Міністерство освіти і науки України

Національний університет “Львівська політехніка”



**Курсова робота**

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-38

Чорнобай Д.М.

**Прийняв:** викладач

Козак Н.Б.

Львів-2020

Зміст

Анотація 3

[1. Завдання до курсової роботи 4](#_Toc61093982)

[2. Огляд методів та способів проектування трансляторів 6](#_Toc61093983)

[3. Формальний опис вхідної мови програмування 9](#_Toc61093984)

[3.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура 9](#_Toc61093985)

[3.2. Опис термінальних символів та ключових слів 11](#_Toc61093986)

[4. Розробка транслятора вхідної мови програмування 12](#_Toc61093987)

[4.1. Вибір технології програмування 12](#_Toc61093988)

[4.2. Проектування таблиць транслятора 13](#_Toc61093989)

[4.3. Розробка лексичного аналізатора 14](#_Toc61093990)

[4.4. Опис лексичного аналізатора 15](#_Toc61093991)

[4.5. Розробка синтаксичного та семантичного аналізатора 19](#_Toc61093992)

[4.6. Розробка генератора коду 21](#_Toc61093993)

[4.7. Опис генератора коду 22](#_Toc61093994)

[5. Опис програми 24](#_Toc61093995)

[5.1. Опис інтерфейсу та інструкція користувачеві 27](#_Toc61093996)

[6. Відлагодження та тестування програми 28](#_Toc61093997)

[6.1. Виявлення лексичних та синтаксичних помилок 28](#_Toc61093998)

[6.2. Загальна перевірка коректності роботи транслятора 29](#_Toc61093999)

[7. Висновки 31](#_Toc61094000)

[8. Список використаних джерел 32](#_Toc61094001)

[9. Додатки 33](#_Toc61094002)

Анотація

В ході виконання курсової роботи буде розроблений транслятор, який дозволяє транслювати вхідну мову(згідно варіанту) в асемблерний код. Функції транслятора: лексичний аналіз, синтаксичний аналіз та генерація коду.

На етапі лексичного аналізу вхідна послідовність символів розбивається на лексеми, які записуються у відповідну таблицю лексем. Кожній лексемі присвоюється число, оскільки числа легше порівнювати, ніж символи. Також у таблицю лексем записується додаткова інформація:

* рядок, у якому розміщується лексема;
* значення, якщо тип лексеми відповідає числу та ін.

В ході синтаксичного аналізу використовується висхідних метод аналізу без повернення. Призначений для побудови дерева розбору, послідовно рухаючись від листків вгору до кореня дерева розбору.

В процесі генерації коду відбувається повторне зчитування таблиці лексем та формується відповідний асемблерний код для кожного блоку лексем. Отриманий код записується у результуючий файл та готовий до виконання. Його можна відкомпілювати за допомогою відповідних програм.

1. Завдання до курсової роботи
2. Цільова мова транслятора асемблер (iх86).
3. Для отримання виконавчого файлу на виході розробленого транслятора скористатися програмами tasm.exe і tlink.exe або tasm32.exe і tlink32.exe.
4. Мова розробки транслятора: ANSI C або C++.
5. Реалізувати оболонку або інтерфейс з командного рядка.
6. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.
7. На виході розробленого транслятора мають створюватись чотири файли: файл з повідомленнями про помилки (або про їх відсутність); файл на мові асемблера; об’єктний файл; виконавчий файл.
8. 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. Згідно мого варіанту: .c20.

Опис вхідної мови програмування:

- типи даних: Int32\_t

- оператор вводу: Get ()

- оператор виводу: Put ()

- блок тіла програми: Startprogram Variable…; Startblock - Endblock

- оператор: For – Downto (Паскаль)

- регістр ключових слів: Up-Low перший символ Up

- регістр ідентифікаторів: Up6 перший символ \_

- операції арифметичні: Add, Sub, Mul, Div, Mod

- операції порівняння: =, <>, >, <

- операції логічні: Not, And, Or

- коментар: $$...

- ідентифікатори змінних, числові константи

- оператор присвоєння: ::=

Для отримання виконавчого файлу, на виході розробленого транслятора скористатися програмами masm32.exe (компілятор мови асемблера) і link.exe (редактор зв’язків).

