

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



Пояснювальна записка

до курсового проєкту "СИСТЕМНЕ ПРОГРАМУВАННЯ"

на тему: "РОЗРОБКА СИСТЕМНИХ ПРОГРАМНИХ МОДУЛІВ ТА КОМПОНЕНТ
СИСТЕМ ПРОГРАМУВАННЯ"

Індивідуальне завдання

"РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ"

Виконав:

ст. гр. КІ-307

Маринович Марко

Перевірив:

Козак Н. Б.

ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ

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.

Деталізація завдання на проектування:

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

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

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

Деталізований опис власної мови програмування:

Опис вхідної мови програмування:

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

- Оператори: IF ELSE (C)

GOTO (C)

FOR-TO-DO (Паскаль)

FOR-DOWNTO-DO (Паскаль)

WHILE (Бейсік)

REPEAT-UNTIL (Паскаль)

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

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

АНОТАЦІЯ

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

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

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

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

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

ЗМІСТ

Анотація	Помилка! Закладку не визначено.
Завдання до курсового проекту.....	Помилка! Закладку не визначено.
Вступ.....	Помилка! Закладку не визначено.
1. Огляд методів та способів проектування трансляторів.....	Помилка! Закладку не визначено.
2. Формальний опис вхідної мови програмування.....	Помилка! Закладку не визначено.
2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура	Помилка! Закладку не визначено.
2.2. Опис термінальних символів та ключових слів	Помилка! Закладку не визначено.
3. Розробка транслятора вхідної мови програмування.....	Помилка! Закладку не визначено.
3.1. Вибір технології програмування	Помилка! Закладку не визначено.
3.2. Проектування таблиць транслятора	Помилка! Закладку не визначено.
3.3. Розробка лексичного аналізатора	Помилка! Закладку не визначено.
3.3.1. Розробка блок-схеми алгоритму	Помилка! Закладку не визначено.
3.3.2. Опис програми реалізації лексичного аналізатора	Помилка! Закладку не визначено.
3.4. Розробка синтаксичного та семантичного аналізатора.....	Помилка! Закладку не визначено.
3.4.1. Опис програми реалізації синтаксичного та семантичного аналізатора	Помилка! Закладку не визначено.
3.4.2. Розробка граф-схеми алгоритму	Помилка! Закладку не визначено.
3.5. Розробка генератора коду	Помилка! Закладку не визначено.
3.5.1. Розробка граф-схеми алгоритму	Помилка! Закладку не визначено.
3.5.2. Опис програми реалізації генератора коду	Помилка! Закладку не визначено.
4. Опис програми	Помилка! Закладку не визначено.
4.1. Опис інтерфейсу та інструкція користувачеві	Помилка! Закладку не визначено.
5. Відлагодження та тестування програми	Помилка! Закладку не визначено.
5.1. Виявлення лексичних та синтаксичних помилок	Помилка! Закладку не визначено.
5.2. Виявлення семантичних помилок	Помилка! Закладку не визначено.
5.3. Загальна перевірка коректності роботи транслятора	Помилка! Закладку не визначено.
5.4. Тестова програма №1	36
5.5. Тестова програма №2	38
5.6. Тестова програма №3	41
Висновки	Помилка! Закладку не визначено.
Список використаної літератури.....	Помилка! Закладку не визначено.
Додатки.....	Помилка! Закладку не визначено.

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 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 мові програмування. У сучасній практиці найчастіше використовуються методи аналізу, такі як LL (1) або LR (1) та їхні варіанти (рекурсивний спуск для LL (1) або LR (1), LR (0), SLR (1), LALR (1) та інші для LR (1)). Рекурсивний спуск застосовується частіше при ручному програмуванні синтаксичного аналізатора, тоді як LR (1) використовується при автоматичній генерації синтаксичних аналізаторів.

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

На етапі контекстного аналізу виявляються взаємозалежності між різними частинами програми, які не можуть бути адекватно описані за допомогою контекстно-

вільної граматики. Ці взаємозалежності, зокрема, включають аналіз типів об'єктів, областей видимості, відповідності параметрів, міток та інших аспектів "опис-використання". У ході контекстного аналізу таблиці об'єктів доповнюються інформацією, пов'язаною з описами (властивостями) об'єктів.

