Міністерство освіти і науки України

Національний університет "Львівська Політехніка"

Кафедра ЕОМ



**Пояснювальна записка**

до курсового проєкту "Системне програмування"

на тему: “Розробка системних програмних модулів та компонент систем програмування”

Індивідуальне завдання

“Розробка транслятора з вхідної мови програмування”

Виконав

студент групи КІ-308

Худьо В.В.

Перевірив:

Козак Н. Б.

Львів-2024

Завдання на курсовий проєкт

1. Цільова мова транслятора – мова програмування С 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 С або асемблера;*

*об’єктний файл;*

*виконуваний файл.*

1. 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.

1. За допомогою оператора вводу можна зчитати з клавіатури значення змінної; за допомогою оператора виводу можна вивести на екран значення змінної, виразу чи цілої константи.
2. 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.
3. В кожному завданні обов’язковим є оператор типу “блок” (складений оператор), його вигляд має бути таким, як і блок тіла програми.
4. 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.
5. Оператори можуть бути довільної вкладеності і в будь-якій послідовності.
6. Для перевірки роботи розробленого транслятора, необхідно написати три тестові програми на вхідній мові програмування.

**Деталізований опис власної мови програмування:**

* Тип даних: Int16\_t
* Блок тіла програми: Mainprogram Data…; Start End
* Оператор вводу: Input ()
* Оператор виводу: Output ()
* 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: Up-Low8 перший символ \_
* Операції арифметичні: +, -, Mul, Div, Mod
* Операції порівняння: Eg, Ne, >, <
* Операції логічні: !!, &&, ||
* Коментар: #\*... \*#
* Ідентифікатори змінних, числові константи
* Оператор присвоєння: ==>

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.

Зміст

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 проєктування 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> ==> Mainprogram Data <varsBlok> ; <codeBlok>

<varsBlok> ==> Int16\_t <identifier> [{<commaAndIdentifier>}];

<identifier> ==> \_<up\_letter> {<low\_letter>|<number>} {8}

<commaAndIdentifier> ==> , <identifier>

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

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

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

<write> ==> Output ( <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> ==> Mul | Div | Mod

<logic> ==> && | ||

<notOperation> ==> !!

<compare> ==> Eg | Ne | < | >

<stringRule> ==> “ <string> “

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

<LComment> ==> #\*

<RComment> ==> \*#

<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

**2.2 Опис термінальних символів та ключових слів**

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

|  |  |
| --- | --- |
| Термінальний символ або ключове слово | Значення |
| Mainprogram | Початок програми |
| Start | Початок тексту програми |
| Data | Початок блоку опису змінних |
| End | Кінець розділу операторів |
| Input | Оператор вводу змінних |
| Output | 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 |
| Mul | Оператор множення |
| Div | Оператор ділення |
| Mod | Оператор знаходження залишку від ділення |
| Eg | Оператор перевірки на рівність |
| Ne | Оператор перевірки на нерівність |
| < | Оператор перевірки чи менше |
| > | Оператор перевірки чи більше |
| !! | Оператор логічного заперечення |
| && | Оператор кон’юнкції |
| || | Оператор диз’юнкції |
| Int16\_t | 16-ти розрядні знакові цілі |
| #\*…\*# | Коментар |
| , | Розділювач |
| ; | Ознака кінця оператора |
| ( | Відкриваюча дужка |
| ) | Закриваюча дужка |

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

1. ТЕСТОВІ ПРОГРАМИ ДЛЯ ПЕРЕВІРКИ РОБОТИ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА

**3.1 Тестова програма №1 «Лінійний алгоритм»**

1. Ввести два числа А і В (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання).

2. Вивести на екран:

А + В (результат операції додавання);

А - В (результат операції віднімання);

А \* В (результат операції множення);

А / В (результат операції ділення);

А % В (результат операції отримання залишку від ділення).

3. Обрахувати значення виразів

Х = (А - В) \* 10 + (А + В) / 10

У = Х + Х % 10

4. Вивести значення Х і У на екран.

***Текст програми***

#\*Prog1\*#

Mainprogram

Data Int16\_t \_Aaaaaaaaa,\_Bbbbbbbbb,\_Xxxxxxxxx,\_Yyyyyyyyy;

Start

Output("Input \_Aaaaaaaaa: ");

Input(\_Aaaaaaaaa);

Output("Input \_Bbbbbbbbb: ");

Input(\_Bbbbbbbbb);

Output("\_Aaaaaaaaa + \_Bbbbbbbbb: ");

Output(\_Aaaaaaaaa + \_Bbbbbbbbb);

Output("\n\_Aaaaaaaaa - \_Bbbbbbbbb: ");

Output(\_Aaaaaaaaa - \_Bbbbbbbbb);

Output("\n\_Aaaaaaaaa \* \_Bbbbbbbbb: ");

Output(\_Aaaaaaaaa Mul \_Bbbbbbbbb);

Output("\n\_Aaaaaaaaa / \_Bbbbbbbbb: ");

Output(\_Aaaaaaaaa Div \_Bbbbbbbbb);

Output("\n\_Aaaaaaaaa % \_Bbbbbbbbb: ");

Output(\_Aaaaaaaaa Mod \_Bbbbbbbbb);

\_Xxxxxxxxx==>(\_Aaaaaaaaa - \_Bbbbbbbbb) Mul 10 + (\_Aaaaaaaaa + \_Bbbbbbbbb) Div 10;

\_Yyyyyyyyy==>\_Xxxxxxxxx + (\_Xxxxxxxxx Mod 10);

Output("\n\_Xxxxxxxxx = (\_Aaaaaaaaa - \_Bbbbbbbbb) \* 10 + (\_Aaaaaaaaa + \_Bbbbbbbbb) / 10\n");

Output(\_Xxxxxxxxx);

Output("\n\_Yyyyyyyyy = \_Xxxxxxxxx + (\_Xxxxxxxxx % 10)\n");

Output(\_Yyyyyyyyy);

End

**3.2 Тестова програма №2 «Алгоритм з розгалуженням»**

1. Ввести три числа А, В, С (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання). Використання вкладеного умовного оператора:

2. Знайти найбільше з них і вивести його на екран. Використання простого умовного оператора:

3. Вивести на екран число 1, якщо усі числа однакові (логічний вираз в умовному операторі має виглядати так: «(А=В) і (А=С) і (В=С)»), інакше вивести 0.