1. Огляд методів та способів проектування трансляторів

Транслятор - обслуговуюча програма, що перетворює вихідну програму, надану на вхідній мові програмування, у робочу програму, представлену на об'єктному мовою. Наведене визначення відноситься до всіх різновидів транслюють програм. Однак у кожної з таких програм можуть бути свої особливості щодо організації процесу трансляції. В даний час транслятори поділяються на три основні групи: асемблери, компілятори та інтерпретатори.

Асемблер - системна обслуговуюча програма, яка перетворить символічні конструкції в команди машинної мови. Специфічною рисою асемблером є те, що вони здійснюють дослівну трансляцію однієї символічної команди в одну машинну.

Компілятор - це обслуговуюча програма, що виконує трансляцію на машинну мову програми, записаної мовою оригіналу програмування. Також як і асемблер, компілятор забезпечує перетворення програми з однієї мови на іншу (найчастіше, в мову конкретного комп'ютера).

Інтерпретатор - програма або пристрій, що здійснює пооператорну трансляцію і виконання вихідної програми. На відміну від компілятора, інтерпретатор не породжує на виході програму на машинній мові. Розпізнавши команду вихідного мови, він тут же виконує її. Як у компіляторах, так і в інтерпретатора використовуються однакові методи аналізу вихідного тексту програми. Але інтерпретатор дозволяє почати обробку даних після написання навіть однієї команди. Це робить процес розробки і налагодження програм більш гнучким.

Процес трансляції складається з наступних фаз: лексичний аналіз, синтаксичний та семантичний аналіз, оптимізація коду, генерація коду.

На фазі лексичного аналізу вхідна програма, що представляє собою потік літер, розбивається на лексеми - слова у відповідності з визначеннями мови. Основними формалізму, які лежать в основі реалізації лексичних аналізаторів, є кінцеві автомати та регулярні вирази. 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.

У процесі виділення лексем лексичний аналізатор може як самостійно будувати таблиці об'єктів (ідентифікаторів, рядків, чисел і т.д.), так і видавати значення для кожної лексеми при черговому до нього зверненні. У цьому випадку таблиці об'єктів будуються в наступних фазах (наприклад, в процесі синтаксичного аналізу).

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

Основне завдання синтаксичного аналізу - розбір структури програми. Як правило, під структурою розуміється дерево, відповідне розбору в контекстно-вільної граматики мови. В даний час найчастіше використовується або LL (1)-аналіз (і його варіант - рекурсивний спуск), або LR (1) - аналіз та його варіанти (LR (0), SLR (1), LALR (1) та інші) . Рекурсивний спуск частіше використовується при ручному програмуванні синтаксичного аналізатора, LR (1) - при використанні систем автоматичного побудови синтаксичних аналізаторів.

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

На етапі контекстного аналізу виявляються залежності між частинами програми, які не можуть бути описані контекстно-вільною синтаксисом. Це в основному зв'язку «опис-використання», зокрема, аналіз типів об'єктів, аналіз областей видимості, відповідність параметрів, мітки та інші. У процесі контекстного аналізу таблиці об'єктів поповнюються інформацією про описах (властивостях) об'єктів.

Основним формалізмом, що використовується при контекстному аналізі, є апарат атрибутних граматик. Результатом контекстного аналізу є атрибутовані дерево програми. Інформація про об'єкти може бути як розосереджена в самому дереві, так і зосереджена в окремих таблицях об'єктів. У процесі контекстного аналізу також можуть бути виявлені помилки, пов'язані з неправильним використанням об'єктів.

Потім програма може бути переведена у внутрішнє представлення. Це робиться для цілей оптимізації та / або зручності генерації коду. Ще однією метою перетворення програми у внутрішнє представлення є бажання мати портативний компілятор. Тоді тільки остання фаза (генерація коду) є машинно-залежною. В якості внутрішнього подання може використовуватися префіксний або постфіксній запис, орієнтований граф, трійки, четвірки та інші.

Фаз оптимізації може бути декілька. Оптимізації зазвичай ділять на машинно-залежні та машинно-незалежні, локальні і глобальні. Частина машинно-залежною оптимізації виконується на фазі генерації коду. Глобальна оптимізація намагається взяти до уваги структуру всієї програми, локальна - тільки невеликих її фрагментів. Глобальна оптимізація грунтується на глобальному потоковий аналіз, який виконується на графі програми і представляє по суті перетворення цього графа. При цьому можуть враховуватися такі властивості програми, як міжпроцедурний аналіз, міжмодульних аналіз, аналіз галузей життя змінних і т.д.