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

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

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

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

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

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

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

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

2. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ

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

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

$\langle \text{topRule} \rangle ::= \text{STARTPROGRAM VARIABLE } \langle \text{varsBlok} \rangle ; \langle \text{codeBlok} \rangle$

$\langle \text{varsBlok} \rangle ::= \text{INT_4 } \langle \text{identifier} \rangle [\{ \langle \text{commaAndIdentifier} \rangle \}] ;$

$\langle \text{identifier} \rangle ::= _ \langle \text{low_letter} \rangle \{ \langle \text{low_letter} \rangle | \langle \text{number} \rangle \} \{ 3 \}$

$\langle \text{commaAndIdentifier} \rangle ::= , \langle \text{identifier} \rangle$

$\langle \text{codeBlok} \rangle ::= \text{STARTBLOK } \langle \text{write} \rangle | \langle \text{read} \rangle | \langle \text{assignment} \rangle | \langle \text{ifStatement} \rangle$

$| \langle \text{goto_statement} \rangle | \langle \text{labelRule} \rangle | \langle \text{forToOrDownToDoRule} \rangle | \langle \text{while} \rangle | \langle \text{repeatUntil} \rangle \text{ENDBLOK}$

$\langle \text{read} \rangle ::= \text{READ } (\langle \text{identifier} \rangle) ;$

$\langle \text{write} \rangle ::= \text{WRITE } (\langle \text{equation} \rangle | \langle \text{stringRule} \rangle) ;$

$\langle \text{assignment} \rangle ::= \langle \text{identifier} \rangle ::= \langle \text{equation} \rangle ;$

$\langle \text{ifStatement} \rangle ::= \text{IF } (\langle \text{equation} \rangle) \langle \text{codeBlok} \rangle \langle \text{elseStatement} \rangle$

$\langle \text{elseStatement} \rangle ::= \text{ELSE } \langle \text{codeBlok} \rangle$

$\langle \text{goto_statement} \rangle ::= \text{GOTO } \langle \text{ident} \rangle ;$

$\langle \text{labelRule} \rangle ::= \langle \text{identifier} \rangle :$

$\langle \text{forToOrDownToDoRule} \rangle ::= \text{FOR } \langle \text{assignment} \rangle \text{ TO } | \text{DOWNTO } \langle \text{equation} \rangle \text{ DO } \langle \text{codeBlok} \rangle$

$\langle \text{while} \rangle ::= \text{WHILE } (\langle \text{equation} \rangle) \langle \text{codeBlok} \rangle$

$\langle \text{repeatUntil} \rangle ::= \text{REPEAT } \langle \text{codeBlok} \rangle \text{ UNTIL } (\langle \text{equation} \rangle)$

< forCRule > <== for (<assignment> ; <equation> ; <assignment>) <codeBlok>

<equation> <== <signedNumber> | <identifier> | <notRule> [{
<operationAndIdentOrNumber> | <equation> }]

< notRule > <== <notOperation> <signedNumber> | <identifier> | <equation>

<operationAndIdentOrNumber> <== <mult> | <arithmetic> | <logic> | <compare>
<signedNumber> | <identifier> | <equation>

<arithmetic> <== ADD | SUB

<mult> <== MUL | DIV | MOD

<logic> <== & | |

<notOperation> <== !

<compare> <== EQ | NE | LT | GT

<stringRule> <== “ <string> “

<comment> <== <LComment> <string> <RComment>

<LComment> <== #*

<RComment> <== *#

<string> <== { <low_letter> | <up_letter> | <number> }

<signedNumber> <== [<sign>] <digit>[{digit}]

<sign> <== + | -

<low_letter> <== a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<up_letter> <== A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<digit> <== 0|1|2|3|4|5|6|7|8|9

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

Термінальний символ або ключове слово	Значення
STARTPROGRAM	Початок програми
STARTBLOK	Початок тексту програми
VARIABLE	Початок блоку опису змінних
ENDBLOK	Кінець розділу операторів
READ	Оператор вводу змінних
WRITE	Оператор виводу (змінних або рядкових констант)
<==	Оператор присвоєння
IF	Оператор умови
ELSE	Оператор умови
GOTO	Оператор переходу
LABEL	Мітка переходу
FOR	Оператор циклу
TO	Інкремент циклу
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
LT	Оператор перевірки чи менше
GT	Оператор перевірки чи більше

!	Оператор логічного заперечення
&	Оператор кон'юнкції
	Оператор диз'юнкції
INT_4	32-ох розрядні знакові цілі
#*...*#	Коментар
,	Розділювач
;	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 символів та ключових слів

Токен	Значення
Program	STARTPROGRAM
Start	STARTBLOK
Vars	VARIABLE
End	ENDBLOK
VarType	INT_4
Read	READ
Write	WRITE
Assignment	<==
If	IF
Else	ELSE
Goto	GOTO
Colon	:
Label	
For	FOR
To	TO
DownTo	DOWNTTO
Do	DO
While	WHILE
Repeat	REPEAT
Until	UNTIL

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

3.3. Розробка лексичного аналізатора.

