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 №20**

**Виконав:** ст. гр. КІ-309

Стадник М. В.

**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 C. 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](#_Toc153318762)

[Завдання до курсового проекту 4](#_Toc153318763)

[Вступ 6](#_Toc153318764)

[1. Огляд методів та способів проектування трансляторів 7](#_Toc153318765)

[2. Формальний опис вхідної мови програмування 10](#_Toc153318766)

[2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура 10](#_Toc153318767)

[2.2. Опис термінальних символів та ключових слів 12](#_Toc153318768)

[3. Розробка транслятора вхідної мови програмування 14](#_Toc153318769)

[3.1. Вибір технології програмування 14](#_Toc153318770)

[3.2. Проектування таблиць транслятора 15](#_Toc153318771)

[3.3. Розробка лексичного аналізатора 17](#_Toc153318772)

[3.3.1. Розробка блок-схеми алгоритму 18](#_Toc153318773)

[3.3.2. Опис програми реалізації лексичного аналізатора 18](#_Toc153318774)

[3.4. Розробка синтаксичного та семантичного аналізатора 20](#_Toc153318775)

[3.4.1. Опис програми реалізації синтаксичного та семантичного аналізатора 21](#_Toc153318776)

[3.4.2. Розробка граф-схеми алгоритму 21](#_Toc153318777)

[3.5. Розробка генератора коду 22](#_Toc153318778)

[3.5.1. Розробка граф-схеми алгоритму 23](#_Toc153318779)

[3.5.2. Опис програми реалізації генератора коду 24](#_Toc153318780)

[4. Опис програми 25](#_Toc153318781)

[4.1. Опис інтерфейсу та інструкція користувачеві 28](#_Toc153318782)

[5. Відлагодження та тестування програми 29](#_Toc153318783)

[5.1. Виявлення лексичних та синтаксичних помилок 29](#_Toc153318784)

[5.2. Виявлення семантичних помилок 30](#_Toc153318785)

[5.3. Загальна перевірка коректності роботи транслятора 30](#_Toc153318786)

[5.4. Тестова програма №1 32](#_Toc153318787)

[5.5. Тестова програма №2 33](#_Toc153318788)

[5.6. Тестова програма №3 34](#_Toc153318789)

[Висновки 36](#_Toc153318790)

[Список використаної літератури 37](#_Toc153318791)

[Додатки 38](#_Toc153318792)

Завдання до курсового проекту

**Варіант 20**

Завдання на курсовий проект

1. Цільова мова транслятора – мова програмування С.

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 .s20

Опис вхідної мови програмування:

* Тип даних: Integer\_4
* Блок тіла програми: Name <name>; Body Data…; 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: Up16 перший символ \_
* Операції арифметичні: +, -, Mul, Div, Mod
* Операції порівняння: =, <>, Ge, Le
* 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 відміну 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 C. Наведене визначення застосовне до різноманітних транслюючих 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 C. 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 C. 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 C, 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.

1. Формальний опис вхідної мови програмування
   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 = "**Name**", identifier , ";", "**Body**", varsBlok, ";", operators, "**End**";

varsBlok = "**Data**", "**Integer\_4**", identifier, [{ commaAndIdentifier }];

identifier = “\_”, up\_letter, { up\_letter | number } {15};

commaAndIdentifier = ",", identifier;

codeBlok = "**Body**", write | read | assignment | ifStatement | goto\_statement | labelRule | forToOrDownToDoRule | while | repeatUntil, "**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, ")", "**Body**", 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 = "**=**" | "**<>**" | "**Le**" | "**Ge**";

comment = "**LComment**", text;

LComment = "**//**";

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";

* 1. Опис термінальних символів та ключових слів

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

|  |  |
| --- | --- |
| Термінальний символ або ключове слово | Значення |
| Name | Початок програми |
| Body | Початок тексту програми |
| Data | 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 |
| = | Оператор перевірки на рівність |
| <> | Оператор перевірки на нерівність |
| Le | Оператор перевірки чи менше |
| Ge | Оператор перевірки чи більше |
| ! | Оператор логічного заперечення |
| And | Оператор кон’юнкції |
| Or | Оператор диз’юнкції |
| Integer\_4 | 32‑ох розрядні знакові цілі |
| //… | Коментар |
| , | Розділювач |
| ; | Ознака кінця оператора |
| ( | Відкриваюча дужка |
| ) | Закриваюча дужка |

