Івано-Франківський національний технічний університет нафти і газу

В. Б. Кропивницька, Т. В. Гуменюк

СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

ЛАБОРАТОРНИЙ ПРАКТИКУМ
Частина 2

2012

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ,
МОЛОДІ ТА СПОРТУ УКРАЇНИ**

**Івано-Франківський національний технічний
університет нафти і газу**

Кафедра комп'ютерних систем і мереж

В. Б. Кропивницька, Т. В. Гуменюк

СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

ЛАБОРАТОРНИЙ ПРАКТИКУМ
Частина2
Для студентів напряму підготовки
6.050102 - «Комп'ютерна інженерія»

Рекомендовано методичною радою університету

Івано-Франківськ
2012

УДК 004.45
ББК 32.973.23
К83

Рецензент: доц. кафедри комп'ютерних систем і мереж Заячук Я. І.

Рекомендовано методичною радою університету
(протокол № від .2012 р).

Кропивницька В. Б.
К 83 Системне програмне забезпечення: лабораторний практикум / В. Б. Кропивницька, Т. В. Гуменюк – Івано-Франківськ: ІФНТУНГ, 2012. – с.

МВ _____ – _____ – 2012

Розроблений відповідно до робочої програми навчальної дисципліни.

Лабораторний практикум містить методичні вказівки для проведення лабораторних занять з дисципліни «Системне програмне забезпечення». Призначений для підготовки бакалаврів напрямку 6.050102 - «Комп'ютерна інженерія».

Голова навчально-методичного об'єднання спеціальності «Комп'ютерні системи і мережі»	М. І. Горбійчук
Завідувач кафедри комп'ютерних систем і мереж	М. І. Горбійчук
Член експертно-рецензійної комісії університету	М. Й. Федорів
Нормоконтролер	Г. Я. Онуфрик
Інженер І категорії	Л. З. Костюк

МВ _____ – _____ – 2012

УДК 004.45
ББК 32.973.23

УДК 004.45
ББК 32.973.23
К 83

Рецензент: доц. кафедри комп'ютерних систем і мереж Заячук Я.І..

Рекомендовано методичною радою університету
(протокол № від .2012 р).

Кропивницька В. Б.
К 83 Системне програмне забезпечення: лабораторний практикум / В. Б. Кропивницька, Т. В. Гуменюк – Івано-Франківськ: ІФНТУНГ, 2012. – __ с.

МВ _____ – _____ – 2012

Розроблений відповідно до робочої програми навчальної дисципліни.
Лабораторний практикум містить методичні вказівки для проведення лабораторних занять з дисципліни «Системне програмне забезпечення». Призначений для підготовки бакалаврів напрямку 6.050102 - «Комп'ютерна інженерія».

МВ _____ – _____ – 2012

УДК 004.45
ББК 32.973.23

ЗМІСТ

ВСТУП

Практикум містить другу частину лабораторних робіт курсу “Системне програмування”. В перших роботах детально описуються дії при проведенні роботи. В подальшому детальність опису порядку виконання роботи зменшується, зважаючи на досвід, отриманий студентами при виконанні попередніх робіт.

В даній частині лабораторного практикуму розглядається етапи створення трансляторів та компіляторів. У всіх лабораторних роботах пропонується написати програму (відповідно до обраного варіанту).

До лабораторних занять студент повинен попередньо підготуватися, використовуючи рекомендовану літературу.

В результаті проведення робіт студент повинен закріпити отримані теоретичні знання, а також відповідним чином оформити звіт з матеріалами, отриманими при дослідженні складеної програми, який повинен містити:

- титульний листок;
- завдання на лабораторну роботу;
- короткий теоретичний опис використаного методу та функцій;
- блок-схему та опис функціонування програми;
- тексти програм (C++, Pascal, Basic, тощо);
- висновок.

Текст програми повинен містити коментарі та пояснення до основних функціональних блоків та частин програми.

До кожного завдання повинен бути сформований висновок у відповідності з вимогами, що ставляться до виконання роботи.

ЛАБОРАТОРНА РОБОТА №1-2. ПОДАННЯ РЕГУЛЯРНИХ ТА КОНТЕКСТНО-ВІЛЬНИХ ГРАМАТИК З У ФОРМІ БЕКАСА-НАУРА. ОБРОБКА ЛАНЦЮГІВ ВИВОДУ ДЛЯ КОНТЕКСТНО-ВІЛЬНИХ ГРАМАТИК З ВИКОРИСТАННЯМ ТЕКСТОВИХ ФАЙЛІВ.

Мета роботи: Навчитися працювати з текстовими файлами. Створювати, редагувати, об'єднувати текстові файли.

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Файлом називається будь-яка інформація, записана в зовнішню (магнітну) пам'ять. Обмін даними між оперативною та магнітною пам'яттю відбувається через спеціально відведену для цього частину пам'яті, яка називається буферною.

При використанні файлів обов'язковими атрибутами програми є: виділення пам'яті для дескриптору файлу, функції відкриття та закриття файлу, а також функції вводу/виводу. Виділення пам'яті для дескриптору можна виконати наступним чином:

FILE *fl,*f2,…,*fn;

де `*fl,*f2,…,*fn` – покажчики на відповідні блоки пам'яті. Кожному файлу відповідає блок пам'яті (дескриптор), який можна отримати через відповідний покажчик, така відповідність встановлюється в момент відкриття файлу.

Деякі функції роботи з файлами (бібліотеки **stdio.h**, **stdlib.h**):

f1=FOPEN(”**ім'я файлу**”, ”**тип**”)**;** – відкриття файлу з іменем (”**ім'я файлу**”) і отримання його дескриптору (**f1**). Ім'я файлу складається з назви та розширення, розділених крапкою. Назва може мати до 8-ми символів. Розширення має довжину до трьох символів і не є обов'язковим.. Для задавання типу служать символи:

r – зчитування з файлу;

w – запис у файл;

a – доточування файлу.

Функція повертає значення **NULL**, якщо при відкритті виникла помилка.

FCLOSE(f1,…,fn); – закриття файлів. В момент закриття відбувається запис у файл залишку буферної пам'яті, якщо він був відкритий для запису.

FSEEK(f1,n,k); – в режимі прямого доступу до файлу, встановлює покажчик поточної позиції, на яку вказує **f1**. При вводі/виводі контролюється поточний номер байта, який саме вводитися чи виводиться, покажчиком (лічильником номера байта в файлі, не путати з покажчиком на файл). Параметр **n** визначає на скільки байтів слід змістити цей покажчик. Параметр **k** задає початкову позицію, від якої потрібно перемістити покажчик і може приймати такі значення:

1 – від поточної позиції;

2 – від кінця файлу.

a=FGETC(f1); – читання одного символа з файлу, на який вказує **f1**, присвоєння його значення змінній **a** типу **int** або **char**.

FPUTC(a,f1); – запис у фай із вказувачем **f1** одного символу.

FREAD(*r,size,N,f1); – читання з файлу на який вказує **f1**.

FWRITE(*r,size,N,f1); – занесення даних у файл.

FSCANF(f1, ”сф”,VarVar); – форматований ввід, читання з файлу даних, перетворення їх типів відповідно до списку форматів (”**сф**”) і занесення у відповідну змінну (**VarVar**).

FPRINTF(f1, ”сф”,VarVar); – форматований вивід, занесення в файл змінних, заданих списком **VarVar**.

Всі функції читання файлу повертають значення **EOF** або **NULL**, якщо був досягнений кінець файлу.

Незалежно від типів даних, занесених у файл різними засобами, на магнітні носії записуються символьні дані в ASCII форматі.

Для отримання і обробки параметрів з командного рядка використовуються наступні змінні:

int argc – цілого типу, містить кількість параметрів, що було передано програмі в момент запуску;

char **argv – посилання на покажчики символьного типу, містить усі передані в програму параметри, як масив символьних рядків.

Кожна програма має, як мінімум, один вхідний параметр. Це пристрій і шлях звідки вона була запущена.

В наведеному нижче тексті, програма виводить номер параметру і параметр, який був переданий програмі в момент запуску з командного рядка.

```
#include <conio.h>
int main(int argc, char **argv)
   {
     int i;
     clrscr();
     for(i=0;i<argc;i++)
        cprintf("%i -> %s\r\n",i,*(argv+i));
   }
```

Для більшого розуміння рядок програми:

```
cprintf("%i -> %s\r\n",i,*(argv+i));
```

можна замінити на аналогічний за призначенням:

```
cprintf("%i -> %s\r\n",i,argv[i]);
```

Необхідно пам'ятати, що елемент масиву ***(argv+i)** чи **argv[i]** – це символи, тобто для перетворення їх в числовий вигляд необхідно здійснювати додаткові операції.

ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Написати програму на мові С++ (чи іншій за згодою викладача) яка виконує вказані операції з файлом(и) відповідно до заданого варіанту, таблиця 1.1.

Таблиця 1.1.

Варіант	Завдання
1	Об'єднує два заданих в командному рядку файли і результат зберігає в третьому файлі, теж заданому в командному рядку.
2	Підраховує кількість рядків у файлі. Назва файлу задається в командному рядку. Кількість рядків вивести на екран.
3	Здійснює заміну опорного слова на інше у заданому файлі. Результат зберегти в тому ж файлі. Назва файлу та слово, яке підлягає заміні і слово, на яке здійснюється заміна, задаються в командному рядку.
4	Для заданого в командному рядку файлі видалити всі рядки, які починаються із "*". Результат зберегти в тому ж файлі.
5	В командному рядку задається назва файлу зі словами. Кожне слово знаходиться на окремому рядку. Необхідно відсортувати слова по алфавіту. Результат зберегти в тому ж файлі.
6	В командному рядку задається назва файлу. Підрахувати кількість слів у файлі. Результат вивести на екран.

7	Для заданого в командному рядку файлу здійснити нумерацію рядків. Формат запису <номер рядка>: <пропуск> (наприклад для п’ятого рядка ”135dsf” результат буде ”5: 135dsf”). Результат зберегти в тому ж файлі.
8	В командному рядку дано файл та список слів, розділених пропусками. Необхідно в заданому файлі знайти всі перераховані в командному рядку слова та охопити їх квадратними дужками [ ]. Результат зберегти в тому ж файлі.
9	Підрахувати кількість символів у файлі. Символи з кодами ≤32, управляючі та пробіли не рахувати. Результат вивести на екран.
10	Для заданого в командному рядку файлі вилучити всі надлишкові пропуски і результат записати в тому ж файлі. (Надлишковими є символи пропуску, табуляції та переходу на новий рядок, коли їх знаходиться більше одного підряд.)
11	Підрахувати кількість одно-, два- і т.д. та десятибуквенних слів у заданому командному рядку файлі. Результат вивести на екран.
12	Підрахувати кількість чисел у файлі, заданому в командному рядку. Числом вважати послідовність символів, яка починається з цифри і не містить інших символів, крім цифр. Результат вивести на екран.
13	Зробити заміну всіх великих букв у файлі на маленькі. Результат зберегти в тому ж файлі.
14	Підрахувати кількість дужок [, { і (у файлі та порівняти, чи їх кількість дорівнює кількості дужок },] і). Загальну кількість дужок та результат порівняння для кожного виду дужок вивести на екран.
15	Перевірити чи два заданих файли однакові. У випадку виявлення відмінностей, вказати в якому рядку та позиції виявлена відмінність. Результати вивести на екран.
16	Заданий в командному рядку файл розділити на декілька за наступними правилами: - якщо в рядку виявлена послідовність символів, що відповідає числам від 340 до 360, то рядок зберегти в окремому файлі із назвою заданого файлу та розширенням, що відповідає заданій послідовності; - якщо в рядку знайдено кілька послідовностей, то використати ту, що зустрічається першою від початку рядка; - якщо рядок не містить жодної з послідовностей, то він пропускається. Наприклад, для файлу abc.txt : 1-ий рядок: ит 53145 → пропустити 2-ий рядок: ро345346 350 → зберегти у файл abc.345 3-ій рядок: ап 12 е30310350123 → зберегти у файл abc.350

2. Здійснити компіляцію програми.

3. Оформити звіт в згідно вимог.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке дескриптор файлу і для чого він потрібен?
2. Чим відрізняються покажчик позиції у файлі і покажчик на файл?
3. Що таке прямий доступ до файлу і як він здійснюється?
4. Чи можна файл, який містить дані літерного і числового типів і створений функцією посимвольного вводу, читати за допомогою інших функцій? Чи потрібні будуть тут якісь перетворення?

ЛАБОРАТОРНА РОБОТА №3. ОСНОВИ ПРЕПРОЦЕСІНГУ

Мета роботи: Навчитися працювати з деректівами файлів, обробляти коментарі.

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Препроцесорна обробка – це попередня фаза трансляції, що здійснює обробку тексту програми не вдаючись в її зміст (фаза трансляції на рівні перетворення вихідного тексту програми). Під час виконання цієї фази відбувається заміна одних частин програми на інші при цьому сама програма залишається у вихідному (текстовому) вигляді. Для прикладу на мові С++ директиви процесора виділяються окремими символами.

#define <ідентифікатори> <рядок тексту>

Дана директива забезпечує заміну по тексту програми ідентифікатора на відомий рядок тексту, що найчастіше використовують для позначення констант:

До препроцесорної обробки

```
#define size 80
int a[size]
for (i=0;i<size;i++)
     {
       Ξ
     }
```

Після препроцесорної обробки

```
int a[80]
for (i=0;i<80;i++)
     {
       Ξ
     }
```

Крім того директива дещо схожа на визначення (оголошення) функції з формальними параметрами, де замість коду функції (тіла функції) використовують рядок тексту:

#define <ідентифікатори (параметри)> <рядок з параметрами>

Якщо препроцесор знаходить по тексту програми вказаний ідентифікатор зі списком фактичних параметрів в дужках він здійснює підстановку (заміну) і підставляє замість цього рядка, рядок із дерективи define із заміною в рядку формальних параметрів на фактичні.

Основна відмінність такого запису від функції полягає в тому, що функція реалізує описані дії, підстановку параметрів і виклик під час роботи програми, а препроцесор їх здійснює ще до трансляції. Крім того дерективи такого виду дозволяють оформляти таким чином будь-яку частину програми незалежно від того чи це закінчена частина мови чи її фрагмент.

Таблиця 3.1.

До препроцесорної обробки	Після препроцесорної обробки

#define for(i,n)
 	
1.1.1.1.1.1 Ξ
for (k,20) a[k]=0
	for(i=0;i,n;i++)
	Ξ
	for (k=0;k<20;k++) a[k]=0
	Ξ

Ξ
for (j,m+2)
{
Ξ
}
	for (j=0;j<m+2;j++)
	{
	Ξ
	}

В даному випадку директива **define** представляє собою макровизначення. Крім того в препроцесорній обробці використовують макропідстановки, які здійснюють заміну по тексту програми ідентифікатора з параметрами на відповідний рядок.

#include <ім'я файлу>

В даному випадку в текст програми замість вказаної дерективи включається (підставляється) текст відповідного файлу, що знаходиться в системні або явно вказаній директорії. Найчастіше такі файли містять необхідну для транслятора інформацію про зовнішні функції, що знаходяться в інших об'єктних модулях і бібліотеках

#include <ім'я файлу>
#include <conio.h>
#include <stdio.h>

В процесі обробки даних деректив препроцесор включає в текст програми текст заголовних файлів, які містять оголошення (описи) функцій бібліотеки. Аналогічні засоби в інших мовах програмування називають макропроцесором і відповідно макрозасобами.

ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Згідно заданого варіанту (таблиця 2.2) виконати відповідні операції з файлом, ім'я якого задається в командному рядку, а результат зберегти у файлі із такою ж назвою, як у вхідного але розширенням ".tmp". Мова програмування С++ (чи інша за згодою викладача).

Таблиця 2.2.

Варіант	Завдання
1	Виконати включення всіх файлів, описаних за допомогою директиви включення файлу #include <назва файлу> або #include "назва файлу". Якщо файл не знайдено, то видати повідомлення про помилку у вхідному файлі, вказавши номер рядка, та файл, що не вдалося відкрити. При цьому роботу по формуванню результуючого файлу не переривати.
2	Виключити з файлу коментарії, які починаються із '//'.
3	Виключити із файлу коментарії, які взяті у символи '/*' та '*/'. Вкладені коментарії не дозволені.
4	Виключити із файлу коментарії, які взяті у символи '/*' та '*/'. Дозволено використання вкладених коментаріїв.
5	Виключити із файлу коментарії, які починаються із '*' в першій позиції рядку, або '&&' в любій позиції рядка до кінця рядка.
6	Виключити із файлу коментарії, які взяті у фігурні дужки '{' та '}'. В командному рядку передбачити параметр, який дозволяє використання вкладених коментарів.
7	Виключити із файлу коментарі, які починаються із REM або знака апостроф, перед якими можуть знаходитися тільки пробіли та закінчуються із кінцем рядка.
8	Здійснити заміну у відповідності до директиви #define (C/C++). Відомо, що у файлі використовуються тільки прості define, які задають заміну ідентифікатора деяким значенням.
9	Здійснити заміну відповідно до директиви EQU ассемблера.
10	Залишити в файлі один із варіантів тексту, що обмежений конструкцією

	виду: #ifdef VARIANT1 <текст 1> #else <текст 2> #endif Причому у файлі повинен залишитися <текст 1>, якщо до цієї конструкції зустрічався рядок #define VARIANT1, та <текст 2> в іншому випадку.
11	Здійснити заміну знайдених ідентифікаторів __LINE__ та __FILE__ відповідно на номер рядка, в якому зустрівся ідентифікатор __LINE__ та назву файлу, що обробляється, у подвійних лапках.
12	У випадку виявлення директиви #error <текст> завершити обробку файлу, та вивести на екран текст, який вказаний після цієї директиви.
13	Залишити в файлі один із варіантів тексту, що обмежений конструкцією виду: #ifndef VARIANT1 <текст 1> #else <текст 2> #endif Причому у файлі повинен залишитися <текст 2>, якщо до цієї конструкції зустрічався рядок #define VARIANT1, та <текст 1> у іншому випадку.
14	Здійснити заміну знайдених ідентифікаторів __LINE__ та __FILE__ відповідно на номер рядка, в якому зустрілась __LINE__ та назву файлу, що обробляється, у подвійних лапках. У випадку виявлення директиви #line всі наступні __LINE__ та __FILE__ приймають значення відповідно із першого та другого параметру директиви.
15	У випадку виявлення директиви #stdout <текст> не завершуючи обробки файлу, вивести на екран текст, який вказаний після цієї директиви.
16	Виключити з файлу коментарії, які починаються з ключового слова COMMENT і охоплені фігурними дужками { }.

2. Здійснити компіляцію програми.

3. Оформити звіт в згідно вимог.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Які дії повинна виконувати процедура препроцесоної обробки?
2. Чи можуть виконувати різного роду математичні чи інші операції під час фази препроцесорної обробки?
3. Чи можна обійтись без препроцесорної обробки при реалізації трансляторів чи компіляторів?

ЛАБОРАТОРНА РОБОТА №4. ГРАФИ, ДЕРЕВА, СПИСКИ, СТЬОКИ І ЧЕРГИ (РОБОТА ЗІ СТРУКТУРОЮ ДАНИХ ТРАНСЛЯТОРА)

Мета роботи: Навчитися формувати та обробляти основні компонентні структури трансляторів (компіляторів): графи, дерева, списки, стьоки і черги при.

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Рядки – це послідовність символів, що належать до кінцевої множини (алфавіту) A{0,1}, то послідовність виду 0010110110 є рядком над А. Для виділення рядків один від одного використовують або спеціальні символи, або змінюють алфавіт.

Рядки мають наступні властивості:

- довжина рядка є змінною, хоча алфавіт завжди фіксований;
- звертання до елементів рядка відбувається з якогось кінця, тобто важливу роль відіграє впорядкованість послідовності, а не її індексація;
- рядок представляє собою кінцеве речення деякої мови, що зумовлює існування обмежень від того, які символи можуть зустрічатися разом і в якій послідовності (задається синтаксисом мови).

Орієнтований граф – формально визначається як (X, U), де X – множина елементів, що називаються вершинами, U – множина дуг, або ребер графу ($U \rightarrow X \cdot X$).

Якщо *а* і *в* маленькі вершини, то *а*(*а*, *в*) є ребром, яке напрямлене з вершини *а* у вершину *в*.

Неорієнтований граф – граф в якому відсутня орієнтація ребер, тобто (*а*, *в*) → U і (*в*, *а*) → U.

Зважений граф – якщо в графі додатково задана функція, що визначає довжину (вагомість) кожного з ребер в графі $W{:}U \rightarrow R$.

Регулярний граф – граф у якого з кожної вершини виходить і в кожну вершину входить одна і та ж кількість ребер.

Кореневе дерево (орієнтоване) – це фактично орієнтований граф в якому:

- усі вершини крім однієї (першої) знаходяться в основі тільки однієї дуги;
- коренева (основна) вершина дерева не знаходиться в основі якоїсь іншої дуги;
- коренева вершина зв'язана з кожною вершиною дерева.

Двійкове дерево – це дерево в якому в кожній вершині містяться не більше двох потомків.

Обхід дерева – це впорядкована послідовності вершин дерева в якій кожна вершина зустрічається тільки один раз.

Стьоки і черги є динамічними структурами, що призначені для добавлення і видалення елементів з такої множини в певному порядку.

Стьок – це множина (пам'ять), що організована за принципом останній зайшов, перший вийшов LIFO.

Черга – це аналогічна множина, яка організована за принципом перший зайшов, перший вийшов FIFO.

Списки – послідовність елементів кожен компонент якої містить інформацію двох типів:

- внутрішню, що відповідає вмісту даного елементу;
- зовнішню, про зв'язок даного вузла з іншими елементами списку.

Зовнішня інформація про зв'язок елементів має вид показників, тобто адрес вузлів, що зв'язані з даними (поточними).

ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Згідно заданого варіанту (таблиця 4.1) виконати відповідні операції з файлом, ім’я якого задається в командному рядку, а результат зберегти у файлі із такою ж назвою, як у вхідного але розширенням ”.tmp”. Мова програмування С++ (чи інша за згодою викладача).

Таблиця 4.1.