4. Вивести на екран число -1, якщо хоча б одне з чисел від’ємне (логічний вираз в умовному операторі має виглядати так: «(А<0) або (В<0) або (С<0)»), інакше вивести 0.

5. Вивести на екран число 10, якщо число А більше за суму чисел В і С (логічний вираз в умовному операторі має виглядати так: «!(А<(В+С))»), інакше вивести 0.

***Текст програми***

#\*Prog2\*#

Mainprogram

Data Int16\_t \_Aaaaaaaaa,\_Bbbbbbbbb,\_Ccccccccc;

Start

Output("Input \_Aaaaaaaaa: ");

Input(\_Aaaaaaaaa);

Output("Input \_Bbbbbbbbb: ");

Input(\_Bbbbbbbbb);

Output("Input \_Ccccccccc: ");

Input(\_Ccccccccc);

If(\_Aaaaaaaaa > \_Bbbbbbbbb)

Start

If(\_Aaaaaaaaa > \_Ccccccccc)

Start

Goto \_Temporal0;

End

Else

Start

Output(\_Ccccccccc);

Goto \_Gotoofif0;

\_Temporal0:

Output(\_Aaaaaaaaa);

Goto \_Gotoofif0;

End

End

If(\_Bbbbbbbbb < \_Ccccccccc)

Start

Output(\_Ccccccccc);

End

Else

Start

Output(\_Bbbbbbbbb);

End

\_Gotoofif0:

Output("\n");

If((\_Aaaaaaaaa Eg \_Bbbbbbbbb) && (\_Aaaaaaaaa Eg \_Ccccccccc) && (\_Bbbbbbbbb Eg \_Ccccccccc))

Start

Output(1);

End

Else

Start

Output(0);

End

Output("\n");

If((\_Aaaaaaaaa < 0) || (\_Bbbbbbbbb < 0) || (\_Ccccccccc < 0))

Start

Output(-1);

End

Else

Start

Output(0);

End

Output("\n");

If(!!(\_Aaaaaaaaa < (\_Bbbbbbbbb + \_Ccccccccc)))

Start

Output(10);

End

Else

Start

Output(0);

End

End

**3.3 Тестова програма №3 «Циклічний алгоритм»**

1. Ввести два числа А і В, причому А<В (імена змінних можуть бути іншими і мають

відповідати правилам запису ідентифікаторів згідно індивідуального завдання).

Використання простого оператора циклу:

2. Вивести на екран квадрати чисел від А до В включно.

Використання вкладеного оператора циклу:

3. Обрахувати Х=А\*В за наступним алгоритмом:

Х = 0

Цикл від 1 до А з кроком 1

Цикл від 1 до В з кроком 1

Х = Х + 1

4. Вивести значення Х на екран.

***Текст програми***

#\*Prog3\*#

Mainprogram

Data Int16\_t \_Aaaaaaaaa,\_Aaaaaaaa2,\_Bbbbbbbbb,\_Xxxxxxxxx,\_Cccccccc1,\_Cccccccc2;

Start

Output("Input \_Aaaaaaaaa: ");

Input(\_Aaaaaaaaa);

Output("Input \_Bbbbbbbbb: ");

Input(\_Bbbbbbbbb);

Output("For To do");

For \_Aaaaaaaa2==>\_Aaaaaaaaa To \_Bbbbbbbbb Do

Start

Output("\n");

Output(\_Aaaaaaaa2 Mul \_Aaaaaaaa2);

End

Output("\nFor Downto do");

For \_Aaaaaaaa2==>\_Bbbbbbbbb Downto \_Aaaaaaaaa Do

Start

Output("\n");

Output(\_Aaaaaaaa2 Mul \_Aaaaaaaa2);

End

Output("\nWhile \_Aaaaaaaaa \* \_Bbbbbbbbb: ");

\_Xxxxxxxxx==>0;

\_Cccccccc1==>0;

While(\_Cccccccc1 < \_Aaaaaaaaa)

Start

\_Cccccccc2==>0;

While (\_Cccccccc2 < \_Bbbbbbbbb)

Start

\_Xxxxxxxxx==>\_Xxxxxxxxx + 1;

\_Cccccccc2==>\_Cccccccc2 + 1;

End

\_Cccccccc1==>\_Cccccccc1 + 1;

End

Output(\_Xxxxxxxxx);

Output("\nRepeat Until \_Aaaaaaaaa \* \_Bbbbbbbbb: ");

\_Xxxxxxxxx==>0;

\_Cccccccc1==>1;

Repeat

\_Cccccccc2==>1;

Repeat

\_Xxxxxxxxx==>\_Xxxxxxxxx+1;

\_Cccccccc2==>\_Cccccccc2+1;

Until(!!(\_Cccccccc2 > \_Bbbbbbbbb))

\_Cccccccc1==>\_Cccccccc1+1;

Until(!!(\_Cccccccc1 > \_Aaaaaaaaa))

Output(\_Xxxxxxxxx);

End