

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 №3

Виконала: ст. гр. КІ-308
Бохонок О. П.
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 С. 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 коду на мові С для кожного блоку лексем. Отриманий код записується у результуючий файл, готовий для виконання.

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.....	12
3. Розробка транслятора вхідної мови програмування.....	14
3.1. Вибір технології програмування.....	14
3.2. Проектування таблиць транслятора.....	15
3.3. Розробка лексичного аналізатора.....	17
3.3.1. Розробка блок-схеми алгоритму.....	19
3.3.2. Опис програми реалізації лексичного аналізатора	20
3.4. Розробка синтаксичного та семантичного аналізатора	22
3.4.1. Опис програми реалізації синтаксичного та семантичного аналізатора	23
3.4.2. Розробка граф-схеми алгоритму	25
3.5. Розробка генератора коду	25
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.....	32
5. Відлагодження та тестування програми.....	33
5.1. Виявлення лексичних та синтаксичних помилок	33
5.2. Виявлення семантичних помилок	34
5.3. Загальна перевірка коректності роботи транслятора.....	34
5.4. Тестова програма №1.....	36
5.5. Тестова програма №2.....	37
5.6. Тестова програма №3.....	38
Висновки.....	40
Список використаної літератури	41
Додатки.....	42

Завдання до курсового проекту

Варіант 3

Завдання на курсовий проект

1. Цільова мова транслятора – мова програмування C.
2. Для отримання виконавчого файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio або будь-яким іншим.
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 C;*
 - *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 .b03

Опис вхідної мови програмування:

- Тип даних: Int16
- Блок тіла програми: Program <name>; Var...; Begin End
- Оператор вводу: Get ()
- Оператор виводу: Put ()
- 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: Up6
- Операції арифметичні: +, -, Mul, Div, Mod
- Операції порівняння: Eg, Ne, >>, <<
- Операції логічні: !, And, Or
- Коментар: !!... !!
- Ідентифікатори змінних, числові константи

- Оператор присвоєння: ==>

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

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 іншій цільовій мові програмування, такій як С. Наведене визначення застосовне до різноманітних транслуючих 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.

Компілятор - обслуговуюча програма, яка виконує трансляцію програми, написаної мовою оригіналу програмування, в іншу мову, наприклад, мову С. Схоже до асемблера, компілятор виконує перетворення програми з однієї мови в іншу, часто генеруючи код, який може бути виконаний іншими компіляторами.

Інтерпретатор - це програма чи пристрій, що виконує пооператорну трансляцію та виконання вихідної програми. Відмінно від компілятора, інтерпретатор не створює на виході нову програму мовою С. 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 створення коду мовою С, з урахуванням ефективності його виконання.

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

Під час процесу виділення лексем лексичний аналізатор може виконувати дві основні функції: автоматичну побудову таблиць об'єктів (таких як ідентифікатори, рядки, числа тощо) 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 у внутрішнє представлення. Це здійснюється з метою оптимізації та/або для полегшення генерації коду мовою C. Крім того, перетворення програми у внутрішнє представлення може бути використано для створення переносимого транслятора. У цьому випадку, тільки остання фаза (генерація коду) є залежною від конкретної архітектури. В якості внутрішнього представлення може використовуватися 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-незалежні оптимізації орієнтовані на спрощення коду або видалення надлишкових обчислень, тоді як машинно-залежні оптимізації проводяться на етапі генерації коду.

Фінальна фаза трансляції - генерація коду мовою C. На цьому етапі можуть застосовуватися деякі локальні оптимізації для полегшення генерації ефективного та читабельного коду.

Важливо відзначити, що фази транслятора можуть бути відсутніми або об'єднаними залежно від конкретної реалізації. У простіших випадках, таких як однопрохідні транслятори, може бути відсутній окремий етап генерації проміжного представлення та оптимізації, а інші фази можуть бути об'єднані в одну, без створення 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).

```
topRule = "Program", identifier, ";", varsBlok, ";", "Begin", operators, "End";
varsBlok = "Var", "Int16", identifier, [{ commaAndIdentifier }];
identifier = up_letter, up_letter, up_letter, up_letter, up_letter, up_letter ;
commaAndIdentifier = ",", identifier;
codeBlok = "Begin", operators, "End";
operators = write | read | assignment | ifStatement | goto_statement | labelRule
| forToOrDownToDoRule | while | repeatUntil;
read = "Get", "(", identifier, ")";
write = "Put", "(", equation | stringRule, ")";
assignment = identifier, "==>", equation;
cycle_counter = identifier;
cycle_counter_last_value = equation;
ifStatement = "If", "(", equation, ")", codeBlok, ["Else", codeBlok];
goto_statement = "Goto", ident ;
labelRule = identifier, ":";
forToOrDownToDoRule = "For", cycle_counter, "==>", equation , "To" |
"Downto", cycle_counter_last_value, "Do", codeBlok;
while = "While", "(", equation, ")", "Begin", operators | whileContinue | whileExit,
"End", "While";
whileContinue = "Continue", "While";
whileExit = "Exit", "While";
repeatUntil = "Repeat", operators, "Until", "(", equation, ")";
```

```

equation = signedNumber | identifier | notRule [{ operationAndIdentOrNumber
| equation }];

notRule = notOperation, signedNumber | identifier | equation;

operationAndIdentOrNumber = mult | arithmetic | logic | compare
signedNumber | identifier | equation;

arithmetic = "+" | "-";

mult = "Mul" | "Div" | "Mod";

logic = "And" | "Or";

notOperation = "!";

compare = "Eg" | "Ne" | "<<" | ">>";

comment = "LComment", text , "RComment" ;

LComment = "!!";

RComment = "!!";

text = { 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";

