

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 №24

Виконав: ст. гр. КІ-307

Петренко В.А.

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 скопіювати за допомогою відповідних програм (наприклад, LINK, ML і т. д.).

Зміст

Анотація.....	2
Завдання до курсового проекту.....	4
Вступ.....	6
1. Огляд методів та способів проектування трансляторів.....	7
2. Формальний опис вхідної мови програмування.....	10
2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура...10	
2.2. Опис термінальних символів та ключових слів.....	15
3. Розробка транслятора вхідної мови програмування.....	17
3.1. Вибір технології програмування.	17
3.2. Проектування таблиць транслятора.....	18
3.3. Розробка лексичного аналізатора.	20
3.3.1. Розробка блок-схеми алгоритму.....	22
3.3.2. Опис програми реалізації лексичного аналізатора.....	22
3.4. Розробка синтаксичного та семантичного аналізатора.	24
3.4.1. Опис програми реалізації синтаксичного та семантичного аналізатора	25
3.4.2. Розробка граф-схеми алгоритму.....	25
3.5. Розробка генератора коду.....	26
3.5.1. Розробка граф-схеми алгоритму.....	27
3.5.2. Опис програми реалізації генератора коду.	28
4. Опис програми.....	29
4.1. Опис інтерфейсу та інструкція користувачеві.....	34
5. Відлагодження та тестування програми..	35
5.1. Виявлення лексичних та синтаксичних помилок.....	35
5.2. Виявлення семантичних помилок.....	37
5.3. Загальна перевірка коректності роботи транслятора.....	37
5.4. Тестова програма №1.....	39
5.5. Тестова програма №2.....	41
5.6. Тестова програма №3.....	43
Висновки.....	47
Список використаної літератури.....	48
Додатки.....	49

Завдання до курсового проекту

Варіант 24

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 .p24

Опис вхідної мови програмування:

- Тип даних: integer16
- Блок тіла програми: Maimprogram Start Data...; End
- 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-Low перший символ Up
- Регістр ідентифікаторів: Low6 перший символ _
- Операції арифметичні: ++, --, **, Div, Mod
- Операції порівняння: =, <>, Et, 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 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 (extended Backus/Naur Form - EBNF).

```
labeled_point = label , ":"
goto_label = tokenGOTO, label, ";"
program_name = ident, ";"
value_type = tokenINTEGER16
other_declaration_ident = tokenCOMMA , ident
declaration = value_type , ident , {other_declaration_ident}
unary_operator = tokenNOT | tokenMINUS | tokenPLUS
unary_operation = unary_operator , expression
binary_operator = tokenAND | tokenOR | tokenEQUAL | tokenNOTEQUAL | tokenLESSOREQUAL
| tokenGREATEROREQUAL | tokenPLUS | tokenMINUS | tokenMUL | tokenDIV | tokenMOD
binary_action = binary_operator , expression
left_expression = group_expression | unary_operation | ident | value
expression = left_expression , {binary_action}
group_expression = tokenGROUPEXPRESSIONBEGIN , expression , tokenGROUPEXPRESSIONEND
//
bind_right_to_left = ident , tokenRLBIND , expression
bind_left_to_right = expression , tokenLRBIND , ident
//
if_expression = expression
body_for_true = {statement} , ";"
body_for_false = tokenELSE , {statement} , ";"
cond_block = tokenIF , tokenGROUPEXPRESSIONBEGIN , if_expression ,
tokenGROUPEXPRESSIONEND , body_for_true , [body_for_false];
//
cycle_begin_expression = expression
cycle_counter = ident
cycle_counter_rl_init = cycle_counter , tokenRLBIND , cycle_begin_expression
cycle_counter_lr_init = cycle_begin_expression , tokenLRBIND , cycle_counter
cycle_counter_init = cycle_counter_rl_init | cycle_counter_lr_init
cycle_counter_last_value = value
cycle_body = tokenDO , statement , {statement}
forto_cycle = tokenFOR , cycle_counter_init , tokenTO , cycle_counter_last_value , cycle_body ,
";"
continue_while = tokenCONTINUE , tokenWHILE
```

```

exit_while = tokenEXIT , tokenWHILE
statement_in_while_body = statement | continue_while | exit_while
while_cycle_head_expression = expression
while_cycle = tokenWHILE , while_cycle_head_expression , {statement_in_while_body} ,
tokenEND , tokenWHILE
//
repeat_until_cycle_cond = group_expression
repeat_until_cycle = tokenREPEAT , {statement} , tokenUNTIL , repeat_until_cycle_cond
input = tokenGET , tokenGROUPEXPRESSIONBEGIN , ident , tokenGROUPEXPRESSIONEND
output = tokenPUT , tokenGROUPEXPRESSIONBEGIN , expression , tokenGROUPEXPRESSIONEND
statement = bind_right_to_left | bind_left_to_right | cond_block | forto_cycle | while_cycle |
repeat_until_cycle | labeled_point | goto_label | input | output
program = tokenNAME , program_name , tokenSEMICOLON , tokenBODY , tokenDATA ,
[declaration] , tokenSEMICOLON , {statement} , tokenEND
//
digit = digit_0 | digit_1 | digit_2 | digit_3 | digit_4 | digit_5 | digit_6 | digit_7 | digit_8 | digit_9
non_zero_digit = digit_1 | digit_2 | digit_3 | digit_4 | digit_5 | digit_6 | digit_7 | digit_8 |
digit_9
unsigned_value = ((non_zero_digit , {digit}) | digit_0)
value = [sign] , unsigned_value
/-- hello wolrd
letter_in_lower_case = 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
    letter_in_upper_case = 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
    ident = tokenUNDERSCORE , letter_in_lower_case , letter_in_lower_case ,
letter_in_lower_case , letter_in_lower_case , letter_in_lower_case
    label = letter_in_lower_case , {letter_in_lower_case}
//
sign = sign_plus | sign_minus
sign_plus = '-'
sign_minus = '+'
//
digit_0 = '0'
digit_1 = '1'
digit_2 = '2'
digit_3 = '3'
digit_4 = '4'
digit_5 = '5'
digit_6 = '6'
digit_7 = '7'
digit_8 = '8'
digit_9 = '9'
//
tokenCOLON = ":"

