

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 №6

Виконав: ст. гр. КІ-307
Вітик С.А.
Перевірив:
Козак Н.Б

Анотація

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

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

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

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

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

Зміст

| | |
|---|----|
| Анотація..... | 2 |
| Завдання до курсового проекту | 4 |
| Вступ | 6 |
| Огляд методів та способів проектування трансляторів | 7 |
| Формальний опис вхідної мови програмування | 10 |
| 1.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура | 10 |
| 1.2. Опис термінальних символів та ключових слів | 11 |
| Розробка транслятора вхідної мови програмування | 13 |
| 1.3. Вибір технології програмування | 13 |
| 1.4. Проектування таблиць транслятора | 14 |
| 1.5. Розробка лексичного аналізатора | 16 |
| 1.5.1. Розробка блок-схеми алгоритму..... | 17 |
| 1.5.2. Опис програми реалізації лексичного аналізатора | 17 |
| 1.6. Розробка синтаксичного та семантичного аналізатора | 19 |
| 1.6.1. Опис програми реалізації синтаксичного та семантичного аналізатора..... | 20 |
| 1.6.2. Розробка граф-схеми алгоритму | 20 |
| 1.7. Розробка генератора коду..... | 21 |
| 1.7.1. Розробка граф-схеми алгоритму..... | 22 |
| 1.7.2. Опис програми реалізації генератора коду | 23 |
| Налагодження та тестування розробленого транслятора | 24 |
| 1.8. Опис інтерфейсу та інструкції користувачу. | 24 |
| 1.9. Виявлення лексичних і синтаксичних помилок. | 25 |
| 1.10. Перевірка роботи транслятора за допомогою тестових задач..... | 26 |
| Висновки | 28 |
| СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ..... | 29 |

Завдання до курсового проекту

Варіант 6

Завдання на курсовий проект

1. Цільова мова транслятора – мова програмування C або асемблер для 32/64 розрядного процесора.
2. Для отримання виконуваного файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio або будь-яким іншим.
3. Мова розробки транслятора: C/C++.
4. Реалізувати графічну оболонку або інтерфейс з командного рядка.
5. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.
6. На виході розробленого транслятора мають створюватись такі файли:

файл з лексемами;

файл з повідомленнями про помилки (або про їх відсутність);

файл на мові C або асемблера;

об'єктний файл;

виконуваний файл.

7. Назва вхідної мови програмування утворюється від першої букви у прізвищі студента та останніх двох цифр номера його варіанту. Саме таке розширення повинні мати текстові файли, написані на цій мові програмування.

Деталізація завдання на проектування:

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

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

3. За допомогою оператора вводу можна зчитати з клавіатури значення змінної; за допомогою оператора виводу можна вивести на екран значення змінної, виразу чи цілої константи.
4. В кожному завданні обов'язковим є оператор присвоєння за допомогою якого можна реалізувати обчислення виразів з використанням заданих операцій і операції круглі дужки (); у якості операндів можуть бути цілі константи, змінні, а також інші вирази.

5. В кожному завданні обов'язковим є оператор типу "блок" (складений оператор), його вигляд має бути таким, як і блок тіла програми.
6. Необхідно реалізувати задані варіантом оператори, синтаксис операторів наведено у таблиці 1.1. Синтаксис вхідної мови має забезпечити реалізацію обчислень лінійних алгоритмів, алгоритмів з розгалуженням і циклічних алгоритмів. Опис формальної мови студент погоджує з викладачем.
7. Оператори можуть бути довільної вкладеності і в будь-якій послідовності.
8. Для перевірки роботи розробленого транслятора, необхідно написати три тестові програми на вхідній мові програмування.

Деталізований опис власної мови програмування:

- **Start**— початок програми
- **Var**— оголошення змінних
- **Finish**— кінець програми
- **Int_4**— тип даних
- **Scan**— оператор вводу
- **Print**— оператор виводу
- **if, else if, else**— умовний оператор
- **<—** оператор присвоєння
- **++**— додавання
- **--** - віднімання
- ******— множення
- **Div**— ділення
- **Ge**— більше
- **Le**— менше
- **Eg**— рівність
- **Ne**— нерівність
- **Not**— заперечення
- **And** — логічне І
- **Or**— логічне АБО
- **;**— кінець оператора
- **,**— розділювач змінних
- **(**— відкрита дужка
- **)**— закрыта дужка
- **??...**— початок коментаря
- **a...z**— маленькі латинські букви
- **0...9**— цифри
- **Goto**
- **For-to**
- **For-downto**
- **While**
- **Repeat-Untill**
- символи табуляції, переходу на новий рядок, пробіл

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 – кожен 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.