```

2.2. Опис термінальних символів та ключових слів

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

Термінальний символ або ключове слово	Значення
Program	Початок програми
Begin	Початок тексту програми
Var	Початок блоку опису змінних
End	Кінець розділу операторів
Get	Оператор вводу змінних
Put	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
Mul	Оператор множення

Div	Оператор ділення
Mod	Оператор знаходження залишку від ділення
Eg	Оператор перевірки на рівність
Ne	Оператор перевірки на нерівність
<<	Оператор перевірки чи менше
>>	Оператор перевірки чи більше
!	Оператор логічного заперечення
And	Оператор кон'юнкції
Or	Оператор диз'юнкції
Int16	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 наступні:

- 1) Таблиця лексем з елементами, які мають таку структуру:

```
struct Token
{
    char name[16];          // ім'я лексеми
    int value;              // значення лексеми (для цілих констант)
    int line;              // номер рядка
    TypeOfTokens type;     // тип лексеми
};
```

- 2) Таблиця лексичних класів

```
enum TypeOfTokens
{
    Mainprogram,
    StartProgram,
    Variable,
    Type,
    EndProgram,
    Input,
    Output,

    If,
    Else,

    Goto,
    Label,

    For,
    To,
    DownTo,
    Do,

    While,
    Exit,
    Continue,
    End,

    Repeat,
    Until,

    Identifier,
    Number,
    Assign,
    Add,
    Sub,
    Mul,
    Div,
    Mod,
    Equality,
    NotEquality,
    Greate,
    Less,
    Not,
    And,
    Or,
    LBraket,
    RBraket,
```

```
Semicolon,  
Colon,  
Comma,  
Unknown  
};
```

Якщо у стовпці «Значення» відсутня інформація про токен, то це означає що його значення визначається користувачем під час написання коду на створеній мові програмування.

Таблиця 2 Опис термінальних символів та ключових слів

Токен	Значення
Program	Program
Start	Begin
Vars	Var
End	End
VarType	Int16
Read	Get
Write	Put
Assignment	==>
If	If
Else	Else
Goto	Goto
Colon	:
Label	
For	For
To	To
DownTo	Downto
Do	Do
While	While
Continue	Continue
Exit	Exit
Repeat	Repeat
Until	Until

Addition	+
Subtraction	-
Multiplication	Mul
Division	Div
Mod	Mod
Equal	Eg
NotEqual	Ne
Less	<<
Greate	>>
Not	!
And	And
Or	Or
Identifier	
Number	
Unknown	
Comma	,
Semicolon	;
LBracket	(
RBracket)
LComment	!!
RComment	!!
Comment	

3.3. Розробка лексичного аналізатора

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

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

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

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

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

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

За варіантом розділено лексеми на типи або лексичні класи:

- Ключові слова (0-Program, 1-Var, 2-Begin, 3-End, 4-Get, 5-Put, 6-Int16, 7-If, 8-Else, 9-Goto, 10-For, 11-To, 12-Do, 13-Downto, 14-While, 15-Repeat, 16-Until, 17-Exit)
- Ідентифікатори (18-максимум 6 великих літер)
- Числові константи (19-ціле число без знаку)
- Оператор присвоєння (20- ==>)
- Знаки операції (21- +, 22- -, 23- Mul, 24- Div, 25- Mod, 26- >>, 27- <<, 28- Eg, 29- Ne, 30- !, 31- And, 32- Or)
- Розділювачі (33- ;, 34- ,)
- Дужки (35- (, 36-))
- Невідома лексема (37- символи і ланцюжки символів, які не підпадають під вище описані правила)

3.3.1. Розробка блок-схеми алгоритму

Лексичний аналізатор виконує обробку тексту з вхідного файлу та визначає типи лексем за наступними етапами. Процес роботи можна описати наступним чином::

- **Початок роботи:**
алгоритм починається зі зчитування чергового слова або символу з файлу (`Str`).
- **Перевірка на кінець файлу:**
якщо зчитане значення `Str` дорівнює `EndOfFile`, тип лексеми визначається як `EndOfFile`, після чого завершується обробка.
- **Ключове слово або символ:**
якщо зчитане значення є ключовим словом або символом, виконується обробка лексеми, і тип визначається як `Ключове слово або символ`.
- **Розпізнавання слова (ідентифікатора):**
якщо значення є словом, проводиться обробка ідентифікатора, а тип встановлюється як `Identifier`.
- **Розпізнавання числа:**
якщо значення є числом, тип лексеми визначається як `Number`.
- **Невідомий тип:**
якщо жодна з вищезазначених умов не виконана, тип лексеми визначається як `Unknown`.
- **Завершення роботи:**
після визначення типу лексеми алгоритм повертається до зчитування наступного слова або символу, доки не буде досягнуто стану `EndOfFile`.