```

```

tokenGOTO = "Goto"
tokenINTEGER16 = "Integer16"
tokenCOMMA = ","
tokenNOT = "!"
tokenAND = "&"
tokenOR = "|"
tokenEQUAL = "="
tokenNOTEQUAL = "<>"
tokenLESSOREQUAL = "Lt"
tokenGREATEROREQUAL = "Et"
tokenPLUS = "++"
tokenMINUS = "--"
tokenMUL = "***"
tokenDIV = "Div"
tokenMOD = "Mod"
tokenGROUPEXPRESSIONBEGIN = "("

tokenGROUPEXPRESSIONEND = ")"
tokenRLBIND = "<-"
tokenLRBIND = ","
tokenELSE = "Else"
tokenIF = "If"
tokenDO = "Do"
tokenFOR = "For"
tokenTO = "To"
tokenWHILE = "While"
tokenCONTINUE = "Continue"
tokenEXIT = "Exit"
tokenREPEAT = "Repeat"
tokenUNTIL = "Until"
tokenGET = "Read"
tokenPUT = "Write"
tokenNAME = "MainProgram"
tokenBODY = "Start"
tokenDATA = "Body"
tokenEND = "End"
tokenSEMICOLON = ""
//
tokenUNDERSCORE = " _"
//
A = "A"
B = "B"
C = "C"
D = "D"
E = "E"

```

F = "F"
G = "G"
H = "H"
I = "I"
J = "J"
K = "K"
L = "L"
M = "M"
N = "N"
O = "O"
P = "P"
Q = "Q"
R = "R"
S = "S"
T = "T"
U = "U"
V = "V"
W = "W"
X = "X"
Y = "Y"
Z = "Z"
//
a = "a"
b = "b"
c = "c"
d = "d"
e = "e"
f = "f"
g = "g"
h = "h"
i = "i"
j = "j"
k = "k"
l = "l"
m = "m"
n = "n"
o = "o"
p = "p"
q = "q"
r = "r"
s = "s"
t = "t"
u = "u"
v = "v"
w = "w"

```
x = "x"  
y = "y"  
z = "z"  
//
```

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
Maimprogram	Початок програми
Start	Початок тексту програми
Data	Початок блоку опису змінних
End	Кінець розділу операторів
Read	Оператор вводу змінних
Write	Оператор виводу (змінних або рядкових констант)
<=	Оператор присвоєння
If	Оператор умови
Else	Оператор умови
Goto	Оператор переходу
Label	Мітка переходу
For	Оператор циклу
To	Інкремент циклу
DownTo	Декремент циклу
Do	Початок тіла циклу
While	Оператор циклу
Continue	Оператор циклу
Exit	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
Et	Оператор перевірки чи більше
!	Оператор логічного заперечення
&	Оператор кон'юнкції
	Оператор диз'юнкції
integer16	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	Maimprogram
Start	Start
Vars	Data
End	End
VarType	integer16
Read	Read
Write	Write
Assignment	<-
If	If
Else	Else
Goto	Goto
Colon	:
Label	

For	For
To	To
DownTo	Downto
Do	Do
While	While
WhileContinue	Continue
WhileExit	Exit
Repeat	Repeat
Until	Until
Addition	++
Subtraction	--
Multiplication	**
Division	Div
Mod	Mod
Equal	=
NotEqual	<>
Less	Lt
Greate	Et
Not	!
And	&
Or	
Plus	+
Minus	-
Identifier	
Number	
String	
Undefined	

Unknown	
Comma	,
Quotes	“
Semicolon	;
LBracket	(
RBracket)
LComment	\$\$
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

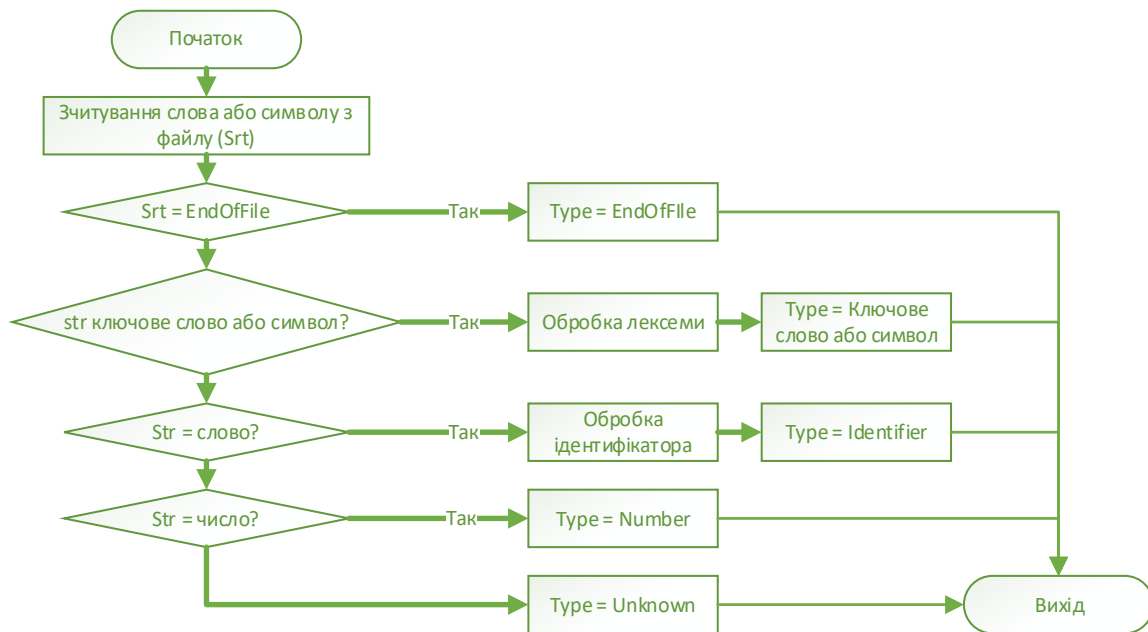


Рис. 3.1 Блок-схема роботи лексичного аналізатора

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, літерали і термінальні символи (операції, ключові слова).