Огляд методів та способів проектування трансляторів

Термін "транслятор" визначає обслуговуючу програму, що проводить трансляцію вихідної програми, представленої на вхідній мові програмування, у робочу програму, яка відображена на об'єктній мові. Наведене визначення застосовне до різноманітних транслують програм. Однак кожна з таких програм може виявляти свої особливості в організації процесу трансляції. В сучасному контексті транслятори поділяються на три основні групи: асемблери, компілятори та інтерпретатори.

Асемблер - це системна обслуговуюча програма, яка перетворює символічні конструкції в команди машинної мови. Типовою особливістю асемблерів є дослівна трансляція однієї символічної команди в одну машинну.

Компілятор - обслуговуюча програма, яка виконує трансляцію програми, написаної мовою оригіналу програмування, в машинну мову. Схоже до асемблера, компілятор виконує перетворення програми з однієї мови в іншу, найчастіше - у мову конкретного комп'ютера.

Інтерпретатор - це програма чи пристрій, що виконує пооператорну трансляцію та виконання вихідної програми. Відмінно від компілятора, інтерпретатор не створює на виході програму на машинній мові. Розпізнавши команду вихідної мови, він негайно її виконує, забезпечуючи більшу гнучкість у процесі розробки та налагодження програм.

Процес трансляції включає фази лексичного аналізу, синтаксичного та семантичного аналізу, оптимізації коду та генерації коду. Лексичний аналіз розбиває вхідну програму на лексеми, що представляють слова відповідно до визначень мови. Синтаксичний аналіз визначає структуру програми, створюючи синтаксичне дерево. Семантичний аналіз виявляє залежності між частинами програми, недосяжні контекстно-вільним синтаксисом. Оптимізація коду та генерація коду спрямовані на оптимізацію та створення машинно-залежного коду відповідно.

Зазначені фази можуть об'єднуватися або відсутні у трансляторах в залежності від їхньої реалізації. Наприклад, у простих однопрохідних трансляторах може 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: автоматично побудову таблиць об'єктів (таких як ідентифікатори, рядки, числа і т. д.) і видачу значень для кожної лексеми при кожному новому зверненні до нього. У цьому контексті таблиці об'єктів формуються в подальших етапах, наприклад, під час синтаксичного аналізу.

На етапі лексичного аналізу виявляються деякі прості помилки, такі як неприпустимі символи або невірний формат чисел та ідентифікаторів.

Основним завданням синтаксичного аналізу є розбір структури програми. Зазвичай під структурою розуміється дерево, яке відповідає розбору в контекстно-вільній

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

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

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

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

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

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

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

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

Фінальна фаза трансляції - генерація коду, результатом якої є або асемблерний модуль, або об'єктний (або завантажувальний) модуль. На цьому етапі можуть

застосовуватися деякі локальні оптимізації для полегшення генерації вартісного та ефективного коду.

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

Формальний опис вхідної мови програмування

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

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

програма = 'Start' 'Var' <оголошення змінних> ';' <тіло програми> 'Finish'

оголошення змінних = [<тип даних> <список змінних>]

тип даних = 'Int_4'

список змінних = <ідентифікатор> { ',' <ідентифікатор> }

ідентифікатор = <буква> { <буква або цифра> }

буква або цифра = <буква> | <цифра>

буква = 'a' | 'b' | 'c' | 'd' | ... | 'z'

цифра = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

тіло програми = <оператор> ';' { <оператор> ';' }

оператор = <присвоєння> | <ввід> | <вивід> | <умовний оператор> | <складений оператор> | <цикл> | <перехід>

перехід = 'Goto' <мітка>

мітка = <ідентифікатор>

цикл = <цикл For> | <цикл While> | <цикл Repeat-Until>

цикл For = 'For' <ідентифікатор> '<-' <число> 'To' <число> <оператор>

| 'For' <ідентифікатор> '<-' <число> 'Downto' <число> <оператор>

цикл While = 'While' <логічний вираз> <оператор>

цикл Repeat-Until = 'Repeat' <тіло програми> 'Until' <логічний вираз>

умовний оператор = 'If' <логічний вираз> <оператор> ['Else If' <оператор>] ['Else' <оператор>]