Алгоритм працює на основі автомату з наступними станами:

- **Start** – початковий стан, зчитування символу.
- **EndOfFile** – завершення обробки файлу.
- **Ключове слово або символ** – обробка зарезервованих слів або символів.
- **Identifier** – обробка лексем, що є ідентифікаторами.
- **Number** – обробка числових значень.
- **Unknown** – визначення невідомих лексем.

Ця модель дозволяє покроково аналізувати вхідний файл, виділяючи ключові слова, ідентифікатори, числа та інші елементи, забезпечуючи коректну обробку кожної лексеми.

Алгоритм роботи лексичного аналізатора можна зобразити у вигляді блок-схеми.

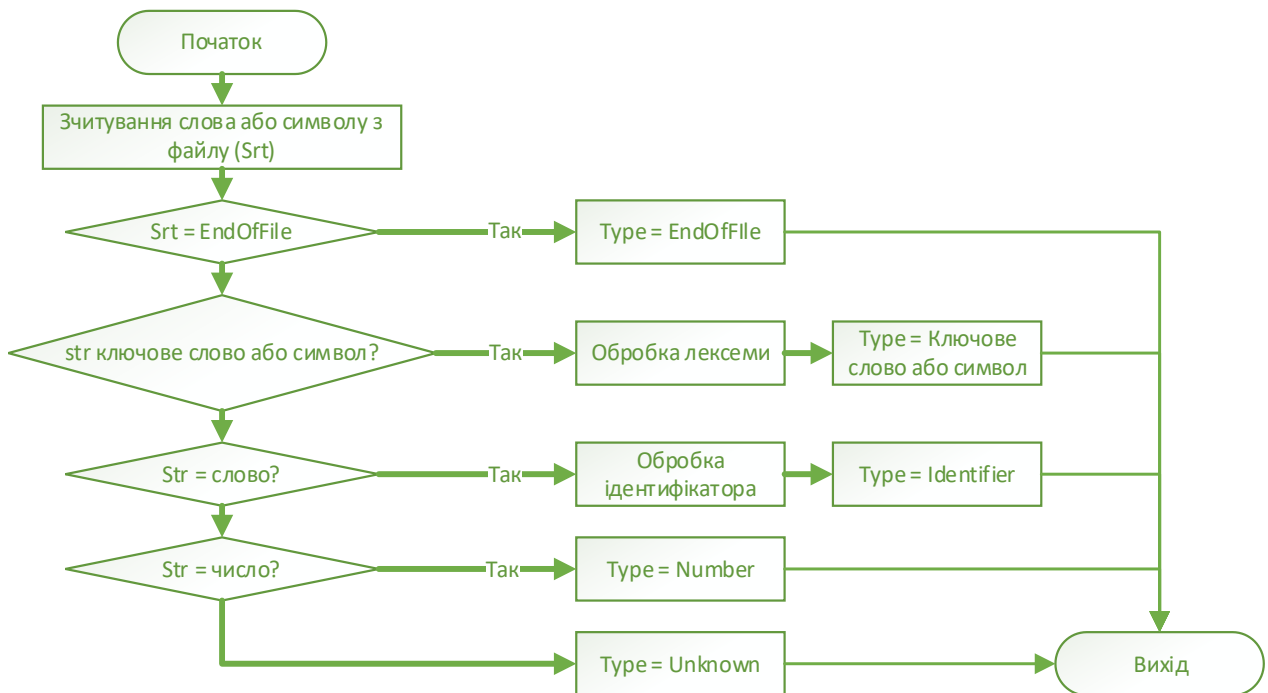


Рис. 3.1 Блок-схема роботи лексичного аналізатора

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 `Parser()`. Вона зчитує з файлу його вміст та кожну лексему порівнює з зарезервованими словами якщо є співпадіння то присвоює лексемі відповідний тип або значення, якщо це числова константа.

При виділенні лексеми вона розпізнається та записується у таблицю за допомогою відповідного типу лексеми, що є унікальним для кожної лексеми із усього можливого їх набору. Це дає можливість наступним фазам компіляції звертатись до лексеми не як до послідовності символів, а як до унікального типу лексеми, що значно спрощує роботу синтаксичного аналізатора: легко перевіряти належність лексеми до відповідної синтаксичної конструкції та є

можливість легкого перегляду програми, як вгору, так і вниз, від поточної позиції аналізу. Також в таблиці лексем ведуться записи, щодо рядка відповідної лексеми – для місця помилки – та додаткова інформація.

При лексичному аналізі виявляються і відзначаються лексичні помилки (наприклад, недопустимі символи і неправильні ідентифікатори). Лексична фаза відкидає також коментарі, оскільки вони не мають ніякого впливу на виконання програми, отже й на синтаксичний розбір та генерацію коду.