Варіант	Завдання
1	Сформувати список частоти використання кожного із символів в заданому файлі.
2	Сформувати список частоти використання розділових знаків в заданому файлі.
3	Сформувати список частоти використання голосних та приголосних англійських і кириличних символів в заданому файлі.
4	Сформувати список слів та кількість їх використання в заданому файлі.
5	Сформувати список довжин слів та частоту їх використання в заданому файлі.
6	Сформувати список довжин рядків та частоту їх використання в заданому файлі.
7	Сформувати файл з обернено(зеркально) записаними словами.
8	Сформувати файл з обернено(зеркально) записаними рядками.
9	Сформувати файл з в якому слова замінити числами, що відповідають довжинам цих слів.
10	Провести перевірку правильності розстановки дужок наступних типів: (), [], {} в заданому файлі.
11	Провести перевірку наявності циклу у файлі посилань на умовні операції, що представлені у вигляді з назвою поточної і номером наступної операції.
12	Сформувати двійковий файл на основі вихідної послідовності довжиною N, де кожний наступний елемент визначається на основі додавання першого і останнього попередніх N символів за наступним правилом: (1+1=0+0→0, 1+0=0+1→1). Формування завершити при повторенні вхідної послідовності.
13	З послідовно взятих N двійкових символів вихідного файлу, символ, що зустрічається частіше записується у результуючий файл. Кожна наступна послідовність N двійкових символів формується на основі зсуву попередньої на один символ. Виконувати поки не досягнуто кінець файлу.
14	З послідовно взятих N цифр вихідного файлу, число, що визначається як сума цих цифр записується у результуючий файл. Кожна наступна послідовність N цифр формується на основі зсуву попередньої на одну цифру. Виконувати поки не досягнуто кінець файлу.
15*	Побудувати дерево для пошуку чисел в заданому файлі, що включає поточну позицію числа, позиції меншого і більшого чисел, або признак кінця.
16*	Побудувати дерево для пошуку символів в заданому файлі, що включає поточну позицію символу, позиції старшого і молодшого за алфавітом символів, або признак кінця.

* – завдання підвищеної складності.

2. Здійснити компіляцію програми.

3. Оформити звіт в згідно вимог.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Пояснити особливості організації структур черг, списків та стоьків?
2. Що таке граф, які його основні компоненти?
3. Як організовується дерево, які вони бувають і як обробляються?

ЛАБОРАТОРНА РОБОТА № 5. ОРГАНІЗАЦІЯ ТАБЛИЦЬ ІДЕНТИФІКАТОРІВ

Мета работи: вивчити основні методи організації таблиць ідентифікаторів, отримати уявлення про переваги і недоліки, властиві різним методам організації таблиць ідентифікаторів.

Для виконання лабораторної роботи потрібно написати програму, яка отримує на вході набір ідентифікаторів, організовує таблиці ідентифікаторів за допомогою заданих методів, дозволяє здійснити багатократний пошук довільного ідентифікатора в таблицях і порівняти ефективність методів організації таблиць. Список ідентифікаторів вважати заданим у вигляді текстового файлу. Довжина ідентифікаторів обмежена 32 символами.

Короткі теоретичні відомості
Призначення таблиць ідентифікаторів

При виконанні семантичного аналізу, генерації коду і оптимізації результуючої програми компілятор повинен оперувати характеристиками основних елементів початкової програми - змінних, констант, функцій і інших лексичних одиниць вхідної мови. Ці характеристики можуть бути отримані компілятором на етапі синтаксичного аналізу вхідної програми (найчастіше при аналізі структури блоків описів змінних і констант), а також доповнені на етапі підготовки до генерації коду (наприклад при розподілі пам'яті).

Набір характеристик, відповідний кожному елементу початкової програми, залежить від типу цього елементу, від його сенсу (семантики) і, відповідно, від тієї ролі, яку він виконує в початковій і результуючій програмах. У кожному конкретному випадку цей набір характеристик може бути свій залежно від синтаксису і семантики вхідної мови, від архітектури цільової обчислювальної системи і від структури компілятора. Але є типові характеристики, які найчастіше властиві тим або іншим елементам початкової програми. Наприклад для змінній - це її тип і адреса елементу пам'яті, для константи - її значення, для функції - кількість і типи формальних аргументів, тип повертаного результату, адреса виклику коду функції. Докладнішу інформацію про характеристики елементів початкової програми, їх аналіз і використання можна знайти в [1, 3, 7].

Головною характеристикою будь-якого елементу початкової програми є його ім'я. Саме з іменами змінних, констант, функцій і інших елементів вхідної мови оперує розробник програми - тому і компілятор повинен уміти аналізувати ці елементи по їх іменах.

Ім'я кожного елементу повинне бути унікальним. Багато сучасних мов програмування допускають збіги (ієуникальность) імен змінних і функцій залежно від їх області видимості і інших умов початкової програми. В цьому випадку унікальність імен повинен забезпечувати сам компілятор - про те, як вирішується ця проблема, можна дізнатися в [1-3, 7], тут же вважатимемо, що імена елементів початкової програми завжди є унікальними.

Таким чином, завдання компілятора полягає в тому, щоб зберігати деяку інформацію, пов'язану з кожним елементом початкової програми, і мати доступ до цієї інформації по імені елементу. Для вирішення цього завдання компілятор організовує спеціальні сховища даних, звані таблицями ідентифікаторів, або таблицями символів. Таблиця ідентифікаторів складається з набору полий даних (записів), кожне з яких може

відповідати одному елементу початкової програми. Запис містить всю необхідну компілятору інформацію про даний елемент і може поповнюватися у міру роботи компілятора. Кількість записів залежить від способу організації таблиці ідентифікаторів, але у будь-якому випадку їх не може бути менше, ніж елементів в початковій програмі. В принципі, компілятор може працювати не з однією, а з декількома таблицями ідентифікаторів - їх кількість і структура залежать від реалізації компілятора [1,2].

Принципи організації таблиць ідентифікаторів

Компілятор поповнює записи в таблиці ідентифікаторів у міру аналізу початкової програми і виявлення в ній нових елементів, що вимагають розміщення в таблиці. Пошук інформації в таблиці виконується всякий раз, коли компілятору необхідні зведення про той або інший елемент програми. Причому слід відмітити, що пошук елементу в таблиці виконуватиметься компілятором істотно частіший, ніж переміщення в неї нових елементів. Так відбувається тому, що описи нових елементів в початковій програмі, як правило, зустрічаються набагато рідше, ніж ці елементи використовуються. Крім того, кожному додаванню елементу в таблицю ідентифікаторів у будь-якому випадку передуватиме операція пошуку - щоб переконатися, що такого елементу в таблиці немає.

На кожну операцію пошуку елементу в таблиці компілятор витрачатиме час, і оскільки кількість елементів в початковій програмі велика (від одиниць до сотень тисяч залежно від об'єму програми), цей час істотно впливатиме на загальний час компіляції. Тому таблиці ідентифікаторів повинні бути організовані так, щоб компілятор мав можливість максимально швидко виконувати пошук потрібного йому запису таблиці по імені елементу, з яким пов'язаний цей запис.

Можна виділити наступні способи організації таблиць ідентифікаторів:

- прості і впорядковані списки;
- бінарне дерево;
- хеш-адресація з рехешированням;
- хеш-адресація по методу ланцюжків;
- комбінація хеш-адресації із списком або бінарним деревом.

Далі буде дано короткий опис всіх вищеперелічених способів організації таблиць ідентифікаторів. Докладнішу інформацію можна знайти в [3, 7].

Прості методи побудови таблиць ідентифікаторів

У простому випадку таблиця ідентифікаторів є лінійним неврегульованим списком, або масивом, кожен осередок якого містить дані про відповідний елемент таблиці. Розміщення нових елементів в такій таблиці виконується шляхом запису інформації в черговий осередок масиву або списку у міру виявлення нових елементів в початковій програмі.

Пошук потрібного елементу в таблиці в цьому випадку виконуватиметься шляхом послідовного перебору всіх елементів і порівняння їх імені з ім'ям шуканого елементу, поки не буде знайдений елемент з таким же ім'ям. Тоді якщо за одиницю часу прийняти час, що витрачається компілятором на порівняння двох рядків (у сучасних обчислювальних системах таке порівняння найчастіше виконується однією командою), то для таблиці що містить, N елементів, в середньому буде виконано N/2 порівнянь.

Час, потрібний на додавання нового елементу в таблицю ($T_д$), не залежить від числа елементів в таблиці (N). Але якщо N велике, то пошук зажадає значних витрат часу. Час пошуку ($T_п$) в такій таблиці можна оцінити як $T_п = O(N)$. Оскільки саме пошук в таблиці ідентифікаторів є найбільш часто виконуваною компілятором операцією, такий

спосіб організації таблиць ідентифікаторов є неефективним. Він застосовний тільки для найпростіших компіляторів, що працюють з невеликими програмами.

Пошук може бути виконаний ефективніше, якщо елементи таблиці відсортовані (впорядковані) природним чином. Оскільки пошук здійснюється по імені, найбільш природним рішенням буде розташувати елементи таблиці в прямому або зворотному алфавітному порядку. Ефективним методом пошуку у впорядкованому списку з N елементів є бінарний, або логарифмічний пошук.

Алгоритм логарифмічного пошуку полягає в наступному: шуканий символ порівнюється з елементом (N + 1) /2 в середині таблиці; якщо цей елемент не є шуканим, то ми повинні проглянути тільки блок елементів, пронумерованих від 1 до (N + 1) /2 – 1 , або блок елементів від (N + 1) /2 + 1 до N в залежності від того, менше або більше шуканий елемент того, з яким його порівняли. Потім процес повторюється над потрібним блоком в два рази меншого розміру. Так продовжується до тих пір, поки або шуканий елемент не буде знайдений, або алгоритм не дійде до чергового блоку, що містить один або два елементи (з якими можна виконати пряме порівняння шуканого елементу).

Оскільки на кожному кроці число елементів, які можуть містити шуканий елемент, скорочується в два рази, максимальне число порівнянь рівне $1 + \log_2 N$. Тоді час пошуку елементу в таблиці ідентифікаторів можна оцінити як $T_п=O(\log_2 N)$. Для порівняння: при N = 128 бінарний пошук вимагає найбільше 8 порівнянь, а пошук в неврегульованій таблиці - в середньому 64 порівняння. Метод називають «бінарним пошуком», оскільки на кожному кроці об'єм даної інформації скорочується в два рази, а «логарифмічним» - оскільки час, що витрачається на пошук потрібного елементу в масиві, має логарифмічну залежність від загальної кількості елементів у ньому.

Недоліком логарифмічного пошуку є вимога впорядковування таблиці ідентифікаторів. Оскільки масив інформації, в якому виконується пошук, повинен бути впорядкований, час його заповнення вже залежатиме від числа елементів в масиві. Таблиця ідентифікаторів часто є видимим компілятором ще до того, як вона заповнена, тому потрібний, щоб умова впорядкованості виконувалася на всіх етапах звернення до неї. Отже, для побудови такої таблиці можна користуватися тільки алгоритмом прямого впорядкованого включення елементів.

Якщо користуватися стандартними алгоритмами, вживаними для організації впорядкованих масивів даних, то середній час, необхідне на приміщення всіх елементів в таблицю, можна оцінити таким чином:

$$T_д = O(N\cdot\log_2 N) + k\cdot O(N^2).$$

Тут k - деякий коефіцієнт, що відображає співвідношення між часом, що витрачаються комп'ютером на виконання операції порівняння і операції перенесення даних.

При організації логарифмічного пошуку в таблиці ідентифікаторів забезпечується істотне скорочення часу пошуку потрібного елементу за рахунок збільшення часу на приміщення нового елементу в таблицю. Оскільки додавання нових елементів в таблицю ідентифікаторів відбувається істотно рідше, ніж звернення до них, цей метод слід визнати ефективнішим, ніж метод організації неврегульованої таблиці. Проте в реальних компіляторах цей метод безпосередньо також не використовується, оскільки існують ефективніші методи.

Побудова таблиць ідентифікаторів по методу бінарного дерева

Можна скоротити час пошуку шуканого елементу в таблиці ідентифікаторів, не збільшуючи значно час, необхідний на її заповнення. Для цього треба відмовитися від організації таблиці у вигляді безперервного масиву даних.

Існує метод побудови таблиць, при якій таблиця має форму бінарного дерева. Кожен вузол дерева є елемент таблиці, причому кореневим вузлом стає перший елемент, зустрінутий компілятором при заповненні таблиці. Дерево називається бінарним, оскільки кожна вершина в нім може мати не більше двох гілок. Для визначеності називатимемо дві гілки «права» і «ліва».

Розглянемо алгоритм заповнення бінарного дерева. Вважатимемо, що алгоритм працює з потоком вхідних даних, що містить ідентифікатори. Перший ідентифікатор, як вже було сказано, поміщається у вершину дерева. Всі подальші ідентифікатори потрапляють в дерево по наступному алгоритму:

1. Вибрати черговий ідентифікатор з вхідного потоку даних. Якщо чергового ідентифікатора немає, то побудова дерева закінчена.

2. Зробити поточним вузлом дерева кореневу вершину.

3. Порівняти ім'я чергового ідентифікатора з ім'ям ідентифікатора, що міститься в поточному вузлі дерева.

4. Якщо ім'я чергового ідентифікатора менше, то перейти до кроку 5, якщо рівно - припинити виконання алгоритму (двох однакових ідентифікаторів бути не повинно!), інакше - перейти до кроку 7.

5. Якщо у поточного вузла існує ліва вершина, то зробити її поточним вузлом і повернутися до кроку 3, інакше - перейти до кроку 6.

6. Створити нову вершину, помістити в неї інформацію про черговий ідентифікатор, зробити цю нову вершину лівою вершиной поточного вузла і повернутися до кроку 1.

7. Якщо у поточного вузла існує права вершина, то зробити її поточним вузлом і повернутися до кроку 3, інакше - перейти до кроку 8.

8. Створити нову вершину, помістити в неї інформацію про черговий ідентифікатор, зробити цю нову вершину правою вершиной поточного вузла і повернутися до кроку 1.

Розглянемо як приклад послідовність ідентифікаторів Ga, Dl, M22, E, A12, BC, F. На рис. 1.1 проілюстрований весь процес побудови бінарного дерева для цієї послідовності ідентифікаторів.

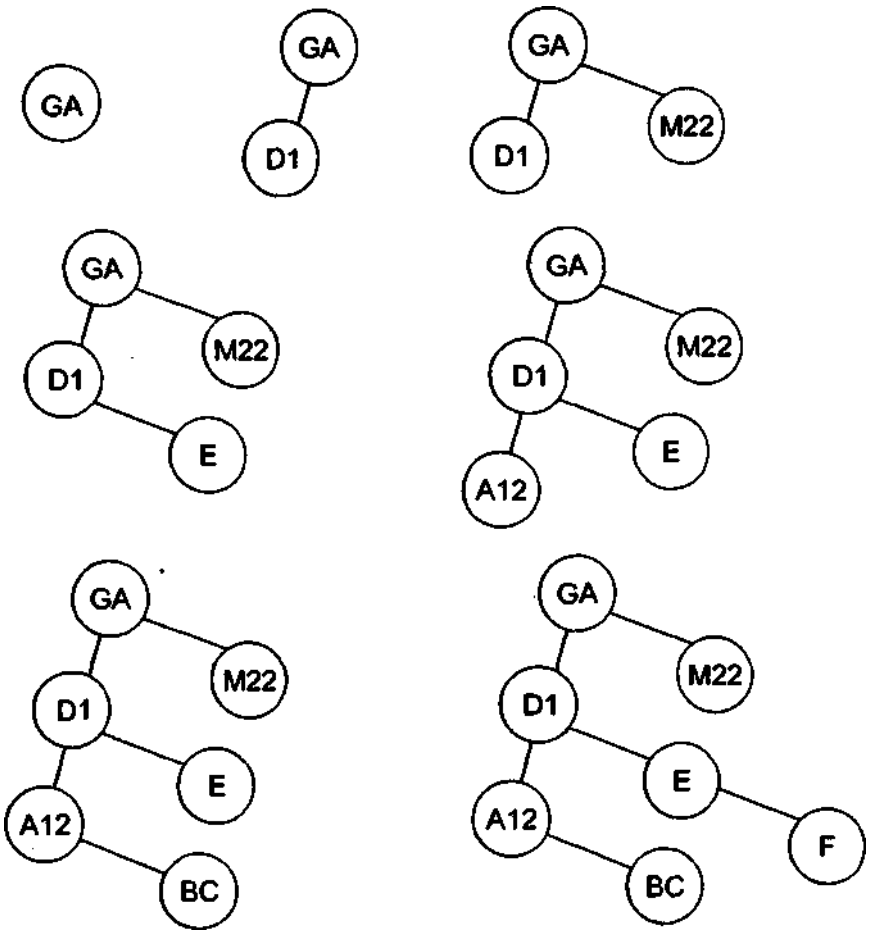


Рис. 1. Заповнення бінарного дерева для послідовності ідентифікаторів
Ga, D1, M22, E, A12, BC, F

Пошук елементу в дереві виконується по алгоритму, схожому з алгоритмом заповнення дерева:

1. Зробити поточним вузлом дерева кореневу вершину.
2. Порівняти ім'я шуканого ідентифікатора з ім'ям ідентифікатора, що міститься в поточному вузлі дерева.
3. Якщо імена співпадають, то шуканий ідентифікатор знайдений, алгоритм завершується, інакше треба перейти до кроку 4.
4. Якщо ім'я чергового ідентифікатора менше, то перейти до кроку 5, інакше - перейти до кроку 6.
5. Якщо у поточного вузла існує ліва вершина, то зробити її поточним вузлом і повернутися до кроку 2, інакше - шуканий іделтіфікатор не знайдений, алгоритм завершується.
6. Якщо у поточного вузла існує права вершина, то зробити її поточним вузлом і повернутися до кроку 2, інакше - шуканий ідентифікатор не знайдений, алгоритм завершується.

Для даного методу число необхідних порівнянь і форма дерева, що вийшло, залежать від того порядку, в якому поступають ідентифікатори. Наприклад, якщо в розглянутому вище прикладі замість послідовності ідентифікаторів Ga, D1, M22, E, A12, BC, F узяти послідовність A12, BC, Dl, E, F, Ga, M22, те дерево виродиться у впорядкований однонаправлений зв'язний список. Ця особливість є недоліком даного методу організації таблиць ідентифікаторів. Іншими недоліками методу є: необхідність зберігати два додаткові посилання на ліву і праву гілку в кожному елементі дерева і робота з динамічним виділенням пам'яті при побудові дерева.

Якщо припустити, що послідовність ідентифікаторів в початковій програмі є статистично неврегульованою (що в цілому відповідає дійсності), то можна вважати, що побудоване бінарне дерево буде невиродженим. Тоді середній час на заповнення дерева ($T_д$) і на пошук елементу в нім ($T_п$) можна оцінити таким чином [3, 7]:

$$T_д=N\cdot O(\log_2 N);$$

$$T_п=O(\log_2 N).$$

Не дивлячись на вказані недоліки, метод бінарного дерева є досить вдалим механізмом для організації таблиць ідентифікаторів. Він знайшов своє застосування у ряді компіляторів. Іноді компілятори будують декілька різних дерев для ідентифікаторів різних типів і різної довжини [1, 2,3, 7].

Хеш-функції і хеш-адресація

У реальних початкових програмах кількість ідентифікаторів така велика, що навіть логарифмічну залежність часу пошуку від їх числа не можна визнати задовільною. Необхідні ефективніші методи пошуку інформації в таблиці ідентифікаторів. Кращих результатів можна досягти, якщо застосувати методи, зв'язані з використанням хеш-функцій і хеш-адресації.

Хеш-функцією F називається деяке відображення безлічі вхідних елементів R на безліч цілих ненегативних чисел Z: F(r) = n, r є R, n є Z. Сам термін «хеш-функція» походить від англійського терміну «hash function» (hash - «заважати», «змішувати», «плутати»).

Безліч допустимих вхідних елементів R називається областю визначення хеш-функції. Безліччю значень хеш-функції F називається підмножина M з безлічі цілих невід'ємних чисел Z: M $\subseteq$ Z, що містить всі можливі значення, повернені функцією F:$\forall$rIR: F(r) є M і $\forall$m є M:$\exists$r є R: F(r) = m. Процес відображення області визначення хеш-функції на безліч значень називається хешуванням.

При роботі з таблицею ідентифікаторів хеш-функція повинна виконувати відображення імен ідентифікаторів на безліч цілих невід'ємних чисел. Областю визначення хеш-функції буде безліч всіх можливих імен ідентифікаторів.

Хеш-адресація полягає у використанні значення, повертаного хеш-функцією, як адреса ячейки з деякого масиву даних. Тоді розмір масиву даних повинен відповідати області значень використовуваної хеш-функції.

Отже, в реальному компіляторі область значень хтш-функції ніяк не повинна перевищувати розмір доступного адресного простору комп'ютера.

Метод організації таблиць ідентифікаторів, заснований на використанні хеш-адресації, полягає в переміщенні кожного елементу таблиці в ячейку, адресу якого повертає хеш-функція, обчислена для цього елементу. Тоді в ідеальному випадку для переміщення будь-якого елементу в таблицю ідентифікаторів досить тільки обчислити його хеш-функцію і звернутися до потрібної ячейки масиву даних. Для пошуку елементу в таблиці також необхідно обчислити хеш-функцію для шуканого елементу і перевірити, чи не є задана нею ячейка масиву порожньою (якщо вона не порожня - елемент знайдений, якщо порожня - не знайдений). Спочатку таблиця ідентифікаторів повинна бути заповнена інформацією, яка дозволила б говорити про те, що всі її ячейки є порожніми.

Цей метод вельми ефективний, оскільки як час розміщення елементу в таблиці, так і час його пошуку визначаються тільки часом, що витрачається на обчислення хеш-функції, яке в загальному випадку незіставимо менше часу, необхідного для багатократних порівнянь елементів таблиці.

Метод має два очевидні недоліки. Перший з них - неефективне використання об'єму пам'яті під таблицю ідентифікаторів: розмір масиву для її зберігання повинен відповідати всій області значень хеш-функції, тоді як ідентифікаторів, що реально зберігаються в таблиці, може бути істотно менше. Другий недолік - необхідність відповідного розумного вибору хеш-функції. Цей недолік є настільки істотним, що не

дозволяє безпосередньо використовувати хеш-адресацію для організації таблиць ідентифікаторів.

Проблема вибору хеш-функції не має універсального рішення. Хешування зазвичай відбувається за рахунок виконання над ланцюжком символів деяких простих арифметичних і логічних операцій. Найпростішою хеш-функцією для символу є код внутрішнього представлення в комп'ютері літери символу. Цю хеш-функцію можна використовувати і для ланцюжка символів, вибираючи перший символ в ланцюжку.