присвоєння = <ідентифікатор> '<-' <арифметичний вираз>

арифметичний вираз = <доданок> { '++' <доданок> | '--' <доданок> }

доданок = <множник> { '**' <множник> | 'Div' <множник> }

множник = <ідентифікатор> | <число> | '(' <арифметичний вираз> ')'

число = <цифра> { <цифра> }

ввід = 'Scan' <ідентифікатор>

вивід = 'Print' <ідентифікатор>

логічний вираз = <вираз l> { 'Or' <вираз l> }

вираз l = <порівняння> { 'And' <порівняння> }

порівняння = <операція порівняння> | 'Not' '(' <логічний вираз> ')'
| '(' <логічний вираз> ')'

операція порівняння = <арифметичний вираз> <менше-більше> <арифметичний вираз>

менше-більше = 'Ge' | 'Le' | 'Eg' | 'Ne'

складений оператор = 'Start' <тіло програми> 'Finish'

1.2. Опис термінальних символів та ключових слів

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

| Термінальний символ або ключове слово | Значення |
|---------------------------------------|---|
| | |
| Start | Початок тексту програми |
| Var | Початок блоку опису змінних |
| Finish | Кінець розділу операторів |
| Scan | Оператор вводу змінних |
| Print | Оператор виводу (змінних або рядкових констант) |
| <- | Оператор присвоєння |
| If | Оператор умови |
| Else | Оператор умови |
| Goto | Оператор переходу |
| | |
| For | Оператор циклу |
| To | Інкремент циклу |
| Downto | Декремент циклу |

| | |
|--------|----------------------------------|
| Do | Початок тіла циклу |
| While | Оператор циклу |
| Repeat | Початок тіла циклу |
| Until | Оператор циклу |
| ++ | Оператор додавання |
| -- | Оператор віднімання |
| ** | Оператор множення |
| DIV | Оператор ділення |
| | |
| | |
| Ne | Оператор перевірки на нерівність |
| Le | Оператор перевірки чи менше |
| Ge | Оператор перевірки чи більше |
| NOT | Оператор логічного заперечення |
| AND | Оператор кон'юнкції |
| OR | Оператор диз'юнкції |
| INT_4 | 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.

Розробка транслятора вхідної мови програмування

1.3. 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

1.4. Проектування таблиць транслятора

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

1) Мульти мапа для лексеми, значення та рядка кожного токена.

```
std::multimap<int, std::shared_ptr<IToken>> m_priorityTokens;

std::string m_lexeme; //Лексема
std::string m_value;  //Значення
int m_line = -1;      //Рядок
```

2) Таблиця лексичних класів

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

Таблиця 2 Опис термінальних символі та ключових слів

| Термінальний символ або ключове слово | Значення |
|---------------------------------------|---|
| | |
| Start | Початок тексту програми |
| Var | Початок блоку опису змінних |
| Finish | Кінець розділу операторів |
| Scan | Оператор вводу змінних |
| Print | Оператор виводу (змінних або рядкових констант) |
| <- | Оператор присвоєння |
| If | Оператор умови |
| Else | Оператор умови |
| Goto | Оператор переходу |
| | |
| For | Оператор циклу |
| To | Інкремент циклу |
| Downto | Декремент циклу |
| Do | Початок тіла циклу |

| | |
|--------|----------------------------------|
| While | Оператор циклу |
| Repeat | Початок тіла циклу |
| Until | Оператор циклу |
| ++ | Оператор додавання |
| -- | Оператор віднімання |
| ** | Оператор множення |
| DIV | Оператор ділення |
| | |
| | |
| Ne | Оператор перевірки на нерівність |
| Le | Оператор перевірки чи менше |
| Ge | Оператор перевірки чи більше |
| NOT | Оператор логічного заперечення |
| AND | Оператор кон'юнкції |
| OR | Оператор диз'юнкції |
| INT_4 | знакові цілі |
| ??... | Коментар |
| , | Розділювач |
| ; | Ознака кінця оператора |
| (| Відкриваюча дужка |
|) | Закриваюча дужка |