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.

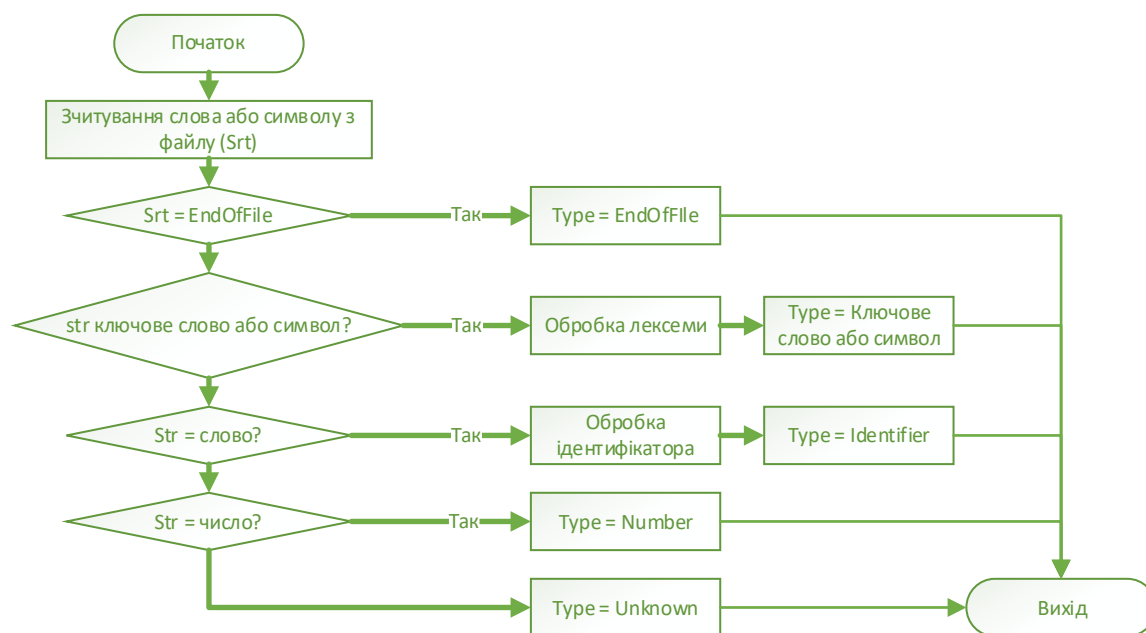


Рис. 3.1 Блок-схема роботи лексичного аналізатора

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 кожен 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

лексми не як до послідовності символів, а як до унікального типу лексми, що значно спрощує роботу синтаксичного аналізатора: легко перевіряти належність лексми 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 лексми – 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 лексми 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 проєкті синтаксичний аналіз можна виконувати лише після виконання лексичного аналізу, він являється окремим етапом трансляції.

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

3.4.1. Розробка дерева граматичного розбору.

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 алгоритму роботи синтаксичного і семантичного аналізатора.