Очевидно, що така примітивна хеш-функція буде незадовільною: при її використанні виникне проблема - двом різним ідентифікаторам, що починаються з однієї і тієї ж букви, відповідатиме одне і те ж значення хеш-функції. Тоді при хеш-адресації в одну і ту ж ячейку таблиці ідентифікаторів повинні бути поміщені два різні ідентифікатори, що явно неможливе. Така ситуація, коли двом або більш ідентифікаторам відповідає одне і те ж значення хеш-функції, називається колізією.

Природно, що хеш-функція, що допускає колізії, не може бути використана для хеш-адресації в таблиці ідентифікаторів. Причому досить отримати хоч би один випадок колізії на всій безлічі ідентифікаторів, щоб такою хеш-функцією не можна було користуватися. Але чи можливо побудувати хеш-функцію, яка б повністю виключала виникнення колізій? Для повного виключення колізій хеш-функція повинна бути взаємно однозначною: кожному елементу з області визначення хеш-функції повинне відповідати одне значення з її безлічі значень, і навпаки - кожному значенню з безліч значень цієї функції повинен відповідати тільки один елемент з її області визначення. Тоді будь-яким двом довільним елементам з області визначення хеш-функції завжди відповідатимуть два різних її значення. Теоретично для ідентифікаторів таку хеш-функцію побудувати можна, оскільки і область визначення хеш-функції (всі можливі імена ідентифікаторів), і область її значень (цілі невід'ємні числа) є нескінченними рахунковими множинами, тому можна організувати взаємно однозначне відображення однієї множини на інше.

Але на практиці існує обмеження, що робить створення взаємно однозначній хеш-функції для ідентифікаторів неможливим. Річ у тому, що в реальності область значень будь-якої хеш-функції обмежена розміром доступного адресного простору комп'ютера. Безліч адрес будь-якого комп'ютера з традиційною архітектурою може бути велика, але завжди звичайно, тобто обмежено. Організувати взаємне однозначне відображення нескінченної множини на кінцеве навіть теоретично неможливо. Можна, звичайно, врахувати, що довжина частини імені ідентифікатора, що приймається до уваги, в реальних компіляторах на практиці також обмежена - зазвичай вона лежить в межах від 32 до 128 символів (тобто і область визначення хеш-функції кінцева). Але і тоді кількість елементів в кінцевій множині, що становить область визначення хеш-функції, перевищуватиме їх кількість в кінцевій безлічі області її значень (кількість всіх можливих ідентифікаторів більше кількості допустимих адрес в сучасних комп'ютерах). Таким чином, створити взаємно однозначну хеш-функцію на практиці неможливо. Отже, неможливо уникнути виникнення колізій.

Тому не можна організувати таблицю ідентифікаторів безпосередньо на основі однієї тільки хеш-адресації. Але існують методи, що дозволяють використовувати хеш-функції для організації таблиць ідентифікаторів навіть за наявності колізій.

Хеш-адресація з рехешированням

Для вирішення проблеми колізії можна використовувати багато способів. Одним з них є метод рехешування (або розстановки). Згідно цьому методу, якщо для елементу A адреса $n_0 = h(A)$, обчислений за допомогою хеш-функції h, вказує на вже зайняту ячейку, то необхідно обчислити значення функції $n_1 = h_1(A)$ і перевірити зайнятість осередку за адресою n_1. Якщо і вона зайнята, то обчислюється значення $h_2(A)$, і так до тих пір, поки або не буде знайдена вільна ячейка, або чергове значення $h_i(A)$ не співпаде з $h(A)$. У останньому випадку вважається, що таблиця ідентифікаторів заповнена і місця в ній більше немає - видається інформація про помилку розміщення ідентифікатора в таблиці.

Тоді пошук елементу A в таблиці ідентифікаторів, організованій таким чином, виконуватиметься по наступному алгоритму:

1. Обчислити значення хеш-функції n = h(A) для шуканого елементу A.
2. Якщо ячейка за адресою n порожня, то елемент не знайдений, алгоритм завершений, інакше необхідно порівняти ім'я елементу в ячейці n з ім'ям шуканого елементу A. Якщо вони співпадають, то елемент знайдений і алгоритм завершений, інакше i := 1 і перейти до кроку 3.
3. Обчислити $n_i = h_i(A)$. Якщо ячейка за адресою n_i порожня або $n = n_i$,то елемент не знайдений і алгоритм завершений, інакше - порівняти ім'я елементу в ячейці n_i, з ім'ям шуканого елементу A. Якщо вони співпадають, то елемент знайдений і алгоритм завершений, інакше i := i + 1 і повторити крок 3.

Алгоритми розміщення і пошуку елементу схожі по виконуваних операціях. Тому вони матимуть однакові оцінки часу, необхідного для їх виконання.

При такій організації таблиць ідентифікаторів у разі виникнення колізії алгоритм поміщає елементи в порожні елементи таблиці, вибираючи їх певним чином. При цьому елементи можуть потрапляти в ячейки з адресами, які потім співпадатимуть із значеннями хеш-функції, що приведе до виникнення нових, додаткових колізій. Таким чином, кількість операцій, необхідних для пошуку або розміщення в таблиці елементу, залежить від заповненої таблиці.

Для організації таблиці ідентифікаторів по методу рехешировання необхідно визначити всі хеш-функції h_i для всіх i. Найчастіше функції h_i визначають як деякі модифікації хеш-функції h. Наприклад, найпростішим методом обчислення функції $h_i(A)$ є її організація у вигляді $h_i(A) = (h(A) + p_i) \bmod N_m$, де p_i - деяке обчислюване ціле число, а N_m - максимальне значення з області значень хтш-функції h. У свою чергу, найпростішим підходом тут буде $p_i = i$. Тоді отримуємо формулу $h_i(A) = (h(A) + i) \bmod N_m$. В цьому випадку при збігу значень хеш-функції для яких-небудь елементів пошук вільної ячейки в таблиці починається послідовно від поточної позиції, заданою хеш-функцією h(A).

Цей спосіб не можна визнати особливо вдалим: при збігу хеш-адрес елементи в таблиці починають групуватися навколо них, що збільшує число необхідних порівнянь при пошуку і розміщенні. Але навіть такий примітивний метод рехешировання є достатньо ефективним засобом організації таблиць ідентифікаторів при неповному заповненні таблиці. Середній час на приміщення одного елементу в таблицю і на пошук елементу в таблиці можна понизити, якщо застосувати більш довершений метод рехешировання. Одним з таких методів є використання як p_i для функції $h_i(A) = (h(A) + p_i) \bmod N_m$ послідовності псевдовипадкових цілих чисел p_1 , p_2 , ... , p_k. При хорошому виборі генератора псевдовипадкових чисел довжина послідовності $k = N_m$.

Існують і інші методи організації функцій рехешированія /г(Л), засновані на квадратичних обчисленнях або, наприклад, на обчисленні твору за формулою: $h_i(A) = (h(A)N \cdot i) \bmod N'_m$, де N'_m - найближче просте число, менше N_m. В цілому рехешировання дозволяє добитися непоганих результатів для ефективного пошуку елементу в таблиці (кращих, ніж бінарний пошук і бінарне дерево), але ефективність методу сильно залежить від заповненої таблиці ідентифікаторів і якості використовуваної хеш-функції - чим рідше виникають колізії, тим вище ефективність методу. Вимога неповного заповнення таблиці веде до неефективного використання об'єму доступної пам'яті.

Оцінки часу розміщення і пошуку елементу в таблицях ідентифікаторів при використанні різних методів рехешировання можна знайти в [1, 3,7].

Хеш-адресація з використанням методу ланцюжків

Неповне заповнення таблиці ідентифікаторів при застосуванні рехешировання веде до неефективного використання всього об'єму пам'яті, доступного компілятору. Причому об'єм невживаної пам'яті буде тим вище, чим більше інформації зберігається для кожного ідентифікатора. Цього недоліку можна уникнути, якщо доповнити таблицю ідентифікаторів деякою проміжною хеш-таблицею.

У ячейках хеш-таблиці може зберігатися або порожнє значення, або значення покажчика на деяку область пам'яті з основної таблиці ідентифікаторів. Тоді хеш-функція обчислює адресу, по якій відбувається звернення спочатку до хеш-таблиці, а потім вже через неї за знайденою адресою - до самої таблиці ідентифікаторів. Якщо відповідний елемент таблиці ідентифікаторів порожній, то ячейка хеш-таблиці міститиме порожнє значення. Тоді зовсім не обов'язково мати в самій таблиці ідентифікаторів ячейку для кожного можливого значення хеш-функції - таблицю можна зробити динамічною, так щоб її об'єм ріс у міру заповнення (спочатку таблиця ідентифікаторів не містить жодної ячейки, а всі ячейки хеш-таблиці мають порожнє значення).

Такий підхід дозволяє добитися двох позитивних результатів: по-перше, немає необхідності заповнювати порожніми значеннями таблицю ідентифікаторів - це можна зробити тільки для хеш-таблиці; по-друге, кожному ідентифікатору відповідатиме строго одна ячека в таблиці ідентифікаторів. Порожні ячейки у такому разі будуть тільки в хеш-таблиці, і об'єм невживаної пам'яті не залежатиме від об'єму інформації, що зберігається для кожного ідентифікатора, - для кожного значення хеш-функції витрачатиметься тільки пам'ять, необхідна для зберігання одного покажчика на основну таблицю ідентифікаторів.
На основі цієї схеми можна реалізувати ще один спосіб організації таблиць ідентифікаторів за допомогою хеш-функції, званий методом ланцюжків. В цьому випадку в таблицю ідентифікаторів для кожного елементу додається ще одне поле, в якому може міститися посилання на будь-який елемент таблиці. Первинно це поле завжди порожнє (нікуди не указує). Також необхідно мати одну спеціальну змінну, яка завжди указує на перший вільний елемент основної таблиці ідентифікаторів (спочатку вона указує на початок таблиці).
Метод ланцюжків працює по наступному алгоритму:
У всі ячейки хеш-таблиці помістити порожнє значення, таблиця ідентифікаторів порожня, змінна FreePtr (покажчик першого вільного осередку) указує на початок таблиці ідентифікаторів.

1. Обчислити значення хеш-функції n для нового елементу A. Якщо ячейка хеш-таблиці за адресою n порожня, то помістити в неї значення змінної FreePtr і перейти до кроку 5, інакше перейти до кроку 3.
2. Вибрати з хеш-таблиці адресу елементу таблиці ідентифікаторів m і перейти до кроку 4.
3. Для ячейки таблиці ідентифікаторів за адресою m перевірити значення поля посилання. Якщо воно порожнє, то записати в нього адресу із змінної FreePtr і перейти до кроку 5; інакше вибрати з поля посилання нову адресу m і повторити крок 4.
4. Додати в таблицю ідентифікаторів нову ячейку, записати в неї інформацію для елементу A (поле посилання повинне бути порожнім), в змінну FreePtr помістити адресу за кінцем доданої ячейки. Якщо більше немає ідентифікаторів, які треба помістити в таблицю, то виконання алгоритму закінчене, інакше перейти до кроку 2.

Пошук елементу в таблиці ідентифікаторів, організованій таким чином, виконуватиметься по наступному алгоритму:

1. Обчислити значення хеш-функції n для шуканого елементу A. Якщо хеш-таблиці за адресою n порожні, то елемент не знайдений і алгоритм завершений, інакше вибрати з хеш-таблиці адресу елементу таблиці ідентифікаторів m.
2. Порівняти ім'я елементу в ячейці таблиці ідентифікаторів за адресою m з ім'ям шуканого елементу A. Якщо вони співпадають, то шуканий елемент знайдений і алгоритм завершений, інакше перейти до кроку 3.
3. Перевірити значення поля посилання в ячейці таблиці ідентифікаторів за адресою m. Якщо воно порожнє, то шуканий елемент не знайдений і алгоритм завершений;

інакше вибрати з поля посилання адресу m і перейти до кроку 2.

При такій організації таблиць ідентифікаторів у разі виникнення колізії алгоритм поміщає елементи в ячейки таблиці, пов'язуючи їх один з одним послідовно через поле посилання. При цьому елементи не можуть потрапляти в ячейки з адресами, які потім співпадатимуть із значеннями хеш-функції. Таким чином, додаткові колізії не виникають. У результаті в таблиці виникають своєрідні ланцюжки зв'язаних елементів, звідки і відбувається назва даного методу - «метод ланцюжків».

На рис. 1.2 проілюстровано заповнення хеш-таблиці і таблиці ідентифікаторів для ряду ідентифікаторів: A_1, A_2, A_3, A_4, A_5 за умови, що $h(A_1) = h(A_2) = h(A_5) = n_1$; $h(A_3) = n_2$; $h(A_4) = n_4$. Після розміщення в таблиці для пошуку ідентифікатора A_1, буде потрібно одне порівняння, для A_2 - два порівняння, для A_3 - одне порівняння, для A_4 - одне порівняння і для A_5 - три порівняння (спробуйте порівняти ці дані з результатами, отриманими з використанням простого рехе-ширования для тих же ідентифікаторів).

Метод ланцюжків є дуже ефективним засобом організації таблиць ідентифікаторів. Середній час на розміщення одного елементу і на пошук елементу в таблиці для нього залежить тільки від середнього числа колізій, що виникають при обчисленні хеш-функції. Накладні витрати пам'яті, пов'язані з необхідністю мати одне додаткове поле покажчика в таблиці ідентифікаторів на кожен її елемент, можна визнати цілком виправданими, оскільки виникає економія використовуваної пам'яті за рахунок проміжної хеш-таблиці. Цей метод дозволяє економніше використовувати пам'ять, але вимагає організації роботи з динамічними масивами даних.

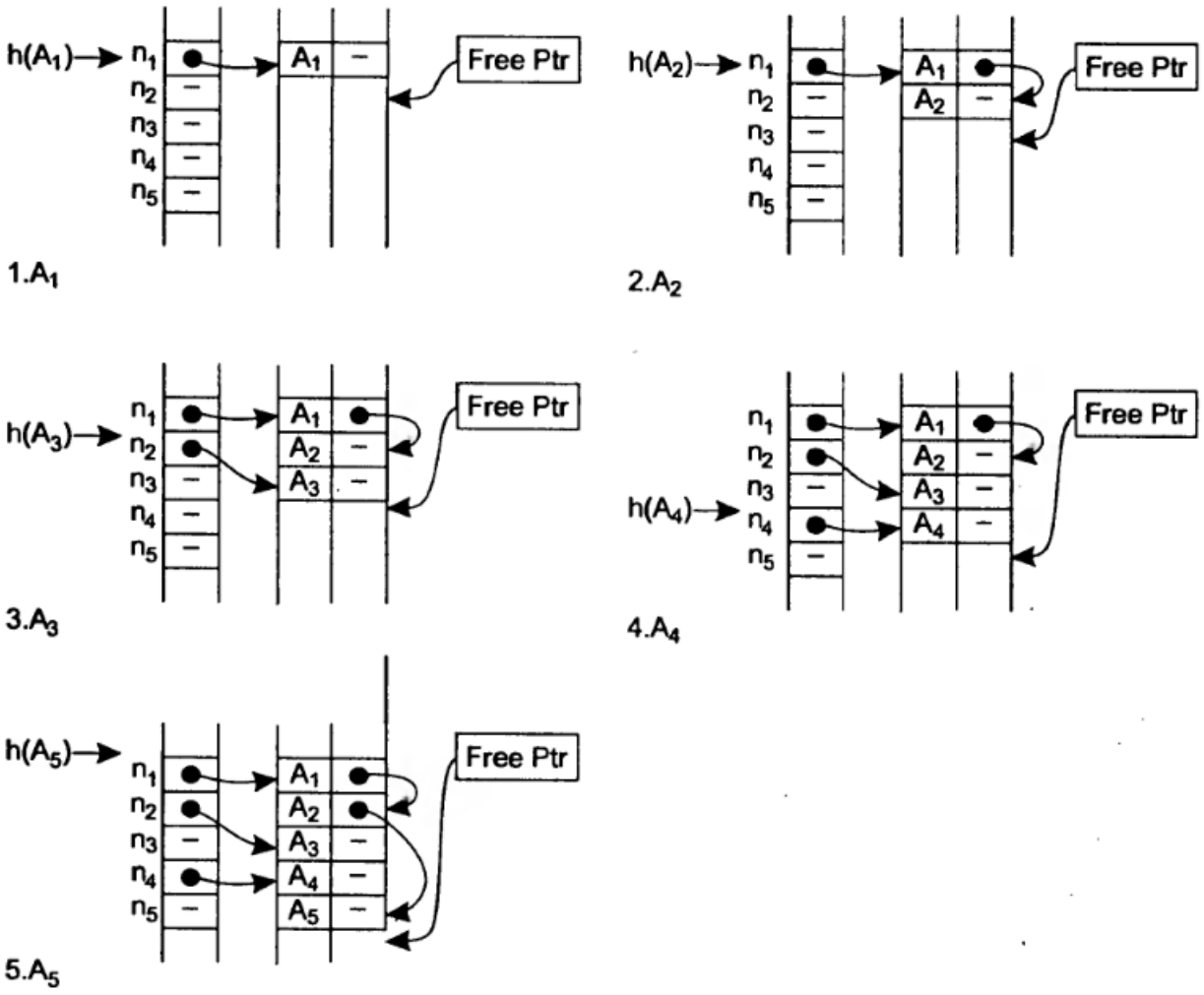


Рис. 2. Заповнення таблиці ідентифікаторів при використанні методу ланцюжків

Комбіновані способи побудови таблиць ідентифікаторів

Окрім рехешированія і методу ланцюжків можна використовувати комбіновані методи для організації таблиць ідентифікаторів за допомогою хеш-адресації. В цьому випадку для виключення колізій хеш-адресація поєднується з одним з раніше розглянутих методів - простим списком, впорядкованим списком або бінарним деревом, який використовується як додатковий метод впорядковування ідентифікаторів, для яких виникають колізії. Причому, оскільки при якісному виборі хеш-функції кількість колізій зазвичай невелика (одиниці або десятки випадків), навіть простий список може бути цілком задовільним рішенням при використанні комбінованого методу.

При такому підході можливі два варіанти: у першому випадку, як і для методу ланцюжків, в таблиці ідентифікаторів організовується спеціальне додаткове поле посилання. Але на відміну від методу ланцюжків воно має трохи інше значення: за відсутності колізій для вибірки інформації з таблиці використовується хеш-функція, поле посилання залишається порожнім. Якщо ж виникає колізія, то через поле посилання організовується пошук ідентифікаторів, для яких значення хеш-функції співпадають, - це поле повинне указувати на структуру даних для додаткового методу: початок списку, перший елемент динамічного масиву або кореневий елемент дерева.

У другому випадку використовується хеш-таблиця, аналогічна хеш-таблиці для методу ланцюжків, Якщо за даною адресою хеш-функції ідентифікатор відсутній, то ячейка хеш-таблиці порожня. Коли з'являється ідентифікатор з даним значенням хеш-функції, то створюється відповідна структура для додаткового методу, в хеш-таблицю записується посилання на цю структуру, а ідентифікатор поміщається в створену структуру за правилами вибраного додаткового методу. У першому варіанті за відсутності колізій пошук виконується швидше, але другий варіант кращий, оскільки за рахунок використання проміжної хеш-таблиці забезпечується ефективніше використання пам'яті. Як і для методу ланцюжків, для комбінованих методів час розміщення і час пошуку елементу в таблиці ідентифікаторів залежить тільки від середнього числа колізій, що виникають при обчисленні хеш-функції. Накладні витрати пам'яті при використанні проміжної хеш-таблиці мінімальні. Очевидно, що якщо як додатковий метод використовувати простий список, то вийде алгоритм, повністю аналогічний методу ланцюжків. Якщо ж використовувати впорядкований список або бінарне дерево, то метод ланцюжків і комбіновані методи матимуть приблизно рівну ефективність при незначному числі колізій (одиничні випадки), але із зростанням кількості колізій ефективність комбінованих методів в порівнянні з методом ланцюжків зростатиме.

Недоліком комбінованих методів є складніша організація алгоритмів пошуку і розміщення ідентифікаторів, необхідність роботи з динамічно розподілюваними областями пам'яті, а також великі витрати часу на розміщення нового елементу в таблиці ідентифікаторів в порівнянні з методом ланцюжків.

То, який конкретно метод застосовується в компіляторі для організації таблиць ідентифікаторів, залежить від реалізації компілятора. Один і той же компілятор може мати навіть декілька різних таблиць ідентифікаторів, організованих на основі різних методів. Як правило, застосовуються комбіновані методи.

Створення ефективної хеш-функції - це окреме завдання розробників компіляторів, і отримані результати, як правило, тримаються в секреті. Хороша хеш-функція розподіляє ідентифікатори, що поступають на її вхід, рівномірно на що все є у розпорядженні адреси, щоб звести до мінімуму кількість колізій. В даний час існує безліч хеш-функцій, але, як було показано вище, ідеального хешування досягти неможливо.

Хеш-адресація - це метод, який застосовується не тільки для організації таблиць ідентифікаторів в компіляторах. Даний метод знайшов своє застосування і в операційних системах, і в системах управління базами даних [5, 6, 11].

Вимоги до виконання роботи

Порядок виконання роботи

У всіх варіантах завдання потрібно розробити програму, яка може забезпечити порівняння двох способів організації таблиці ідентифікаторів за допомогою хеш-адресації. Для порівняння пропонуються способи, засновані на використанні рехешированія або комбінованих методів. Програма повинна прочитувати ідентифікатори з вхідного файлу, розміщувати їх в таблицях за допомогою заданих методів і виконувати пошук вказаних ідентифікаторів на вимогу користувача. В процесі розміщення і пошуку ідентифікаторів в таблицях програма повинна підраховувати середнє число виконаних операцій порівняння для зіставлення ефективності використовуваних методів. Для організації таблиць пропонується використовувати просту хеш-функцію, яку розробник програми повинен вибрати самостійно. Хеш-функція повинна забезпечувати роботу не менше чим з 200 ідентифікаторами, допустима довжина ідентифікатора повинна бути не меншого 32 символів. Забороняється використовувати в роботі хеш-функції, узяті з прикладу виконання роботи.

Лабораторна робота повинна виконуватися в наступному порядку:

1. Отримати варіант завдання у викладача.
2. Вибрати і описати хеш-функцію.
3. Описати структури даних, використовувані для заданих методів організації таблиць ідентифікаторів.
4. Підготувати і захистити звіт.
5. Написати і відладити програму на ЕОМ.
6. Здати працюючу програму викладачеві.