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

3.4. Розробка синтаксичного та семантичного аналізатора

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

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

Аналізатор працює за принципом рекурсивного спуску, де кожне правило граматики реалізується окремою функцією.

Основні етапи роботи аналізатора:

1. **Ініціалізація:** Виклик функції `Parser()`, яка починає аналіз програми.
2. **Аналіз програми:** Функція `program()` аналізує основну структуру програми, включаючи оголошення змінних та тіло програми.
3. **Аналіз операторів:** Функція `statement()` визначає тип оператора (ввід, вивід, умовний оператор, присвоєння тощо) та викликає відповідну функцію для його аналізу.
4. **Аналіз виразів:** Функції `arithmetic_expression()`, `term()`, `factor()` аналізують арифметичні вирази, включаючи операції додавання, віднімання, множення та ділення.
5. **Аналіз умов:** Функції `logical_expression()`, `and_expression()`, `comparison()` аналізують логічні вирази та операції порівняння.

Основні функції

- **`program()`:** Аналізує основну структуру програми.
- **`variable_declaration()`:** Аналізує оголошення змінних.
- **`variable_list()`:** Аналізує список змінних.
- **`program_body()`:** Аналізує тіло програми.
- **`statement()`:** Визначає тип оператора та викликає відповідну функцію для його аналізу.
- **`assignment()`:** Аналізує оператор присвоєння.

- **arithmetic_expression()**: Аналізує арифметичний вираз.
- **term()**: Аналізує доданок у виразі.
- **factor()**: Аналізує множник у виразі.
- **input()**: Аналізує оператор вводу.
- **output()**: Аналізує оператор виводу.
- **conditional()**: Аналізує умовний оператор.
- **goto_statement()**: Аналізує оператор переходу.
- **label_statement()**: Аналізує мітку.
- **for_to_do()**: Аналізує цикл for з інкрементом.
- **for_downto_do()**: Аналізує цикл for з декрементом.
- **while_statement()**: Аналізує цикл while.
- **repeat_until()**: Аналізує цикл repeat until.
- **logical_expression()**: Аналізує логічний вираз.
- **and_expression()**: Аналізує логічний вираз з операцією AND.
- **comparison()**: Аналізує операції порівняння.
- **compound_statement()**: Аналізує складений оператор.

Цей аналізатор забезпечує перевірку синтаксичної коректності програми та виявлення синтаксичних помилок. Якщо виявляється помилка, аналізатор виводить повідомлення про помилку та завершує роботу.

3.4.2. Розробка граф-схеми алгоритму

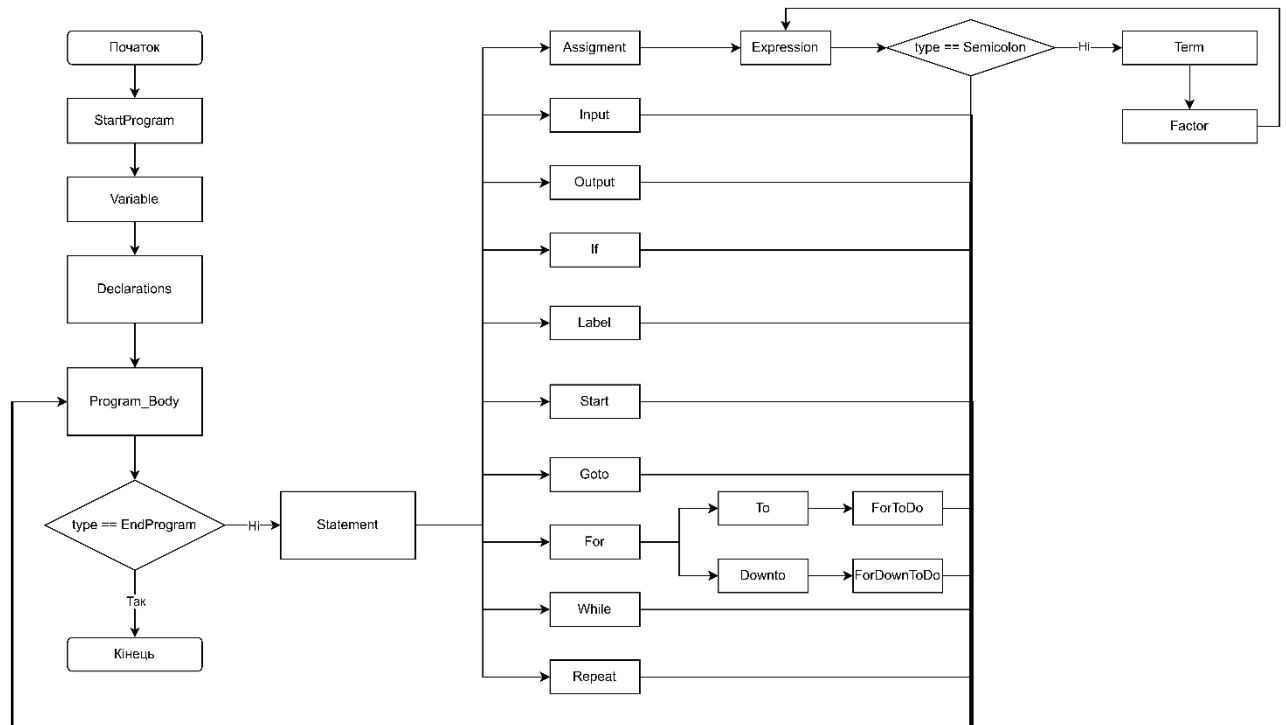


Рис. 3.2 Граф-схема роботи синтаксичного аналізатора

3.5. Розробка генератора коду

Синтаксичне дерево в чистому вигляді несе тільки інформацію про структуру програми. Насправді в процесі генерації коду потрібна також інформація про змінні, операції, 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 операції.