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

1. Розробка транслятора вхідної мови програмування
   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: Х – початкова, Y – об`єктна та Z – інструментальна. Транслятор перекладає програми мовою Х 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

* 1. Проектування таблиць транслятора

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

1. Таблиця лексем з елементами, які мають таку структуру:

struct Token

{

char name[32]; // ім'я лексеми

int value; // значення лексеми (для цілих констант)

int line; // номер рядка

TypeOfTokens type; // тип лексеми

};

1. Таблиця лексичних класів

enum TypeOfTokens

{

Mainprogram,

StartProgram,

Variable,

Type,

EndProgram,

Input,

Output,

If,

Then,

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 символі та ключових слів

|  |  |
| --- | --- |
| **Токен** | **Значення** |
| Program | Name |
| Start | Body |
| Vars | Data |
| End | End |
| VarType | Integer\_4 |
| 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 | = |
| NotEqual | <> |
| Less | Le |
| Greate | Ge |
| Not | ! |
| And | And |
| Or | Or |
| Identifier |  |
| Number |  |
| Unknown |  |
| Comma | , |
| Semicolon | ; |
| LBraket | ( |
| RBraket | ) |
| LComment | // |
| Comment |  |

* 1. 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;
  + 1. Розробка блок-схеми алгоритму



Рис. 3.1 Блок-схема роботи лексичного аналізатора

* + 1. 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. Вхiдна програма проглядається послідовно з початку до кінця. Базовi елементи, або лексичнi одиницi, роздiляються пробілами, знаками операцiй i спецiальними символами (новий рядок, знак табуляції), i таким чином видiляються та розпізнаються iдентифiкатори, лiтерали i термiнальнi символи (операцiї, 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 зарезервованою словами якщо є співпадіння то присвоює лексемі відповідний тип або значення, якщо це числова константа.

При виділенні лексеми вона розпізнається та записується у таблиця за допомогою відповідного типу лексеми, що є унікальним для кожної лексеми із усього можливого їх набору. Це дає можливість наступним фазам компiляції 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 аналiзі виявляються i вiдзначаються лексичнi помилки (наприклад, недопустимi символи i неправильнi iдентифiкатори). Лексична фаза вiдкидає також коментарi, оскiльки вони не мають нiякого 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.

* 1. 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-граматик, 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.

* + 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.

* + 1. Розробка граф-схеми алгоритму



Рис. 3.2 Граф-схема роботи синтаксичного аналізатора

* 1. 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. 3азвичай входом для генератора коду служить проміжна форма представлення програми, а на виході може з’являтися об’єктний код або модуль завантаження.

Генератор C 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 C код відповідно операторам які були в програмі, другий файл містить таблицю змінних.

* + 1. Розробка граф-схеми алгоритму



Рис. 3.3 Блок схема генератора коду

* + 1. Опис програми реалізації генератора коду

У компілятора, реалізованого в даному курсовому проекті, вихідна мова - програма на мові С. 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 С, 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 С. Для цього з таблиці лексем вибирається ім’я змінної (типи змінних відповідають типам у С, 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 С, із врахуванням пріоритетів операцій.

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

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

1. Опис програми

Дана програма написана мовою С++ з використанням визначень нових типів та перечислень:

enum TypeOfTokens

{

Mainprogram,

StartProgram,

Variable,

Type,

EndProgram,

Input,

Output,

If,

Then,

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

TypeOfTokens 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 анадізувати програми великого обсягу.

* 1. Опис інтерфейсу та інструкція користувачеві

Вхідним файлом для даної програми є звичайний текстовий файл з розширенням s20. 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\_s20.exe <ім’я програми>.s20"

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>.с.

1. 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 перевірці коректності коду, що генерується, коректності знаходження помилок та розбивки на лексеми.