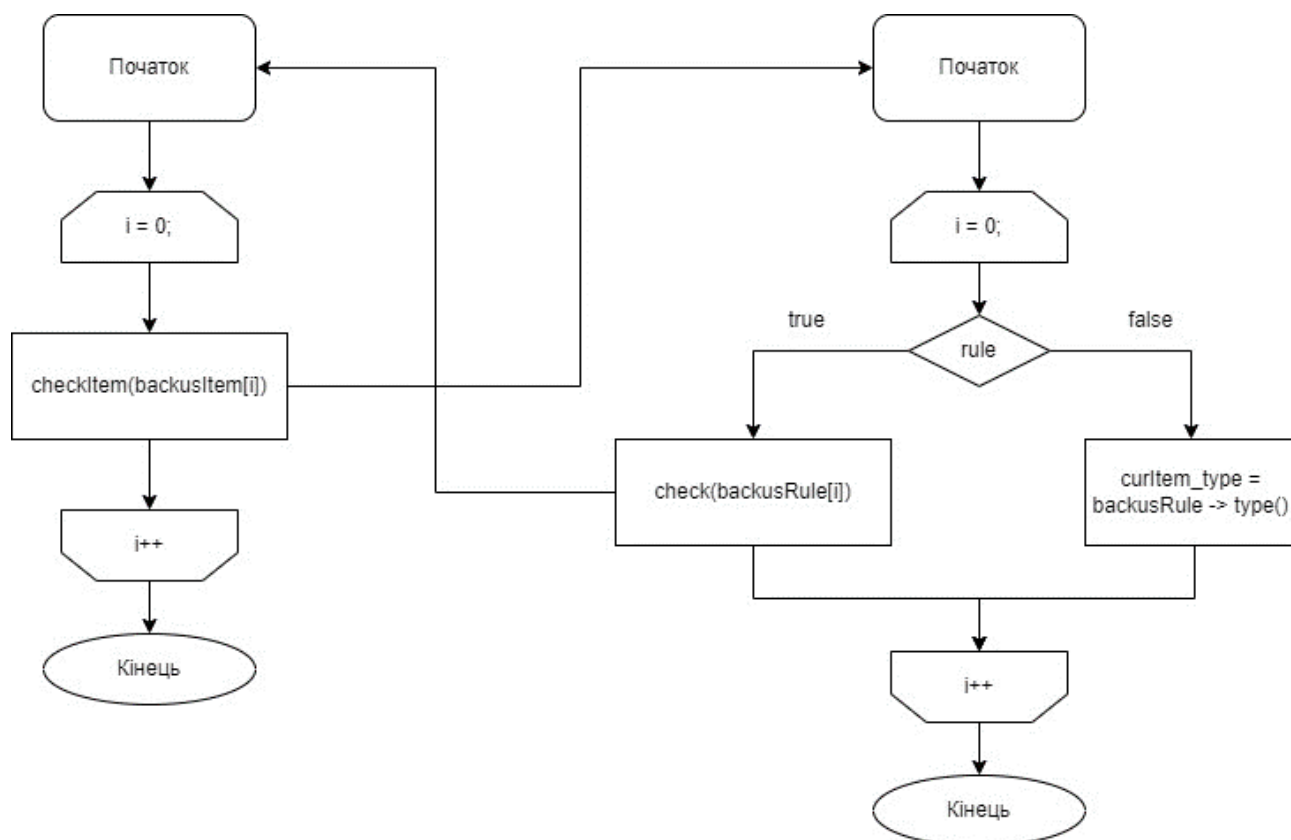


Рис. 3.2 Граф-схема роботи синтаксичного аналізатора

3.4.3. Опис програми реалізації синтаксичного та семантичного 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 заводитьися 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.

3.5.1. Розробка алгоритму роботи генератора коду.

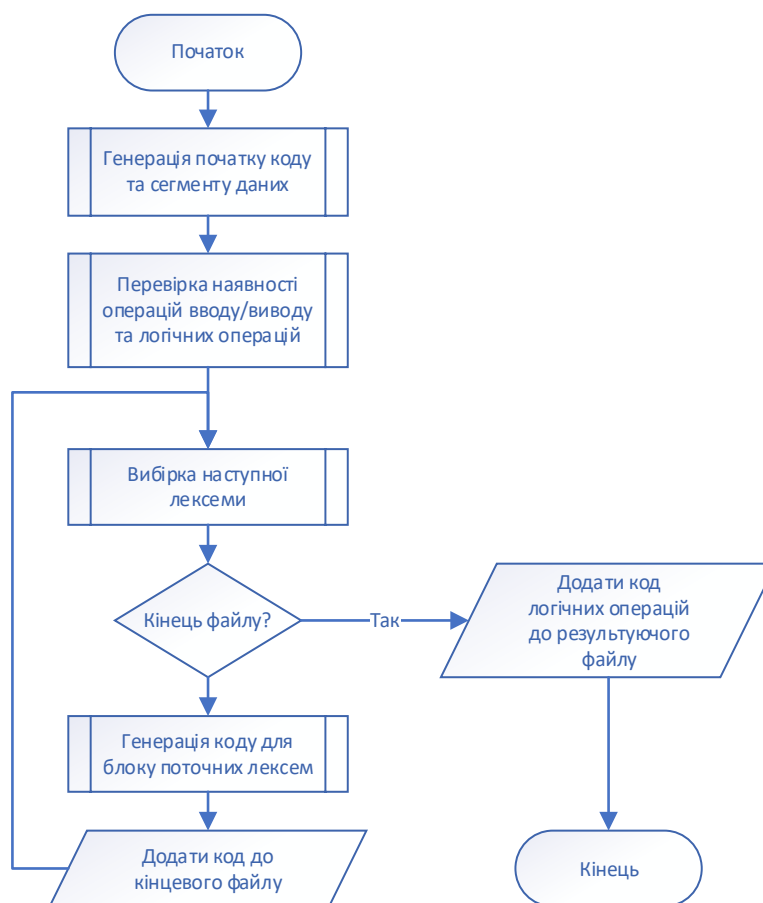


Рис. 3.3 Блок схема генератора коду

3.5.2. Опис програми реалізації генератора коду.