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

В даному курсовому проекті генерація коду реалізується як окремий етап. Можливість його виконання є лише за умови, що попередньо успішно виконався етап синтаксичного аналізу. І використовує результат виконання попереднього аналізу, тобто два файли: перший містить згенерований С код відповідно операторам які були в програмі, другий файл містить таблицю змінних.

3.5.1. Розробка граф-схеми алгоритму

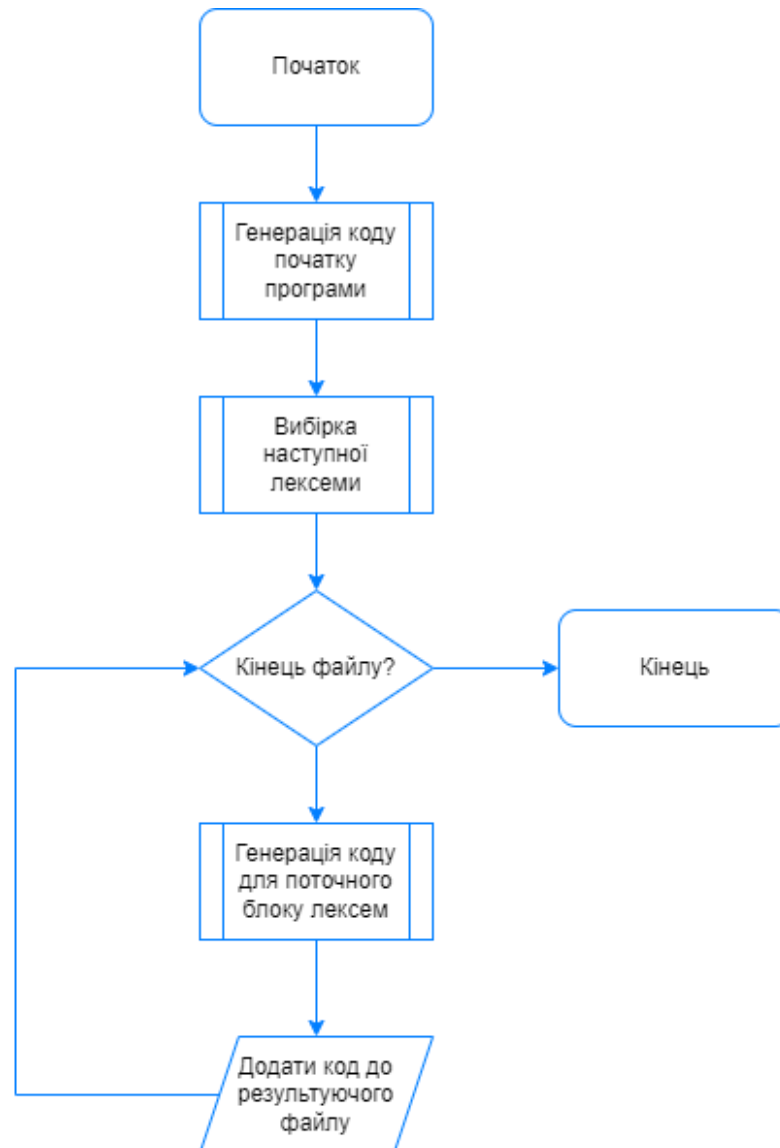


Рис. 3.3 Блок схема генератора коду

3.5.2. Опис програми реалізації генератора коду

У компілятора, реалізованого в даному курсовому проєкті, вихідна мова - програма на мові C. Ця програма записується у файл, що має таку ж саму назву, як і файл з вхідним текстом, але розширення “.c”. 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 заголовки, необхідні для програми на C, та визначається основна функція `main()`. Далі виконується аналіз коду та визначаються змінні, які використовуються.

Проаналізувавши змінні, які є у програмі, генератор формує секцію оголошення змінних для програми на C. Для цього з таблиці лексем вибирається ім'я змінної (типи змінних відповідають типам у C, наприклад `int`), та записується її початкове значення, якщо воно задано.

Аналіз наявних операторів необхідний у зв'язку з тим, що введення/виведення, виконання арифметичних та логічних операцій виконуються як окремі конструкції, 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 функції `printf`, яка формує вихідний текст. Якщо це арифметична операція, то у вихідний файл записується вираз, що відповідає правилам C, із врахуванням пріоритетів операцій.

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

В кінці своєї роботи генератор формує завершення програми на C, додаючи повернення значення 0 з основної функції.