Вимоги до оформлення звіту

Звіт по лабораторній роботі повинен містити наступні розділи:

- завдання по лабораторній роботі;
- опис вибраної хеш-функції;
- схеми організації таблиць ідентифікаторів (відповідно до варіанту завдання);
- опис алгоритмів пошуку в таблицях ідентифікаторів (відповідно до варіанту завдання);
- текст програми (оформляється після виконання програми на ЕОМ);
- результати обробки заданого набору ідентифікаторів (вхідного файлу) за допомогою методів організації таблиць ідентифікаторів, зазначених у варіанті завдання;
- аналіз ефективності використовуваних методів організації таблиць ідентифікаторов і висновки щодо виконаної роботи.

Основні контрольні запитання

- Що таке таблиця символів і для чого вона призначена? Яка інформація може зберігатися в таблиці символів?
- Які цілі переслідуються при організації таблиці символів?
- Якими характеристиками можуть володіти лексичні елементи вихідної програми? Які характеристики є обов'язковими?
- Які існують способи організації таблиць символів?
- У чому полягає алгоритм логарифмічного пошуку? Які переваги він дає у порівнянні з простим перебором і які він має недоліки?
- Розкажіть про деревоподібну організацію таблиць ідентифікаторів. У чому її переваги і недоліки?
- Що таке хеш-функції і для чого вони використовуються? У чому суть хеш-адресації?
- Що таке колізія? Чому вона відбувається? Чи можна повністю уникнути колізій?
- Що таке рехешування? Які методи рехешування існують?
- Розкажіть про переваги і недоліки організації таблиць ідентіфікаторів за допомогою хеш-адресації та рехешування.

- У чому полягає метод ланцюжків?
- Розкажіть про переваги і недоліки організації таблиць ідентіфікаторів за допомогою хеш-адресації та методу ланцюжків.
- Як можуть бути скомбінувані різні методи організації хеш-таблиць?
- Розкажіть про переваги і недоліки організації таблиць ідентіфікаторів за допомогою комбінованих методів.

Варіанти завдань

У табл. 1.1 перераховані методи організації таблиць ідентифікаторів, що використовуються в завданнях.

Таблиця 1.1 Методи організацій таблиць ідентифікаторів

№ метода	Спосіб вирішення колізії
1	Просте рехешування
2	Рехешування з використанням псевдовипадкових чисел
3	Рехешування за допомогоб проекції
4	Метод ланцюжків
5	Простий список
6	Упорядкований список
7	Бінарне дерево

У табл. 1.2 подані варіанти завдань на основі методів організації таблиць ідентифікаторів, перерахованих в табл. 1.1.

Таблиця 1.2 Варіанти завдань

№ варіанту	Перший метод організації таблиць	Другий метод організації таблиць
1	1	5
2	1	6
3	1	7
4	2	1
5	2	5
6	2	6
7	3	5
8	3	6
9	3	7
10	7	5
11	4	6
12	4	7
13	1	4
14	2	4
15	3	4
16	2	3

Приклад виконання роботи

Завдання для прикладу

В якості прикладу виконання лабораторної роботи візьмемо зіставлення двох методів: хеш-адресації з рехешуванням на основі псевдоймовірних чисел і комбінації хеш-адресації з бінарним деревом. Якщо звернутися до наведеної вище табл. 1.1, то такий варіант завдання буде відповідати комбінації методів 2 і 7 (в табл. 1.2 серед варіантів завдань така комбінація відсутня).

Вибір і опис хеш-функції

Для хеш-адресації з рехешуванням в якості хеш-функції візьмемо функцію, яка буде отримувати на вході рядок, а в результаті видавати суму кодів першого, середнього й останнього елементів рядка. Причому якщо рядок міститиме менше трьох символів, то

один і той же символ буде взятий і в якості першого, і в якості середнього, і в якості останнього.

Будемо вважати, що прописні і рядкові букви в ідентіфікаторах різні. У якості кодів символів візьмемо коди таблиці ASCII, яка використовується в обчислювальних системах на базі ОС типу Microsoft Windows. Тоді, якщо вважати, що рядок з області визначення хеш-функції містить лише цифри і букви англійского алфавіту, то мінімальним значенням хеш-функції буде сума трьох кодів цифри «0», а максимальним значенням - сума трьох кодів літери «z».

Таким чином, область значень обраної хеш-функції в термінах мови Object Pascal може бути описана як:

```
(Ord('0') + Ord('0') + Ord('0')) .. (Ord('z') + Ord('z') + Ord('z'))
```

Діапазон області значень складає 223 елемента, що задовольняє вимоги завдання (не менше ніж 200 елементів). Довжина вхідних ідентифікаторів в даному випадку нічим не обмежена. Для зручності користування опишемо дві константи, що задають границю області значень хеш-функції:

```
HASH_MIN = Ord('0') + Ord('0') + Ord('0');
HASH_MAX = Ord('z') + Ord('z') + Ord('z').
```

Сама хеш-функція без урахування рехешування буде обчислювати наступний вираз:

```
Ord(sName[1])+Ord(sName[(Length(sName)+1)div2])+Ord(sName[Length(sName)]
```

тут sName - це вхідний рядок (аргумент хеш-функції).

Для рехешування візьмемо простий генератор послідовності псевдовипадкових чисел, побудований на основі формули $F = i \cdot H_1 \bmod H_2$, де H_1 і H_2 - прості числа, обрані таким чином, щоб H_1 було в діапазоні від $H_2/2$ до H_2. Причому так, щоб цей генератор видавав максимально довгу послідовність – у всьому діапазоні від HASH_MIN до HASH_MAX, H_2 має бути максимально наближене до величини HASH_MAX – HASH_MIN + 1. У даному випадку діапазон містить 223 елемента, і оскільки 223 - просте число, то візьмемо $H_2 = 223$ (якщо б розмір діапазону не був простим числом, то в якості H_2 потрібно було б взяти найближче до нього менше просте число). В якості H_1 візьмемо 127: $H_1 = 127$. Опишемо відповідні константи:

```
REHASH1 = 127;
REHASH2 = 223.
```

Тоді хеш-функція з урахуванням рехешуванням буде мати такий вигляд:

```
function VarHash(const sName:string; Num:integer):longint;
begin
     Result:=(Ord(sName[1])+Ord(sName[(Length(sName)+1) div 2])
       + Ord(sName[Length(sName)]) - HASH_MIN
       + iNum * REHASH1 mod REHASH2)
       mod (HASH_MAX - HASH_MIN+1) + HASH_MIN;
     if Result < HASH_MIN then Result := HASH_MIN;
end;
```

Вхідні параметри цієї функції: sName - ім'я ідентифікатора, що хешується, iNum — індекс рехешування (якщо iNum = 0, то рехешування відсутнє). Рядок перевірки величини результату (Result < HASHMIN) доданий, щоб виключити помилки в тих випадках, коли на вхід функції подається рядок, що містить символи не з діапазону '0' .. 'z' (оскільки контроль вхідних ідентифікаторів відсутній, це має сенс).

Для комбінації хеш-адресації та бінарного дерева можна використовувати більш просту хеш-функцію - суму кодів першого та середнього символів вхідного рядка. Діапазон значень такої хеш-функції в термінах мови Object Pascal буде виглядати так:

```
(Ord('0')+Ord('0'))..(Ord('z')+Ord('z'))
```

Цей діапазон містить менше ніж 200 елементів, однак функція буде задовільнти вимоги завдання, так як в комбінації з бінарним деревом вона буде забезпечувати обробку необмеженої кількості ідентифікаторів (максимальна кількість ідентифікаторів буде обмежена тільки об'ємом доступної оперативної пам'яті комп'ютера).

Без застосування рехешування ця хеш-функція буде виглядати значно простіший, ніж описана вище хеш-функція з урахуванням рехешування:

```
function VarHash(const sName: string): longint; begin
     Result:=(0rd(sName[1])+0rd(sName[(Length(sName)+1) div 2])
       - HASH_MIN) mod (HASH_MAX-HASH_MIN+1) + HASH_MIN;
     if Result < HASH_MIN then Result := HASH_MIN;
end.
```

Опис структур даних таблиць ідентифікаторів

У першу чергу необхідно описати структуру даних, яка буде використана для зберігання інформації про ідентифікатори у таблицях ідентифікаторів. Для обох таблиць (з рехешуванням на основі генератора псевдовипадкових чисел і в комбінації з бінарним деревом) будемо використовувати одну й ту ж структуру. У цьому випадку в таблицях будуть зберігатися дані, що не використовуються, але програмний код буде простіший. В якості навчального прикладу такий підхід виправданий.

Структура даних таблиці ідентифікаторів (назвемо її TVarInfo) повинна містити в обов'язковому порядку поля імені ідентифікатора (поле sName: string), а також поля додаткової інформації про ідентіфікатор (на розгляд розробників компілятора). У лабораторної роботі не передбачено зберігання будь-якої додаткової інформації про ідентіфікатори, тому в якості ілюстрації інформаційного поля добавимо у структуру TVarInfo додаткову інформаційну структуру TAddVarlnfo (поле pInfo: TAddVarInfo).

Оскільки в мові Object Pascal для полів і змінних, що описані як class, зберігаються тільки посилання на відповідну структуру, такий підхід не призведе до значних витрат пам'яті, але дозволить в майбутньому зберігати будь-яку інформацію, пов'язану з кодом, в окремій структурі даних (оскільки передбачається використовувати створювані програмні модулі в подальших лабораторних роботах). У даному випадку інший підхід неможливий, так як заздалегідь невідомо, які дані необхідно буде зберігати в таблицях ідентифікаторів. Але розробник реального компілятора, як правило, знає, яку інформацію потрібно зберігати, і може використовувати інший підхід - безпосередньо включити всі необхідні поля в структуру даних таблиці ідентифікаторів (в даному випадку - в структуру TVarInfo) без використання проміжних структур даних та посилань.

Перший підхід, реалізований в даному прикладі, забезпечує більш економне використання оперативної пам'яті, але є більш складним і вимагає роботи з динамічними структурами, другий підхід більш простий у реалізації, але менш ощадливо використовує пам'ять. Який з двох підходів вибрати, вирішує розробник компілятора в кожному конкретному випадку (другий підхід буде проілюстрований пізніше в прикладі до лабораторної роботи № 4).

Для роботи зі структурою даних TVarInfo будуть потрібні наступні функції:

- функції створення структури даних та звільнення займаної пам'яті - реалізовані як constructor Create і destructor Destroy;
- функції доступу до додаткової інформації - у цій реалізації це procedure Setlnfo і procedure Clearlnfo.

Ці функції будуть загальними для таблиці ідентифікаторів з рехешуванням і для комбінованої таблиці ідентифікаторів.

Однак, для комбінованої таблиці ідентифікаторів в структуру даних TVarInfo буде потрібно також включити додаткові поля даних та функції, що забезпечують організацію бінарного дерева:

- посилання на ліву («меншу») та праву («велику») гілку дерева – реалізовані як поля даних minEl, maxEl: TVarInfo;

- функції додавання елемента в дерево – function AddElCnt і function AddElem;
- функції пошуку елемента в дереві - function FindEICnt і function FindElem;
- функція очищення інформаційних полів у всьому дереві – procedure ClearAllInfo;
- функція виведення вмісту бінарного дерева в один рядок (для отримання списку всіх ідентифікаторів) - function GetElList.

Функції пошуку та розміщення елементу в дереві реалізовані у двох примірниках, так як одна з них виконує підрахунок кількості порівнянь, а інша - ні.

Оскільки на функції і процедури не витрачається оперативна пам'ять, в результаті ми одержали, що при використанні однієї і тієї ж структури даних для різних таблиць ідентифікаторів в таблиці з рехешуванням буде витрачатися пам'ять тільки на зберігання двох зайвих посилань (minEl і maxEl).

Повністю вся структура даних TVarlnfo та пов'язані з нею процедури та функції описані в програмному модулі TblElem. Повний текст цього програмного модуля наведено в лістингу П3.1 в додатку 3.

Треба звернути увагу на один важливий момент в реалізації функції пошуку ідентифікатора у дереві (function TVarInfo.FindElCnt). Якщо виконувати порівняння двох рядків (в даному випадку - імені шуканого ідентифікатора sN та імені ідентифікатора в цьому вузлі дерева sName) за допомогою стандартних методів порівняння рядків мови Object Pascal, то фрагмент програмного коду виглядав би приблизно так:

```
if sN < sName then
begin
……
end
else
if sN > sName then
begin
……
end
else...
```

У цьому фрагменті порівняння рядків виконується двічі: спочатку перевіряється відношення «менше» (sN < sName), а потім - «більше» (sN > sName). І хоча в програмному коді явно це не вказано, для кожного з цих операторів буде викликана бібліотечна функція порівняння рядків (тобто операція порівняння може виконуватися двічі!). Щоб цього уникнути, в реалізації, запропонованій в прикладі виконується явний виклик функції порівняння рядків, а потім обробляється отриманий результат:

```
i := StrComp(PChar(sN),PChar(sName));
if i < 0 then
begin
……
end
else
if i > 0 then
begin
……
end
else...
```

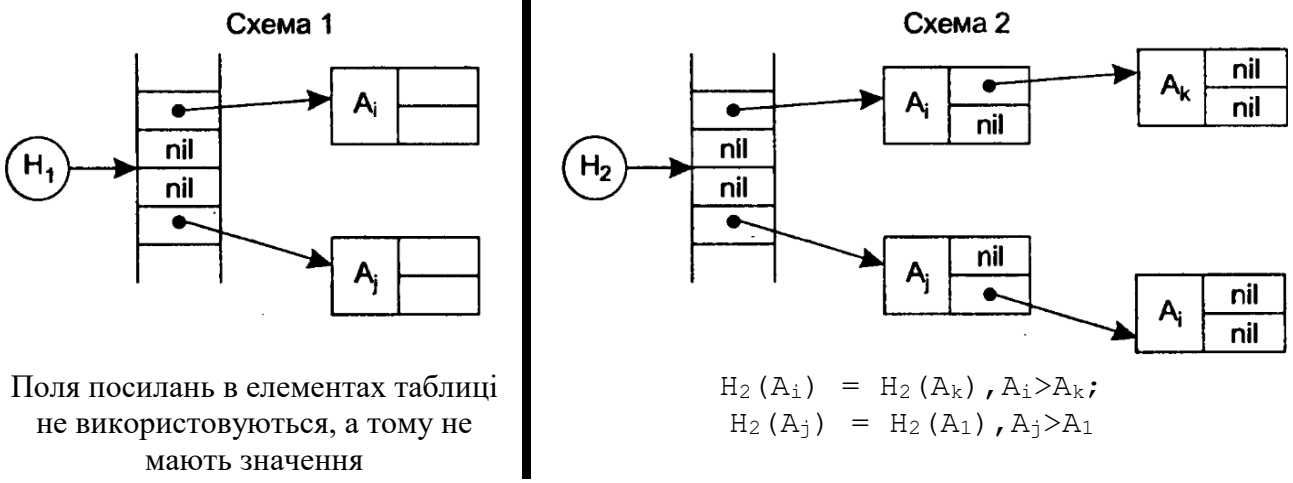
У такому варіанті двічі може бути виконано лише порівняння цілого числа з нулем, а порівняння рядків завжди виконується тільки один раз, що істотно збільшує ефективність процедури пошуку.

Організація таблиць ідентифікаторів

Коди таблиці реалізовані у вигляді статичних масивів розміром HASHMIN .. HASHMAX, елементами яких є структури даних типу TVarInfo. У мові Object Pascal, як було зазначено вище, для структур таких типів зберігаються посилання. Тому для позначення порожніх комірок у таблицях ідентифікаторів буде використовуватися порожнє посилання - nil.

Оскільки в пам'яті зберігаються посилання, описані масиви будуть відігравати роль хеш-таблиць, посилання з яких вказують безпосередньо на інформацію в таблицях ідентифікаторів.

На рис. 1.3 показані умовні схеми, які наочно ілюструють організацію таблиць ідентифікаторів. Схема 1 ілюструє таблицю ідентифікаторів з ре-хешуванням на основі генератора псевдовипадкових чисел, схема 2 - таблицю ідентифікаторів на основі комбінації хеш-адресації з бінарним деревом. Комірки з написом «nil» відповідають незаповненим коміркам хеш-таблиці.



Поля посилань в елементах таблиці не використовуються, а тому не мають значення

$H_2(A_i) = H_2(A_k), A_i > A_k;$
$H_2(A_j) = H_2(A_1), A_j > A_1$

H_1 і H_2 – відповідні хеш-функції

Рис. 1.3. Схема організації таблиць ідентифікаторів

Для кожної таблиці ідентифікаторів реалізовані наступні функції:

- функції початкової ініціалізації хеш-таблиці – InitTreeVar та InitHashVar;
- функції звільнення пам'яті хеш-таблиці – ClearTreeVar та ClearHashVar;
- функції видалення додаткової інформації в таблиці – ClearTreeInfo та ClearHashInfo;
- функції додавання елемента в таблицю ідентифікаторів – AddTreeVar та AddHashVar;
- функції пошуку елемента в таблиці ідентифікаторів – GetTreeVar та GetHashVar;
- функція, що повертає кількість виконаних операцій порівняння при розміщенні або пошуку елемента в таблиці – GetTreeCount і GetHashCount.

Алгоритми пошуку та розміщення ідентифікаторів для двох даних методів організації таблиць були описані вище в розділі «Короткі теоретичні відомості», тому наводити їх тут повторно немає сенсу. Вони реалізовані у вигляді чотирьох перерахованих вище функцій (AddTreeVar та AddHashVar – для розміщення елемента; GetTreeVar та GetHashVar – для пошуку елемента). Функції пошуку й розміщення елементів в таблиці в якості результату повертають посилання на елемент таблиці (структура якого описана в модулі TblElem) у разі успішного виконання і нульове посилання – в іншому випадку.

Треба відзначити, що функції розміщення ідентифікатора в таблиці організовані таким чином, що у випадку, коли на момент внесення нового ідентифікатора в таблиці вже є ідентифікатор з таким же ім'ям, то функція не додає новий ідентифікатор в таблицю, а повертає в якості результату посилання на раніше поміщений в таблицю ідентифікатор.

Таким чином, в таблиці не може бути двох і більше ідентифікаторів з однаковим ім'ям. При цьому наявність однакових ідентифікаторів у вхідному файлі не сприймається як помилка - це припустимо, так як в завданні не передбачено обмеження на наявність імен ідентифікаторів, що співпадають.

Всі перераховані функції описані в двох програмних модулях: FncHash – для таблиці ідентифікаторів, яка побудована на основі рехешування з використанням генератора псевдовипадкових чисел, і FncTree – для таблиці ідентифікаторів, яка побудована на основі комбінації хеш-адресації та бінарного дерева. Крім масивів даних для організації таблиць ідентифікаторів і функцій роботи з ними ці модулі містять також опис змінних, що використовуються для підрахунку кількості виконаних операцій порівняння при розміщенні і пошуку ідентифікатора у таблицях.

Повні тексти обох модулів (FncHash і FncTree) можна знайти на веб-сайті видавництва, в файлах FncHash.pas і FncTree.pas. Крім того, текст модуля FncTree наведено в лістингу П3.2 в додатку 3.

Зверніть увагу на те, що в розділах ініціалізації (initialization) обох модулів викликається функція початкового заповнення таблиці ідентифікаторів, а в розділах завершення (finalization) обох модулів – функція звільнення пам'яті. Це гарантує коректну роботу модулів при будь-якому порядку виклику інших функцій, оскільки Object Pascal сам забезпечує своєчасний виклик програмного коду в розділах ініціалізації та завершення модулів.

Текст програми

Крім перерахованих вище модулів необхідний ще модуль, що забезпечує інтерфейс з користувачем. Цей модуль (FormLabl) реалізує графічне вікно Tlab1Form на основі класу TForm бібліотеки VCL. Він забезпечує інтерфейс засобами Graphical User Interface (GUI) в ОС типу Windows на основі стандартних елементів управління з системних бібліотек цієї ОС. Крім програмного коду (файл FormLabl. Pas) модуль включає в себе опис ресурсів користувацького інтерфейсу (файл FormLabl.dfm). Більш докладно принципи організації користувацького інтерфейсу на основі GUI і робота систем програмування з ресурсами інтерфейсу описані в [3, 5,6,7].

Крім опису інтерфейсної форми та елементів її управління модуль FormLabl містить три змінні (iCountNum, iCountHash, iCountTree), що служить для накопичення статистичних результатів по мірі розміщення та пошуку ідентифікаторів у таблицях, а також функцію (procedure ViewStatistic) для відображення накопиченої статистичної інформації на екрані.

Інтерфейсна форма, описана в модулі, містить наступні основні елементи управління:

- поле введення імені файлу (EditFi1e), кнопка вибору імені файлу з каталогів файлової системи (BtnFile), кнопка читання файлу (BtnLoad);
- багаторядкове поле для відображення прочитаного файлу (ListIdents);
- поле вводу імені шуканого ідентифікатора (EditSearch);
- кнопка для пошуку введеного ідентифікатора (BtnSearch) – цією кнопкою однократно викликається процедура пошуку (procedure SearchStr);
- кнопка автоматичного пошуку всіх ідентифікаторів (BtnAllSearch) - цією кнопкою процедура пошуку ідентифікатора (procedure SearchStr) викликається циклічно для всіх зчитаних з файлу ідентифікаторів (для всіх, перерахованих в полі ListIdents);
- кнопка скидання накопиченої статистичної інформації (BtnReset);
- поля для відображення статистичної інформації;
- кнопка завершення роботи з програмою (BtnExit).

Зовнішній вигляд цієї форми наведено на рис. 1.4.