У компілятора, реалізованого в даному курсовому проекті, вихідна мова - програма на мові 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. 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 tokenів 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 << "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 << "Greate_ =====\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 m20. У цьому файлі необхідно набрати бажану для трансляції програму та зберегти її. Синтаксис повинен відповідати вхідній мові.

Створений транслятор є консольною програмою, що запускається з командної стрічки з параметром: "CWork_m20.exe <ім'я програми>.m20"

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

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

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

Для отримання виконавчого файлу необхідно скористатись програмою Masm32.exe

4.2. Виявлення лексичних і синтаксичних помилок.

Виявлення лексичних помилок відбувається на стадії лексичного аналізу. Під час розбиття вхідної програми на окремі лексеми відбувається перевірка чи відповідає

вхідна лексема граматиці. Якщо ця лексема є в граматиці то вона ідентифікується і в таблиці лексем визначається. У випадку неспівпадіння лексемі присвоюється тип "невпізнаної лексеми". Повідомлення про такі помилки можна побачити лише після виконання процедури перевірки таблиці лексем, яка знаходиться в файлі.

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

Приклад виявлення:

Текст програми з помилками

```
##Prog1*#

STARTPROGRAM

VARIABLE INT_4 _a aaa,_bbbb,_xxxx,_yyyy;
```

```
STARTBLOK

WRITdE("Input A: ");

READ(_aaaa);

WRITE("Input B: ");

READ(_bbbb);

WRITE("A + B: ");

WRITE(_aaaa ADD _bbbb);

WRITE("\nA - B: ");

WRITE(_aaaa SUB _bbbb);

WRITE("\nA * B: ");

WRITE(_aaaa MUL _bbbb);

WRITE("\nA / B: ");

WRITE(_aaaa DIV _bbbb);

WRITE("\nA % B: ");

WRITE(_aaaa MOD _bbbb);
```



```

_xxxx<-( _aaaa SUB _bbbb) MUL 10 ADD ( _aaaa ADD _bbbb) DIV 10;

_yyyy<-_xxxx ADD (_xxxx MOD 10);

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

WRITE(_xxxx);

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

WRITE(_yyyy);

ENDBLOK

```

Текст файлу з повідомленнями про помилки

List of errors

=====

There are 5 lexical errors.

There are 1 syntax errors.

There are 0 semantic errors.

Line 3: Lexical error: Unknown token: VARI

Line 3: Lexical error: Unknown token: ABLE

Line 3: Lexical error: Unknown token: _a

Line 3: Lexical error: Unknown token: aaa

Line 3: Syntax error: Expected: Vars before VARI

Line 5: Lexical error: Unknown token: WRITdE

4.3. Перевірка роботи транслятора за допомогою тестових задач.

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

Текст коректної програми

```

#*Prog1*#

STARTPROGRAM

VARIABLE INT_4 _aaaa,_bbbb,_xxxx,_yyyy;

STARTBLOK

WRITE("Input A: ");

READ(_aaaa);

WRITE("Input B: ");

READ(_bbbb);

WRITE("A + B: ");

WRITE(_aaaa ADD _bbbb);

WRITE("\nA - B: ");

WRITE(_aaaa SUB _bbbb);

WRITE("\nA * B: ");

WRITE(_aaaa MUL _bbbb);

WRITE("\nA / B: ");

WRITE(_aaaa DIV _bbbb);

WRITE("\nA % B: ");

WRITE(_aaaa MOD _bbbb);

_xxxx<-( _aaaa SUB _bbbb) MUL 10 ADD ( _aaaa ADD _bbbb) DIV 10;

_yyyy<-_xxxx ADD (_xxxx MOD 10);

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

WRITE(_xxxx);

```

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

```
WRITE(_yyyy);
```

```
ENDBLOK
```

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 отримаємо наступний результат роботи програми:

```
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. 4.1 Результат виконання коректної програми

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

Тестова програма №1

Текст програми