Нарешті, генерація коду - остання фаза трансляції. Результатом її є або асемблерний модуль, або об'єктний (або завантажувальний) модуль. У процесі створення коду можуть виконуватися деякі локальні оптимізації, такі як розподіл регістрів, вибір довгих або коротких переходів, облік вартості команд при виборі конкретної послідовності команд. Для генерації коду розроблені різні методи, такі як таблиці рішень, зіставлення зразків, що включає динамічне програмування, різні синтаксичні методи.

Звичайно, ті чи інші фази транслятора можуть або відсутні зовсім, або об'єднуватися. У найпростішому випадку однопрохідного транслятора немає явної фази генерації проміжного представлення та оптимізації, інші фази об'єднані в одну, причому немає і 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 застосувала розширену нотацію Бекуса-Наура (Backus/Naur Form - BNF).

<program>::= Startprogram Variable <VAR\_blok>; Startblock <code\_blok> Endblock

<VAR\_blok> ::=<declarations> [{;<declarations>} ];.

<name>::=<letter>[<l \_or\_n>].

<declarations>::=<type\_i><declaration\_i> [{,<declaration\_i>}] .

<type\_i> ::= Int32\_t.

<declaration\_i> ::= <ident> [ ::= <const\_i>] .

<ident> ::= <letter>[<l \_or\_n>].

<l\_or\_n> ::= <letter>|<number> .

<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 .

<number> ::= 0|1|2|3|4|5|6|7|8|9 .

<const\_i> ::= [-|+]<number>[{number}] .

<code\_blok> ::= <statement> [{<statement>}] .

<statement> ::= <equation>|<if\_statement>|<get>|<put> .

<equation> ::= <ident> :: <expression\_i><expression\_b> .

<expression\_i>::= <term> [{add <term> | sub <term>}]|<term1>

<term1>::=[{<ident>|<const\_i>}] <expression\_b> [{<ident>|<const\_i>}]? <number>:<number>

<term>::=<ident>|<const\_i>|<factor>|<factor\_b>;

<factor>::=[{mul <term> | div <term>| mod <term> | <brackets>}].

<brackets>::=(<expression\_i>).

<expression\_b>::= <term\_b> [{== <term\_b> | != <term\_b> | >> <term\_b>| << <term\_b>}].

<term\_b>::=<ident>|<const\_b>|<factor\_b>.

<factor\_b>::=<term\_b> [{ |<term\_b> | <and> |<not>|<or>}].

<and>::= [{ And <term\_b>}]

<or>::= [{ Or <term\_b>}]

<not>::= Not <term>

<for>::= ::= For (<ident><is><ident>|<number>; Downto <ident>|<number>;) Do.

<get> ::= get(<ident>); .

<put> ::= put(<expression\_i>|<string>); .

<string> ::= “<l\_or\_n>[{<l\_or\_n>}]” .

* 1. Опис термінальних символів та ключових слів

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

Startprogram

Startblock

Variable

Int32\_t

Endblock

Get

Put

For

Downto

Do

(

)

;

,

::=

Add

Sub

Mul

Div

Mod

=

<>

>

<

Not

And

Or

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 кілька таблиць:

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

Lexema LexTable[MAX\_LEXEMS]; // Таблиця лексем

typedef struct Lexem

{

char name[50]; // Ім’я

int value; // Значення

int line; // Рядок

TypeOfLex type; // Тип

}Lexema;

1. Таблиця лексичних класів

enum TypeOfLex

{

LProgram, // Startprogram

LStart, // Startblock

LData, // Variable

LVarType, // Int32\_t

LIdentifier, // NameVar

LEnd, // Endblock

LScan, // Get

LPrint, // Put

LFor, // For

LDownTo, // Downto

LDo, // Do

LNewValue, // ::=

LAdd, // Add

LSub, // Sub

LMul, // Mul

LDiv, // Div

LMod, // Mod

LEqu, // =

LNotEqu, // <>

LGreate, // >

LLess, // <

LNot, // Not

LAnd, // And

LOr, // Or

LLBraket, // (

LRBraket, // )

LNumber, // number

LSeparator, // ;

LComma, // ,

LEOF, // EndFile

LUnknown

};