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

При виділенні лексеми вона розпізнається та записується у список `m_tokens` за допомогою відповідного типу лексеми, що є унікальним для кожної

лексми 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 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

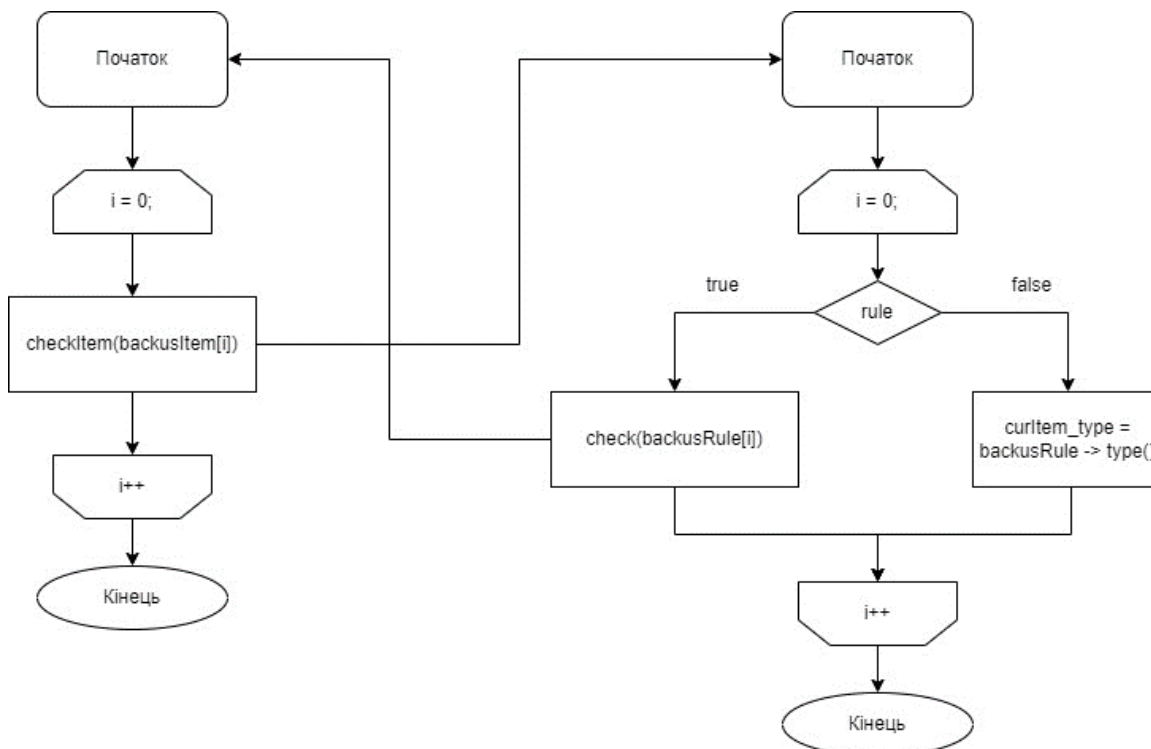
На вхід синтаксичного аналізатора подіється таблиця лексем створена на етапі лексичного аналізу. Аналізатор проходить по ній і перевіряє чи набір лексем відповідає раніше описаним формам нотації Бекуса-Наура. І разі не відповідності у файл з помилками виводиться інформація про помилку і про рядок на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



3.5 Розробка генератора коду

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

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

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

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

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

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

3.5.1 Розробка граф-схеми алгоритму

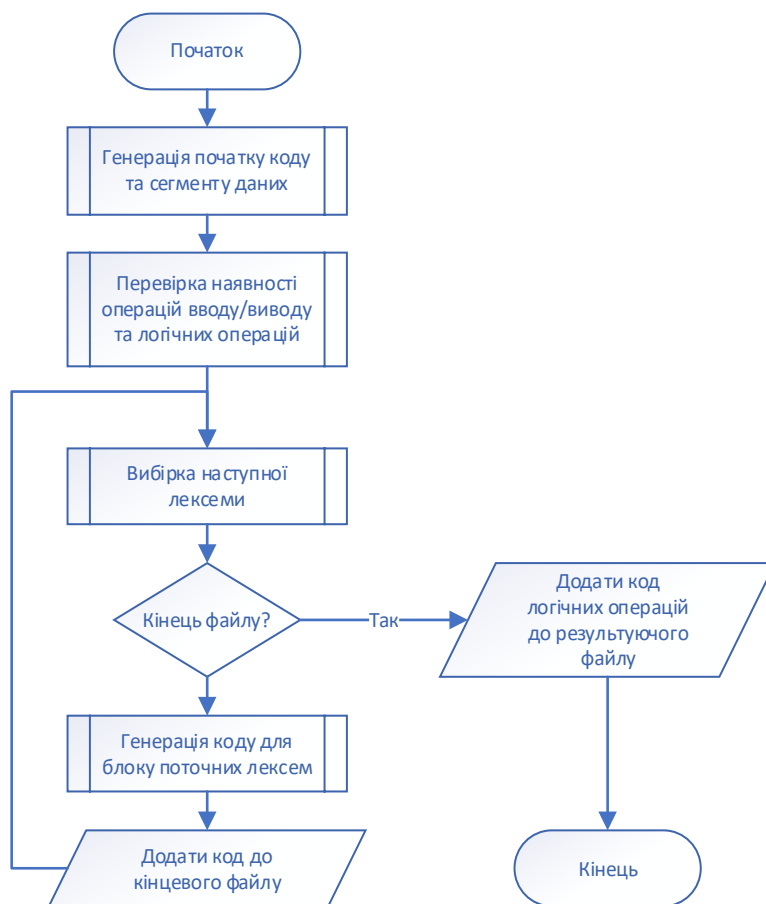


Рис. 3.3 Блок схема генератора коду

3.5.2 Опис програми реалізації генератора коду

У компілятора, реалізованого в даному курсовому 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. 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 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({ 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 трактуються програмою як «АБО», тобто повинен бути лише один з описаних tokenів. Наприклад для write послідовно необхідний token Write після якого йде ліва дужка, далі може бути або певний вираз або рядок тексту який необхідно вивести. І закінчується правило токеном правої дужки.