```

#*Prog1*#

STARTPROGRAM

VARIABLE INT_4 _aaaa,_bbbb,_xxxx,_yyyy;

STARTBLOK

WRITE("Input A: ");

READ(_aaaa);

WRITE("Input B: ");

READ(_bbbb);

WRITE("A + B: ");

WRITE(_aaaa ADD _bbbb);

WRITE("\nA - B: ");

WRITE(_aaaa SUB _bbbb);

WRITE("\nA * B: ");

WRITE(_aaaa MUL _bbbb);

WRITE("\nA / B: ");

WRITE(_aaaa DIV _bbbb);

WRITE("\nA % B: ");

WRITE(_aaaa MOD _bbbb);

_xxxx<-( _aaaa SUB _bbbb) MUL 10 ADD ( _aaaa ADD _bbbb) DIV 10;

_yyyy<-_xxxx ADD ( _xxxx MOD 10);

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

WRITE(_xxxx);

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

```

WRITE(_yyyy);

ENDBLOK

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. 4.2 Результат виконання тестової програми №1

Тестова програма №2

Текст програми

```
##*Prog2*#

STARTPROGRAM

VARIABLE INT_4 _aaaa,_bbbb,_cccc;

STARTBLOK

WRITE("Input A: ");

READ(_aaaa);

WRITE("Input B: ");

READ(_bbbb);

WRITE("Input C: ");

READ(_cccc);

IF(_aaaa GT _bbbb)

STARTBLOK

    IF(_aaaa GT _cccc)

STARTBLOK

    GOTO _temp;

ENDBLOK

ELSE

STARTBLOK

    WRITE(_cccc);

    GOTO _outi;

    _temp:

    WRITE(_aaaa);
```

```

        GOTO _outi;

ENDBLOK

ENDBLOK

    IF(_bbbb LT _cccc)

        STARTBLOK

            WRITE(_cccc);

        ENDBLOK

    ELSE

        STARTBLOK

            WRITE(_bbbb);

        ENDBLOK

_outi:

WRITE("\n");

IF((_aaaa EQ _bbbb) & (_aaaa EQ _cccc) & (_bbbb EQ _cccc))

    STARTBLOK

        WRITE(1);

    ENDBLOK

ELSE

    STARTBLOK

        WRITE(0);

    ENDBLOK

WRITE("\n");

IF((_aaaa LT 0) | (_bbbb LT 0) | (_cccc LT 0))

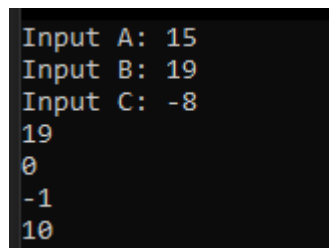
    STARTBLOK

        WRITE(- 1);

```

```
ENDBLOK  
  
ELSE  
  
STARTBLOK  
  
    WRITE(0);  
  
ENDBLOK  
  
WRITE("\n");  
  
IF(!_aaaa LT (_bbbb ADD _cccc))  
  
STARTBLOK  
  
    WRITE(10);  
  
ENDBLOK  
  
ELSE  
  
STARTBLOK  
  
    WRITE(0);  
  
ENDBLOK  
  
ENDBLOK
```

Результат виконання

A screenshot of a terminal window with a black background and white text. The text shows the input values for variables A, B, and C, followed by the output of the program's logic. The output consists of five lines: '19', '0', '-1', and '10'.

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

Рис. 4.3 Результат виконання тестової програми №2

Тестова програма №3

Текст програми

```

#*Prog3*#

STARTPROGRAM

VARIABLE INT_4 _aaaa,_aaa2,_bbbb,_xxxx,_ccc1,_ccc2;

STARTBLOK

WRITE("Input A: ");

READ(_aaaa);

WRITE("Input B: ");

READ(_bbbb);

WRITE("FOR TO DO");

FOR _aaa2<=_aaaa TO _bbbb DO

STARTBLOK

    WRITE("\n");

    WRITE(_aaa2 MUL _aaa2);

ENDBLOK

WRITE("\nFOR DOWNT TO DO");

FOR _aaa2<=_bbbb DOWNT TO _aaaa DO

STARTBLOK

    WRITE("\n");

    WRITE(_aaa2 MUL _aaa2);

ENDBLOK

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

```

```

_xxxx<-0;

_ccc1<-0;

WHILE(_ccc1 LT _aaaa)

STARTBLOK

    _ccc2<-0;

    WHILE (_ccc2 LT _bbbb)