1. Таблиця ідентифікаторів з елементами типу Identificator та додатковою змінною цілочисельного типу в якій зберігається кількість визначених змінних. Структура елементів така:

Identificator IdTable[MAX\_IDENT]; //Таблиця ідентифікаторів

typedef struct ID

{

char name[50]; // Ім’я

int value; // Кількість

}Identificator;

* 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 своєї програми я вибрав другий варіант. Тобто, спочатку буду виконувати фазу лексичного аналізу. Результатом цієї фази буде файл з списком лексем. Але лексеми записуються у файл не як послідовність символів. Кожній лексемі присвоюється унікальний номер. Цей номер і записується у файл. Такий підхід дозволяє спростити роботу синтаксичного аналізатора.

Також на етапі лексичного аналізу виявляються 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. Опис 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).

При виділенні лексеми вона розпізнається та записується у таблицю лексем за допомогою відповідного номера 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 номера 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 текучої позиції аналізу. Також в таблиці лексем ведуться записи, щодо рядка відповідної лексеми – для місця помилки – та додаткова інформація.

При лексичному аналiзі виявляються i виявляються лексичнi помилки (наприклад, недопустимi символи i неправильнi iдентифiкатори). Лексична фаза вiдкидає також i коментарi, оскiльки вони не мають нiякого впливу на виконання програми, отже й на синтаксичний розбір та генерацію коду.

Лексичний аналізатор (сканер) не обов’язково обробляє всю програму до початку всіх інших фаз. Якщо лексичний аналіз не виділяється як окрема фаза компіляції, а є частиною синтаксичного аналізу, то лексична обробка тексту програми виконується по мірі необхідності по запиту синтаксичного аналізатора.

На вході лексичного аналізатора є текстовий файл, який містить текст програми для аналізу. Для зчитування з файлу наступної лексеми, створюються змінні: ch (містить перший символ лексеми), c (містить другий символ лексем, використовується у випадку, коли лексема складається більше ніж з 1 символа), buf (містить набір символів, які формують лексему, використовується коли лексема є рядком символів).

Перед початком виділення кожної наступної лексеми з файлу зчитується символ у змінну ch. Маючи у магазині перший символ, аналізатор намагається виділити лексему, початок якої збігається з символом у змінній. Якщо співпадіння має місце, з файлу зчитується наступний символ. Після чого, лексичний аналізатор перевіряє чи вибрана лексема на першому (попередньому) кроці підходить під отриманий опис. Якщо підходить, то з вхідного файлу зчитується ще один символ. Це відбувається до тих пір, доки вхідний ланцюжок символів не буде відповідати одному з описів лексем. Якщо на певному кроці, сформована послідовність символів не відповідає лексемі, яка була вибрана, лексичний аналізатор повертає зчитаний символ у файл і намагається знову підставити вхідний ланцюжок символів під один з описів лексем. Якщо це знову не вдалося, в вхідний файл повертається ще один символ. Так відбувається доки вхідний ланцюжок символів не буде ідентичним з описом лексеми. Якщо набір вхідний символів не співпадає з жодним описом лексем, то аналізатор повертає повідомлення про помилку.

Сформувавши лексему, аналізатор записує інформацію про неї у таблицю лексем. У цій таблиці зберігається інформація про тип лексеми (числове значення), рядок, у якому вона знаходиться, її величина (для цифр і змінних).

Зробивши запис у таблиці лексем, аналізатор заново ініціалізує змінні, та намагається зчитати з файлу наступну лексему. Аналіз завершується, коли аналізатор зчитує з вхідного файлу символ, що відповідає кінцю файлу.

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



Рис. Блок-схема роботи лексичного аналізатора

* 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 вхідного 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.

Розпізнавач дає відповідь на питання про те, належить або немає ланцюжок вхідних символів заданому мови. Однак, як у випадки лексичного аналізатора, завдання синтаксичного розбору не обмежується тільки перевіркою приналежності ланцюжка заданому мови. Синтаксичний аналізатор виконує ще ряд важливих функцій і вже не є різновидом МП-автомата - його функції можна трактувати ширше. Синтаксичний аналізатор повинен мати якийсь вихідний мова, за допомогою якого він передає наступним фаз компіляції всю інформацію про знайдені і розібраних синтаксичних структурах. У такому випадки він вже є перетворювачем з магазинної пам'яттю - МП-перетворювачем.

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.