* 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

Name prog1;

Body

Data Integer\_4 \_AAAAAAAAAAAAAA,\_BBBBBBBBBBBBBBB,\_XXXXXXXXXXXXXXX,\_YYYYYYYYYYYYYYY;

Get \_AAAAAAA AAAAAAAA;

Get \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA + \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA - \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA Mul \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA Div \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA Mod \_BBBBBBBBBBBBBBB;

\_XXXXXXXXXXXXXXX<==(\_AAAAAAAAAAAAAAA - \_BBBBBBBBBBBBBBB) Mul 10 + (\_AAAAAAAAAAAAAAA + \_BBBBBBBBBBBBBBB) Div 10;

\_YYYYYYYYYYYYYYY<==\_XXXXXXXXXXXXXXX + (\_XXXXXXXXXXXXXXX Mod 10);

Put \_XXXXXXXXXXXXXXX;

Put \_YYYYYYYYYYYYYYY;

End

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

Lexical Error: line 4, lexem \_AAAAAAAAAAAAAA is Unknown

Lexical Error: line 5, lexem \_AAAAAAA is Unknown

Lexical Error: line 5, lexem AAAAAAAA is Unknown

Syntax error in line 4 : another type of lexeme was expected.

Syntax error: type Unknown

Expected Type: Identifier

* 1. Виявлення семантичних помилок

Суттю виявлення семантичних помилок є перевірка числових констант на відповідність типу Integer\_4, тобто знаковому цілому числу з відповідним діапазоном значень і перевірку на коректність використання змінних Integer\_4 у цілочисельних і логічних виразах.

* 1. 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

Name prog1;

Body

Data Integer\_4 \_AAAAAAAAAAAAAAA,\_BBBBBBBBBBBBBBB,\_XXXXXXXXXXXXXXX,\_YYYYYYYYYYYYYYY;

Get \_AAAAAAAAAAAAAAA;

Get \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA + \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA - \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA Mul \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA Div \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA Mod \_BBBBBBBBBBBBBBB;

\_XXXXXXXXXXXXXXX<==(\_AAAAAAAAAAAAAAA - \_BBBBBBBBBBBBBBB) Mul 10 + (\_AAAAAAAAAAAAAAA + \_BBBBBBBBBBBBBBB) Div 10;

\_YYYYYYYYYYYYYYY<==\_XXXXXXXXXXXXXXX + (\_XXXXXXXXXXXXXXX Mod 10);

Put \_XXXXXXXXXXXXXXX;

Put \_YYYYYYYYYYYYYYY;

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:

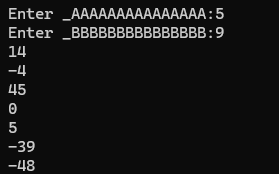


Рис. 5.1 Результат виконання коректної програми

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

* 1. Тестова програма №1

***Текст програми***

//Prog1

Name prog1;

Body

Data Integer\_4 \_AAAAAAAAAAAAAAA,\_BBBBBBBBBBBBBBB,\_XXXXXXXXXXXXXXX,\_YYYYYYYYYYYYYYY;

Get \_AAAAAAAAAAAAAAA;

Get \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA + \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA - \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA Mul \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA Div \_BBBBBBBBBBBBBBB;

Put \_AAAAAAAAAAAAAAA Mod \_BBBBBBBBBBBBBBB;

\_XXXXXXXXXXXXXXX<==(\_AAAAAAAAAAAAAAA - \_BBBBBBBBBBBBBBB) Mul 10 + (\_AAAAAAAAAAAAAAA + \_BBBBBBBBBBBBBBB) Div 10;

\_YYYYYYYYYYYYYYY<==\_XXXXXXXXXXXXXXX + (\_XXXXXXXXXXXXXXX Mod 10);

Put \_XXXXXXXXXXXXXXX;

Put \_YYYYYYYYYYYYYYY;

End

***Результат виконання***

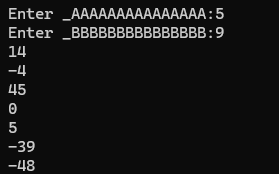


Рис. 5.2 Результат виконання тестової програми №1

* 1. Тестова програма №2

***Текст програми***

//Prog2

Name prog2;