    STARTBLOK

        _xxxx<-_xxxx ADD 1;

        _ccc2<-_ccc2 ADD 1;

    ENDBLOK

    _ccc1<-_ccc1 ADD 1;

ENDBLOK

WRITE(_xxxx);

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

_xxxx<-0;

_ccc1<-1;

REPEAT

    _ccc2<-1;

    REPEAT

        _xxxx<-_xxxx ADD 1;

        _ccc2<-_ccc2 ADD 1;

    UNTIL(!(_ccc2 GT _bbbb))

    _ccc1<-_ccc1 ADD 1;

UNTIL(!(_ccc1 GT _aaaa))

WRITE(_xxxx);

```

ENDBLOK

Результат виконання

```
Input A: 5
Input B: 9
FOR TO DO
25
36
49
64
81
FOR DOWNT0 DO
81
64
49
36
25
WHILE A MUL B: 45
REPEAT UNTIL A MUL B: 45
```

Рис. 4.4 Результат виконання тестової програми №3

ВИСНОВКИ

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування m20, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.
 2. Створено компілятор мови програмування m20, а саме:
 - 2.1. Розроблено прямий лексичний аналізатор, орієнтований на розпізнавання лексем, що є заявлені в формальному описі мови програмування.
 - 2.2. Розроблено синтаксичний аналізатор на основі низхідного методу. Складено деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура
 - 2.3. Розроблено генератор коду, відповідні процедури якого викликаються після перевірки синтаксичним аналізатором коректності запису чергового оператора, мови програмування m20. Вихідним кодом генератора є програма на мові Assembler(x86).
 3. Проведене тестування компілятора на тестових програмах за наступними пунктами:
 - 3.1. На виявлення лексичних помилок.
 - 3.2. На виявлення синтаксичних помилок.
 - 3.3. Загальна перевірка роботи компілятора.
- Тестування не виявило помилок в роботі компілятор, і всі помилки в тестових програмах на мові m20 були успішно виявлені і відповідно оброблені.

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

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. 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)

Див. файли «додаток_А_ТЛ_Проґ1.txt», «додаток_А_ТЛ_Проґ2.txt»,
«додаток_А_ТЛ_Проґ3.txt»

Додаток Б(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

includelib masm32\lib\kernel32.lib

includelib masm32\lib\masm32.lib

includelib masm32\lib\user32.lib

includelib masm32\lib\msvcrt.lib

.DATA

;===User

Data=====

=====

```
_aaaa_dd    0
_bbbb_      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 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 _aaaaaaaa_, 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 _aaaaaaaa_

push _bbbbbbbb_

call Add_

call Output_

invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3 - 1, 0, 0

push _aaaaaaaa_

push _bbbbbbbb_

call Sub_

call Output_

invoke WriteConsoleA, hConsoleOutput, ADDR String_4, SIZEOF String_4 - 1, 0, 0

push _aaaaaaaa_

push _bbbbbbbb_

call Mul_

call Output_

invoke WriteConsoleA, hConsoleOutput, ADDR String_5, SIZEOF String_5 - 1, 0, 0

push _aaaaaaaa_

push _bbbbbbbb_

call Div_

call Output_

invoke WriteConsoleA, hConsoleOutput, ADDR String_6, SIZEOF String_6 - 1, 0, 0

push _aaaaaaaa_

push _bbbbbbbb_

call Mod_

call Output_

push _aaaaaaaa_

push _bbbbbbbb_

call Sub_


```

push dword ptr 10
call Mul_
push _aaaaaaa_
push _bbbbbbbb_
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 _yyyyyyyy_
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 _yyyyyyyy_
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=====
```

```
====
```

```
    _aaaa_dd    0
```

```
    _bbbb_      dd    0
```

```
    _cccc_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 _aaaaaaaa_, eax
```

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

```
    call Input_
```

```
    mov _bbbbbbbbb_, eax
```