Виходом лексичного аналізатора є таблиця лексем. Ця таблиця утворює вхід лексичного аналізатора, який досліджує тільки один компонент кожної лексеми - її тип. Інша інформація про лексеми використовується на більш пізніх фазах компіляції при семантичному аналізі, підготовці до генерації та генерації коду результуючої програми. Синтаксичний аналіз - це процес, в якому досліджується таблиця лексем і встановлюється, чи задовольняє вона структурним умовам, явно сформованим у визначенні синтаксису мови.

Синтаксичний аналізатор сприймає вихід лексичного аналізатора і розбирає його відповідно до граматикою вхідного мови. Однак у граматиці вхідного мови програмування звичайно не уточнюється, які конструкції слід вважати лексемами. Прикладами конструкцій, які звичайно розпізнаються під час лексичного аналізу, служать ключові слова, константи і ідентифікатори. Але ці ж конструкції можуть розпізнаватися і синтаксичним аналізатором. На практиці немає жорсткого правила для того, щоб визначити, які конструкції повинні розпізнаватися на етапі лексичного, а які на етапі синтаксичного аналізу. Зазвичай це визначає розробник компілятора виходячи з технологічних аспектів програмування, а також синтаксису та семантики вхідного мови.

В даній курсовій роботі 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. Перевірка вхідного ланцюга лексем перевіряється методом рекурсивного спуску.

Даний аналізатор працює до першої виявленої помилки, тобто при знаходженні першої помилки перевірка одразу припиняється і виводиться повідомлення користувачу про місце її знаходження.

* 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 (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 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. Опис генератора коду

У компілятора, реалізованого в даній курсовій роботі, 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, зміні, 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 своє роботи, генератор формує код завершення асемблерної програми.



Рис. Блок схема генератора коду

1. Опис програми

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

#define MAX\_LEXEMS 1000

#define MAX\_IDENT 100

#define MAX\_BUF\_SIZE 100

#define STACK\_SIZE 200

#define MAX\_LENGTH\_TYPES 20

enum TypeOfLex

{

LProgram, // Program

LProgramName, // ProgramName

LStart, // Start

LData, // Var

LVarType, // Int\_4

LIdentifier, // NameVar

LEnd, // Finish

LScan, // Get

LPrint, // Put

LFor, // For

LTo, // To

LDo, // Do

LNewValue, // >>

LAdd, // ++

LSub, // --

LMul, // \*\*

LDiv, // Div

LMod, // Mod

LEqu, // Eg

LNotEqu, // Ne

LGreate, // >=

LLess, // <=

LNot, // !

LAnd, // &

LOr, // |

LLBraket, // (

LRBraket, // )

LNumber, // number

LSeparator, // ;

LComma, // ,

LEOF, // EndFile

LUnknown

};

//DATA

typedef struct Lexem

{

char name[50];

int value;

int line;

TypeOfLex type;

}Lexema;

typedef struct ID

{

char name[50];

int value;

}Identificator;

typedef struct L

{

Lexema LexTable[MAX\_LEXEMS]; //Таблиця лексем

int LexNum; //

Identificator IdTable[MAX\_IDENT]; //Таблиця ідентифікаторів

int IdNum; //

bool IsPresentInput, //Ознаки присутності операторів

IsPresentOutput,

IsPresentMod,

IsPresentAnd,

IsPresentOr,

IsPresentNot,

IsPresentEqu,

IsPresentGreate,

IsPresentLess;

int numberErrors;

char InputFileName[50];

char OutputFileName[50];

int bufExprPostfixForm[MAX\_BUF\_SIZE]; //Буфер для виразу в постфіксній формі

}DataType;

StackT Stack; //Стек

StackT startStack;

StackT forStack;

DataType Data; //Структура-реєстр в якій зберігаються всі дані програми

FILE\* InF, \* OutF, \* ErrorF;

В програмі також використовуються багато функцій тому вона є доволі структурованою:

int main(int argc, char\* argv[]) – Основна підпрограма. З неї починає своє виконання програма. В ній прямо чи опосередковано викликаються наступні функції:

int AnalisisLexems(FILE\* FIn) - Аналізує вхідний файл і розбиває його на лексеми