Body

Data Integer\_4 \_AAAAAAAAAAAAAAA,\_BBBBBBBBBBBBBBB,\_CCCCCCCCCCCCCCC;

Get \_AAAAAAAAAAAAAAA;

Get \_BBBBBBBBBBBBBBB;

Get \_CCCCCCCCCCCCCCC;

If(\_AAAAAAAAAAAAAAA Ge \_BBBBBBBBBBBBBBB)

Body

If(\_AAAAAAAAAAAAAAA Ge \_CCCCCCCCCCCCCCC)

Body

Goto Abigger;

End

Else

Body

Put \_CCCCCCCCCCCCCCC;

Goto Outofif;

Abigger:

Put \_AAAAAAAAAAAAAAA;

Goto Outofif;

End

End

If(\_BBBBBBBBBBBBBBB Le \_CCCCCCCCCCCCCCC)

Body

Put \_CCCCCCCCCCCCCCC;

End

Else

Body

Put \_BBBBBBBBBBBBBBB;

End

Outofif:

If((\_AAAAAAAAAAAAAAA = \_BBBBBBBBBBBBBBB) And (\_AAAAAAAAAAAAAAA = \_CCCCCCCCCCCCCCC) And (\_BBBBBBBBBBBBBBB = \_CCCCCCCCCCCCCCC))

Body

Put 1;

End

Else

Body

Put 0;

End

If((\_AAAAAAAAAAAAAAA Le 0) Or (\_BBBBBBBBBBBBBBB Le 0) Or (\_CCCCCCCCCCCCCCC Le 0))

Body

Put -1;

End

Else

Body

Put 0;

End

If(!(\_AAAAAAAAAAAAAAA Le (\_BBBBBBBBBBBBBBB + \_CCCCCCCCCCCCCCC)))

Body

Put(10);

End

Else

Body

Put(0);

End

End

***Результат виконання***

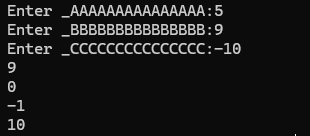


Рис. 5.3 Результат виконання тестової програми №2

* 1. Тестова програма №3

***Текст програми***

//Prog3

Name prog3;

Body

Data Integer\_4 \_AAAAAAAAAAAAAAA,\_DDDDDDDDDDDDDDD,\_BBBBBBBBBBBBBBB,\_XXXXXXXXXXXXXXX,\_CCCCCCCCCCCCCCC,\_ZZZZZZZZZZZZZZZ;

Get \_AAAAAAAAAAAAAAA;

Get \_BBBBBBBBBBBBBBB;

For \_DDDDDDDDDDDDDDD<==\_AAAAAAAAAAAAAAA To \_BBBBBBBBBBBBBBB Do

Put \_DDDDDDDDDDDDDDD Mul \_DDDDDDDDDDDDDDD;

For \_DDDDDDDDDDDDDDD<==\_BBBBBBBBBBBBBBB To \_AAAAAAAAAAAAAAA Do

Put \_DDDDDDDDDDDDDDD Mul \_DDDDDDDDDDDDDDD;

\_XXXXXXXXXXXXXXX<==0;

\_CCCCCCCCCCCCCCC<==0;

While \_CCCCCCCCCCCCCCC Le \_AAAAAAAAAAAAAAA

Body

\_ZZZZZZZZZZZZZZZ<==0;

While \_ZZZZZZZZZZZZZZZ Le \_BBBBBBBBBBBBBBB

Body

\_XXXXXXXXXXXXXXX<==\_XXXXXXXXXXXXXXX + 1;

\_ZZZZZZZZZZZZZZZ<==\_ZZZZZZZZZZZZZZZ + 1;

End

End While

\_CCCCCCCCCCCCCCC<==\_CCCCCCCCCCCCCCC + 1;

End

End While

Put \_XXXXXXXXXXXXXXX;

\_XXXXXXXXXXXXXXX<==0;

\_CCCCCCCCCCCCCCC<==1;

Repeat

Body

\_ZZZZZZZZZZZZZZZ<==1;

Repeat

Body

\_XXXXXXXXXXXXXXX<==\_XXXXXXXXXXXXXXX+1;