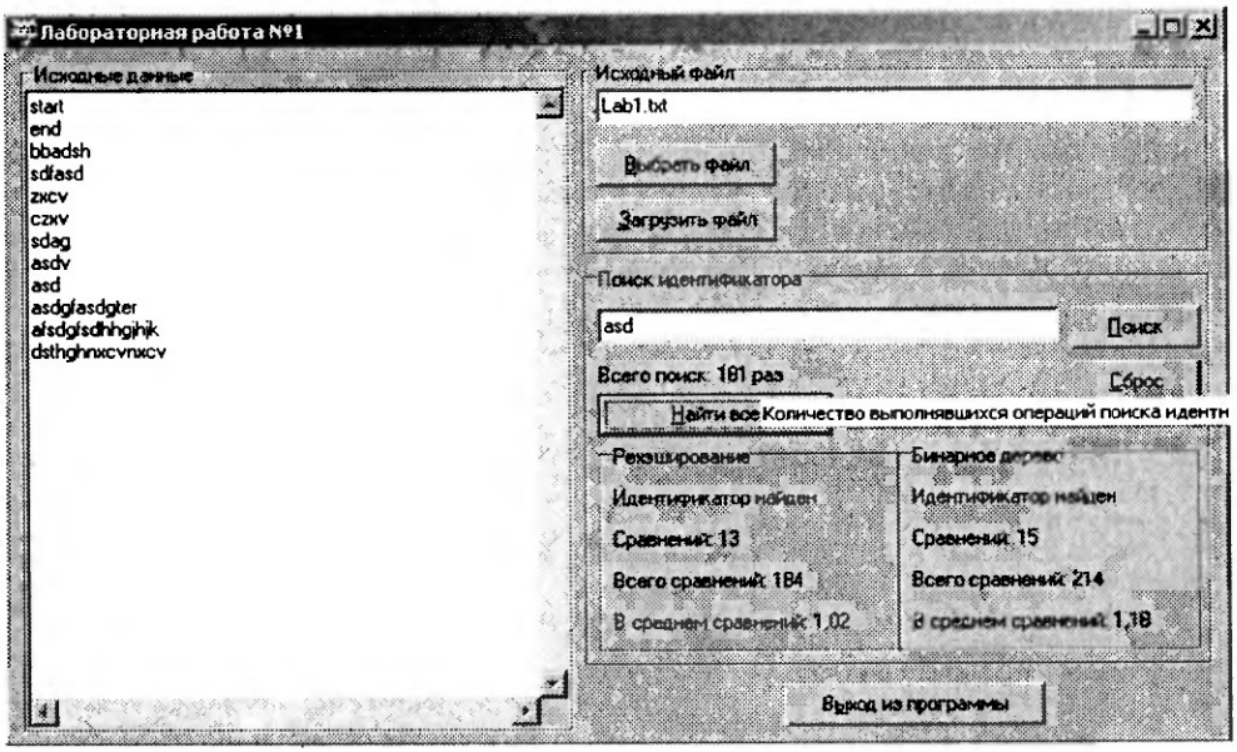


Рис. 4. Зовнішній вигляд інтерфейсної форми для лабораторної роботи №1

Функція читання вмісту файлу з ідентифікаторами (procedure TLablForm.BtnLoadClick) викликається клацанням по кнопці BtnLoad. Вона організована таким чином, що спочатку вміст файлу читається в багаторядкове поле ListIdents, а потім всі прочитані ідентифікатори записуються у дві таблиці ідентифікаторів. Кожен рядок файлу вважається окремим кодом, пробіли на початку і в кінці рядка ігноруються. При помилці розміщення ідентифікатора в одній з таблиць видається попереджувальне повідомлення (наприклад, у разі, коли буде зчитано більше, ніж 223 різних ідентифікаторів, тоді рехешування стане неможливим і буде видане повідомлення про помилку).

Функція пошуку ідентифікатора (procedure TLablForm.SearchStr) викликається одноразово клацанням по кнопці BtnSearch (процедура procedure TLablForm.BtnSearchClick) або багаторазово клацанням по кнопці BtnAllSearch (процедура procedure TLablForm.BtnAllSearchClick). Пошук ведеться відразу в двох таблицях, результати пошуку та накопичена статистична інформація відображаються у відповідних полях.

Повний текст програмного коду модуля інтерфейсу з користувачем і опис ресурсів користувацького інтерфейсу знаходяться в архіві, які розташовуються на веб-сайті видавництва, в файлах FormLabl.Pas і FormLabl.Dfm відповідно.

Повний текст всіх програмних модулів, що реалізують розглянутий приклад для лабораторної роботи № 1, можна знайти в архіві, який розташовується на веб-сайті, у підкаталогах LABS і COMMON (до підкаталогу COMMON внесено ті програмні модулі, вихідний текст яких не залежить від вхідної мови та завдання на лабораторну роботи). Головним файлом проекту є файл LAB1.DPR в субдиректорії LABS. Крім того, текст модуля FncTree наведено в лістингу П3.1 в додатку 3.

Висновки по виконаній роботі.

В результаті виконання програмного коду для ряду тестових файлів було встановлено, що при заповненні таблиці ідентифікаторів до 20% (до 45 ідентифікаторів) для пошуку та розміщення ідентифікатора з використанням рехешування на основі генератора псевдовипадкових чисел в середньому необхідно менше число порівнянь, ніж при використанні хеш-адресації в комбінації з бінарним деревом. При заповненні таблиці

від 20% до 40% (приблизно 45-90 ідентифікаторів) обидва методи мають приблизно рівні показники, але при заповненні таблиці більше, ніж на 40% (90-223 ідентифікаторів), ефективність комбінованого методу в порівнянні з методом рехешування різко зростає. Якщо на вході є більше, ніж 223 ідентифікатори, рехешування повністю перестає працювати.

Таким чином, встановлено, що комбінований метод працездатний навіть при наявності найпростішої хеш-функції і дає непогані результати (в середньому 3-5 порівнянь на вхідних файлах, що містять 500-700 ідентифікаторів), в той час як метод на основі рехешування для реальної роботи вимагає більш складної хеш-функції з діапазоном значень в кілька тисяч чи навіть десятків тисяч.

ЛАБОРАТОРНА РОБОТА № 6. ПРОЕКТУВАННЯ ЛЕКСИЧНОГО АНАЛІЗАТОРА

Мета роботи: вивчення основних понять теорії регулярних граматик, ознайомлення з призначенням і принципами роботи лексичних аналізаторів (сканерів), отримання практичних навиків побудови сканера на прикладі заданої простої вхідної мови.

Для виконання лабораторної роботи потрібно написати програму, яка виконує лексичний аналіз вхідного тексту відповідно до завдання і породжує таблицю лексем з вказівкою їх типів і значень. Текст на вхідній мові задається у вигляді символьного (текстового) файлу. Програма повинна видавати повідомлення про наявність у вхідному тексті помилок, які можуть бути виявлені на етапі лексичного аналізу.
Довжину ідентифікаторів і рядкових констант вважати за обмежену 32 символами.
Програма повинна допускати наявність коментарів необмеженої довжини у вхідному файлі. Форму організації коментарів пропонується вибрати самостійно.

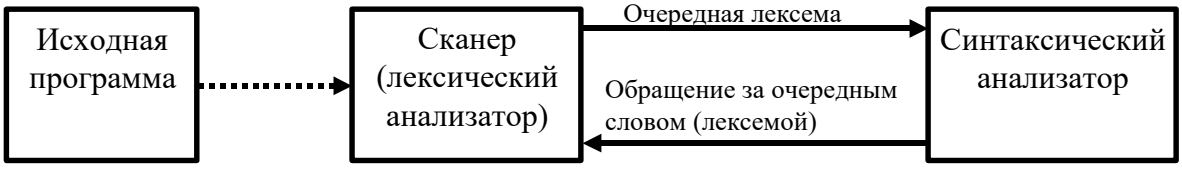
КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Лексичний аналізатор (або сканер) - це частина компілятора, яка читає літери програми на початковій мові і будує з них слова (лексеми) початкової мови. На вхід лексичного аналізатора надходить текст початкової програми, а вихідна інформація передається для подальшої обробки компілятором на етапі синтаксичного аналізу і розбору.
З теоретичної точки зору лексичний аналізатор не є обов'язковою, необхідною частиною компілятора. Його функції можуть виконуватися на етапі синтаксичного розбору. Проте існує декілька причин, виходячи з яких до складу практично всіх компіляторів включають лексичний аналіз. Ці причини полягають в наступному:

- спрощується робота з текстом початкової програми на етапі синтаксичного розбору і скорочується об'єм оброблюваної інформації, оскільки лексичний аналізатор структурує початковий текст програми, що поступає на вхід, і викидає всю незначущу інформацію;
- для виділення в тексті і розбору лексем можливо застосовувати просту, ефективну і теоретично техніку аналізу, що добре пропрацювала, тоді як на етапі синтаксичного аналізу конструкцій початкової мови використовуються досить складні алгоритми розбору;
- сканер відокремлює складний по конструкції синтаксичний аналізатор від роботи безпосередньо з текстом початковий програми, структура якого може варіюватися залежно від версії вхідної мови - при такій конструкції компілятора при переході від однієї версії мови до іншої досить тільки перебудувати відносно простий сканер.

Функції, що виконуються лексичним аналізатором,, і склад лексем, які він виділяє в тексті початкової програми, можуть мінятися залежно від версії компілятора. В основному лексичні аналізатори виконують виключення з тексту початкової програми коментарів і незначущих пропусків, а також виділення лексем наступних типів: ідентифікаторів, рядкових, символьних і числових констант, ключових (службових) слів вхідної мови.
У простому випадку фази лексичного і синтаксичного аналізу можуть виконуватися компілятором *послідовно* – лексичний аналізатор повністю обробляє текст початкової програми і лише після цього починає працювати синтаксичний аналізатор.
Але для багатьох мов програмування інформації на етапі лексичного аналізу може бути недостатньо для однозначного визначення типа і кордонів чергової лексеми. Ілюстрацією такого випадку може служити приклад оператора програми на мові Фортран, коли по частці тексту *DO 10 I=1*... неможливо визначити типа оператора (а відповідно, і кордони лексем). В разі DO *10 I=1.15* - це буде привласнення речовій змінній *DO10I* значення константи *1.15* (пропуски у Фортрані ігноруються), а в разі *DO 10 I=1,15* - це цикл з перерахуванням від 1 до 15 по цілочисельній змінній *I* до мітки *10.*

В цьому випадку лексичний і синтаксичний аналізатори - це взаємозв'язані частки, які функціонують *паралельно*. Лексичний розбір початкового тексту в такому варіанті виконується поетапно так, що синтаксичний аналізатор, виконавши розбір чергової конструкції мови, звертається до сканера за наступною лексемою. При цьому він може повідомити інформацію про те, яку лексему слід чекати. В процесі розбору може навіть відбуватися «відхід назад», щоб виконати аналіз тексту на іншій основі.
Роботу синтаксичного і лексичного аналізаторів при паралельній обробці тексту початкової програми можна змалювати у вигляді схеми на мал. 2.



Мал. 2. Взаємодія синтаксичного і лексичного аналізаторів.

Надалі виходитимемо з припущення, що всі лексеми можуть бути однозначно виділені сканером на етапі лексичного розбору – лексичний і синтаксичний аналізатори працюють послідовно (для більшості сучасних мов програмування це справедливо).
Ось приклад фрагмента тексту програми на мові Паскаль і відповідної йому таблиці лексем (таблиця. 1):

```
...
begin
   for i:=1 to N do
      fg := fg * 0.5
...
```

Таблиця 1
Таблиця лексем програми

Лексема	**Тип лексеми**	**Значення**
begin	Ключове слово	X1
for	Ключове слово	X2
i	Ідентифікатор	i : 1
:=	Знак привласнення	
1	Цілочисельна константа	1
to	Ключове слово	X3
N	Ідентифікатор	N : 2
do	Ключове слово	X4
fg	Ідентифікатор	fg : 3
:=	Знак привласнення	
fg	Ідентифікатор	fg : 3
*	Знак арифметичної операції	
0.5	Речова константа	0.5

Вид представлення інформації після виконання лексичного аналізу цілком залежить конструкції компілятора. Але в загальному виді її можна уявляти як таблицю лексем, яка в кожній строчці повинна містити інформацію про вид лексеми, її тип і, можливо, значення. Зазвичай така таблиця має два стовпці: перший - рядок лексеми, другої, - покажчик на інформацію про лексему, може бути включений і третій стовпець - тип лексем.

Лексичний аналізатор має справу з таким об'єктами, як різного роду константи і ідентифікатори (до останніх відносяться і ключові слова). Мова констант і ідентифікаторів в більшості випадків є регулярною - тобто, може бути описаний за допомогою регулярних (праволінійних або ліволінійних) граматик [1,3,4]. Розпізнавачами для регулярних мов є скінченні автомати. Існують правила, за допомогою яких для будь-якої регулярної граматики може бути побудований недетермінований скінченний автомат, що розпізнає ланцюжки мови, заданої цією граматикою.

Недетермінований скінченний автомат задається п'ятіркою:

M=(Q,,,q0,Fδ)

де:

Q – кінцева множина станів автомата;

Σ - скінченна множина допустимих вхідних символів;

δ - задане відображення безлічі **Q*Σ** в множина підмножин P**P(Q)** δ: **Q*Σ→P(Q)** (інколи δназивають функцією переходів автомата);

$q_0 \in$ **Q** -початковий стан автомата;

F$\subseteq$ **Q** - безліч завершальних станів автомата.

Робота автомата виконується по тактах. На кожному черговому такті і автомат, знаходячись в деякому стані $q_i \in Q$, прочитує черговий символ $w \in$ **Σ** з вхідного ланцюжка символів і змінює своє перебування на $q_{i+1}=\delta(q_i,w)$, , після чого покажчик в ланцюжку вхідних символів пересувається на наступний символ і зачинається такт i+1. Так продовжується до тих пір, поки ланцюжок вхідних символів не закінчиться. Кінець ланцюжка символів часто позначають особливим символом ⊥. Вважається також, що перед тактом 1 автомат знаходиться в початковому стані q_0.

Говорять, що автомат допускає ланцюжок $\alpha \in \Sigma^*$, якщо в результаті виконання всіх тактів роботи над цим ланцюжком він опиниться в стані **q∈ F** Мова, визначувана автоматом, є множиною всіх ланцюжків, що допускаються автоматом. Для аналізу ланцюжка за допомогою кінцевого автомата досить подати її на вхід автомата, виконати всі такти його роботи і визначити, чи перейшов автомат в результаті роботи в один із заданих кінцевих станів.

Графічно автомат зображується навантаженим однонаправленим графом, в якому вершини представляють стани, дуги відображають переходи з одного стану в інший, а символи навантаження (позначки) дуг відповідають функції переходу. Якщо функція переходу передбачає перехід з стану q в q’ по декількох символах, то між ними будується одна дуга, яка позначається всіма символами, по яких відбувається перехід з q в q’.

Недетермінований автомат незручний для аналізу ланцюжків, оскільки в нім можуть зустрічатися стани, що допускають неоднозначність, тобто такі, з яких виходить дві або більш за 1-ну дугу, позначених одним і тим же символом. Очевидно, що програмування роботи такого автомата - нетривіальне завдання.

Доведено, що будь-який недетермінований автомат може бути перетворений в детермінований так, щоб їх мови збігалися [1,2,3] (говорять, що автомати еквівалентні).

Детермінований скінченний автомат задається п'ятіркою:

M’=(**Q’**,**Σ**,δ’,q_0’,**F’**),

де:

Q’ - кінцева множина станів автомата;

Σ - скінченна множина допустимих вхідних символів автомата;

q_0’$\in$ **Q’** початковий стан автомата;

δ’ - задане відображення безлічі **Q’* Σ**в безліч **Q’** δ: **Q’*Σ→Q’**;

F$\subseteq$ **Q’** - множина закінчуючих станів автомата.

Після побудови скінченний детермінований автомат може бути мінімізований, тобто для нього може бути побудований еквівалентний йому автомат з мінімальним числом станів.

Можна написати функцію, що відображає функціонування будь-якого детермінованого кінцевого автомата. Щоб запрограмувати таку функцію, досить мати змінну, яка б відображувала поточний стан автомата, а переходи автомата з одного стану в інше на основі символів вхідного ланцюжка можуть бути побудовані за допомогою операторів вибору. Робота функції повинна тривати до тих пір, поки не буде досягнутий кінець вхідного ланцюжка. Для обчислення результату функції необхідно по її завершенню проаналізувати стан автомата. Якщо це один з скінченних станів, то функція виконана успішно, і вхідний ланцюжок приймається, якщо ні - то вхідний ланцюжок не належить заданій мові.

Розгледимо приклад аналізу лексем, що є цілочисельними константами без знаку у форматі мови Сі. Відповідно до вимог мови, такі константи можуть бути десятковими, вісімковими, або шіснадцятковтми. Вісімковою константою вважається число, що починається з 0 і містить цифри від 0 до 7; шіснадцяткова константа повинна починатися з послідовності символів 0x і може містити цифри і букви від A до F (розглядатимемо тільки прописні букви). Решта чисел вважається за десяткові (правила їх написання нагадувати, напевно, непотрібно). Вважатимемо, що всяке число завершується символом кінця рядка ⊥.

Ті, що розглядали вище за правило можуть бути записані у формі Бекуса-наура в граматиці цілочисельних констант без знаку мови Сі (для уникнення плутанини термінальні символи граматики виділені жирним шрифтом).

G({S,G,X,H,Q,Z},{**0**...**9**,**A**...**F,** ⊥},P,S)

P: S→ G⊥|Z⊥|H⊥|Q⊥
G→**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**|G**0**|G**1**|G**2**|G**3**|G**4**|G**5**|G**6**|G**7**|G**8**|G**9**|Z**8**|Z**9**|Q**8**|Q**9**
H→X**0**|X**1**|X**2**|X**3**|X**4**|X**5**|X**6**|X**7**|X**8**|X**9**|X**A**|X**B**|X**C**|X**D**|X**E**|X**F**|
H**0**|H**1**|H**2**|H**3**|H**4**|H**5**|H**6**|H**7**|H**8**|H**9**|H**A**|H**B**|H**C**|H**D**|H**E**|H**F**
X→Z**x**
Q→Z**0**|Z**1**|Z**2**|Z**3**|Z**4**|Z**5**|Z**6**|Z**7**|Q**0**|Q**1**|Q**2**|Q**3**|Q**4**|Q**5**|Q**6**|Q**7**
Z→ **0**

Добре видно, що дана граматика є регулярною граматикою (ліволінійною). Кінцевий детермінований автомат **M'**({N,Z,X,H,Q,G,S,ER},{0...9,A...F,⊥},δ,N,{S}), який розпізнає мову, задану цією граматикою, змальований на мал. 3. Початковим станом автомата є стан *N*. У автомат додатково введений особливий стан *ER*, що позначає помилку в розпізнаванні ланцюжка символів. На графі автомата дуги, що йдуть в цей стан, не

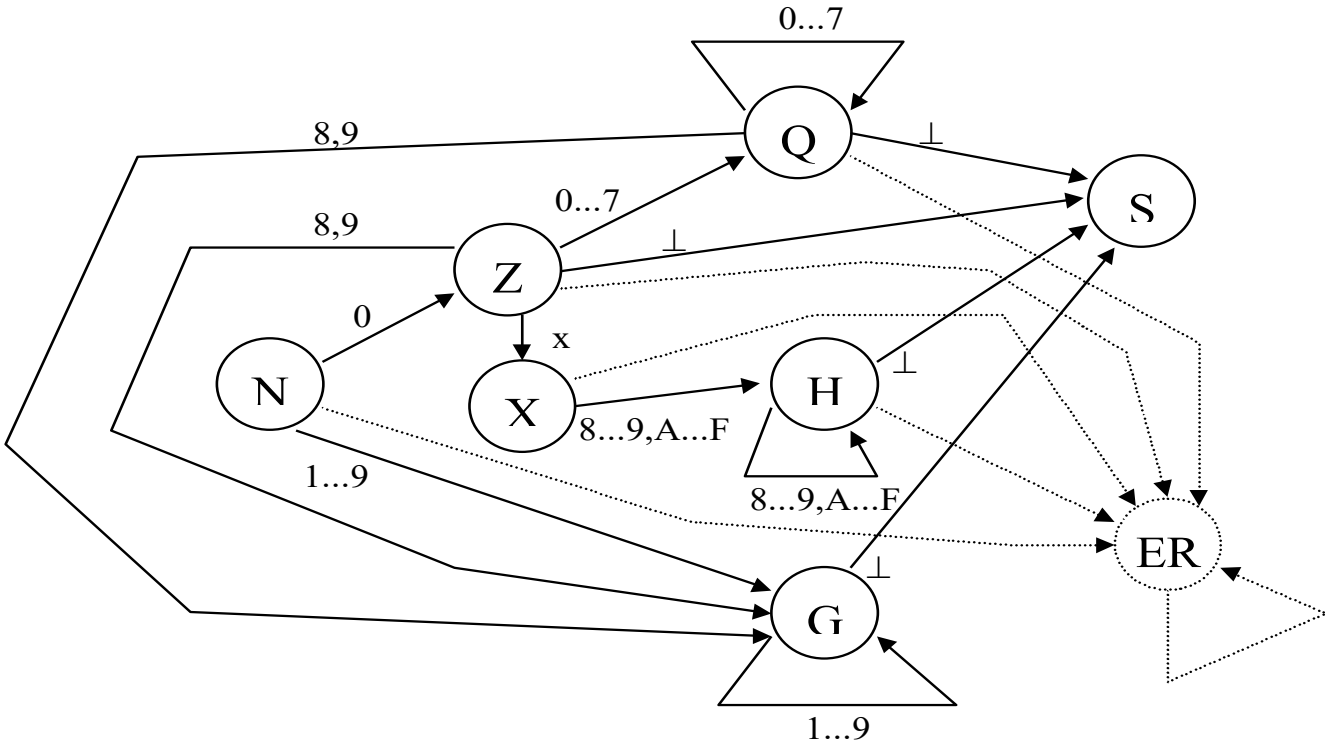


Рис. 3. Граф кінцевого детермінованого автомата, що розпізнає граматику цілих чисел мови Сі.