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

Кожен token це окремий клас що наслідує 3 інтерфейси:

- IToken
- IBackusRule

- IGeneratorItem

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

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

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

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

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

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

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

```

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

```

```

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

    out << "Greate_ PROC\\n";
    out << "\\tpushf\\n";
    out << "\\tpop cx\\n\\n";
    out << "\\tmov " << args.regPrefix << "ax, [esp + " << args.posArg0 << "]\\n";
    out << "\\tcmp " << args.regPrefix << "ax, [esp + " << args.posArg1 << "]\\n";
    out << "\\tjle greate_false\\n";
    out << "\\tmov " << args.regPrefix << "ax, 1\\n";
    out << "\\tjmp greate_fin\\n";
    out << "greate_false:\\n";
    out << "\\tmov " << args.regPrefix << "ax, 0\\n";
    out << "greate_fin:\\n";
    out << "\\tpush cx\\n";
    out << "\\tpopf\\n\\n";

```



```

GeneratorUtils::PrintResultToStack(out, args);

out << "\tret\n";

out << "Greate_ ENDP\n";

out <<
";=====
=====\\n";

}

```

Така структура програми дозволяє без проблем аналізувати великі програми, написані на вхідній мові програмування. Також використання правил Бекуса-Наура дозволяє ефективно 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 p24. 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_p24.exe <ім'я програми>.p24"

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 в подальшому виправити слабкі сторони, якщо вони є.

Відлагодження даної програми здійснюється за допомогою набору кількох програм, які відповідають заданій граматиці. Та перевірка 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
Mainprogram
Start
Data Integer16 _aaaaaa,_bbbb,_xxxxxx,_yyyyyy;
Write("Read _aaaaaa: ");
Re ad(_aaaaaa);
Write("Read _bbbbbb: ");
Read(_bbbbbb);
```

```

Write("_aaaaaa + _bbbbbb: ");
Write(_aaaaaa ++ _bbbbbb);
Write("\n_Aaaaaaaa - _bbbbbb: ");
Write(_aaaaaa -- _bbbbbb);
Write("\n_Aaaaaaaa * _bbbbbb: ");
Write(_aaaaaa ** _bbbbbb);
Write("\n_Aaaaaaaa / _bbbbbb: ");
Write(_aaaaaa Div _bbbbbb);
Write("\n_Aaaaaaaa % _bbbbbb: ");
Write(_aaaaaa Mod _bbbbbb);
_xxxxxx<-( _aaaaaa -- _bbbbbb) ** 10 ++ ( _aaaaaa ++ _bbbbbb) Div 10;
_yyyyyy<-_xxxxxx ++ (_xxxxxx Mod 10);
Write("\n_xxxxxxx = ( _aaaaaa - _bbbbbb) * 10 + ( _aaaaaa + _bbbbbb) / 10\n");
Write(_xxxxxx);
Write("\n_yyyyyyy = _xxxxxx + (_xxxxxx % 10)\n");
Write(_yyyyyy);
End

```

Текст файлу з повідомленнями про помилки

List of errors

```

=====
=====

```

There are 4 lexical errors.

There are 2 syntax errors.

There are 0 semantic errors.

Line 4: Lexical error: Unknown token: _aaaaaa

Line 4: Lexical error: Unknown token: _bbbb

Line 4: Syntax error: Expected: VarsBlok before _aaaaaa

Line 4: Syntax error: Expected: IdentRule before _aaaaaa

Line 6: Lexical error: Unknown token: Re

Line 6: Lexical error: Unknown token: ad

5.2 Виявлення семантичних помилок

Суттю виявлення семантичних помилок є перевірка числових констант на відповідність типу `integer16`, тобто знаковому цілому числу з відповідним діапазоном значень і перевірку на коректність використання змінних `integer16` 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

Mainprogram

Start

Data Integer16 _aaaaaa, _bbbbbb, _xxxxxx, _yyyyyy;

Write("Read _aaaaaa: ");

Read(_aaaaaa);

Write("Read _bbbbbb: ");

Read(_bbbbbb);

Write("_aaaaaa + _bbbbbb: ");

Write(_aaaaaa ++ _bbbbbb);

Write("\n_Aaaaaaaa - _bbbbbb: ");

Write(_aaaaaa -- _bbbbbb);

Write("\n_Aaaaaaaa * _bbbbbb: ");

Write(_aaaaaa ** _bbbbbb);

Write("\n_Aaaaaaaa / _bbbbbb: ");

Write(_aaaaaa Div _bbbbbb);

Write("\n_Aaaaaaaa % _bbbbbb: ");

Write(_aaaaaa Mod _bbbbbb);

```

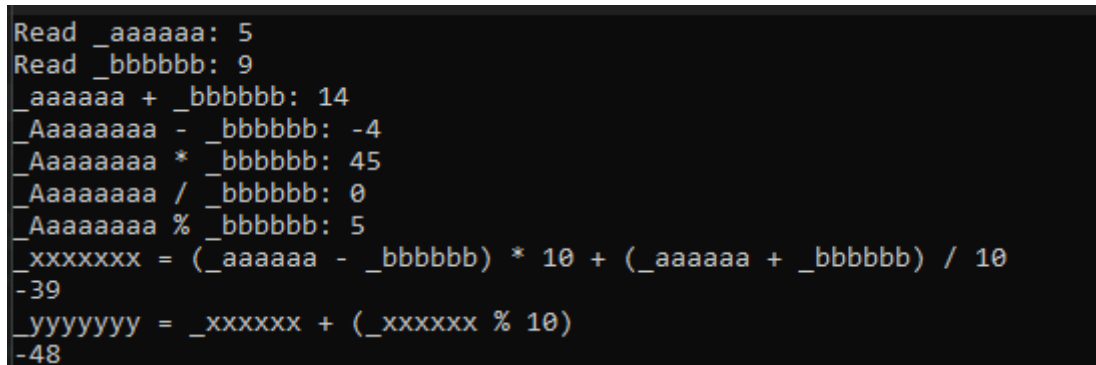
_xxxxxxx<-( _aaaaaa -- _bbbbbb) ** 10 ++ ( _aaaaaa ++ _bbbbbb) Div 10;
_yyyyyyy<-_xxxxxxx ++ ( _xxxxxxx Mod 10);
Write("\n_xxxxxxx = ( _aaaaaa - _bbbbbb) * 10 + ( _aaaaaa + _bbbbbb) / 10\n");
Write(_xxxxxxx);
Write("\n_yyyyyyy = _xxxxxxx + ( _xxxxxxx % 10)\n");
Write(_yyyyyy);
End

```

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:



```

Read _aaaaaa: 5
Read _bbbbbb: 9
_aaaaaa + _bbbbbb: 14
_Aaaaaaaa - _bbbbbb: -4
_Aaaaaaaa * _bbbbbb: 45
_Aaaaaaaa / _bbbbbb: 0
_Aaaaaaaa % _bbbbbb: 5
_xxxxxxx = ( _aaaaaa - _bbbbbb) * 10 + ( _aaaaaa + _bbbbbb) / 10
-39
_yyyyyyy = _xxxxxxx + ( _xxxxxxx % 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

Mainprogram

Start

Data Integer16 _aaaaaa,_bbbbbb,_xxxxxx,_yyyyyy;

Write("Read _aaaaaa: ");

Read(_aaaaaa);

Write("Read _bbbbbb: ");

Read(_bbbbbb);

Write("_aaaaaa + _bbbbbb: ");

Write(_aaaaaa ++ _bbbbbb);

Write("\n_Aaaaaaaa - _bbbbbb: ");

Write(_aaaaaa -- _bbbbbb);

Write("\n_Aaaaaaaa * _bbbbbb: ");

Write(_aaaaaa ** _bbbbbb);

Write("\n_Aaaaaaaa / _bbbbbb: ");

Write(_aaaaaa Div _bbbbbb);

Write("\n_Aaaaaaaa % _bbbbbb: ");

Write(_aaaaaa Mod _bbbbbb);

_xxxxxx<-(_aaaaaa -- _bbbbbb) ** 10 ++ (_aaaaaa ++ _bbbbbb) Div 10;

_yyyyyy<-_xxxxxx ++ (_xxxxxx Mod 10);

Write("\n _xxxxxx = (_aaaaaa - _bbbbbb) * 10 + (_aaaaaa + _bbbbbb) / 10\n");

Write(_xxxxxx);

Write("\n _yyyyyy = _xxxxxx + (_xxxxxx % 10)\n");

Write(_yyyyyy);

End

Результат виконання

```
Read _aaaaaa: 5
Read _bbbbbb: 9
_aaaaaa + _bbbbbb: 14
_Aaaaaaaa - _bbbbbb: -4
_Aaaaaaaa * _bbbbbb: 45
_Aaaaaaaa / _bbbbbb: 0
_Aaaaaaaa % _bbbbbb: 5
_xxxxxxx = (_aaaaaa - _bbbbbb) * 10 + (_aaaaaa + _bbbbbb) / 10
-39
_yyyyyyy = _xxxxxx + (_xxxxxx % 10)
-48
```

Рис. 5.2 Результат виконання тестової програми №1

5.5 Тестова програма №2

Текст програми

\$\$Prog2

Mainprogram

Start

Data Integer16 _aaaaaa,_bbbbbb,_cccccc;

Write("Read _aaaaaa: ");

Read(_aaaaaa);

Write("Read _bbbbbb: ");

Read(_bbbbbb);

Write("Read _cccccc: ");

Read(_cccccc);

If(_aaaaaa Et _bbbbbb)

Start

 If(_aaaaaa Et _cccccc)

 Start

 Goto _avalue;

 End

 Else

 Start

 Write(_cccccc);

 Goto _outoif;

 _avalue:

 Write(_aaaaaa);

 Goto _outoif;

 End

End

```

        If(_bbbbbb Lt _cccccc)
        Start
            Write(_cccccc);
        End
    Else
        Start
            Write(_bbbbbb);
        End
_outoif:
Write("\n");
If((_aaaaaa = _bbbbbb) & (_aaaaaa = _cccccc) & (_bbbbbb = _cccccc))
Start
    Write(1);
End
Else
Start
    Write(0);
End
Write("\n");
If((_aaaaaa Lt 0) | (_bbbbbb Lt 0) | (_cccccc Lt 0))
Start
    Write(-1);
End
Else
Start
    Write(0);
End
Write("\n");

```

```
If(!_aaaaaa Lt (_bbbbbb ++ _cccccc)))
```

```
Start
```

```
    Write(10);
```

```
End
```

```
Else
```

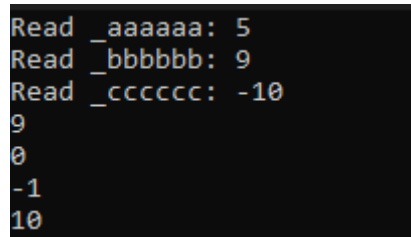
```
Start
```

```
    Write(0);
```

```
End
```

```
End
```

Результат виконання



```
Read _aaaaaa: 5
Read _bbbbbb: 9
Read _cccccc: -10
9
0
-1
10
```

Рис. 5.3 Результат виконання тестової програми №2

5.6 Тестова програма №3

Текст програми

```
$$Prog3
```

```
Mainprogram
```

```
Start
```

```
Data Integer16 _aaaaaa,_aaaaa2,_bbbbbb,_xxxxxx,_cccc1,_cccc2;
```

```
Write("Read _aaaaaa: ");
```

```
Read(_aaaaaa);
```

```
Write("Read _bbbbbb: ");
```

```
Read(_bbbbbb);
```

```
Write("For To do");
```

```

For _aaaaa2<-_aaaaaa To _bbbbbb Do
Start
    Write("\n");
    Write(_aaaaa2 ** _aaaaa2);
End
Write("\nFor Downto do");
For _aaaaa2<-_bbbbbb Downto _aaaaaa Do
Start
    Write("\n");
    Write(_aaaaa2 ** _aaaaa2);
End

Write("\nWhile _aaaaaa * _bbbbbb: ");
_xxxxxx<-0;
_ccccc1<-0;
While(_cccc1 Lt _aaaaaa)
Start
    _cccc2<-0;
    While (_cccc2 Lt _bbbbbb)
    Start
        _xxxxxx<-_xxxxxx ++ 1;
        _cccc2<-_cccc2 ++ 1;
    End
    _cccc1<-_cccc1 ++ 1;
End
Write(_xxxxxx);

Write("\nRepeat Until _aaaaaa * _bbbbbb: ");

