAAAAAAAAA / Bbbbb: ", 0  
String\_6 db 13, 10, "\_AAAAAAAAAAAAAAAAAAAA % Bbbbb: ", 0  
String\_7 db 13, 10, "\_XXXXXXXXXXXXXXXXXXXX = (Aaaaa - Bbbbb) \* 10 +  
(Aaaaa + Bbbbb) / 10", 13, 10, 0  
String\_8 db 13, 10, "\_YYYYYYYYYYYYYYYYYY = Xxxx + (Xxxx % 10)", 13,  
10, 0

;===Addition

Data=====

hConsoleInput dd ?  
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 Aaaaa\_, eax

invoke WriteConsoleA, hConsoleOutput, ADDR String\_1, SIZEOF String\_1 - 1, 0, 0

call Input\_

mov Bbbbb\_, eax

invoke WriteConsoleA, hConsoleOutput, ADDR String\_2, SIZEOF String\_2 - 1, 0, 0

push Aaaaa\_

push Bbbbb\_

call Add\_

call Output\_

invoke WriteConsoleA, hConsoleOutput, ADDR String\_3, SIZEOF String\_3 - 1, 0, 0

push Aaaaa\_

push Bbbbb\_

call Sub\_

call Output\_

invoke WriteConsoleA, hConsoleOutput, ADDR String\_4, SIZEOF String\_4 - 1, 0, 0

push Aaaaa\_

push Bbbbb\_

call Mul\_

call Output\_

invoke WriteConsoleA, hConsoleOutput, ADDR String\_5, SIZEOF String\_5 - 1, 0, 0

push Aaaaa\_

push Bbbbb\_

call Div\_

call Output\_

invoke WriteConsoleA, hConsoleOutput, ADDR String\_6, SIZEOF String\_6 - 1, 0, 0

push Aaaaa\_

push Bbbbb\_

call Mod\_

call Output\_

push Aaaaa\_

push Bbbbb\_

call Sub\_

push dword ptr 10

call Mul\_

push Aaaaa\_

push Bbbbb\_

call Add\_

```

push dword ptr 10
call Div_
call Add_
pop Xxxx_
push Xxxx_
push Xxxx_
push dword ptr 10
call Mod_
call Add_
pop Yyyy_
invoke WriteConsoleA, hConsoleOutput, ADDR String_7, SIZEOF String_7 - 1, 0, 0
push Xxxx_
call Output_
invoke WriteConsoleA, hConsoleOutput, ADDR String_8, SIZEOF String_8 - 1, 0, 0
push Yyyy_
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
include lib masm32\lib\kernel32.lib
include lib masm32\lib\masm32.lib
include lib masm32\lib\user32.lib
include lib masm32\lib\msvcrt.lib

.DATA
;===User
Data=====
=====

    Aaaaa_      dd      0
    Bbbbb_      dd      0
    Ccccc_dd    0

    String_0     db      "INPUT Aaaaa: ", 0
    String_1     db      "INPUT Bbbbb: ", 0
    String_2     db      "INPUT Ccccc: ", 0
    String_3     db      13, 10, 0
    String_4     db      13, 10, 0
    String_5     db      13, 10, 0

```

;===Addition

Data=====

=====

```
hConsoleInput dd      ?
hConsoleOutput dd      ?
endBuf        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 Aaaaa\_, eax

invoke WriteConsoleA, hConsoleOutput, ADDR String\_1, SIZEOF String\_1 - 1, 0, 0

call Input\_

mov Bbbbb\_, eax

invoke WriteConsoleA, hConsoleOutput, ADDR String\_2, SIZEOF String\_2 - 1, 0, 0

call Input\_

mov Ccccc\_, eax

push Aaaaa\_

push Bbbbb\_

call Create\_

pop eax

cmp eax, 0

je endIf2

push Aaaaa\_

push Ccccc\_

call Create\_

pop eax

cmp eax, 0

je elseLabel1

jmp Aibig\_

jmp endIf1

elseLabel1:

push Ccccc\_

call Output\_

jmp Outif\_

Aibig\_:

push Aaaaa\_

call Output\_