4. Опис програми

Дана програма написана мовою C++ з використанням визначень нових типів та перелічень:

```
enum TypeOfTokens
{
    Mainprogram,
    StartProgram,
    Variable,
    Type,
    EndProgram,
    Input,
    Output,

    If,
    Else,

    Goto,
    Label,

    For,
    To,
    DownTo,
    Do,

    While,
    Exit,
    Continue,
    End,

    Repeat,
    Until,

    Identifier,
    Number,
    Assign,
    Add,
    Sub,
    Mul,
    Div,
    Mod,
    Equality,
    NotEquality,
    Greate,
    Less,
    Not,
    And,
    Or,
    LBraket,
    RBraket,
```

```

        Semicolon,
        Colon,
        Comma,
        Unknown
    };

    // структура для зберігання інформації про лексему
    struct Token
    {
        char name[16];        // ім'я лексеми
        int value;            // значення лексеми
        int line;            // номер рядка
        TokenType type;      // тип лексеми
    };

    // структура для зберігання інформації про ідентифікатор
    struct Id
    {
        char name[16];
    };

    // перерахування, яке описує стани лексичного аналізатора
    enum States
    {
        Start,              // початок виділення чергової лексеми
        Finish,             // кінець виділення чергової лексеми
        Letter,             // опрацювання слів (ключові слова і ідентифікатори)
        Digit,              // опрацювання цифри
        Separators,         // видалення пробілів, символів табуляції і переходу на
        новий рядок
        Another,            // опрацювання інших символів
        EndOfFile,          // кінець файлу
        SComment,           // початок коментаря
        Comment             // видалення коментаря
    };
};

```

Спочатку вхідна програма за допомогою функції `unsigned int GetTokens(FILE* F, Token TokenTable[])` розбивається на відповідні токени для запису у таблицю та подальше їх використання в процесі синтаксичного аналізу та генерації коду.

Далі відбувається синтаксичний аналіз вхідної програми за допомогою функції `void Parser()`. Всі правила запису як різноманітних операцій так і програми в цілому відбувається за нотатками Бекуса-Наура, за допомогою яких можна легко описати синтаксис всіх операцій.

Нище наведено опис структури програми за допомогою нотаток Бекуса-Наура.

```
void program()
```

```

{
    match(Mainprogram);
    match(StartProgram);
    match(Variable);
    variable_declaration();
    match(Semicolon);
    program_body();
    match(EndProgram);
}

```

Наступним етапом є генерація С коду. Алгоритм генерації працює за принципом синтаксичного аналізу але при вибірці певної лексеми або операції генерує відповідний С код який записується у вихідний файл.

Нище наведено генерацію С коду на прикладі операції присвоєння.

```

void assignment(FILE* outFile)
{
    fprintf(outFile, "    ");
    fprintf(outFile, TokenTable[pos++].name);
    fprintf(outFile, " = ");
    pos++;
    arithmetic_expression(outFile);
    pos++;
    fprintf(outFile, ";\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.

4.1. Опис інтерфейсу та інструкція користувачеві

Вхідним файлом для даної програми є звичайний текстовий файл з розширенням b03. 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_b03.exe <ім'я програми>.b03"

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>.c.

5. Відлагодження та тестування програми

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

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

5.1. Виявлення лексичних та синтаксичних помилок

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

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

Приклад виявлення:

Текст програми з помилками

```
!!Prog1!!
Program prog1;
Var Int16 AAAAAA,BBB BBB,XXXXXX,YYYYYY;
Begin
Get AAAAAA;
Get BBBBBB
Put AAAAAA + BBBBBB;
Put AAAAAA - BBBBBB;
Put AAAAAA Mul BBBBBB;
Put AAAAAA Div BBBBBB;
Put AAAAAA Mod BBBBBB;

XXXXXX==>(AAAAAA - BBBBBB) Mul 10 + (AAAAAA + BBBBBB) Div 10;
YYYYYY==>XXXXXX + (XXXXXX Mod 10);
Put XXXXXX;
Put YYYYYY;
```

End

Текст файлу з повідомленнями про помилки

Lexical Error: line 3, lexem BBB is Unknown

Lexical Error: line 3, lexem BBB is Unknown

Syntax error in line 3 : another type of lexeme was expected.

Syntax error: type Unknown

Expected Type: Identifier

5.2. Виявлення семантичних помилок

Суттю виявлення семантичних помилок є перевірка числових констант на відповідність типу Int16, тобто знаковому цілому числу з відповідним діапазоном значень і перевірку на коректність використання змінних Int16 у цілочисельних і логічних виразах.

5.3. Загальна перевірка коректності роботи транслятора

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

Текст коректної програми

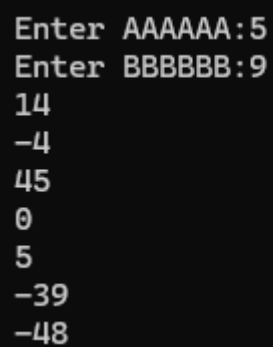
```
!!Prog1!!
Program prog1;
Var Int16 AAAAAA,BBBBBB,XXXXXX,YYYYYY;
Begin
Get AAAAAA;
Get BBBBBB;
Put AAAAAA + BBBBBB;
Put AAAAAA - BBBBBB;
Put AAAAAA Mul BBBBBB;
Put AAAAAA Div BBBBBB;
Put AAAAAA Mod BBBBBB;

XXXXXX==>(AAAAAA - BBBBBB) Mul 10 + (AAAAAA + BBBBBB) Div 10;
YYYYYY==>XXXXXX + (XXXXXX Mod 10);
Put XXXXXX;
Put YYYYYY;
End
```

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

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

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

A screenshot of a terminal window with a black background and white text. The text shows two input prompts followed by a series of numerical outputs.