навантажені символами. За прийнятою угодою вони позначають функцію переходу по будь-якому символу, окрім тих символів, якими вже помічені інші дуги, що виходять з тієї ж вершини графа. Така угода прийнята, щоб не захаращувати позначеннями граф автомата (на мал. 3 таких дуги і стан *ER* виділені пунктиром).
Можна написати програму, що моделює роботу вказаного автомата. Нижче приводиться текст функції на мові Паскаль, що його реалізовує. Результат функції достеменний (*True*), якщо вхідний ланцюжок символів належить вхідній мові автомата. За кордон ланцюжка вважається символ з кодом 0 (*#0*), у функції він штучно додається в кінець ланцюжка.
У програмі змінна iState відображає поточний стан автомата, змінна i є лічильником символів вхідного рядка. Звичайно, програма, що розглядалась, може бути оптимізована (наприклад, можна відразу ж припиняти розбір по виявленню помилки), але в даному прикладі оптимізація не виконувалася, щоб можна було чітко відстежити відповідність між програмою і побудованим автоматом.

```
type
   TAutoState = ( AUTO_N, AUTO_Z, AUTO_X,
                  AUTO_Q, AUTO_H, AUTO_G,
                  AUTO_ER, AUTO_S );

function RunAuto (sInput: string): Boolean;
var
   iState : TAutoState;
   i : integer;
begin
   sInput := sInput + #0;
   iState := AUTO_N;
   i := 0;
   repeat
       i := i + 1;
      case iState of
      AUTO_N:
         case sInput[i] of
            '0':       iState := AUTO_Z;
            '1'..'9':  iState := AUTO_G;
            else       iState := AUTO_ER;
         end;
      AUTO_Z:
         case sInput[i] of
            '0'..'7':  iState := AUTO_Q;
            '8','9':   iState := AUTO_G;
            'x':       iState := AUTO_X;
            #0:        iState := AUTO_S;
            else       iState := AUTO_ER;
         end;
      AUTO_X:
         case sInput[i] of
            '0'..'9':  iState := AUTO_H;
            'A'..'F': iState := AUTO_H;
            else       iState := AUTO_ER;
         end;
      AUTO_Q:
         case sInput[i] of
```

```
            '0'..'7':  iState := AUTO_Q;
            '8','9':   iState := AUTO_G;
            #0:        iState := AUTO_S;
            else       iState := AUTO_ER;
          end;
        AUTO_H:
          case sInput[i] of
            '0'..'9':  iState := AUTO_H;
            'A'..'F': iState := AUTO_H;
            #0:        iState := AUTO_S;
            else       iState := AUTO_ER;
          end;
        AUTO_G:
          case sInput[i] of
            '0'..'9':  iState := AUTO_G;
            #0:        iState := AUTO_S;
            else       iState := AUTO_ER;
          end;
        AUTO_ER:    iState := AUTO_ER;
        end {case};
    until (sInput[i]= #0);
    RunAuto := (iState = AUTO_S);
end; { RunAuto }
```

Проте в спільному випадку завдання сканера декілька ширше, ніж просто перевірка ланцюжка символів лексеми на відповідність її вхідній мові. Сканер повинен виконати ті або інші дії із запам'ятовування розпізнаної лексеми (занесення її в таблицю лексем). Набір дій визначається реалізацією компілятора. Зазвичай ці дії виконуються відразу ж по виявленню кінця розпізнаваної лексеми, тому їх нескладно вставити у відповідні місця програми-сканера, що розглядалась вище (у тих операторів, де виявляється символ *#0*).

Друга проблема, яка вже обговорювалася вище, це виділення меж лексем. Адже у вхідному тексті лексеми не обмежені спеціальними символами. Якщо говорити в термінах програми-сканера, то визначення меж лексем - це виділення тих рядків в загальному потоці вхідних символів, для яких треба виконувати розпізнавання. У загальному випадку це завдання може бути складним, але для простих вхідних мов межі лексем розпізнаються по заданих термінальних символах. Ці символи - пропуски, знаки операцій, символи коментарів, а також роздільники (коми, крапки з комою і ін.). Набір таких термінальних символів може варіюватися залежно від вхідної мови. Поважно відзначити, що знаки операцій самі також є лексемами, і необхідно не пропустити їх при розпізнаванні тексту.

Таким чином, алгоритм роботи простого сканера можна описати так:

- є видимим вхідний потік символів програми на початковій мові до виявлення чергового символу, що обмежує лексему;
- для вибраної частини вхідного потоку виконується функція розпізнавання лексеми;
- при успішному розпізнаванні інформація про виділену лексему заноситься в таблицю лексем, і алгоритм повертається до першого етапу;
- при невдалому розпізнаванні видається повідомлення про помилку, а подальші дії залежать від реалізації сканера - або його виконання припиняється, або робиться спроба розпізнати наступну лексему (йде повернення до першого етапу алгоритму).

Робота програми-сканера продовжується до тих пір, поки не будуть проглянуті всі символи програми на початковій мові з вхідного потоку.

ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Отримати варіант завдання у викладача.
2. Розробити КС-граматику вхідної мови відповідно до завдання.
3. Підготувати і захистити звіт.
4. Написати і налаштувати програму на ЕОМ.
5. Здати працюючу програму викладачеві.

ВИМОГИ ДО ОФОРМЛЕННЯ ЗВІТУ

Звіт повинен містити наступні розділи:

- Завдання по лабораторній роботі.
- Опис КС-граматики вхідної мови у формі Бекуса-наура.
- Опис алгоритму роботи сканера або граф скінченного автомата для розпізнавання ланцюжків (відповідно до варіанту завдання).
- Текст програми (оформляється після виконання програми на ЕОМ).
- Висновки по виконаній роботі.

ОСНОВНІ КОНТРОЛЬНІ ПИТАННЯ

1. Що таке трансляція, компіляція, транслятор, компілятор?
2. З яких процесів складається компіляція? Розкажіть про загальну структуру компілятора.
3. Яку роль виконує лексичний аналіз в процесі компіляції?
4. Як зв'язані лексичний і синтаксичний аналіз?
5. Дайте визначення ланцюжка, мови. Що таке синтаксис і семантика мови?
6. Які існують методи задавання мов? Які додаткові питання необхідно вирішити при задаванні мови програмування?
7. Що таке граматика? Дайте визначення граматики.
8. Як виглядає опис граматики у формі Бекуса-наура.
9. Які класи граматик існують? Що таке регулярні граматики?
10. Дайте визначення контекстно-вільній граматиці, виводимості ланцюжка, безпосередньої виводимості, довжини виводу.
11. Що таке лексема? Розкажіть, які типи лексем існують в мовах програмування.
12. Що таке скінченний автомат? Дайте визначення детермінованого і недетермінованого скінченних автоматів.
13. Розкажіть про можливість перетворення недетермінованого кінцевого автомата в детермінований.
14. Які проблеми необхідно вирішити при побудові сканера на основі скінченного автомата?

ВАРІАНТИ ЗАВДАНЬ

1. Вхідна мова містить арифметичні вирази, розділені символом **;**(крапка з комою). Арифметичні вирази складаються з ідентифікаторів, десяткових чисел з плаваючою крапкою (у звичайній і логарифмічній формі), знаку привласнення (:=), знаків операцій +, - *, / і круглих дужок.

2. Вхідна мова містить логічні вирази, розділені символом **;**(крапка з комою). Логічні вирази складаються з ідентифікаторів, констант **true** і **false**, знаку привласнення (**:=**), знаків операцій **or, xor, and, not** і круглих дужок.

3. Вхідна мова містить операторів умови типа **if . then . else i if** . **then**, розділені символом **;**(крапка з комою). Операторів умови містять ідентифікатори, знаки порівняння <, >, =, десяткові числа з плаваючою крапкою (у звичайній і логарифмічній формі), знак привласнення (:=).

4. Вхідна мова містить операторів циклу типу **for (**.; .; .) **do**, розділені символом **;**(крапка з комою). Операторів циклу містять ідентифікатори, знаки порівняння <, >, =, десяткові числа з плаваючою крапкою (у звичайній і логарифмічній формі), знак привласнення (:=).

5. Вхідна мова містить арифметичні вирази, розділені символом **;**(крапка з комою). Арифметичні вирази складаються з ідентифікаторів, римських чисел, знаку привласнення (:=), знаків операцій +, - *, / і круглих дужок.

6. Вхідна мова містить логічні вирази, розділені символом **;**(крапка з комою). Логічні вирази складаються з ідентифікаторів, констант 0 і 1, знаку привласнення (:=), знаків операцій **or, xor, and, not** і круглих дужок.

7. Вхідна мова містить операторів умови типу **if . then . else і if** . **then**, розділені символом **;**(крапка з комою). Операторів умови містять ідентифікатори, знаки порівняння <, >, =, римські числа, знак привласнення (:=).

8. Вхідна мова містить операторів циклу типа **for (**.; .; .) **do**, розділені символом **;**(крапка з комою). Операторів циклу містять ідентифікатори, знаки порівняння <, >, =, римські числа, знак привласнення (:=).

9. Вхідна мова містить арифметичні вирази, розділені символом **;**(крапка з комою). Арифметичні вирази складаються з ідентифікаторів, шіснадцяткових чисел, знаку привласнення (:=), знаків операцій +, - *, / і круглих дужок.

10. Вхідна мова містить логічні вирази, розділені символом **;**(крапка з комою). Логічні вирази складаються з ідентифікаторів, шіснадцяткових чисел, знаку привласнення (:=), знаків операцій **or**, **xor, and, not** і круглих дужок.

11. Вхідна мова містить операторів умови типа **if** . **then . else і if . then**, розділені символом **;**(крапка з комою). Операторів умови містять ідентифікатори, знаки порівняння <, >, =, шіснадцяткові числа, знак привласнення (:=).

12. Вхідна мова містить операторів циклу типа **for (**.; .; .) **do**, розділені символом **;**(крапка з комою). Операторів циклу містять ідентифікатори, знаки порівняння <, >, =, шіснадцяткові числа, знак привласнення (:=).

13. Вхідна мова містить арифметичні вирази, розділені символом **;**(крапка з комою). Арифметичні вирази складаються з ідентифікаторів, символьних констант (один символ в одинарних лапках), знаку привласнення (:=), знаків операцій +, - *, / і круглих дужок.

14. Вхідна мова містить логічні вирази, розділені символом **;**(крапка з комою). Логічні вирази складаються з ідентифікаторів, символьних констант ‘ T’ і ‘ F’, знаку привласнення (**:=**), знаків операцій **or**, **xor, and, not** і круглих дужок.

15. Вхідна мова містить операторів умови типа **if** . **then . else і if . then**, розділені символом **;**(крапка з комою). Операторів умови містять ідентифікатори, знаки порівняння <, >, =, рядкові константи (послідовність символів в подвійних лапках), знак привласнення (:=).

16. Вхідна мова містить операторів циклу типа **for (**.; .; .) **do**, розділені символом **;**(крапка з комою). Операторів циклу містять ідентифікатори, знаки порівняння <, >, =, рядкові константи (послідовність символів в подвійних лапках), знак привласнення (:=).

Примітка:

- за римські числа вважати послідовності великих латинських букв **X**, **V і I;**
- шіснадцятковими числами вважати послідовність цифр і символів ‘a’, ‘b’, ‘c’,’d’, ’e’ і ‘f’, що починається з цифри (наприклад: 89, 45ac9, 0abc4);
- завдання по лабораторній роботі №2 взаємозв'язано із завданням по лабораторній роботі №3, для уточнення складу вхідної мови можна глянути граматику, задану в роботі №3 по відповідному варіанту.

ЛАБОРАТОРНА РОБОТА № 7. ПОБУДОВА ПРОСТОГО ДЕРЕВА ВИВЕДЕННЯ

Мета роботи: вивчення основних понять теорії граматик простого і операторного передування, ознайомлення з алгоритмами синтаксичного аналізу (розбору) для деяких класів КС-граматик, отримання практичних навиків створення простого синтаксичного аналізатора для заданої граматики операторного передування.

Для виконання лабораторної роботи потрібно написати програму, яка виконує лексичний аналіз вхідного тексту відповідно до завдання, породжує таблицю лексем і виконує синтаксичний розбір тексту по заданій граматиці з побудовою дерева розбору. Текст на вхідній мові задається у вигляді символьного (текстового) файлу. Допускається виходити з умови, що текст містить не більш за одну пропозицію вхідної мови. Програма повинна видавати повідомлення про наявність у вхідному тексті помилок.
Рекомендується програму розбити на три складові частини: лексичний аналіз, побудова ланцюжка виводу і побудова дерева виводу. Лексичний аналізатор повинен виділяти в тексті лексеми мови і замінювати їх на термінальний символ граматики (який в завданні позначений як "**a").** Отриманий після лексичного аналізу ланцюжок повинен в другій частці програми розглядуватися відповідно до алгоритму розбору. При невдалому завершенні алгоритму видається повідомлення про помилку, при вдалому – будується ланцюжок виводу. Після побудови ланцюжка виводу на її основі будується дерево розбору, в якому символи **a** послідовно замінюються на лексеми з таблиці лексем. Довжину ідентифікаторів і рядкових констант вважати обмежену 32 символами. Програма повинна допускати наявність коментарів необмеженої довжини у вхідному файлі. Форму організації коментарів пропонується вибрати самостійно.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

За ієрархією граматики Хомського виділяють 4 основних групи мов
(і граматик, що описують їх). При цьому найбільший інтерес представляють регулярні і контексно-вільні (КВ) граматики і мови. Вони використовуються при описі синтаксису мов програмування. За допомогою регулярних граматик можна описати лексеми мови - ідентифікатори, константи, службові слова та інші. На основі КВ-граматик будуються крупніші синтаксичні конструкції - описи типів і змінних, арифметичні і логічні вирази, оператори, що управляють, і, нарешті, повністю вся програма на початковій мові.
Вхідні ланцюжки регулярних мов розпізнаються за допомогою кінцевих автоматів (КА). Вони лежать в основі сканерів, що виконують лексичний аналіз і виділення слів в тексті програми на вхідній мові. Результатом роботи сканера є перетворення початкової програми в список або таблицю лексем. Подальшу її обробку виконує інша частина компілятора - синтаксичний аналізатор. Його робота заснована на використанні правил КВ-граматики, що описують конструкції початкової мови.
Взаємодія лексичного і синтаксичного аналізатора розглядувалася в попередній лабораторній роботі, тут же вивчатимемо алгоритми, які лежать в основі синтаксичного аналізу. Перед синтаксичним аналізатором стоять два основні завдання: перевірити правильність конструкцій програми, яка представляється у вигляді вже виділених слів вхідної мови, і перетворити її у вигляд, зручний для подальшої семантичної (смислової) обробки і генерації коду. Одним з таких способів уявлення є дерево синтаксичного розбору.

Розпізнавання ланцюжків КВ-мов

Клас КВ-мов допускає розпізнавання за допомогою недетермінованого кінцевого автомата із стековою (або магазинною) пам'яттю - МП-автомата. Контекстно-залежні мови - останній клас мов, які можна ефективно розпізнавати за допомогою ЕОМ. Вони

допускаються двосторонніми недетермінованими лінійно обмеженими автоматами. Для ланцюжків мови без обмежень, в загальному випадку, потрібний універсальний обчислювач (машина Тюрінга, машина з необмеженим числом регістрів тощо). Контекстно-залежні мови і мови без обмежень при створенні структур мов програмування не використовуються.

МП-автомат на відміну від звичайного КА має стек (магазин), в який можна поміщати спеціальні "магазинні" символи (звичайно це термінальні і нетермінальні символи граматики мови). Перехід з одного стану в інше залежить не лише від вхідного символу, але і від одного або декількох верхніх символів стека. Таким чином, конфігурація автомата визначається трьома параметрами: станом автомата, поточним символом вхідного ланцюжка (положенням покажчика в ланцюжку) і вмістом стека.

При виконанні переходу із стека віддаляються верхні символи, відповідні умові переходу, і додається ланцюжок, відповідний правилу переходу. Перший символ ланцюжка стає верхівкою стека. Допускаються переходи, при яких вхідний символ ігнорується (і, тим самим, він буде вхідним символом при наступному переході). Ці переходи називаються (λ-переходами. МП-автомат називається недетермінованим, якщо при одній і тій ж його конфігурації можливий більш ніж один перехід. Якщо при закінченні ланцюжка автомат знаходиться в одному із заданих кінцевих станів, а стек порожній, ланцюжок вважається прийнятий (після закінчення ланцюжки можуть бути зроблені (λ-переходи). Інакше ланцюжок символів не приймається.

Після КВ-граматики **G**(VN,VT,P,S), **V=VT**∪**VN** можна побудувати недетермінований МП-автомат, який допускає ланцюжки мови цієї граматики. Він має тільки однин стан і наступний набір переходів:

- λ(A)÷(α) - для кожного правила граматики A→α ∈ **P**, де A∈VN, **α∈V***;
- a(a)÷() - для кожного терміналу a∈VT;

Тут в круглих дужках показаний вміст верхівки стека. Перед дужками стоїть черговий символ вхідного ланцюжка або символ λ для λ-переходу, а символ ÷ розділяє стани автомата до і після виконання переходу (показує такт роботи автомату). На початку роботи автомата в стеку лежить символ S∈VN, в кінці роботи автомата стек має бути порожній.

Можна побудувати і інший автомат, який також містить один стан і має наступні переходи:

- a()÷(a)- для кожного терміналу a∈VT;
- λ(α)÷(A) - для кожного правила граматики A→α ∈ **P**, де A∈VN, α∈**V***;

У такому варіанті на початку розбору стек автомата порожній. Тоді в кінці розбору допустимого ланцюжка в стеку повинен залишитися символ S∈VN.

По недетермінованому МП-автомату завжди можна побудувати КС-граматику. Таким чином, клас КВ-мов і клас мов, МП-автоматами, що допускаються, еквівалентні.

Не кожна КВ-мова допускає розбір за допомогою детермінованого МП-автомата. Недетерміновані МП-автомати можуть розпізнавати ширший клас мов, чим детерміновані - в цьому їх відмінність від звичайних КА. Інтерес для реалізації компіляторів представляють детерміновані КС-мови - власна підмножина КВ-мов, що допускає розпізнавання за допомогою детермінованих МП-автоматів.

Два (недетермінованих) МП-автомата, побудованих вище для довільної КВ-граматики визначають два класи методів розбору КВ-мов: зверху вниз і від низу до верху.

У першому випадку (приклад - рекурсивний спуск) при прогляданні ланцюжка зліва направо природно виходитимуть ліві виводи. При детермінованому розборі проблема полягатиме в тому, яке правило застосувати для розкриття найлівішого нетермінального символу.

У другому випадку детермінований розбір зручно формалізувати в термінах "зсув" (перенесення символу з вхідного ланцюжка в магазин) і "згортка" (застосування до

вершини магазина якого-небудь правила граматики). При проглядданні вхідного ланцюжка зліва направо при цьому виходитимуть праві виводи. Проблема в цьому випадку полягає у виборі між зсувом і згорткою і у виборі правила для згортки.

Граматики передування

Програмування роботи недетермінованого МП-автомата - це складне завдання. Розроблений алгоритм, що дозволяє для довільної КВ-граматики визначити, чи належить їй заданий вхідний ланцюжок (алгоритм Кока-янгера-касамі)[4,5].
Доведено, що час роботи цього алгоритму пропорційний n3, де n - довжина вхідного ланцюжка. Для однозначної КВ-граматики при використанні іншого алгоритму (алгоритм Ерлі) цей час пропорційний n2. Подібна залежність робить ці алгоритми вимогливими до обчислювальних ресурсів, а тому мало придатними для практичних цілей. Але на практиці і не потрібний аналіз ланцюжка довільної КС-мови - більшість конструкцій мов програмування можуть бути віднесені в один з класів КС-мов, для яких розроблені алгоритми розбору, лінійно залежні від довжини вхідного ланцюжка.
КВ-мови діляться на класи відповідно до структури правил їх граматик. У кожному з класів накладаються додаткові обмеження на допустимі правила граматики.
Одним з таких класів є клас граматик передування. Вони використовуються для синтаксичного розбору ланцюжків за допомогою алгоритму “зсув-згортка”. Виділяють наступні типи граматик передування:

- простого передування;
- розширеного передування;
- слабкого передування;
- змішаної стратегії передування;
- операторного передування.

Далі розглядатимуть обмеження на структуру правил і алгоритми розбору для двох типів - граматика простого і операторного передування.
Граматикою простого передування називають таку КС-граматику **G(VN,VT,P,S)**, **V=VT** $\cup$**VN**в якій:

1. Для кожної впорядкованої пари термінальних і нетермінальних символів виконується не більше ніж один з трьох відношень передування:
   - $S_i = S_j$ ($\forall$ $S_i,S_j \in V$), якщо і тільки якщо $\exists$ правило $U \to xS_iS_{jy} \in \mathbf{P}$, де $U \in VN$, $x,y \in V^*$;
   - $S_i < S_j$ ($\forall$ $S_i,S_j \in V$), якщо і тільки якщо $\exists$ правило $U \to xS_iD_y \in \mathbf{P}$ і виведення $D^*Sjz \Rightarrow$, де $U,DV \in N$, $x,y,z \in V^*$;
   - $S_i > S_j$ ($\forall$ $S_i,S_j \in V$), якщо і тільки якщо $\exists$ правило $UxCS_{jy} \in \mathbf{P}$ і виведення C^*zS_i або $\exists$ правило $U \to xCD_y \in \mathbf{P}$ і виводи C^*zS_i і $D^*S_{jw} \Rightarrow$, де $U,C,D \in VN$, $x,y,z,w \in V^*$.
2. Різні правила, що породжують, мають різні праві частини.

Відношення =, < і > називають відношеннями передування для символів. Відношення передування єдине для кожної впорядкованої пари символів. При цьому між якими-небудь двома символами може і не бути відношення передування - це означає, що вони не можуть знаходитися поруч ні в одному елементі розбору синтаксично правильного ланцюжка. Відношення передування залежать від ладу, в якому стоять символи, і в цьому сенсі їх не можна плутати із знаками математичних операцій - наприклад, якщо $S_i > S_j$, то не обов'язково, що $S_j < S_i$ (тому знаки передування інколи позначають спеціальною крапкою: =,· <,· >·)
Метод передування заснований на тому факті, що стосунки передування між двома сусідніми символами розпізнаваного рядка відповідають трьом наступним варіантам:

- $S_i < S_i+1$, якщо символ S_i+1 - крайній лівий символ деякої основи;
- $S_i > S_i+1$, якщо символ S_i - крайній правий символ деякої основи;
- $S_i = S_i+1$, якщо символи S_i і S_i+1 належать одній основі.

Виходячи з цих співвідношень виконується розбір рядка для граматики передування.
На підставі відношень передування будують матрицю передування граматики. Рядки матриці передування позначаються першими символами, стовпці - другими символами відношень передування, а в клітки матриці на перетині відповідного стовпця і рядка поміщаються знаки стосунків. При цьому порожні клітки матриці говорять про те, що між даними символами немає жодного відношення передування.
Матрицю передування граматики можна побудувати, спираючись безпосередньо на визначення відношень передування, але зручніше скористатися двома додатковими множинами - множиною крайніх лівих і множиною крайніх правих символів відносно нетермінальної граматики. Ця множина визначається таким чином:

- **L(U)=** {T | ∃ U*Tz}, ⇒U,T∈V, z∈V* - множина крайніх лівих символів щодо нетермінального символу U (ланцюжок z може бути і порожнім ланцюжком);
- **R(U)=** {T | ∃ U*zT}, ⇒U,T∈V, z∈V* - множина крайніх правих символів щодо нетермінального символу U.