```

```
_xxxxxx<-0;
_ccccc1<-1;
Repeat
  _cccc2<-1;
  Repeat
    _xxxxxx<-_xxxxxx++1;
    _cccc2<-_cccc2++1;
  Until(!(_cccc2 Et _bbbbbb))
  _cccc1<-_cccc1++1;
Until(!(_cccc1 Et _aaaaaa))
Write(_xxxxxx);

End
```

Результат виконання

```
Read _aaaaaa: 5
Read _bbbbbb: 9
For To do
25
36
49
64
81
For Downto do
81
64
49
36
25
While _aaaaaa * _bbbbbb: 45
Repeat Until _aaaaaa * _bbbbbb: 45
```

Рис. 5.4 Результат виконання тестової програми №3

Висновки

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування p24, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.
2. Створено компілятор мови програмування p24, 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 p24. 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 p24 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

includelib masm32\lib\kernel32.lib

includelib masm32\lib\masm32.lib

includelib masm32\lib\user32.lib

includelib masm32\lib\msvcrt.lib

.DATA

;===User

Data=====

aaaaaa dw 0

bbbbbb dw 0

xxxxxx dw 0

yyyyyy dw 0

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

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

String_0 db "Read _aaaaaa: ", 0

```

String_1    db    "Read _bbbbbb: ", 0
String_2    db    "_aaaaaa + _bbbbbb: ", 0
String_3    db    13, 10, "_Aaaaaaaa - _bbbbbb: ", 0
String_4    db    13, 10, "_Aaaaaaaa * _bbbbbb: ", 0
String_5    db    13, 10, "_Aaaaaaaa / _bbbbbb: ", 0
String_6    db    13, 10, "_Aaaaaaaa % _bbbbbb: ", 0
String_7    db    13, 10, "_xxxxxxx = (_aaaaaa - _bbbbbb) * 10 +
(_aaaaaa + _bbbbbb) / 10", 13, 10, 0
String_8    db    13, 10, "_yyyyyyy = _xxxxxx + (_xxxxxx % 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    "%hd", 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 _aaaaaa_, ax
    invoke WriteConsoleA, hConsoleOutput, ADDR String_1, SIZEOF String_1
- 1, 0, 0
    call Input_
    mov _bbbbbb_, ax
    invoke WriteConsoleA, hConsoleOutput, ADDR String_2, SIZEOF String_2
- 1, 0, 0
    push _aaaaaa_
    push _bbbbbb_
    call Add_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3
- 1, 0, 0
    push _aaaaaa_
    push _bbbbbb_
    call Sub_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_4, SIZEOF String_4
- 1, 0, 0
    push _aaaaaa_
    push _bbbbbb_
    call Mul_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_5, SIZEOF String_5
- 1, 0, 0
    push _aaaaaa_

```

```

    push _bbbbbb_
    call Div_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_6, SIZEOF String_6
- 1, 0, 0
    push _aaaaaa_
    push _bbbbbb_
    call Mod_
    call Output_
    push _aaaaaa_
    push _bbbbbb_
    call Sub_
    push word ptr 10
    call Mul_
    push _aaaaaa_
    push _bbbbbb_
    call Add_
    push word ptr 10
    call Div_
    call Add_
    pop _xxxxxx_
    push _xxxxxx_
    push _xxxxxx_
    push word ptr 10
    call Mod_
    call Add_
    pop _yyyyyy_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_7, SIZEOF String_7
- 1, 0, 0

```

```

    push _xxxxxx_
    call Output_
    invoke WriteConsoleA, hConsoleOutput, ADDR String_8, SIZEOF String_8
- 1, 0, 0
    push _yyyyyy_
    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 ax, [esp + 6]
    add ax, [esp + 4]
    mov [esp + 6], ax
    pop ecx
    pop ax
    push ecx
    ret

```

```

Add_ ENDP

```

```

;=====
=====

```

;===Procedure

Div=====

Div_ PROC

pushf

pop cx

mov ax, [esp + 4]

cmp ax, 0

jne end_check

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

jmp exit_label

end_check:

mov ax, [esp + 6]

cmp ax, 0

jge gr

lo:

mov dx, -1

jmp less_fin

gr:

mov dx, 0

less_fin:

mov ax, [esp + 6]

idiv word ptr [esp + 4]

push cx

popf

mov [esp + 6], ax

```

        pop ecx
        pop ax
        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 ax, [esp + 4]
        cmp ax, 0

```

```

        jne end_check

        invoke WriteConsoleA, hConsoleOutput, ADDR ModErrMsg, SIZEOF
ModErrMsg - 1, 0, 0
        jmp exit_label
end_check:
        mov ax, [esp + 6]
        cmp ax, 0
        jge gr
lo:
        mov dx, -1
        jmp less_fin
gr:
        mov dx, 0
less_fin:
        mov ax, [esp + 6]
        idiv word ptr [esp + 4]
        mov ax, dx
        push cx
        popf

        mov [esp + 6], ax
        pop ecx
        pop ax
        push ecx
        ret
Mod_ ENDP
;=====
=====

```


;===Procedure

Mul=====

Mul_ PROC

mov ax, [esp + 6]

imul word ptr [esp + 4]

mov [esp + 6], ax

pop ecx

pop ax

push ecx

ret

Mul_ ENDP

=====

;===Procedure

Output=====

Output_ PROC value: word

invoke wsprintf, ADDR ResMessage, ADDR OutMessage, value

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

ret 2

Output_ ENDP

=====

;===Procedure

Sub=====

Sub_ PROC

mov ax, [esp + 6]

sub ax, [esp + 4]

mov [esp + 6], ax

pop ecx

pop ax

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=====

aaaaaa dw 0

bbbbbb dw 0

cccccc dw 0

String_0 db "Read _aaaaaa: ", 0

String_1 db "Read _bbbbbb: ", 0

String_2 db "Read _cccccc: ", 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 ?

endBuff db 5 dup (?)

msg1310 db 13, 10, 0

CharsReadNum dd ?