1.5. 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;

1.5.1. Розробка блок-схеми алгоритму

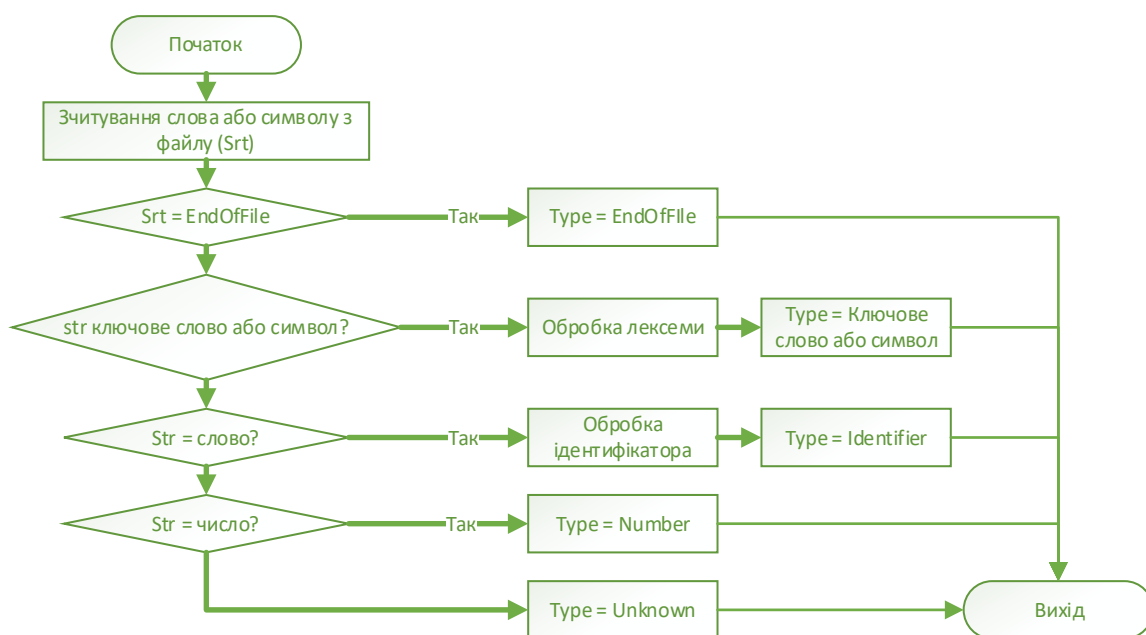


Рис. 3.1 Блок-схема роботи лексичного аналізатора

1.5.2. Опис програми реалізації лексичного аналізатора

Основна задача лексичного аналізу – розбити вихідний текст, що складається з послідовності одиночних символів, на послідовність слів, або лексем, тобто виділити ці слова з безперервної послідовності символів. Всі символи вхідної послідовності з цієї точки зору розділяються на символи, що належать яким-небудь лексемам, і символи, що розділяють лексеми. В цьому випадку використовуються звичайні засоби обробки рядків. Вхідна програма проглядається послідовно з початку до кінця. Базові елементи, або лексичні одиниці, розділяються пробілами, знаками операцій і спеціальними символами (новий рядок, знак табуляції), і таким чином виділяються та розпізнаються ідентифікатори, літерали і термінальні символи (операції, ключові слова).

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

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

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

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

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

1.6. Розробка синтаксичного та семантичного аналізатора

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

В основі синтаксичного аналізатора лежить Розпізнавач тексту вхідної програми на основі граматики вхідного мови. Як правило, синтаксичні конструкції мов програмування можуть бути описані за допомогою КС-грамматик, рідше зустрічаються мови, які можуть бути описані за допомогою регулярних граматик. Найчастіше регулярні граматики застосовні до мов асемблера, а мови високого рівня побудовані на основі КС-мов.

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

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

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

1.6.1. 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.