Lexema\* GetNextLexem(FILE\* f, int ii) - Відділяє наступну лексему

void PrintLexemsInFile() - Друкує лексеми у файл

int ErrorChecking() - функція пошуку помилок у вхідному файлі. Повертає їх кількість

bool IsOperation(TypeOfLex t) – Повертає true якщо лексема є операцією

int IsExpression(int i, FILE\* ef) – Перевірка, чи є дана послідовність лексем виразом

int Balans(int nom, TypeOfLex ends, TypeOfLex ltStart, TypeOfLex ltFinish) – Визначає баланс дужок

void GenerateCode(FILE\* f) - за таблицями ідентифікаторів та лексем генерує вихідний код на мові асемблера

void BeginASMFile(FILE\* fout) - записує до вихідного файлу стандартну „шапку”

void CheckPresent() - Перевіряє наявність операторів

void PrintData(FILE\* f) - Друкує сегмент даних

void BeginCodeSegm(FILE\* fout) - Друкує початок сегменту коду

void PrintCode(FILE\* f) – Видруковує в вихідний файл основний код програми

int ConvertToPostfixForm(int i) - Перетворює вирази в постфіксну форму

bool Prioritet(TypeOfLex t, StackType s) - Перевіряє пріорітет операцій (використовується при перетворенні виразу у посифіксну форму).

void GenASMCode(const char\* str, FILE\* f) - Генерує код для окремих операцій

void PrintEnding(FILE\* f) - Видруковує в вихідний файл стандартне завершення \*.asm-файлу

void PrintAND(FILE\* f) - Видруковує в вихідний файл процедуру And\_

void PrintOR(FILE\* f) - Видруковує в вихідний файл процедуру Or\_

void PrintNOT(FILE\* f) - Видруковує в вихідний файл процедуру Not\_

void PrintEQ(FILE\* f) - Видруковує в вихідний файл процедуру Eq\_

void PrintGE(FILE\* f) - Видруковує в вихідний файл процедуру Greate\_

void PrintLE(FILE\* f) - Видруковує в вихідний файл процедуру Less\_

void PrintInput(FILE\* f) - Видруковує в вихідний файл процедуру вводу

void PrintOutput(FILE\* f) - Видруковує в вихідний файл процедуру виводу

* 1. Опис інтерфейсу та інструкція користувачеві

Вхідним файлом для даної програми є звичайний текстовий файл з розширенням c20. 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\_C20.exe <ім’я програми>.c20"

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>.asm.

Для отримання виконавчого файлу необхідно скористатись програмою Masm32.exe

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 в подальшому виправити слабкі сторони, якщо вони є.

Відлагодження даної програми здійснюється за допомогою набору 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***

Startprogram

Variable Int32\_t \_A,\_B,\_CI,\_D;

Startblock

\_A::=4;

Get(\_Bi);

Geit(\_CI);

Put(\_CI

\_D::=\_B;

For(\_A::=10; Downto 2;) Do

Startblock

Put(\_A);

Endblock

Put(\_B Mul \_D);

Put(\_B);

Put(\_A Add \_B);

Put(1 And 0 Or 0);

Endblock

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

List of bugs in the program

=======================================================================

line 5: Identifier expected!

line 5: Unknown identifier!(\_Bi)

line 6: Unknown identifier!(Geit)

line 7: After identifier must be ')' or ';' or operation!

line 8: After identifier must be ')' or ';' or operation!

line 7: ')' expected!

* 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***

Startprogram

Variable Int32\_t \_A,\_B,\_CI,\_D;

Startblock

\_A::=4;

Get(\_B);

Get(\_CI);

Put(\_CI);

\_D::=\_B;

For(\_A::=10; Downto 2;) Do

Startblock

Put(\_A);

Endblock

Put(\_B Mul \_D);

Put(\_B);

Put(\_A Add \_B);