Тоді відношення передування можна визначити так:

- $S_i = S_j$ (∀ S_i,S_j∈**V**), якщо ∃ правило U→xS_iS_jy ∈**P**, де U∈**VN**, x,y∈$\mathbf{V}^*$;
- $S_i < S_j$ (∀ S_i,S_j∈**V**), якщо ∃ правило U→xS_iDy ∈**P** і S_j∈**L**(D), де U,D∈**VN**, x,y∈$\mathbf{V}^*$;
- $S_i > S_j$ (∀ S_i,S_j∈**V**) , якщо ∃ правило U→xCS_{jy} ∈**P** і S_i∈**R**(C) або ∃ правило U→xCD_y ∈**P** і S_i∈**R**(C), S_j∈**L**(D), де U,C,D∈**VN**, x,y∈$\mathbf{V}^*$.

Таке визначення відношень зручніше на практиці, оскільки не вимагає побудови виводів, а множина **L(U)** і **R(U)** може бути побудована для кожного нетермінального символу UVN по дуже простому алгоритму:
Крок 1. Для кожного нетермінального символу U шукаємо всі правила, U, що містять, в лівій частці. У множині L(U) включаємо найлівіший символ з правої частини правил, а в множині R(U) - самий крайній символ правої частини. Переходимо до кроку 2.
Крок 2. Для кожного нетермінального символу U: якщо множина **L(U)** містить нетермінальні символи граматики U',U"..., то його треба доповнити символами, що входять у відповідну множину L(U'), L(U") ... і що не входять в L(U). Ту ж операцію треба виконати для R(U).
Крок 3. Якщо на попередньому кроці хоч би одна множина **L(U)** або **R(U)** для деякого символу граматики змінилася, то треба повернутися до кроку 2, інакше побудова закінчена.
Після будування множини **L(U)** і **R(U)** за правилами граматики створюється матриця передування. Матрицю передування доповнюють символами $\perp_н$ і $\perp_к$ (початок і кінець ланцюжка). Для них визначені наступні відношення передування:
$\perp_н < a$, ∀ a∈V, якщо ∃ S*ax, де S∈VN, x∈V* або (з іншого боку) якщо a∈L(S);
$\perp_к > a$, ∀ a∈V, якщо ∃ S*xa, де S∈VN, x∈V* або (з іншого боку) якщо a∈R(S).
Граматикою операторного передування називається приведена КС-граматика без λ-правил (е-правил), в якій праві частки продукцій не містять суміжних нетермінальних символів. Для граматики операторного передування відношення передування можна задати на безлічі термінальних символів (включаючи символи $\perp_н$ і до).
Стосунки передування для граматики операторного передування **G**(**VN,VT,P,S**) задаються таким чином:

- a = b, якщо і тільки якщо існує правило U→xaby∈**P** або правило U→xaCby, де a,b∈VT, U,C∈VN, x,y∈V*;
- a < b, якщо і тільки якщо існує правило U→xaCy **P** і виведення C⇒*bz або виведення C⇒*Dbz, де a,b∈VT, U,C,D∈VN, x,y,z∈V*;
- a > b, якщо і тільки якщо існує правило U→xCby∈**P** і виведення C⇒*za або виведення C⇒*zaD, де a,b∈VT, U,C,D∈VN, x,y,z∈V*.

У граматиці операторного передування різні правила, що породжують, мають різні праві частки. Для граматики операторного передування теж будується матриця передування, але вона містить тільки термінальні символи граматики.
Для побудови цієї матриці зручно ввести безліч крайніх лівих і крайніх правих термінальних символів щодо нетермінального символу U - $\mathbf{L_t(U)}$ або $\mathbf{R_t(U)}$:

- $\mathbf{L_t(U)}$= {t | ∃ U⇒*tz або ∃ U⇒*Ctz }, де t∈VT, U,C∈VN, z∈V*;
- $\mathbf{R_t(U)}$= {t | ∃ U⇒*zt або ∃ U⇒*ztC }, де t∈VT, U,C∈VN, z∈V*.

Тоді визначення стосунків операторного передування виглядатимуть так:

- a = b, якщо ∃ правило U→xaby ∈**P** або правило U→xaCby, де a,b∈VT, U,C∈VN, x,y∈V*;
- a < b, якщо ∃ правило U→xaCy ∈**P** і b∈Lt(C), де a,b∈VT, U,C∈VN, x,y∈V*;
- a > b, якщо ∃ правило U→xCby ∈**P** і a∈R_t(C), де a,b∈VT, U,C∈VN, x,y∈V*.

У даних визначеннях ланцюжка символів x,y,z можуть бути і порожніми ланцюжками.
Для знаходження безлічі $\mathbf{L_t(U)}$ і $\mathbf{R_t(U)}$ використовується наступний алгоритм:
Крок 1. Для кожного нетермінального символу граматики U будується безліч **L(U)** і R(U).
Крок 2. Для кожного нетермінального символу граматики U шукаються правила виду U→tz і U→Ctz, де t∈VT, C∈VN, z∈V*; термінальні символи t включаються в безліч Lt(U). Аналогічно для безлічі Rt(U) шукаються правила виду U→zt →і U→ztC.
Крок 3. Є видимою безліч **L(U)**, в яку входять символи U',U"... Безліч $\mathbf{L_t(U)}$ доповнюється символами, що входять в $\mathbf{L_t}$(U'), $\mathbf{L_t}$(U") ... і що не входять в $\mathbf{L_t(U)}$.
Аналогічна операція виконується і для безлічі $\mathbf{R_t(U)}$ на основі безлічі **R(U).**
Для практичного використання матрицю передування доповнюють символами $\perp_н$ і до (зачало і кінець ланцюжка). Для них визначені наступні стосунки передування:
$\perp_н$ < a, ∀ aVT, якщо ∃ S⇒*ax або S⇒*Cax, де S,C∈VN, x∈V* або якщо aL_t(S);
$\perp_{до}$ > a, ∀ aVT, якщо ∃ S⇒*xa або ∃ S⇒*xaC, де S,C∈VN, x∈V* або якщо aR_t(S).

Алгоритм «сув-згортка» для граматик простого і операторного передування

Алгоритм "крок-згортка" для граматики простого передування. Даний алгоритм виконується МП-автоматом з одним станом. Стосунки передування служать для того, щоб визначити в процесі виконання алгоритму, яка дія - зрушення або згортка - повинно виконуватися на даному кроці і однозначно вибрати правило про згортку. У початковому стані автомата голівка обробляє перший символ, в кінець ланцюжка поміщений символ $\perp_{до}$.
Розбір вважається закінченим (алгоритм завершується), якщо голівка автомата яка читається оглядає символ $\perp_н$ і при цьому більше не може бути виконана згортка.
Вирішення про ухвалення ланцюжка залежить від вмісту стека. Автомат приймає ланцюжок, якщо в результаті завершення алгоритму він знаходиться в кінцевому стані, коли в стеку знаходяться початковий символ граматики S і символ $\perp_н$. Виконання алгоритму може бути перерване, якщо на одному з його кроків виникне помилка. Тоді вхідний ланцюжок не приймається.
Алгоритм складається з наступних кроків:
Крок 1. Помістити у верхівку стека символ $\perp_н$.
Крок 2. Порівняти символ, що знаходиться на вершині стека, з поточним символом стрічки.
Крок 3. Якщо має місце відношення < або =, то провести перенесення і повернутися до кроку 2.
Крок 4. Якщо має місце відношення >, то провести згортку, тобто знайти на вершині стека всі символи, зв'язані відношенням = (основу) і замінити їх на ліву частку відповідного правила граматики (якщо символів, зв'язаних відношенням =, на верхівці

стека немає, то для правила використовується один, верхній символ). Якщо розбір не закінчений, то повернутися до кроку 2.

Помилка в процесі виконання алгоритму виникає, коли неможливо виконати черговий крок - наприклад, якщо не встановлено відношення передування між двома порівнюючими символами (на кроках 2 і 4) або якщо не удається знайти потрібне правило в граматиці (на кроці 4). Тоді виконання алгоритму негайно уривається.

Алгоритм "зсув-згортка" для граматики операторного передування в цілому схожий на алгоритм для граматик простого передування. Він виконується тим же МП-автоматом і має ті ж умови завершення і виявлення помилок. Основна відзнака полягає в тому, що при роботі із стеком в цьому алгоритмі не беруться до уваги нетермінальні символи, що знаходяться в нім, і при порівнянні шукається найближчий до верхівки стека термінальний символ. Проте після виконання порівняння і визначення кордонів основи при пошуку правила в граматиці нетермінальні символи слідує, безумовно, брати до уваги.

Крок 1. Помістити у верхівку стека символ $\perp_н$.

Крок 2. Порівняти символ, що знаходиться на вершині стека, ігноруючи всі нетермінальні символи, з поточним символом стрічки.

Крок 3. Якщо має місце відношення < або =, то провести перенесення і повернутися до кроку 2.

Крок 4. Якщо має місце відношення >, то провести згортку, тобто знайти на вершині стека (знову ж таки ігноруючи нетермінальні символи) всі символи, зв'язані відношенням = (основу) і замінити їх на ліву частку відповідного правила граматики (при виборі правила нетермінальні символи повинні враховуватися). Якщо розбір не закінчений, то повернутися до кроку 2.

Кінцева конфігурація автомата збігається з конфігурацією при розпізнаванні ланцюжків граматик простого передування.

Приклад побудови розпізнавача для граматики операторного передування.

Розгледимо як приклад граматиці **G**({S,B,T,J},{**-**,&,^,(,),p},P,S) (термінальні символи виділені жирним шрифтом):

P: S → **-B** (правило 1)
B → T | B&T (правила 2 і 3)
T → J | T^J (правила 4 і 5)
J → (B) | **p** (правила 6 і 7)

Видно, що ця граматика є граматикою операторного передування.

Побудуємо безліч крайніх лівих і крайніх правих символів **L(U)**, R(U) щодо всіх нетермінальних символів граматики. Результат побудови приведений в таблиці. 2.

На основі отриманої безлічі побудуємо безліч крайніх лівих і крайніх правих термінальних символів $\mathbf{L_t(U)}$, $\mathbf{R_t(U)}$ щодо всіх нетермінальних символів граматики. Результат (другий і третій кроки побудови) приведений в таблиці. 3.

Таблиця 2.

Безліч крайніх правих і крайніх лівих символів граматики (по кроках побудови)

Символ (U)	Крок 1 (початок побудови)		Останній крок (результат)	
	L(U)	**R(U)**	**L(U)**	**R(U)**
J	**(p**	**) p**	**(p**	**) p**
T	J T	J	J T **(p**	J **) p**
B	T B	T	T B J **(p**	T J **) p**
S	**-**	B	**-**	B T J **) p**

Таблиця 3.

Безліч крайніх правих і лівих термінальних символів граматики (по кроках побудови)

Символ (U)	Крок 1 (початок побудови)		Останній крок (результат)	
	Lt(U)	**Rt(U)**	**Lt(U)**	**Rt(U)**
J	**(p**	**) p**	**(p**	**) p**
T	^	^	**^ (p**	**^) p**
B	**&**	**&**	**& ^ (p**	**& ^) p**
S	**-**	**-**	**-**	**- & ^) p**

На основі цієї безлічі і правил граматики **G** побудуємо матрицю передування граматики (таблиця. 4).

Таблиця 4.

Матриця передування граматики

Символи	**-**	**&**	^	**(**	**)**	**p**	$\perp_{до}$
-		<	<	<		<	>
&		>	<	<	>	<	>
^		>	>	<	>	<	>
(		<	<	<	=	<	
)		>	>		>		>
p		>	>		>		>
$\perp_{н}$	<						

Переглянемо, як заповнюється матриця передування в таблиці. 4 на прикладі символу **&**. У правилі граматики B → B&T (правило 3) цей символ стоїть зліва від нетермінального символу T. У безліч $\mathbf{L_t(T)}$ входять символи: ^ (**p.** Ставимо знак < у клітках матриці, відповідних цим символам, в рядку для символу **&**. В той же час в цьому ж правилі символ & стоїть праворуч від нетермінального символу B. У безліч $\mathbf{R_t(B)}$ входять символи: **&** ^) **p.** Ставимо знак > у клітках матриці, відповідним цим символам, в стовпці для символу **&**. Більше символ & ні у якому правилі не зустрічається, означає заповнення матриці для нього закінчено, беремо наступний символ і продовжуємо заповнювати матрицю таким же методом.

Алгоритм розбору ланцюжків граматики операторного передування ігнорує нетермінальні символи. Тому має сенс перетворити початкову граматику так, щоб залишити в ній тільки один нетермінальний символ. Тоді отримаємо наступний вигляд правил:

P: E→ **-E** (правило 1)
E→E | E&E (правила 2 і 3)
E→E | E^E (правила 4 і 5)
E→ (E) | **p** (правила 6 і 7)

Це перетворення не веде до створення еквівалентної граматики і виконується тільки після побудови всіх множин і матриці передування. Саме перетворення виконується тільки з метою ефективнішого виконання алгоритму розбору, в який вже закладені необхідні дані про лад вживання правил при створенні матриці передування.

Розгледимо роботу алгоритму розпізнавання на прикладах. Послідовність розбору записуватимемо у вигляді послідовності конфігурацій МП-автомата з трьох складових: не проглянута автоматом частка вхідного ланцюжка, вміст стека і послідовність правил граматики. Оскільки автомат має тільки один стан, то для визначення його конфігурації вистачає дві складові - положення прочитуючої голівки у вхідному ланцюжку і вмісту стека, - але послідовність номерів правил несе додаткову корисну інформацію, яка має

бути потім використана компілятором на подальших етапах для семантичної обробки і для генерації коду об'єктної програми. (Крім того, послідовність застосованих правил робить приклад наочнішим).
Позначатимемо такт автомата символом ÷. Введемо також додаткове позначення ÷п, якщо на даному такті виконувалося перенесення, і ÷с, якщо виконувалася згортка.
Послідовності розбору ланцюжків вхідних символів будуть, таким чином, мати вигляд:
Приклад 1. Вхідний ланцюжок **-p&p^(p)**.
{**-p&p^(p)**⊥к; ⊥н; ∅} ÷п {**p&p^(p)**⊥к; ⊥н-; ∅} ÷п {**&p^(p)**⊥к; ⊥н-**p**; ∅} ÷с {**&p^(p)**⊥к; ⊥н-E; 7} ÷п {**p^(p)**⊥к; ⊥н-E**&**; 7} ÷п {**^(p)**⊥к; ⊥н-E**&p**; 7} ÷с {**^(p)**⊥к; ⊥н-E**&**E; 7,7} ÷п {**(p)**⊥к; ⊥н-E**&**E^; 7,7} ÷п {**p)**⊥к; ⊥н-E**&**E^**(**; 7,7} ÷п {**)**⊥к; ⊥н-E**&**E^**(p**; 7,7} ÷с {**)**⊥к; ⊥н-E**&**E^**(**E; 7,7,7} ÷п
{⊥к; ⊥н-E**&**E^**(**E**)**; 7,7,7} ÷с {⊥к; ⊥н-E**&**E^E; 7,7,7,6} ÷с {⊥к; ⊥н-E**&**E; 7,7,7,6,5} ÷п {⊥к; ⊥н-E; 7,7,7,6,5,3} ÷с {⊥к; ⊥нE; 7,7,7,6,5,3,1}
Приклад 2. Вхідний ланцюжок **-p^p(p)**.
{**-p^p(p)**⊥к; ⊥н; ∅} ÷п {**p^p(p)**⊥к; ⊥н-; ∅} ÷п {**^p(p)**⊥к; ⊥н-**p**; ∅} ÷с {**^p(p)**⊥к; ⊥н-E; 7} ÷п {**p(p)**⊥к; ⊥н-E^; 7} ÷п {**(p)**⊥к; ⊥н-E^**p**; 7} - помилка ! (немає відношення для пари символів {**p(**})
Приклад 3. Вхідний ланцюжок **-p^p&p**.
{**-p^p&p**⊥к; ⊥н; ∅} ÷п {**p^p&p**⊥к; ⊥н-; ∅} ÷п {**^p&p**⊥к; ⊥н-**p**; ∅} ÷с {**^p&p**⊥к; ⊥н-E; 7} ÷п {**p&p**⊥к; ⊥н-E^; 7} ÷п {**&p**⊥к; ⊥н-E^**p**; 7} ÷с {**&p**⊥к; ⊥н-E^E; 7,7} ÷с {**&p**⊥к; ⊥н-E; 7,7,5} ÷п {**p**⊥к; ⊥н-E**&**; 7,7,5} ÷п {⊥к; ⊥н-E**&p**; 7,7,5} ÷с {⊥к; ⊥н-E**&**E; 7,7,5,7} ÷с {⊥к; ⊥н-E; 7,7,5,7,3} ÷с
{⊥к; ⊥нE; 7,7,5,7,3,1}
Приклад 4. Вхідний ланцюжок **-p&p^p**.
{**-p&p^p**⊥к; ⊥н; ∅} ÷п {**p&p^p**⊥к; ⊥н-; ∅} ÷п {**&p^p**⊥к; ⊥н-**p**; ∅} ÷с {**&p^p**⊥к; ⊥н-E; 7} ÷п {**p^p**⊥к; ⊥н-E**&**; 7} ÷п {**^p**⊥к; ⊥н-E**&p**; 7} ÷с {**^p**⊥к; ⊥н-E**&**E; 7,7} ÷п {**p**⊥к; ⊥н-E**&**E^; 7,7} ÷п {⊥к; ⊥н-E**&**E^**p**; 7,7} ÷с {⊥к; ⊥н-E**&**E^E; 7,7,7} ÷с {⊥к; ⊥н-E**&**E; 7,7,7,5} ÷п {⊥к; ⊥н-E; 7,7,7,5,3} ÷с
{⊥к; ⊥нE; 7,7,7,5,3,1}
Два останні приклади наочно демонструють, що пріоритет операцій, встановлений в граматиці, впливає на послідовність розбору і послідовність вживання правил.

Спільний алгоритм роботи синтаксичного аналізатора

Синтаксичний аналізатор працює, спираючись на побудовану матрицю передування. На його вхід поступає оброблений сканером текст початкової програми. Кожен ідентифікатор або константа представляються для нього деяким термінальним символом (у прикладі він позначений як **p**, в завданні - a). Тоді першим кроком спільного алгоритму аналізу має бути побудова таблиці лексем (що вже виконувалося в попередній роботі).
По таблиці лексем і матриці передування виконується розбір ланцюжка. Результатом розбору є перевірка ланцюжка на синтаксичну правильність і для правильних ланцюжків - побудова послідовності правил виводу (для неправильних ланцюжків видається повідомлення про помилку). Отримання послідовності правив - другий крок спільного алгоритму аналізу.
По послідовності правил легко будується ланцюжок виводу і дерево синтаксичного розбору. Дерево будується зверху вниз шляхом послідовного вживання правил. При побудові дерева слід враховувати, що алгоритм розбору породжує правосторонній вивід, тому на кожному кроці на основі правила в ланцюжку слід замінювати крайній правий нетермінальний символ (нижній правий в дереві). Це третій крок спільного алгоритму аналізу.
Ланцюжки виводу для двох з прикладів, що розгледіли вище, матимуть наступний вигляд:
Приклад 3. Вхідний ланцюжок **-p^p&p**.

E → **-E** → -E**&**E → -E**&p** → -E^E**&p** → -E^**p&p** **-p^p&p**

Приклад 4. Вхідний ланцюжок **-p&p^p**.

E → **-E** → -E**&**E → -E**&**E^E → **-E&**E^**p** → **-E&p^p** → **-p&p^p**

Дерева виводу для цих двох прикладів приведені на мал. 4.

Після побудови дерева залишається замінити термінальні символи (**p** або **a**) граматики на відповідні константи і ідентифікатори з таблиці лексем. Для цього досить проглядати таблицю від початку до кінця і, обходячи побудоване дерево від кореня зверху вниз зліва направо, послідовно замінити все листя, помічене шуканим символом, на лексеми з таблиці. Це останній крок спільного алгоритму роботи синтаксичного аналізатора.

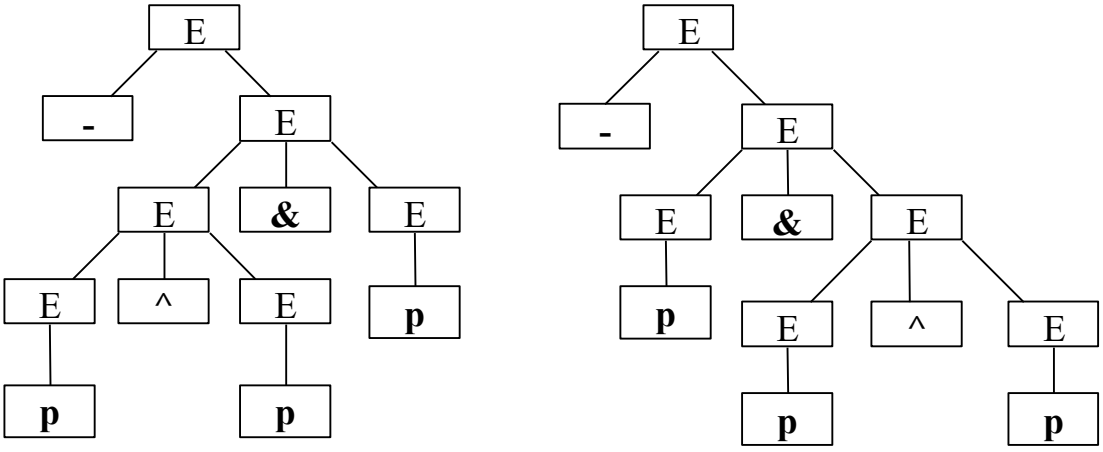


Рис. 4. Дерева виводу для ланцюжків з прикладів 3 і 4 відповідно.

Побудоване дерево і буде деревом синтаксичного розбору пропозиції граматики.

ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Отримати варіант завдання у викладача.
2. Побудувати матрицю передування для заданої граматики.
3. Виконати розбір простого прикладу уручну по правилах заданої граматики.
4. Підготувати і захистити звіт.
5. Написати і настроїти програму на ЕОМ.
6. Здати працюючу програму викладачеві.

ВИМОГИ ДО ОФОРМЛЕННЯ ЗВІТУ

Звіт повинен містити наступні розділи:

- Завдання по лабораторній роботі.
- Короткий виклад меті роботи.
- Запис заданої граматики вхідної мови у формі Бекуса-наура.
- Безліч крайніх правих і крайніх лівих символів з вказівкою кроків побудови.
- Безліч крайніх правих і крайніх лівих термінальних символів.
- Заповнену матрицю передування для граматики.
- Приклад виконання розбору простої пропозиції (по вибору).
- Текст програми (оформляється після виконання програми на ЕОМ).