1.6.2. Розробка граф-схеми алгоритму

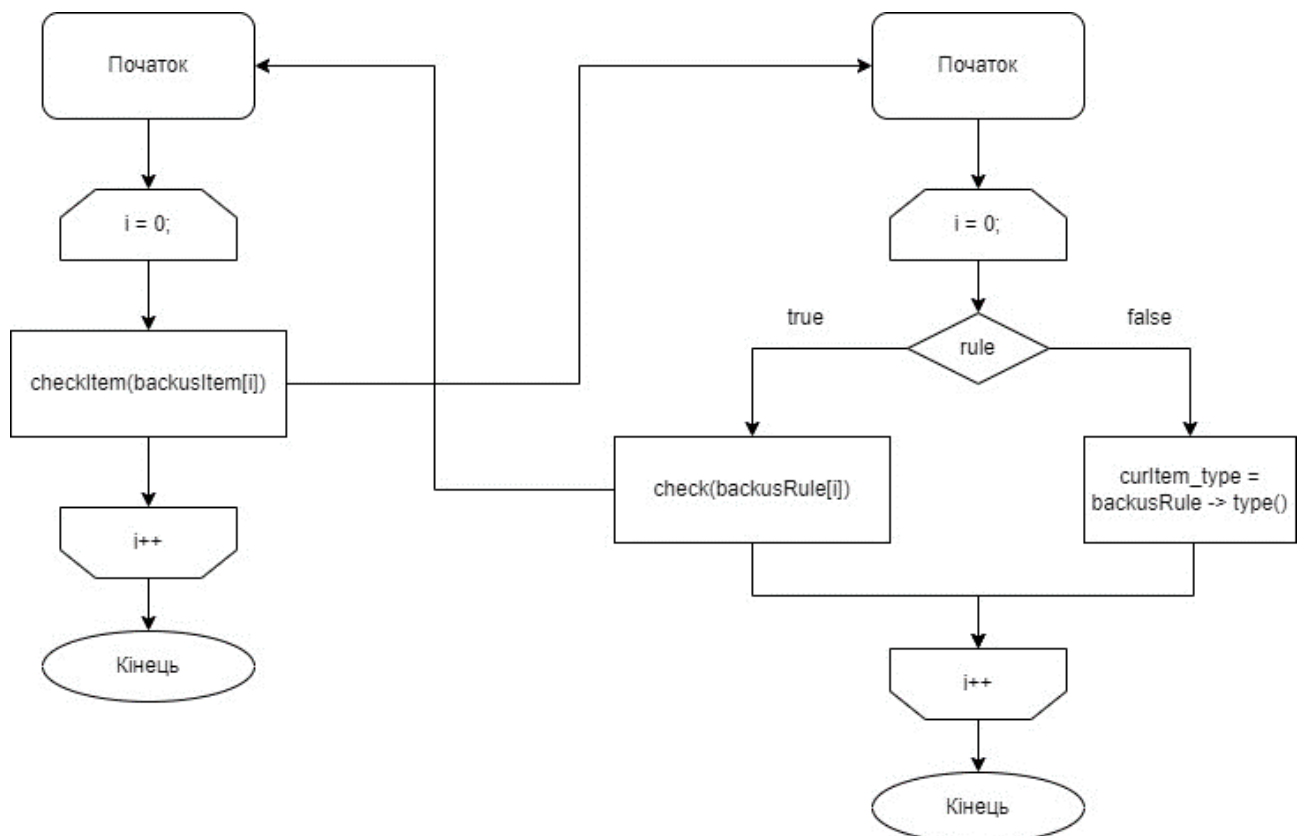


Рис. 3.2 Граф-схема роботи синтаксичного аналізатора

1.7. Розробка генератора коду

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

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

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

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

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

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

1.7.1. Розробка граф-схеми алгоритму

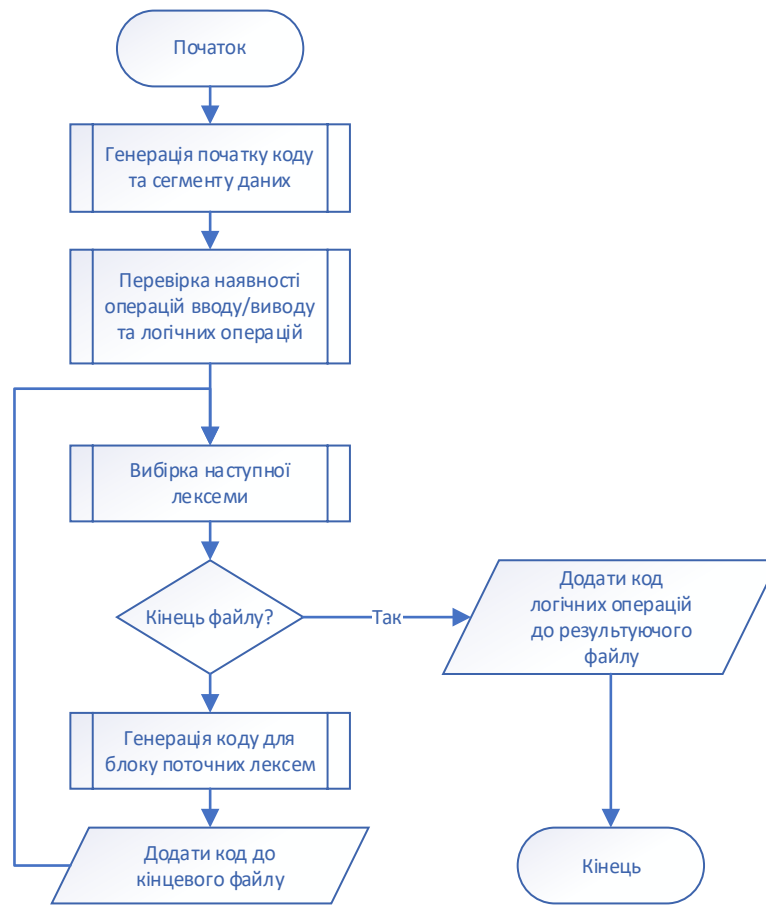


Рис. 3.3 Блок схема генератора коду

1.7.2. Опис програми реалізації генератора коду

У компілятора, реалізованого в даному курсовому проєкті, вихідна мова - програма на мові Assembler. Ця програма записується у файл, що має таку ж саму назву, як і файл з вхідним текстом, але розширення “asm”. Генерація коду відбувається одразу ж після синтаксичного аналізу.

В даному трансляторі генератор коду послідовно викликає окремі функції, які записують у вихідний файл частини коду.

Першим кроком генерації коду записується ініціалізація сегменту даних. Далі виконується аналіз коду, та визначаються процедури, зміни, які використовуються.

Проаналізувавши змінні, які є у програмі, генератор формує код даних для асемблерної програми. Для цього з таблиці лексем вибирається ім'я змінної (типи змінних відповідають 4 байтам), та записується 0, в якості початкового значення.