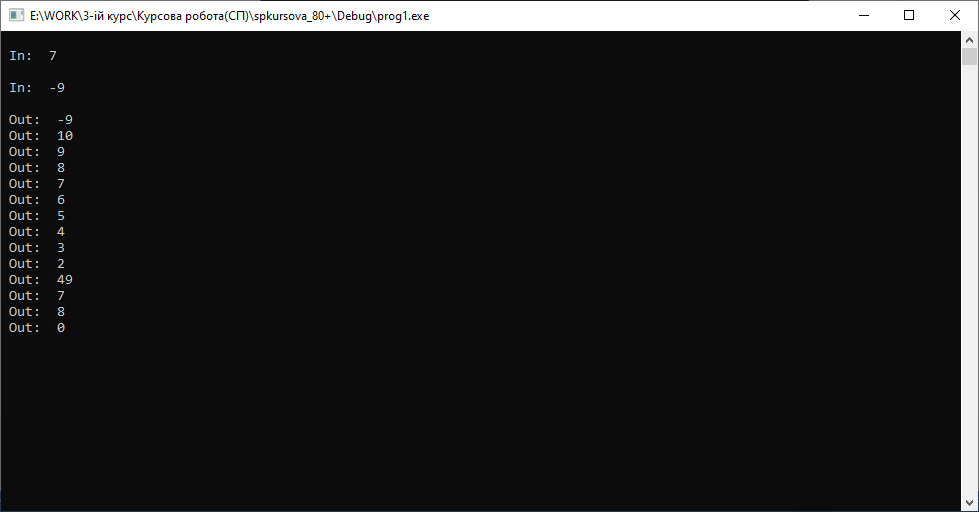
Put(1 And 0 Or 0);

Endblock

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

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

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



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

1. Висновки

В ході виконання курсової роботи я виконав наступні завдання:

1. Склав формальний опис мови програмування C20, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2. Створив компілятор мови програмування C20, а саме:

* Розробив прямий лексичний аналізатор, орієнтований на розпізнавання лексем, що є заявлені в формальному описі мови програмування.
* Розробив синтаксичний аналізатор на основі висхідного методу. Складено деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура.
* Розробив генератор коду, відповідні процедури якого викликаються після перевірки синтаксичним аналізатором коректності запису чергового оператора, мови програмування C20. Вихідним кодом генератора є програма на мові Assembler(x86).

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

* На виявлення лексичних помилок.
* На виявлення синтаксичних помилок.
* Загальна перевірка роботи компілятора.

В процесі тестування не було виявлено помилок в роботі компілятор, всі помилки в тестових програмах на мові C20 були успішно виявлені і відповідно опрацьовані.

В результаті виконання даної курсової роботи я засвоїв методи розробки та реалізації компонент систем програмування.

1. Список використаних джерел
2. Шильдт Г. С++. – Санкт-Петербург: BXV, 2002. – 688 с.
3. Хантер Р. Проектирование и конструирование компиляторов. – М.: Финансы и статистика, 1984. – 232 с.
4. Шамис В. А. С++Builder 4 Техника визуального программирования. – М.: Нолидж, 2000. – 656 с.
5. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978. – т.1, 612 с.
6. Серебряков В.А. Лекции по конструированию компиляторов. – М.: МГУ, 1993.
7. В.С. Проценко, П.Й. Чаленко, А.Б.Ставровський “Техніка програмування мовою Сі”.-Київ “Либідь”, 1993. – 224с.
8. Б.Керниган, Д.Ритчи “Язык программирования Си”. – Москва “Финансы и статистика”, 1992. – 271с.
9. Н.Г.Голубь “Исскуство программирования на асемблере.Лекции и упражнения”. –Киев “DiaSoft”, 2002 – 642с.
10. Л.Бэк “Введение в системное програмирование”. – М.: Мир, 1999
11. Додатки

**Додаток А (Код на мові Асемблер)**

.386

.model flat, stdcall

option casemap :none

include \masm32\include\windows.inc

include \masm32\include\kernel32.inc

include \masm32\include\masm32.inc

include \masm32\include\user32.inc

include \masm32\include\msvcrt.inc

includelib \masm32\lib\kernel32.lib

includelib \masm32\lib\masm32.lib

includelib \masm32\lib\user32.lib

includelib \masm32\lib\msvcrt.lib

CHECK\_DIVISION\_BY\_ZERO MACRO

fldz

fcomp

fstsw ax

sahf

jne end\_check

invoke WriteConsoleA, hConsoleOutput, ADDR ErrorMessage, SIZEOF ErrorMessage, ADDR NumberOfCharsWrite, 0

jmp exit\_label

end\_check:

ENDM

.DATA

;===User Data==============

\_A dd 00h

\_B dd 00h

\_CI dd 00h

\_D dd 00h

;===Addition Data===========================================

hConsoleInput dd ?

hConsoleOutput dd ?

ErrorMessage db "Error: division by zero", 0

InputBuf db 15 dup (?)

InMessage db "In: ", 0

OutMessage db "Out: "," %d",0

ResMessage db 20 dup (?)