InputBuf db 15 dup (?)

OutMessage db "%hd", 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 _aaaaaa_, ax

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

 call Input_

 mov _bbbbbb_, ax

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

 call Input_

 mov _cccccc_, ax

 push _aaaaaa_

 push _bbbbbb_

 call Greate_

 pop ax

 cmp ax, 0

 je endIf2

 push _aaaaaa_

 push _cccccc_

 call Greate_

 pop ax

 cmp ax, 0

```

        je elseLabel1
        jmp _avalue_
        jmp endIf1
elseLabel1:
        push _cccccc_
        call Output_
        jmp _outoif_
_avaue_:
        push _aaaaaa_
        call Output_
        jmp _outoif_
endIf1:
endIf2:
        push _bbbbbb_
        push _cccccc_
        call Less_
        pop ax
        cmp ax, 0
        je elseLabel3
        push _cccccc_
        call Output_
        jmp endIf3
elseLabel3:
        push _bbbbbb_
        call Output_
endIf3:
_outoif_:
        invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3
        - 1, 0, 0

```

```

push _aaaaaa_
push _bbbbbb_
call Equal_
push _aaaaaa_
push _cccccc_
call Equal_
call And_
push _bbbbbb_
push _cccccc_
call Equal_
call And_
pop ax
cmp ax, 0
je elseLabel4
push word ptr 1
call Output_
jmp endIf4

```

elseLabel4:

```

push word ptr 0
call Output_

```

endIf4:

```

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

```

```

push _aaaaaa_
push word ptr 0
call Less_
push _bbbbbb_
push word ptr 0
call Less_

```

```

    call Or_
    push _cccccc_
    push word ptr 0
    call Less_
    call Or_
    pop ax
    cmp ax, 0
    je elseLabel5
    push word ptr -1
    call Output_
    jmp endIf5
elseLabel5:
    push word ptr 0
    call Output_
endIf5:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_5, SIZEOF String_5
- 1, 0, 0
    push _aaaaaa_
    push _bbbbbb_
    push _cccccc_
    call Add_
    call Less_
    call Not_
    pop ax
    cmp ax, 0
    je elseLabel6
    push word ptr 10
    call Output_
    jmp endIf6

```

elseLabel6:

 push word 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 ax, [esp + 6]

 add ax, [esp + 4]

 mov [esp + 6], ax

 pop ecx

 pop ax

 push ecx

 ret

Add_ ENDP

;=====

;===Procedure

And=====

And_ PROC

pushf

pop cx

mov ax, [esp + 6]

cmp ax, 0

jnz and_t1

jz and_false

and_t1:

mov ax, [esp + 4]

cmp ax, 0

jnz and_true

and_false:

mov ax, 0

jmp and_fin

and_true:

mov ax, 1

and_fin:

push cx

popf

mov [esp + 6], ax

pop ecx

pop ax

push ecx

ret

And_ ENDP

;
=====

;===Procedure

Equal=====

Equal_ PROC

pushf

pop cx

mov ax, [esp + 6]

cmp ax, [esp + 4]

jne equal_false

mov ax, 1

jmp equal_fin

equal_false:

mov ax, 0

equal_fin:

push cx

popf

mov [esp + 6], ax

pop ecx

pop ax

push ecx

ret

Equal_ ENDP

=====

====Procedure

Greate=====

Greate_ PROC

pushf

pop cx

mov ax, [esp + 6]

cmp ax, [esp + 4]

jle greate_false

mov ax, 1

jmp greate_fin

greate_false:

mov ax, 0

greate_fin:

push cx

popf

mov [esp + 6], ax

pop ecx

pop ax

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 ax, [esp + 6]
```

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

```
    jge less_false
```

```
    mov ax, 1
```

```
    jmp less_fin
```

```
less_false:
```

```
    mov ax, 0
```

```
less_fin:
```

```
    push cx
```

```
    popf
```

```

        mov [esp + 6], ax
        pop ecx
        pop ax
        push ecx
        ret
Less_ ENDP

;=====
=====

;===Procedure
Not=====
=====

Not_ PROC
    pushf
    pop cx

    mov ax, [esp + 4]
    cmp ax, 0
    jnz not_false
not_t1:
    mov ax, 1
    jmp not_fin
not_false:
    mov ax, 0
not_fin:
    push cx
    popf

```

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

```
    ret
```

```
Not_ ENDP
```

```
=====
```

```
=====Procedure
```

```
Or=====
```

```
Or_ PROC
```

```
    pushf
```

```
    pop cx
```

```
    mov ax, [esp + 6]
```

```
    cmp ax, 0
```

```
    jnz or_true
```

```
    jz or_t1
```

```
or_t1:
```

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

```
    cmp ax, 0
```

```
    jnz or_true
```

```
or_false:
```

```
    mov ax, 0
```

```
    jmp or_fin
```

```
or_true:
```

```
    mov ax, 1
```

```
or_fin:
```

```
push cx
```

```
popf
```

```
mov [esp + 6], ax
```

```
pop ecx
```

```
pop ax
```

```
push ecx
```

```
ret
```

```
Or_ ENDP
```

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

```
;===Procedure
```

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

```
Output_ PROC value: word
```

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

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

```
ret 2
```

```
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=====
=====

```

```

    _aaaaa2_    dw    0
    _aaaaaa_    dw    0
    _bbbbbb_    dw    0
    _cccc1_     dw    0
    _cccc2_     dw    0
    _xxxxxx_    dw    0

```