```
Enter AAAAAA:5
Enter BBBBBB:9
14
-4
45
0
5
-39
-48
```

Рис. 5.1 Результат виконання коректної програми

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

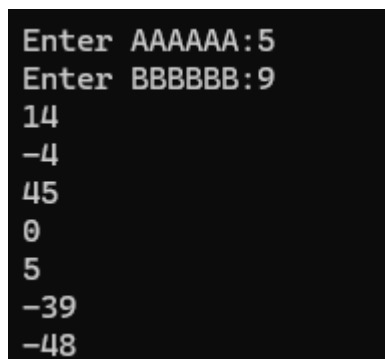
5.4. Тестова програма №1

Текст програми

```
!!Prog1!!
Program prog1;
Var Int16 AAAAAA,BBBBBB,XXXXXX,YYYYYY;
Begin
Get AAAAAA;
Get BBBBBB;
Put AAAAAA + BBBBBB;
Put AAAAAA - BBBBBB;
Put AAAAAA Mul BBBBBB;
Put AAAAAA Div BBBBBB;
Put AAAAAA Mod BBBBBB;

XXXXXX==>(AAAAAA - BBBBBB) Mul 10 + (AAAAAA + BBBBBB) Div 10;
YYYYYY==>XXXXXX + (XXXXXX Mod 10);
Put XXXXXX;
Put YYYYYY;
End
```

Результат виконання



```
Enter AAAAAA:5
Enter BBBBBB:9
14
-4
45
0
5
-39
-48
```

Рис. 5.2 Результат виконання тестової програми №1

5.5. Тестова програма №2

Текст програми

```
!!Prog2!!
Program prog2;
Var Int16 AAAAAA,BBBBBB,CCCCCC;
Begin
  Get AAAAAA;
  GetBBBBB;
  Get CCCCCC;
  If(AAAAAA >> BBBBBB)
  Begin
    If(AAAAAA >> CCCCCC)
    Begin
      Goto Abigger;
    End
    Else
    Begin
      Put CCCCCC;
      Goto Outofif;
    Abigger:
      Put AAAAAA;
      Goto Outofif;
    End
  End
End
If(BBBBBB << CCCCCC)
Begin
  Put CCCCCC;
End
Else
Begin
  PutBBBBB;
End
Outofif:

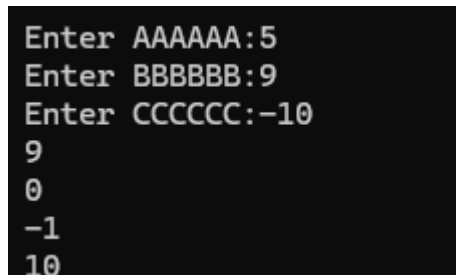
If((AAAAAA Eg BBBBBB) And (AAAAAA Eg CCCCCC) And (BBBBBB Eg CCCCCC))
Begin
  Put 1;
End
Else
Begin
  Put 0;
End
If((AAAAAA << 0) Or (BBBBBB << 0) Or (CCCCCC << 0))
Begin
  Put -1;
End
```

```

Else
Begin
    Put 0;
End
If(!(AAAAAA << (BBBBBB + CCCCCC)))
Begin
    Put(10);
End
Else
Begin
    Put(0);
End
End

```

Результат виконання



```

Enter AAAAAA:5
Enter BBBBBB:9
Enter CCCCCC:-10
9
0
-1
10

```

Рис. 5.3 Результат виконання тестової програми №2

5.6. Тестова програма №3

Текст програми

```

!!Prog3!!
Program prog3;
Var Int16 AAAAAA,AAAAA2,BBBBBB,XXXXXX,CCCCC1,CCCCC2;
Begin
Get AAAAAA;
Get BBBBBB;
For AAAAA2==>AAAAAA To BBBBBB Do
    Put AAAAA2 Mul AAAAA2;

For AAAAA2==>BBBBBB To AAAAAA Do
    Put AAAAA2 Mul AAAAA2;

XXXXXX==>0;
CCCCC1==>0;
While CCCCC1 << AAAAAA
Begin
    CCCCC2==>0;
    While CCCCC2 << BBBBBB
    Begin
        XXXXXX==>XXXXXX + 1;
        CCCCC2==>CCCCC2 + 1;
    End
End

```