NumberOfCharsRead dd ?

NumberOfCharsWrite dd ?

msg1310 dw 13,10

buf dd 0

lb1 dd ?

lb2 dd ?

.CODE

start:

invoke AllocConsole

invoke GetStdHandle, STD\_INPUT\_HANDLE

mov hConsoleInput, EAX

invoke GetStdHandle, STD\_OUTPUT\_HANDLE

mov hConsoleOutput, EAX

finit

mov dword ptr buf,4

fild buf

fistp \_A

call Input

fild buf

fistp \_B

call Input

fild buf

fistp \_CI

fild \_CI

fistp buf

call Output

fild \_B

fistp \_D

mov dword ptr buf,10

fild buf

fistp \_A

forStart1:

mov dword ptr buf,2

fild buf

fistp buf

fild buf

fild \_A

call Less\_

call Not\_

fistp buf

mov eax, dword ptr buf

cmp eax, 0

jne forFinish1

fild \_A

fistp buf

call Output

fild \_A

fld1

fsub

fistp \_A

jmp forStart1

forFinish1:

fild \_B

fild \_D

fmul

fistp buf

call Output

fild \_B

fistp buf

call Output

fild \_A

fild \_B

fadd

fistp buf

call Output

mov dword ptr buf,1

fild buf

mov dword ptr buf,0

fild buf

call And\_

mov dword ptr buf,0

fild buf

call Or\_

fistp buf

call Output

exit\_label:

invoke WriteConsoleA,hConsoleOutput,ADDR msg1310,SIZEOF msg1310,ADDR NumberOfCharsWrite,0

invoke ReadConsoleA,hConsoleInput,ADDR InputBuf,1,ADDR NumberOfCharsRead,0

invoke ExitProcess, 0

;===Procedure Input==========================================================================

Input PROC

invoke WriteConsoleA,hConsoleOutput,ADDR msg1310,SIZEOF msg1310,ADDR NumberOfCharsWrite,0

invoke WriteConsoleA,hConsoleOutput,ADDR InMessage,SIZEOF InMessage,ADDR NumberOfCharsWrite,0

invoke ReadConsoleA,hConsoleInput,ADDR InputBuf,11,ADDR NumberOfCharsRead,0

invoke crt\_atoi, addr InputBuf

mov dword ptr buf, eax

ret

Input ENDP

;============================================================================================

;===Procedure Output=========================================================================

Output PROC

invoke WriteConsoleA,hConsoleOutput,ADDR msg1310,SIZEOF msg1310,ADDR NumberOfCharsWrite,0

invoke wsprintf, addr ResMessage, addr OutMessage, dword ptr buf

invoke WriteConsoleA,hConsoleOutput,ADDR ResMessage,eax,ADDR NumberOfCharsWrite,0

ret

Output ENDP

;============================================================================================

;===Procedure And=========================

And\_ PROC

push eax

push edx

pushf

fistp lb1

fistp lb2

mov eax,lb1

cmp eax,0

jnz and\_t1

jz and\_false

and\_t1:

mov eax,lb2

cmp eax,0

jnz and\_true

and\_false:

fldz

jmp and\_fin

and\_true:

fld1

and\_fin:

popf

pop edx

pop eax

ret

And\_ ENDP

;=========================================

;===Procedure Or==========================

Or\_ PROC

push eax

push edx

pushf

fistp lb1

fistp lb2

mov eax,lb1

cmp eax,0

jnz or\_true

mov eax,lb2

cmp eax,0

jnz or\_true

fldz

jmp or\_fin

or\_true:

fld1

or\_fin:

popf

pop edx

pop eax

ret

Or\_ ENDP

;=========================================

;===Procedure Not=========================

Not\_ PROC

push eax

pushf

fistp lb1

mov eax,lb1

cmp eax,0

jne not\_false

fld1

jmp not\_fin

not\_false:

fldz

not\_fin:

popf

pop eax

ret

Not\_ ENDP

;=========================================

;===Procedure Less========================

Less\_ PROC

push eax

push edx

pushf

fistp lb1

fistp lb2

mov eax, lb1

mov edx, lb2

cmp edx,eax

jg gr

lo:

fld1

jmp less\_fin

gr:

fldz

less\_fin:

popf

pop edx

pop eax

ret

Less\_ ENDP

;=========================================

end start