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

```
    call Input_
```

```
    mov _cccccccc_, eax
```

```
    push _aaaaaaaa_
```

```
    push _bbbbbbbbb_
```

```
    call Create_
```

```
    pop eax
```

```
    cmp eax, 0
```

```
    je endIf2
```

```
    push _aaaaaaaa_
```

```
    push _cccccccc_
```

```
    call Create_
```

```
    pop eax
```

```
    cmp eax, 0
```

```
    je elseLabel1
```

```
    jmp _temporal_
```

```

    jmp endIf1
elseLabel1:
    push _ccccccc_
    call Output_
    jmp _outugoto_
_temporal_:
    push _aaaaaaaa_
    call Output_
    jmp _outugoto_
endIf1:
endIf2:
    push _bbbbbbbb_
    push _ccccccc_
    call Less_
    pop eax
    cmp eax, 0
    je elseLabel3
    push _ccccccc_
    call Output_
    jmp endIf3
elseLabel3:
    push _bbbbbbbb_
    call Output_
endIf3:
_outugoto_:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3 - 1, 0, 0
    push _aaaaaaaa_
    push _bbbbbbbb_
    call Equal_
    push _aaaaaaaa_
    push _ccccccc_
    call Equal_
    call And_
    push _bbbbbbbb_
    push _ccccccc_
    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 _aaaaaaaa_

```

```

    push dword ptr 0
    call Less_
    push _bbbbbbbbb_
    push dword ptr 0
    call Less_
    call Or_
    push _cccccccc_
    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 _aaaaaaaa_
    push _bbbbbbbbb_
    push _cccccccc_
    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=====
=====
    _aaa2_ dd    0
    _aaaa_ dd    0
    _bbbb_ dd    0
    _ccc1_ dd    0
    _ccc2_ dd    0
    _xxxx_ 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 _bbbbbbbbb_, eax
```

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

```
push _aaaaaaa_
```

```
pop _aaaaaaa2_
```

```
forPasStart1:
```

```
push _bbbbbbbbb_
```

```
push _aaaaaaa2_
```

```
call Less_
```

```
call Not_
```

```
pop eax
```

```
cmp eax, 0
```

```
je forPasEnd1
```

```
invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3 - 1, 0, 0
```

```
push _aaaaaaa2_
```

```
push _aaaaaaa2_
```

```
call Mul_
```

```
call Output_
```

```
push _aaaaaaa2_
```

```
push dword ptr 1
```

```
call Add_
```

```
pop _aaaaaaa2_
```

```

    jmp forPasStart1
forPasEnd1:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_4, SIZEOF String_4 - 1, 0, 0
    push _bbbbbbbb_
    pop _aaaaaaa2_
forPasStart2:
    push _aaaaaaaa_
    push _aaaaaaa2_
    call Greate_
    call Not_
    pop eax
    cmp eax, 0
    je forPasEnd2
    invoke WriteConsoleA, hConsoleOutput, ADDR String_5, SIZEOF String_5 - 1, 0, 0
    push _aaaaaaa2_
    push _aaaaaaa2_
    call Mul_
    call Output_
    push _aaaaaaa2_
    push dword ptr 1
    call Sub_
    pop _aaaaaaa2_
    jmp forPasStart2
forPasEnd2:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_6, SIZEOF String_6 - 1, 0, 0
    push dword ptr 0
    pop _xxxxxxx_
    push dword ptr 0
    pop _ccccccc1_
whileStart2:
    push _ccccccc1_
    push _aaaaaaaa_
    call Less_
    pop eax
    cmp eax, 0
    je whileEnd2
    push dword ptr 0
    pop _ccccccc2_
whileStart1:
    push _ccccccc2_
    push _bbbbbbbb_
    call Less_
    pop eax
    cmp eax, 0
    je whileEnd1
    push _xxxxxxx_
    push dword ptr 1
    call Add_
    pop _xxxxxxx_

```

```

    push _ccccccc2_
    push dword ptr 1
    call Add_
    pop _ccccccc2_
    jmp whileStart1
whileEnd1:
    push _ccccccc1_
    push dword ptr 1
    call Add_
    pop _ccccccc1_
    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 _ccccccc1_
repeatStart2:
    push dword ptr 1
    pop _ccccccc2_
repeatStart1:
    push _xxxxxxxx_
    push dword ptr 1
    call Add_
    pop _xxxxxxxx_
    push _ccccccc2_
    push dword ptr 1
    call Add_
    pop _ccccccc2_
    push _ccccccc2_
    push _bbbbbbbbb_
    call Greate_
    call Not_
    pop eax
    cmp eax, 0
    je repeatEnd1
    jmp repeatStart1
repeatEnd1:
    push _ccccccc1_
    push dword ptr 1
    call Add_
    pop _ccccccc1_
    push _ccccccc1_
    push _aaaaaaaa_
    call Greate_
    call Not_
    pop eax

```

```

    cmp eax, 0
    je repeatEnd2
    jmp repeatStart2
repeatEnd2:
    push _xxxxxxx_
    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