```

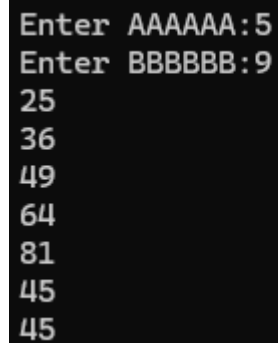
        End
    End While
CCCCC1==>CCCCC1 + 1;
End
End While
Put XXXXXX;

XXXXXX==>0;
CCCCC1==>1;
Repeat
Begin
    CCCCC2==>1;
    Repeat
    Begin
        XXXXXX==>XXXXXX + 1;
        CCCCC2==>CCCCC2 + 1;
    End
    Until !(CCCCC2 >> BBBBBB)
    CCCCC1==>CCCCC1 + 1;
End
Until !(CCCCC1 >> AAAAAA)
Put XXXXXX;

End

```

Результат виконання



```

Enter AAAAAA:5
Enter BBBBBB:9
25
36
49
64
81
45
45

```

Рис. 5.4 Результат виконання тестової програми №3

Висновки

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування b03, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створено компілятор мови програмування b03, 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 b03. Вихідним кодом генератора є програма на мові C.

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 b03 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. C Programming Language Tutorial - GeeksforGeeks
URL: [C Programming Language Tutorial - 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 С)

Prog1.c

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

int main()
{
    int16_t AAAAAA, BBBBBB, XXXXXX, YYYYYY;
    printf("Enter AAAAAA:");
    scanf("%hd", &AAAAAA);
    printf("Enter BBBBBB:");
    scanf("%hd", &BBBBBB);
    printf("%d\n", AAAAAA + BBBBBB);
    printf("%d\n", AAAAAA - BBBBBB);
    printf("%d\n", AAAAAA * BBBBBB);
    printf("%d\n", AAAAAA / BBBBBB);
    printf("%d\n", AAAAAA % BBBBBB);
    XXXXXX = (AAAAAA - BBBBBB) * 10 + (AAAAAA + BBBBBB) / 10;
    YYYYYY = XXXXXX + (XXXXXX % 10);
    printf("%d\n", XXXXXX);
    printf("%d\n", YYYYYY);
    system("pause");
    return 0;
}
```

Prog2.c

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

int main()
{
    int16_t AAAAAA, BBBBBB, CCCCCC;
    printf("Enter AAAAAA:");
    scanf("%hd", &AAAAAA);
    printf("Enter BBBBBB:");
    scanf("%hd", &BBBBBB);
    printf("Enter CCCCCC:");
    scanf("%hd", &CCCCCC);
    if ((AAAAAA > BBBBBB))
    {
        if ((AAAAAA > CCCCCC))
        {
            goto Abigger;
        }
    }
    else
    {
        printf("%d\n", CCCCCC);
    }
}
```

```

    goto Outofif;
Abigger:
    printf("%d\n", AAAAAA);
    goto Outofif;
}
}
if ((BBBBBB < CCCCCC))
{
    printf("%d\n", CCCCCC);
}
else
{
    printf("%d\n", BBBBBB);
}
Outofif:
if (((AAAAAA == BBBBBB) && (AAAAAA == CCCCCC) && (BBBBBB == CCCCCC)))
{
    printf("%d\n", 1);
}
else
{
    printf("%d\n", 0);
}
if (((AAAAAA < 0) || (BBBBBB < 0) || (CCCCCC < 0)))
{
    printf("%d\n", -1);
}
else
{
    printf("%d\n", 0);
}
if (!(AAAAAA < (BBBBBB + CCCCCC)))
{
    printf("%d\n", (10));
}
else
{
    printf("%d\n", (0));
}
system("pause");
return 0;
}

```

Prog3.c

```

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

```

```

int main()
{
    int16_t AAAAAA, AAAAA2, BBBBBB, XXXXXX, CCCCC1, CCCCC2;

```

```

printf("Enter AAAAAA:");
scanf("%hd", &AAAAAA);
printf("Enter BBBBBB:");
scanf("%hd", &BBBBBB);
for (int16_t AAAAA2 = AAAAA; AAAAA2 <= BBBBB; AAAAA2++)
printf("%d\n", AAAAA2 * AAAAA2);
for (int16_t AAAAA2 = BBBBB; AAAAA2 <= AAAAA; AAAAA2++)
printf("%d\n", AAAAA2 * AAAAA2);
XXXXXX = 0;
CCCCC1 = 0;
while (CCCCC1 < AAAAA)
{
{
CCCCC2 = 0;
while (CCCCC2 < BBBBB)
{
{
XXXXXX = XXXXXX + 1;
CCCCC2 = CCCCC2 + 1;
}
}
CCCCC1 = CCCCC1 + 1;
}
}
printf("%d\n", XXXXXX);
XXXXXX = 0;
CCCCC1 = 1;
do
{
CCCCC2 = 1;
do
{
XXXXXX = XXXXXX + 1;
CCCCC2 = CCCCC2 + 1;
}
while (!(CCCCC2 > BBBBB));
CCCCC1 = CCCCC1 + 1;
}
while (!(CCCCC1 > AAAAA));
printf("%d\n", XXXXXX);
system("pause");
return 0;
}

```