```

String_0    db    "Read _aaaaaa: ", 0
String_1    db    "Read _bbbbbb: ", 0
String_2    db    "For To do", 0
String_3    db    13, 10, 0
String_4    db    13, 10, "For Downto do", 0
String_5    db    13, 10, 0
String_6    db    13, 10, "While _aaaaaa * _bbbbbb: ", 0
String_7    db    13, 10, "Repeat Until _aaaaaa * _bbbbbb: ", 0

```


;===Addition

Data=====

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

```
CharsReadNum     dd    ?
InputBuf         db    15 dup (?)
OutMessage       db    "%hd", 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 _aaaaaa_, ax

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

call Input_

mov _bbbbbb_, ax

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

```

    push _aaaaaa_
    pop _aaaaa2_
forPasStart1:
    push _bbbbbb_
    push _aaaaa2_
    call Less_
    call Not_
    pop ax
    cmp ax, 0
    je forPasEnd1
    invoke WriteConsoleA, hConsoleOutput, ADDR String_3, SIZEOF String_3
- 1, 0, 0
    push _aaaaa2_
    push _aaaaa2_
    call Mul_
    call Output_
    push _aaaaa2_
    push word ptr 1
    call Add_
    pop _aaaaa2_
    jmp forPasStart1
forPasEnd1:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_4, SIZEOF String_4
- 1, 0, 0
    push _bbbbbb_
    pop _aaaaa2_
forPasStart2:
    push _aaaaaa_
    push _aaaaa2_

```

```

    call Greate_
    call Not_
    pop ax
    cmp ax, 0
    je forPasEnd2
    invoke WriteConsoleA, hConsoleOutput, ADDR String_5, SIZEOF String_5
- 1, 0, 0
    push _aaaaa2_
    push _aaaaa2_
    call Mul_
    call Output_
    push _aaaaa2_
    push word ptr 1
    call Sub_
    pop _aaaaa2_
    jmp forPasStart2
forPasEnd2:
    invoke WriteConsoleA, hConsoleOutput, ADDR String_6, SIZEOF String_6
- 1, 0, 0
    push word ptr 0
    pop _xxxxxx_
    push word ptr 0
    pop _cccccl_
whileStart2:
    push _cccccl_
    push _aaaaaa_
    call Less_
    pop ax
    cmp ax, 0

```

```

        je whileEnd2
        push word ptr 0
        pop _cccc2_
whileStart1:
        push _cccc2_
        push _bbbbbb_
        call Less_
        pop ax
        cmp ax, 0
        je whileEnd1
        push _xxxxxx_
        push word ptr 1
        call Add_
        pop _xxxxxx_
        push _cccc2_
        push word ptr 1
        call Add_
        pop _cccc2_
        jmp whileStart1
whileEnd1:
        push _cccc1_
        push word ptr 1
        call Add_
        pop _cccc1_
        jmp whileStart2
whileEnd2:
        push _xxxxxx_
        call Output_

```

```
        invoke WriteConsoleA, hConsoleOutput, ADDR String_7, SIZEOF String_7  
- 1, 0, 0
```

```
    push word ptr 0
```

```
    pop _xxxxxx_
```

```
    push word ptr 1
```

```
    pop _cccccl_
```

```
repeatStart2:
```

```
    push word ptr 1
```

```
    pop _cccccl_
```

```
repeatStart1:
```

```
    push _xxxxxx_
```

```
    push word ptr 1
```

```
    call Add_
```

```
    pop _xxxxxx_
```

```
    push _cccccl_
```

```
    push word ptr 1
```

```
    call Add_
```

```
    pop _cccccl_
```

```
    push _cccccl_
```

```
    push _bbbbbb_
```

```
    call Greate_
```

```
    call Not_
```

```
    pop ax
```

```
    cmp ax, 0
```

```
    je repeatEnd1
```

```
    jmp repeatStart1
```

```
repeatEnd1:
```

```
    push _cccccl_
```

```
    push word ptr 1
```

```

    call Add_
    pop _cccc1_
    push _cccc1_
    push _aaaaaa_
    call Greate_
    call Not_
    pop ax
    cmp ax, 0
    je repeatEnd2
    jmp repeatStart2
repeatEnd2:
    push _xxxxxx_
    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 ax, [esp + 6]
    add ax, [esp + 4]
    mov [esp + 6], ax
    pop ecx
    pop ax

```

```

        push ecx
        ret
Add_ ENDP

;=====
=====

;===Procedure
Greate=====
=====

Greate_ PROC
        pushf
        pop cx

        mov ax, [esp + 6]
        cmp ax, [esp + 4]
        jle greate_false
        mov ax, 1
        jmp greate_fin
greate_false:
        mov ax, 0
greate_fin:
        push cx
        popf

        mov [esp + 6], ax
        pop ecx
        pop ax
        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 ax, [esp + 6]

        cmp ax, [esp + 4]

        jge less_false

        mov ax, 1

        jmp less_fin

```


less_false:

mov ax, 0

less_fin:

push cx

popf

mov [esp + 6], ax

pop ecx

pop ax

push ecx

ret

Less_ ENDP

;
=====

;===Procedure

Mul=====

Mul_ PROC

mov ax, [esp + 6]

imul word ptr [esp + 4]

mov [esp + 6], ax

pop ecx

pop ax

push ecx

ret

Mul_ ENDP

;
=====

;===Procedure

Not=====

Not_ PROC

pushf

pop cx

mov ax, [esp + 4]

cmp ax, 0

jnz not_false

not_t1:

mov ax, 1

jmp not_fin

not_false:

mov ax, 0

not_fin:

push cx

popf

mov [esp + 4], ax

ret

Not_ ENDP

=====

```

;===Procedure
Output=====

=====

Output_ PROC value: word
    invoke wsprintf, ADDR ResMessage, ADDR OutMessage, value
    invoke WriteConsoleA, hConsoleOutput, ADDR ResMessage, eax, 0, 0
    ret 2
Output_ ENDP

;=====
=====

;===Procedure
Sub=====

=====

Sub_ PROC
    mov ax, [esp + 6]
    sub ax, [esp + 4]
    mov [esp + 6], ax
    pop ecx
    pop ax
    push ecx
    ret
Sub_ ENDP

;=====
=====

end start

```