```

        jmp Outif_
endIf1:
endIf2:
    push Bbbbb_
    push Ccccc_
    call Less_
    pop eax
    cmp eax, 0
    je elseLabel3
    push Ccccc_
    call Output_
    jmp endIf3
elseLabel3:
    push Bbbbb_
    call Output_
endIf3:
Outif_:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3 - 1, 0, 0
    push Aaaaa_
    push Bbbbb_
    call Equal_
    push Aaaaa_
    push Ccccc_
    call Equal_
    call And_
    push Bbbbb_
    push Ccccc_
    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 Aaaaa_
    push dword ptr 0
    call Less_
    push Bbbbb_
    push dword ptr 0
    call Less_
    call Or_
    push Ccccc_
    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 Aaaaa_
    push Bbbbb_
    push Ccccc_
    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
include lib masm32\lib\kernel32.lib
include lib masm32\lib\masm32.lib
include lib masm32\lib\user32.lib
include lib masm32\lib\msvcrt.lib

.DATA
;===User
Data=====
=====
Aaaa2_      dd      0
Aaaaa_      dd      0
Bbbbb_      dd      0
Cccc1_      dd      0
Cccc2_      dd      0
Xxxxx_      dd      0

String_0     db      "INPUT Aaaaa: ", 0
String_1     db      "INPUT Bbbbb: ", 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 Aaaaa * Bbbbb: ", 0
String_7     db      13, 10, "Repeat UNTIL Aaaaa * Bbbbb: ", 0

;===Addition
Data=====
=====
hConsoleInput dd      ?
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 Aaaaa_, eax
    invoke WriteConsoleA, hConsoleOutput, ADDR String_1, SIZEOF String_1 - 1, 0, 0
    call Input_
    mov Bbbbb_, eax
    invoke WriteConsoleA, hConsoleOutput, ADDR String_2, SIZEOF String_2 - 1, 0, 0
    push Aaaaa_
    pop Aaaa2_
forPasStart1:
    push Bbbbb_
    push Aaaa2_
    call Less_
    call Not_
    pop eax
    cmp eax, 0
    je forPasEnd1
    invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3 - 1, 0, 0
    push Aaaa2_
    push Aaaa2_
    call Mul_
    call Output_
    push Aaaa2_
    push dword ptr 1
    call Add_
    pop Aaaa2_
    jmp forPasStart1
forPasEnd1:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_4, SIZEOF String_4 - 1, 0, 0
    push Bbbbb_
    pop Aaaa2_
forPasStart2:
    push Aaaaa_
    push Aaaa2_
    call Create_
    call Not_
    pop eax
    cmp eax, 0
    je forPasEnd2
    invoke WriteConsoleA, hConsoleOutput, ADDR String_5, SIZEOF String_5 - 1, 0, 0
    push Aaaa2_
    push Aaaa2_
    call Mul_
    call Output_

```

```

    push Aaaa2_
    push dword ptr 1
    call Sub_
    pop Aaaa2_
    jmp forPasStart2
forPasEnd2:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_6, SIZEOF String_6 - 1, 0, 0
    push dword ptr 0
    pop Xxxx_
    push dword ptr 0
    pop Cccc1_
whileStart2:
    push Cccc1_
    push Aaaaa_
    call Less_
    pop eax
    cmp eax, 0
    je whileEnd2
    push dword ptr 0
    pop Cccc2_
whileStart1:
    push Cccc2_
    push Bbbbb_
    call Less_
    pop eax
    cmp eax, 0
    je whileEnd1
    push Xxxx_
    push dword ptr 1
    call Add_
    pop Xxxx_
    push Cccc2_
    push dword ptr 1
    call Add_
    pop Cccc2_
    jmp whileStart1
whileEnd1:
    push Cccc1_
    push dword ptr 1
    call Add_
    pop Cccc1_
    jmp whileStart2
whileEnd2:
    push Xxxx_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_7, SIZEOF String_7 - 1, 0, 0
    push dword ptr 0
    pop Xxxx_
    push dword ptr 1
    pop Cccc1_

```

```

repeatStart2:
    push dword ptr 1
    pop Cccc2_
repeatStart1:
    push Xxxxx_
    push dword ptr 1
    call Add_
    pop Xxxxx_
    push Cccc2_
    push dword ptr 1
    call Add_
    pop Cccc2_
    push Cccc2_
    push Bbbbbb_
    call Greate_
    call Not_
    pop eax
    cmp eax, 0
    je repeatEnd1
    jmp repeatStart1
repeatEnd1:
    push Cccc1_
    push dword ptr 1
    call Add_
    pop Cccc1_
    push Cccc1_
    push Aaaaa_
    call Greate_
    call Not_
    pop eax
    cmp eax, 0
    je repeatEnd2
    jmp repeatStart2
repeatEnd2:
    push Xxxxx_
    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

```