Аналіз наявних процедур необхідний у зв'язку з тим, що процедури введення/виведення, виконання арифметичних та логічних операцій, виконано у вигляді окремих процедур і у випадку їх відсутності немає сенсу записувати у вихідний файл зайву інформацію.

Після цього зчитується лексема з таблиці лексем. Також відбувається перевірка, чи це не остання лексема. Якщо це остання лексема, то функція завершується.

Наступним кроком є аналіз таблиці лексем, та безпосередня генерація коду у відповідності до вхідної програми.

Генератор коду зчитує лексему та генерує відповідний код, який записується у файл. Наприклад, якщо це лексема виведення, то у основну програму записується виклик процедури виведення, попередньо записавши у співпроцесор значення, яке необхідно вивести. Якщо це арифметична операція, так само викликається дана процедура, але як і в попередньому випадку, спочатку у регістри співпроцесора записується інформація, яка вказує над якими значеннями виконувати дії.

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

В кінці своєї роботи, генератор формує код завершення асемблерної програми.

Налагодження та тестування розробленого транслятора

Будь-яке програмне забезпечення необхідно протестувати і налагодити. Після опрацювання синтаксичних і семантичних помилок необхідно переконатися, що розроблене програмне забезпечення функціонує так, як очікувалось.

Для перевірки коректності роботи розробленого транслятора необхідно буде написати тестові задачі на вхідній мові програмування, отримати код на мові програмування C і переконатись, що він працює правильно.

1.8. Опис інтерфейсу та інструкції користувачу.

Розроблений транслятор має простий консольний інтерфейс.