\_ZZZZZZZZZZZZZZZ<==\_ZZZZZZZZZZZZZZZ+1;

End

Until !(\_ZZZZZZZZZZZZZZZ Ge \_BBBBBBBBBBBBBBB)

\_CCCCCCCCCCCCCCC<==\_CCCCCCCCCCCCCCC+1;

End

Until !(\_CCCCCCCCCCCCCCC Ge \_AAAAAAAAAAAAAAA)

Put \_XXXXXXXXXXXXXXX;

End

***Результат виконання***

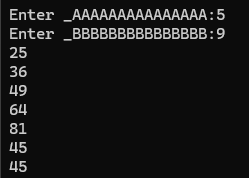


Рис. 5.4 Результат виконання тестової програми №3

Висновки

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування s20, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створено компілятор мови програмування s20, 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 s20. 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 s20 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](https://www.geeksforgeeks.org/c-programming-language/?ref=header_outind)

1. Error Handling in Compiler Design

URL: [Error Handling in Compiler Design - GeeksforGeeks](https://www.geeksforgeeks.org/error-handling-compiler-design/?ref=lbp)

1. Symbol Table in Compiler

URL: [Symbol Table in Compiler - GeeksforGeeks](https://www.geeksforgeeks.org/symbol-table-compiler/?ref=lbp)

1. Вікіпедія

URL: [Wikipedia](https://www.wikipedia.org/)

1. Stack Overflow

URL: [Stack Overflow - Where Developers Learn, Share, & Build Careers](https://stackoverflow.com/)

Додатки

**Додаток А (Код на мові C)**

**Prog1.c**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_AAAAAAAAAAAAAAA, \_BBBBBBBBBBBBBBB, \_XXXXXXXXXXXXXXX, \_YYYYYYYYYYYYYYY;

printf("Enter \_AAAAAAAAAAAAAAA:");

scanf("%d", &\_AAAAAAAAAAAAAAA);

printf("Enter \_BBBBBBBBBBBBBBB:");

scanf("%d", &\_BBBBBBBBBBBBBBB);

printf("%d\n", \_AAAAAAAAAAAAAAA + \_BBBBBBBBBBBBBBB);

printf("%d\n", \_AAAAAAAAAAAAAAA - \_BBBBBBBBBBBBBBB);

printf("%d\n", \_AAAAAAAAAAAAAAA \* \_BBBBBBBBBBBBBBB);

printf("%d\n", \_AAAAAAAAAAAAAAA / \_BBBBBBBBBBBBBBB);

printf("%d\n", \_AAAAAAAAAAAAAAA % \_BBBBBBBBBBBBBBB);

\_XXXXXXXXXXXXXXX = (\_AAAAAAAAAAAAAAA - \_BBBBBBBBBBBBBBB) \* 10 + (\_AAAAAAAAAAAAAAA + \_BBBBBBBBBBBBBBB) / 10;

\_YYYYYYYYYYYYYYY = \_XXXXXXXXXXXXXXX + (\_XXXXXXXXXXXXXXX % 10);

printf("%d\n", \_XXXXXXXXXXXXXXX);

printf("%d\n", \_YYYYYYYYYYYYYYY);

system("pause");

return 0;

}

**Prog2.c**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_AAAAAAAAAAAAAAA, \_BBBBBBBBBBBBBBB, \_CCCCCCCCCCCCCCC;

printf("Enter \_AAAAAAAAAAAAAAA:");

scanf("%d", &\_AAAAAAAAAAAAAAA);

printf("Enter \_BBBBBBBBBBBBBBB:");

scanf("%d", &\_BBBBBBBBBBBBBBB);

printf("Enter \_CCCCCCCCCCCCCCC:");

scanf("%d", &\_CCCCCCCCCCCCCCC);

if ((\_AAAAAAAAAAAAAAA > \_BBBBBBBBBBBBBBB))

{

if ((\_AAAAAAAAAAAAAAA > \_CCCCCCCCCCCCCCC))

{

goto Abigger;

}

else

{

printf("%d\n", \_CCCCCCCCCCCCCCC);

goto Outofif;

Abigger:

printf("%d\n", \_AAAAAAAAAAAAAAA);

goto Outofif;

