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 №23**

**Виконав:** ст. гр. КІ-308

Яконюк Н.О.

**Перевірив:**

Козак Н.Б.

Львів-2024

Анотація

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

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

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

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

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

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 23**

Завдання на курсовий проект

1. Цільова мова транслятора – асемблер для 32-розрядного процесора.

2. Для отримання виконавчого файлу на виході розробленого транслятора скористатися програмами ml.exe і link.exe.

3. Мова розробки транслятора: C++.

4. Реалізувати оболонку або інтерфейс з командного рядка.

5. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.

6. На виході розробленого транслятора мають створюватись такі файли:

* *файл з лексемами;*
* *файл з повідомленнями про помилки (або про їх відсутність);*
* *файл на мові асемблера;*
* *об’єктний файл;*
* *виконавчий файл.*

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

В моєму випадку це .y23

Опис вхідної мови програмування:

* Тип даних: INTEGER
* Блок тіла програми: MAIMPROGRAM DATA…; START 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
* Регістр ідентифікаторів: Up10 перший символ \_
* Операції арифметичні: +, -, \*, DIV, MOD
* Операції порівняння: ==, !=, GT, LT
* 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 об'єктній мові. Наведене визначення застосовне до різноманітних транслюють програм. Однак кожна з таких програм може виявляти свої особливості в організації процесу трансляції. В сучасному контексті транслятори поділяються на три основні групи: асемблери, компілятори та інтерпретатори.

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

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

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

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

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

Під час процесу виділення лексем лексичний аналізатор може виконувати дві основні функції: автоматично побудову таблиць об'єктів (таких як ідентифікатори, рядки, числа і т. д.) і видачу значень для кожної лексеми при кожному новому зверненні до нього. У цьому контексті таблиці об'єктів формуються в подальших етапах, наприклад, під час синтаксичного аналізу.

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

Основним завданням синтаксичного аналізу є розбір структури програми. Зазвичай під структурою розуміється дерево, яке відповідає розбору в контекстно-вільній граматиці мови програмування. У сучасній практиці найчастіше використовуються методи аналізу, такі як LL (1) або LR (1) та їхні варіанти (рекурсивний спуск для LL (1) або LR (1), LR (0), SLR (1), LALR (1) та інші для LR (1)). Рекурсивний спуск застосовується частіше при ручному програмуванні синтаксичного аналізатора, тоді як LR (1) використовується при автоматичній генерації синтаксичних аналізаторів.

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

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

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

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

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

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

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

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

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

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 (Backus/Naur Form - BNF).

<topRule> ==> MAIMPROGRAM DATA <varsBlok> ; START <codeBlok> END

<varsBlok> ==> INTEGER <identifier> [{<commaAndIdentifier>}];

<identifier> ==> \_<up\_letter> {<up\_letter>|<number>} {0,9}

<commaAndIdentifier> ==> , <identifier>

<codeBlok> ==> START <write> | <read> | <assignment> | <ifStatement>

| <goto\_statement> | < labelRule > | < forToOrDownToDoRule> | <while> | <repeatUntil> END

<read> ==> GET ( <identifier> );

<write> ==> PUT ( <equation> | <stringRule>);

<assignment> ==> <identifier> ==> <equation>;

<ifStatement> ==> IF ( <equation> ) <codeBlok> <elseStatement>

<elseStatement> ==> ELSE <codeBlok>

<goto\_statement> ==> GOTO <ident>;

<labelRule> ==> <identifier>:

< forToOrDownToDoRule> ==> FOR <assignment> TO | DOWNTO <equation> DO <codeBlok>

<while> ==> WHILE (<equation>) <codeBlok>

<repeatUntil> ==> REPEAT <codeBlok> UNTIL (<equation<)

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

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

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

<arithmetic> ==> + | -

<mult> ==> \* | DIV | MOD

<logic> ==> AND | OR

<notOperation> ==> !!

<compare> ==> == | != | LT | GT

<stringRule> ==> “ <string> “

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

<LComment> ==> ??

<string> ==> { <low\_letter> | <up\_letter> | <number> }

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

<sign> ==> + | -

<low\_letter> ==> a|b|c|d|e|f|g|h|i|j|k|l|n|m|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|N|M|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 |
| MAIMPROGRAM | Початок програми |
| START | 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 |
| REPEAT | Початок тіла циклу |
| UNTIL | Оператор циклу |
| + | Оператор додавання |
| - | Оператор віднімання |
| \* | Оператор множення |
| DIV | Оператор ділення |
| MOD | Оператор знаходження залишку від ділення |
| == | Оператор перевірки на рівність |
| != | Оператор перевірки на нерівність |
| LT | Оператор перевірки чи менше |
| GT | Оператор перевірки чи більше |
| !! | Оператор логічного заперечення |
| AND | Оператор кон’юнкції |
| OR | Оператор диз’юнкції |
| INTEGER | 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.