При запуску програми файл автоматично відкривається та починається обробка вхідного коду:

```
Input file name not setted. Used default input file name "file1.cwl"

Used default mode

Original source:
-----
Program Rpog ;
Start Var Int_4 Aaaa, Kkkk ;
    Kkkk << 25630
    Scan ( Bbbb )
    Print ( Kkkk ++ Bbbb )
Finish
-----

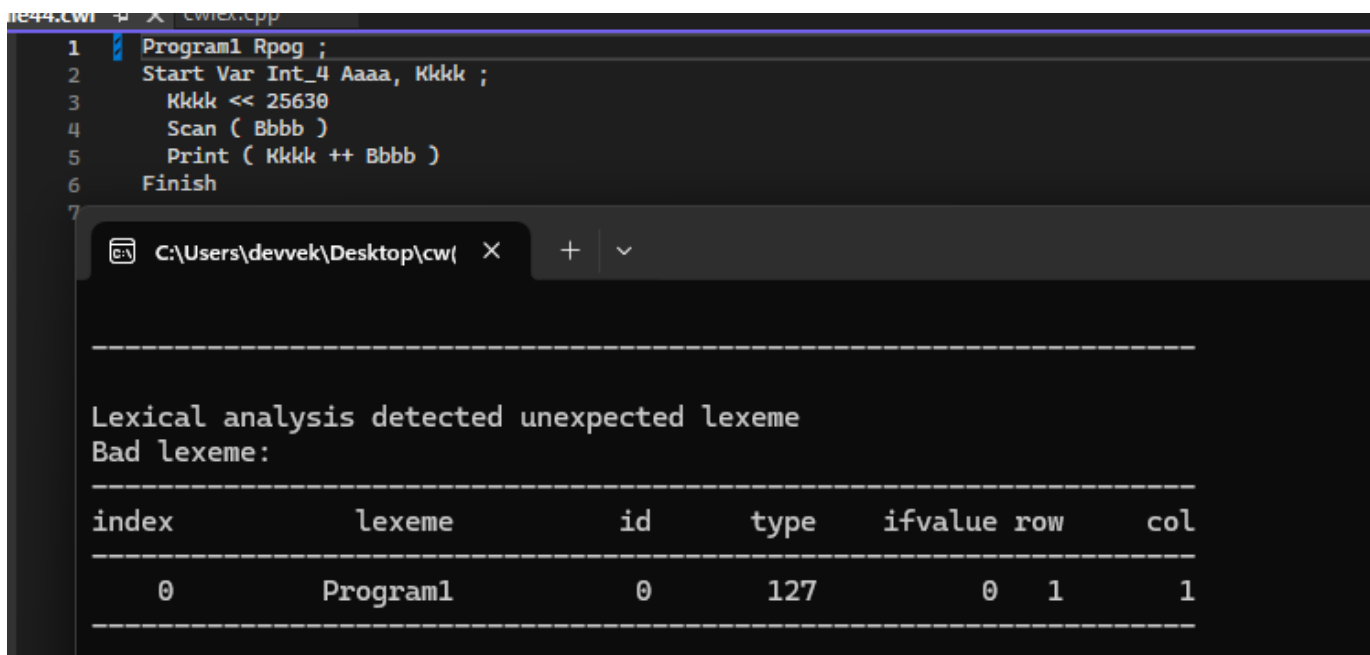
Source after comment removing:
-----
Program Rpog ;
Start Var Int_4 Aaaa, Kkkk ;
    Kkkk << 25630
    Scan ( Bbbb )
    Print ( Kkkk ++ Bbbb )
Finish
-----
```

Рис 4.1. Інтерфейс додатку

1.9. Виявлення лексичних і синтаксичних помилок.

Помилки у вхідній програмі виявляються на етапі синтаксичного і семантичного аналізу.

Наприклад, у програмі зробимо синтаксичну помилку – у 1-му рядку неправильно вкажемо старт програми :



```
1 Program1 Rpog ;
2 Start Var Int_4 Aaaa, Kkkk ;
3 Kkkk << 25630
4 Scan ( Bbbb )
5 Print ( Kkkk ++ Bbbb )
6 Finish
7
```

Lexical analysis detected unexpected lexeme
Bad lexeme:

| index | lexeme | id | type | ifvalue | row | col |
|-------|----------|----|------|---------|-----|-----|
| 0 | Program1 | 0 | 127 | 0 | 1 | 1 |

Рис. 4.2. Вивід інформації про синтаксичну помилку.

1.10. Перевірка роботи транслятора за допомогою тестових задач.