}

}

if ((\_BBBBBBBBBBBBBBB < \_CCCCCCCCCCCCCCC))

{

printf("%d\n", \_CCCCCCCCCCCCCCC);

}

else

{

printf("%d\n", \_BBBBBBBBBBBBBBB);

}

Outofif:

if (((\_AAAAAAAAAAAAAAA == \_BBBBBBBBBBBBBBB) && (\_AAAAAAAAAAAAAAA == \_CCCCCCCCCCCCCCC) && (\_BBBBBBBBBBBBBBB == \_CCCCCCCCCCCCCCC)))

{

printf("%d\n", 1);

}

else

{

printf("%d\n", 0);

}

if (((\_AAAAAAAAAAAAAAA < 0) || (\_BBBBBBBBBBBBBBB < 0) || (\_CCCCCCCCCCCCCCC < 0)))

{

printf("%d\n", -1);

}

else

{

printf("%d\n", 0);

}

if ((!(\_AAAAAAAAAAAAAAA < (\_BBBBBBBBBBBBBBB + \_CCCCCCCCCCCCCCC))))

{

printf("%d\n", (10));

}

else

{

printf("%d\n", (0));

}

system("pause");

return 0;

}

**Prog3.c**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_AAAAAAAAAAAAAAA, \_AAAAAAAAAAAAAA2, \_BBBBBBBBBBBBBBB, \_XXXXXXXXXXXXXXX, \_CCCCCCCCCCCCCC1, \_CCCCCCCCCCCCCC2;

printf("Enter \_AAAAAAAAAAAAAAA:");

scanf("%d", &\_AAAAAAAAAAAAAAA);

printf("Enter \_BBBBBBBBBBBBBBB:");

scanf("%d", &\_BBBBBBBBBBBBBBB);

for (int \_AAAAAAAAAAAAAA2 = \_AAAAAAAAAAAAAAA; \_AAAAAAAAAAAAAA2 <= \_BBBBBBBBBBBBBBB; \_AAAAAAAAAAAAAA2++)

printf("%d\n", \_AAAAAAAAAAAAAA2 \* \_AAAAAAAAAAAAAA2);

for (int \_AAAAAAAAAAAAAA2 = \_BBBBBBBBBBBBBBB; \_AAAAAAAAAAAAAA2 <= \_AAAAAAAAAAAAAAA; \_AAAAAAAAAAAAAA2++)

printf("%d\n", \_AAAAAAAAAAAAAA2 \* \_AAAAAAAAAAAAAA2);

\_XXXXXXXXXXXXXXX = 0;

\_CCCCCCCCCCCCCC1 = 0;

while (\_CCCCCCCCCCCCCC1 < \_AAAAAAAAAAAAAAA)

{

{

\_CCCCCCCCCCCCCC2 = 0;

while (\_CCCCCCCCCCCCCC2 < \_BBBBBBBBBBBBBBB)

{

{

\_XXXXXXXXXXXXXXX = \_XXXXXXXXXXXXXXX + 1;

\_CCCCCCCCCCCCCC2 = \_CCCCCCCCCCCCCC2 + 1;

}

}

\_CCCCCCCCCCCCCC1 = \_CCCCCCCCCCCCCC1 + 1;

}

}

printf("%d\n", \_XXXXXXXXXXXXXXX);

\_XXXXXXXXXXXXXXX = 0;

\_CCCCCCCCCCCCCC1 = 1;

do

{

\_CCCCCCCCCCCCCC2 = 1;

do

{

\_XXXXXXXXXXXXXXX = \_XXXXXXXXXXXXXXX + 1;

\_CCCCCCCCCCCCCC2 = \_CCCCCCCCCCCCCC2 + 1;

}

while (!(\_CCCCCCCCCCCCCC2 > \_BBBBBBBBBBBBBBB));

\_CCCCCCCCCCCCCC1 = \_CCCCCCCCCCCCCC1 + 1;

}

while (!(\_CCCCCCCCCCCCCC1 > \_AAAAAAAAAAAAAAA));

printf("%d\n", \_XXXXXXXXXXXXXXX);

system("pause");

return 0;

}