Тестова програма «Лінійний алгоритм»

1. Ввести два числа Aaaa і Bbbb і виконую дію додавання та множення.

```
Program Rpog ;  
Start Var Int_4 Aaaa, Bbbb, Xxxx ;  
  Scan(Aaaa);  
  Scan(Bbbb);  
  
  Xxxx == Aaaa ++ Bbbb  
  Print(Xxxx);  
  
  Xxxx == Aaaa ** Bbbb  
  Print(Xxxx);  
  
Finish
```

| Microsoft Visual Studio Debug Console | | | | | | |
|---------------------------------------|--------|-----|---|---|----|----|
| 18 | (| 284 | 1 | 0 | 4 | 7 |
| 19 | Bbbb | 2 | 2 | 0 | 4 | 8 |
| 20 |) | 287 | 1 | 0 | 4 | 12 |
| 21 | ; | 256 | 1 | 0 | 4 | 13 |
| 22 | Xxxx | 3 | 2 | 0 | 6 | 3 |
| 23 | == | 275 | 1 | 0 | 6 | 8 |
| 24 | Aaaa | 1 | 2 | 0 | 6 | 11 |
| 25 | + | 265 | 1 | 0 | 6 | 16 |
| 26 | + | 265 | 1 | 0 | 6 | 16 |
| 27 | Bbbb | 2 | 2 | 0 | 6 | 19 |
| 28 | Print | 333 | 1 | 0 | 7 | 3 |
| 29 | (| 284 | 1 | 0 | 7 | 8 |
| 30 | Xxxx | 3 | 2 | 0 | 7 | 9 |
| 31 |) | 287 | 1 | 0 | 7 | 13 |
| 32 | ; | 256 | 1 | 0 | 7 | 14 |
| 33 | Xxxx | 3 | 2 | 0 | 10 | 3 |
| 34 | == | 275 | 1 | 0 | 10 | 8 |
| 35 | Aaaa | 1 | 2 | 0 | 10 | 11 |
| 36 | * | 271 | 1 | 0 | 10 | 16 |
| 37 | * | 271 | 1 | 0 | 10 | 16 |
| 38 | Bbbb | 2 | 2 | 0 | 10 | 19 |
| 39 | Print | 333 | 1 | 0 | 11 | 3 |
| 40 | (| 284 | 1 | 0 | 11 | 8 |
| 41 | Xxxx | 3 | 2 | 0 | 11 | 9 |
| 42 |) | 287 | 1 | 0 | 11 | 13 |
| 43 | ; | 256 | 1 | 0 | 11 | 14 |
| 44 | Finish | 307 | 1 | 0 | 13 | 1 |

cykParse complete.....[ok]: 44 Finish

Рис. 4.5. Результати виконання тестової задачі 1.

Висновки

В процесі виконання курсового проекту було виконано наступне:

1. Складено формальний опис мови програмування p23, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створено компілятор мови програмування p23, 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 p23. Вихідним кодом генератора є програма на мові Assembler(x86).

3. Проведене тестування компілятора на тестових програмах за наступними пунктами:

3.1. На виявлення лексичних помилок.

3.2. На виявлення синтаксичних помилок.

3.3. Загальна перевірка роботи компілятора.

Тестування не виявило помилок в роботі компілятор, і всі помилки в тестових програмах на мові p23 були успішно виявлені і відповідно оброблені.

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

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс]
: навч. посіб. для студ. спеціальності 123 – «Комп’ютерна інженерія» / О.
І. Марченко ; КПІ ім. Ігоря Сікорського. – Київ: КПІ ім. Ігоря Сікорського, 2021.
– 108 с.
2. Формальні мови, граматики та автомати: Навчальний посібник /
Гавриленко С.Ю. – Харків: НТУ «ХПІ», 2021. – 133 с.
3. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії
формальних мов: Навчальний посібник у двох частинах. – Чернівці:
ЧНУ, 2008. – 84 с.
4. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії
компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ,
2008. – 84 с.
5. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers,
principles, techniques, and tools, Second Edition, New York, 2007. – 1038
с.
6. Системне програмування (курсовий проект) [Електронний ресурс] –
Режим доступу до ресурсу:
<https://vns.lpnu.ua/course/view.php?id=11685>.
7. MIT OpenCourseWare. Computer Language Engineering [Електронний
ресурс] – Режим доступу до ресурсу: <https://ocw.mit.edu/courses/6-035-computer-language-engineering-spring-2010>.