ОСНОВНІ КОНТРОЛЬНІ ПИТАННЯ

1. Яку роль виконує синтаксичний аналіз в процесі компіляції?
2. Які типи граматик існують? Як зв'язані типи граматик і мов?
3. Що таке КС-граматики? Розкажіть про їх використання в компіляторі.

4. Дайте визначення приведеної граматики. Які перетворення граматик існують?
5. Поясните правила побудови дерева виведення граматики.
6. Що таке граматики простого передування?
7. Як обчислюються стосунки передування для граматик простого передування ?
8. Що таке граматика операторного передування ?
9. Як обчислюються відношення для граматик операторного передування ?
10. Розкажіть про завдання розбору. Що таке розпізнавач мови?
11. Розкажіть про спільні принципи роботи розпізнавача мови.
12. Що таке перенос, згортка. Для чого необхідний алгоритм «перенос-свертка»?
13. Розкажіть, як працює алгоритм «перенос-свертка» в спільному випадку (з поверненнями).
14. Як працює алгоритм «перенос-свертка» без повернень (поясніть на своєму прикладі)?

ВАРІАНТИ ПОЧАТКОВИХ ГРАМАТИК

Нижче приведені варіанти граматик. У всіх варіантах символ S є початковим символом граматики; S, F, T і E позначають нетермінальні символи. Термінальні символи виділені жирним шрифтом. Замість символу a повинні підставлятися лексеми.

1. S → **a :=** F**;**
F → F**+**T | T
T → T*****E | T**/**E | E
E → **(**F**)** | **-(**F**)** | **a**

2. S → **a :=** F**;**
F → F **or** T | F **xor** T | T
T → T **and** E | E
E → **(**F**)** | **not (**F**)** | **a**

3. S → F**;**
F → **if** E **then** T **else** F | **if** E **then a := a**
T → **if** E **then** T **else** T | **a := a**
E → **a<a** | **a>a** | **a=a**

4. S → F**;**
F → **for** T **do** F | **a := a**
T → **(**F**;** E**;** F**)** | **(;** E**;** F**)** | **(**F**;** E**;)** | **(;** E**;)**
E → **a<a** | **a>a** | **a=a**

ВАРІАНТИ ЗАВДАНЬ

№ варіанту	№ варіанту граматики	Допустимі лексеми вхідної мови
1	1	Ідентифікатори, десяткові числа з плаваючою крапкою
2	2	Ідентифікатори, константи **true** і false
3	3	Ідентифікатори, десяткові числа з плаваючою крапкою
4	4	Ідентифікатори, десяткові числа з плаваючою крапкою
5	1	Ідентифікатори, римські числа
6	2	Ідентифікатори, константи **0** і 1
7	3	Ідентифікатори, римські числа
8	4	Ідентифікатори, римські числа
9	1	Ідентифікатори, шестнадцатеричные числа
10	2	Ідентифікатори, шестнадцатеричные числа
11	3	Ідентифікатори, шестнадцатеричные числа
12	4	Ідентифікатори, шестнадцатеричные числа
13	1	Ідентифікатори, символьні константи (у одинарних лапках)
14	2	Ідентифікатори, символьні константи ‘T’ і ‘F’
15	3	Ідентифікатори, рядкові константи (у подвійних лапках)
16	4	Ідентифікатори, рядкові константи (у подвійних лапках)

Примітка:

- за римські числа вважати послідовності великих латинських букв **X**, V і I;
- за шістнадцяткові числа вважати послідовність цифр і символів ‘a’, ‘b’, ‘c’,’d’, ’e’ і ‘f’, що починається з цифри (наприклад: 89, 45ac9, 0abc4);

– для виконання роботи рекомендується використовувати лексичний аналізатор, ЛІТЕРАТУРА, ЩО РЕКОМЕНДУЄТЬСЯ

ЛАБОРАТОРНА РОБОТА № 8. ТРАНСЛЯЦІЯ АРИФМЕТИЧНИХ ВИРАЗІВ

Мета роботи: Ознайомитися з основними алгоритмами трансляції арифметичних виразів.

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Серед алгоритмів трансляції арифметичних виразів найбільш розповсюдженими є наступні:

Алгоритм Рутисхаузера

Даний алгоритм один з перших і найбільш простих, його особливістю є припущення про повну дужечність в структурі виразу. Під ***дужечністю*** розуміють запис виразу таким чином, що порядок і пріоритет операцій задається розстановкою дужок, а не явний пріоритет операцій при цьому не враховується.

В процесі обробки виразу на основі повної дуженної структури алгоритм присвоює кожному символу оброблюваного рядка номер рівня (пріоритету) за наступними правилами:

Правило 1: Якщо поточний символ – це дужка, що відкривається або заміна, то значення пріоритету зростає на одиницю.

Правило 2 : Якщо поточний символ – це знак операції або дужка що закривається, то пріоритет зменшується на одиницю.

Таким чином, для виразу **(*A* + (*B* * *C*))** присвоєння значення рівнів пріоритету буде здійснюватись наступним чином

№ символу	1	2	3	4	5	6	7	8	9
Символ	**(**	A	+	**(**	B	*	C	**)**	**)**
Рівень пріоритету	1	2	1	2	3	2	3	2	1

Алгоритм трансляції складається з наступних кроків:

Крок 1: Здійснити розстановку пріоритетів;

Крок 2: Виконати пошук триплету (трійки символів) з максимальним значенням рівнів пріоритету;

Крок 3: Виділити трійку (два операнди з максимально ним значенням рівнів пріоритету і операції між ними);

Крок 4: Здійснити обчислення, результат якого занести в проміжну змінну;

Крок 5: З вихідного рядка видалити виділений триплет разом з дужками, а на її місце помістити проміжну змінну, яка позначає результат зі значенням рівня пріоритету на одиницю менше ніж у виділеного триплета (трійки символів);

Крок 6: Виконувати операції 2, 3, 4 і 5 доти поки у вхідній послідовності не лишиться одна змінна, що позначає результат.

Генерований триплет	**Вираз**
+AB→R	**1.1 1.1 1.1 1.1 1.1** **((((*A* + *B*) · *C*) / *D*) - *E*)** 0 1 2 3 4 5 4 5 4 3 4 3 2 3 2 1 2 1 0
****RC→S***	

	(((*) /) -) R C D E 0 1 2 3 4 3 4 3 2 3 2 1 2 1 0
/SD→Q	((/) -) S D E 0 1 2 3 2 3 2 1 2 1 0
-QE→T	(-) Q E 0 1 2 1 2 1 0

Алгоритм Бауера і Замельзона

Ґрунтується на використанні двох стьоків і таблиці функцій переходу. Один строк використовується при трансляції виразу, а другий під час його інтерпретації. Умовно стьоки можна позначити як Т-стьок транслятора і Е-стьок інтерпретатора. В таблиці переходів задаються функції, які повинен виконувати транслятор при роботі виразу. Фактично, можливо в шість реалізацій функцій при читанні операцій з вихідного рядка:

f*1** – переслати операнд з із вхідного рядка в строк ***T і читати наступний символ рядка.

f*2** – виділити триплет (трійка символів) взяти операцію з вершини стьоку ***T і два операнди з вершини стьоку ***E***. Допоміжну змінну, що позначає результат занести в смок ***E***. Переслати операцію із вхідного рядка в строк ***T*** і читати наступний символ вхідного рядка.

***f*3** – виключити символ із стьоку Т і читати наступний символ вхідного рядка.

f*4** – виділити триплет, тобто взяти операцію з вершини стьоку ***T і два операнди з вершини стоку ***E***. Допоміжну змінну, що позначає результат занести в смок ***E***, по таблиці визначити функцію для даного символу вхідного рядка.

***f*5** – сформувати повідомлення про помилку.

***f*6** – завершення роботи.

Для алгебраїчних виразів таблиця переходів може бути представлена наступним чином (таблиця 5.1).

Алгоритм здійснення перегляду виразів із вхідного рядка зліва на право і циклічно виконує наступні дії:

1) Якщо груповий символ із вхідного рядка є операндом, то він безумовно переноситься в строк ***E***.

Таблиця 5.1 – функції переходу.

		ОПЕРАЦІЯ ІЗ ВХІДНОГО РЯДКА						
		$	**(**	**+**	**-**	*****	**/**	**)**
Операнд на вершині стьоку *T*	**$**	6	1	1	1	1	1	5
	(	5	1	1	1	1	1	3
	+	4	1	2	2	1	1	4
	-	4	1	2	2	1	1	4
	*	4	1	4	4	2	2	4
	/	4	1	4	4	2	2	4

$ - позначення стьоку або пустого рядка.

2) Якщо черговий символ операція, то за таблицею функцій переходу визначається номер функції для виконання.

Наприклад, послідовність дій алгоритму для виразу ***A*** **+** **(*B* - *C*)** · ***D*** може бути наступна:

Стьок E	**Стьок T**	**Вхідні символи**	**Номер функції**	**Триплет**
$	$	*A*		
$***A***	$ +	+	1	
$***A***	$ + (	(	1	
$***A***	$ + (	B		
$***AB***	$ + (-	-	1	
$***AB***	$ + (-	C		
$***ABC***	$ + (	)	4	***-BC→***
$***AR***	$ +	)	3	
$***AR***	$ +	*	1	
$***AR***	$ + *	D		
$***ARD***	$ + *	$	4	****RD→Q***
$***AQ***	$ +	$	4	***+AQ→S***
$***S***	$	$	END	

Алгоритм трансляції на основі таблиці передування (“предшествования”)

Фактично таблиці передування задають транслятору пріоритет операції. Таблиця встановлює між операціями передування, а відношення виду **.=.** встановлюється між операціями з однаковим пріоритетом.

Відношення виду **<.** встановлюється після даної операції у виразі у випадку якщо за нею слідує операція з вищим пріоритетом (якщо після + слідує *). Якщо після поточної операції слідує операція з меншим пріоритетом, то між ними встановлюється відношення виду **.>**

Алгоритм на основі використання таблиці передування використовується два стьоки, стьок опера рацій і стьок операндів. Операнди із вхідного рядка безумовно переносяться в строк операндів. Операції в строк операцій переносяться в залежності від того, в якому відношенні передування (“предшествования”) вони знаходяться із символами на вершині стьоку операцій.

Для алгебраїчних виразів така таблиця матиме наступний вигляд:

Таблиця 5.2 – функції передування (“предшествования”).

		ОПЕРАЦІЯ ІЗ ВХІДНОГО РЯДКА						
		$	+	-	*	/	**(**	**)**
Символ операції на вершині стьоку	**$**	End	<.	<.	<.	<.	<.	Erorr
	+	.>	.=.	.=.	<.	<.	<.	.>
	-	.>	.=.	.=.	<.	<.	<.	.>
	*	.>	.>	.>	.=.	.=.	<.	.>
	/	.>	.>	.>	.=.	.=.	<.	.>
	)	Error	<.	<.	<.	<.	<.	Delete

$ - ознака стьоку або пустого рядка.

В процесі аналізу вираз переглядається зліва на право, а операції виконуються поки між символом на вершині стьоку і вхідним символом не виконується одне з відношень **.=.** або **.>** При появі такого відношення виділяється триплет: два символа із стьоку операндів і один із стьоку операцій, а на вершину стьоку операндів заноситься допоміжна змінна в яку поміщається результат. Після чого описані вище дії повторюються до моменту досягнення кінця стьоків операндів та опрацій.

Метод зворотнього польського запису (Лукашевича)

Даний метод ґрунтується на побудові дерева арифметичного виразу і формуванні вихідної послідовності символів для обчислення (на основі обходу дерева) в якій пріоритет операцій розставлений в порядку їх виконання з ліва на право.

Для арифметичного виразу ***(A + B) * (C + D) – E*** проводиться операція, яка дозволяє представити його у вигляді дерева в якому вузлам відповідають операції, а гілкам – операнди. Побудову дерева, як правило починають з кореня в якості вузлової операції вибирають операцію, що виконувалась останньою, відповідно лівій гілці – лівий операнд, а правій – правий, рис.5.1.

Наступним етапом є обхід дерева в результаті чого формується рядок символів вузлів і його дерева. Обхід можна здійснити зліва на право і відповідно до нашого дерева знизу до гори. При обході записується наявні гілки вузла після яких записується значення самого вузла. В результаті отримаємо:

A B + C D + * E

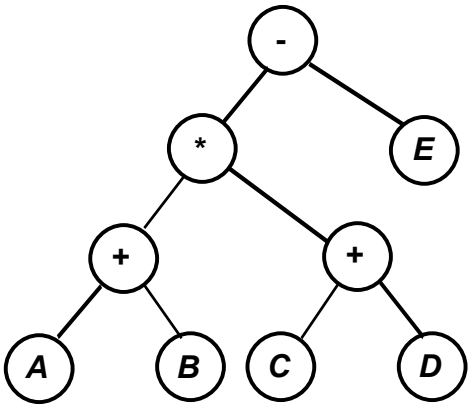


Рис5.1 – Дерево виразу **(A + B) * (C + D) – E**.

Характерною особливістю отриманого рядка є відсутність дужок і розставлені знаків операцій у відповідності до порядку їх виконання зліва на право (зворотній польський запис). Крім того обчислення виразу на основі такого запису можна здійснити шляхом однократного перегляду рядка. Для отриманої відповідної послідовності:

№ п/п	Рядок для аналізу	Дія
0	*A B + C D + * E –*	***R1 = A + B***
1	*R1 C D + * E –*	*R2 = C + D*
2	*R1 R2 * E –*	*R1 = R1 * R2*
3	*R1 E –*	**R*1* = R*1* – E**
4	*R1*	

Отримання виразу у форматі зворотного польського запису здійснюється на основі наступного алгоритму:

Операція	Пріоритет

(	**0**
)	1
+ або **-**	2
* або /	3
** ^	4

В процесі перегляду вхідного рядка символів зліва на право операнди безумовно заносяться у вихідну послідовність, а знаки операцій заносяться в стьок на основі наступних правил:

Правило 1: Якщо стьок пустий, операція із вхідного рядка заноситься в стьок.

Правило 2 : Операція виштовхує із стьоку всі операції із більшим або рівним пріоритетом у вихідний рядок.

Правило 3: Якщо черговий символ із вхідного рядка – дужка, що відкривається, то він записується (проштовхується) в стьок.

Правило 4 : Якщо черговий символ – дужка, що закривається, то вона виштовхує всі операції із стьоку, до найближчої дужки, що відкривається (дужки взаємознищуються і у вихідний рядок не записуються).

№ символу	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1.1.1.1.1.6.1.1.1 хідний рядок	(	A	+	B	)	*	(	C	+	D	)	-	E	
2 Стан стьоку	($	($	+ ($	+ ($	$	* $	(*	(* $	+ ($	+ (* $	* * $	- $	- $	$
3 Вихідний рядок		A		B	+			C		D	+	*	E	-

ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Написати програму яка обчислює арифметичний вираз, заданий в файлі у вигляді текстового рядка методом згідно заданого варіанту (таблиця 5.3). Вхідний файл (розширення "***.clc**") задається через командний рядок, а результати обрахунку зберегти у файлі із тою ж назвою але розширенням "***.res**". Мова програмування С++ (чи інша за згодою викладача).

Таблиця 3

Варіант	**Завдання**
1	Рутисхаузера для операцій (+, -).
2	Рутисхаузера для операцій (*, /).
3	Рутисхаузера для операцій (+, /).
4	Рутисхаузера для операцій (-, *).
5	Бауера і Замельзона для операцій (+, - , * , /).
6	Бауера і Замельзона для операцій (+, - , ^ степінь).

7	Бауера і Замельзона для операцій (+, - , дужки).
8	Бауера і Замельзона для операцій (*, / , дужки).
9	Таблиць передування для операцій (+, - , * , /).
10	Таблиць передування для операцій (+, - , ^ степінь).
11	Таблиць передування для операцій (+, - , дужки).
12	Таблиць передування для операцій (*, / , дужки).
13	Зворотного польського запису для операцій (+, - , *, /)
14	Зворотного польського запису для операцій (+, - , ^ степінь).
15	Зворотного польського запису для операцій (+, - , дужки).
16	Зворотного польського запису для операцій (*, / , дужки).

2. Здійснити компіляцію програми.

3. Оформити звіт в згідно вимог.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Пояснити суть трансляції на основі алгоритму Рутисхаузера?
2. Пояснити суть трансляції на основі алгоритму Бауера і Замельзона?
3. Пояснити суть трансляції на основі алгоритму Зворотного польського запису?

ЛАБОРАТОРНА РОБОТА № 9-10. ГЕНЕРАЦІЯ І ОПТИМІЗАЦІЯ ОБ'ЄКТНГО КОДА

Мета роботи: вивчення основних принципів генерації компілятором об'єктного кода для лінійної ділянки програми, ознайомлення з методами оптимізації результуючого об'єктного кода з допомогою згортки і виключення зайвих операцій.

Для виконання лабораторної роботи потрібно написати програму, яка на підставі дерева синтаксичного розбору створює об'єктний код і виконує потім його оптимізацію. За рахунок початкового дерева синтаксичного розбору рекомендується узяти дерево, яке створює програма, побудована за завданням попередньої лабораторної роботи.

Програму рекомендується побудувати з трьох основних частин: перша частина – створення деревом синтаксичного розбору (за наслідками лабораторної роботи №3), друга частина - реалізація алгоритму створення об'єктного кода по дереву розбору, і третя частина - оптимізація створеного об'єктного кода. Результатом роботи повинна бути побудована на підставі заданої пропозиції граматики програма на об'єктній мові. Як об'єктна мова пропонується узяти мову асемблера для процесорів типу Intel 80x86 в реальному режимі (можливий вибір іншої об'єктної мови за узгодженням з викладачем). Ідентифікатори, що все зустрічаються в початковій програмі, вважати простими скалярними змінними, що не вимагають виконання перетворення типів. Обмеження на довжину ідентифікаторів і констант відповідають вимогам попередньої лабораторної роботи.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Генерація об'єктного кода - це переклад компілятором внутрішнього представлення початкової програми в результуючу об'єктну програму на мові асемблера або безпосередньо на машинній мові (машинних кодах).

Генерація об'єктного кода виконується після того, як виконаний синтаксичний аналіз програми і всі необхідні дії з підготовки до генерації коду: розподілений адресний простір під функції і змінні, перевірена відповідність імен і типів змінних, констант і функцій в синтаксичних конструкціях початкової програми і так далі.

Оптимізація програми - це обробка, пов'язана з переупорядковуванням і зміною операцій в компільованій програмі з метою отримання ефективнішої результуючої об'єктної програми. Оптимізація виконується на етапах підготовки до генерації і безпосередньо при генерації об'єктного кода.

Кращі оптимізуючі компілятори можуть отримувати об'єктні програми з складних початкових програм, написаних на мовах високого рівня, які не поступаються за якістю програмам на мові асемблера. Тимчасові і трудові витрати на створення такої програми істотно менші, ніж при її реалізації на асемблері. У сучасних компіляторів існують можливості вибору тих або інших критеріїв оптимізації, виходячи з яких оцінюється ефективність об'єктної програми. Так, з одного боку, можлива оптимізація з мінімізацією розміру програми, з іншого боку - оптимізація із збільшенням швидкості її виконання. При цьому не потрібно змінювати текст програми на початковій мові.

Всі ці переваги говорять на користь застосування оптимізації. Єдиним, але істотним недоліком оптимізації є необхідність ретельного її опрацьовування при створенні компілятора. Використовувані методи оптимізації ні за яких умов не повинні приводити до зміни “сенсу” початкової програми (тобто до таких ситуацій, коли результат виконання програми змінюється після її оптимізації). На жаль, не всі методи оптимізації, використовувані творцями компіляторів, можуть бути теоретично обгрунтовані і доведені для всіх можливих видів початкових програм. Тому більшість компіляторів передбачає можливість відключати ті або інші з можливих методів оптимізації. (Часто при оптимізації компілятори видають попередження розробникові програми, якщо та або інша

її ділянка викликає підозри відносно правильності його “сенсу”). Застосування оптимізації також недоцільно в процесі відладки початкової програми.

Розрізняються дві основні категорії оптимізуючих перетворень:

- перетворення початкової програми (у формі її внутрішнього уявлення в компіляторі), не залежні від результуючої об'єктної мови;
- перетворення результуючої об'єктної програми.

Останній тип перетворень може залежати не тільки від властивостей об'єктної мови (що очевидно), але і від архітектури обчислювальної системи, на якій виконуватиметься результуюча програма. (Так, наприклад, при оптимізації може враховуватися об'єм кеш-пам'яті і методи організації конвеєрних операцій центрального процесора). Цей тип перетворень тут розглядатися не буде. Саме ці перетворення можуть вплинути на “сенс” початкової програми. В більшості випадків вони є “ноу-хау” виробників компіляторів і строго орієнтовані на певну архітектуру обчислювальних машин.

Методи перетворення програми залежать від типів синтаксичних конструкцій початкової мови. Теоретично розроблені методи оптимізації для багатьох типових конструкцій мов програмування. Далі будуть розглянуті тільки методи оптимізації лінійних ділянок - вони зустрічаються в будь-якій програмі і складають істотну частину програмного коду.

Лінійна ділянка програми - це виконувана по порядку послідовність операцій що має один вхід і один вихід. Найчастіше лінійну ділянку містить послідовність арифметичних операцій і операторів привласнення значень змінним.

Перш ніж перейти до питань оптимізації лінійних ділянок розглянемо їх внутрішньо уявлення в компіляторі.

Алгоритм генерації об'єктного кода по дереву виводу

Можливі різні форми внутрішнього представлення синтаксичних конструкцій початкової програми в компіляторі. На етапі синтаксичного розбору часто використовується форма, що іменується деревом виводу (методи його побудови розглядалися в попередніх лабораторних роботах). Але форми уявлення, використовувані на етапах синтаксичного аналізу, виявляються незручними в роботі при генерації і оптимізації об'єктного кода. Тому перед оптимізацією і безпосередньо генерацією об'єктної коди внутрішнє представлення програми перетвориться в одну з відповідних форм запису.

Прикладами таких форм запису є:

- зворотний польський запис операцій;
- тетради операцій;
- тріади операцій;
- власне команди асемблера.

Зворотній польський запис - це постфіксний запис операцій. Перевагою її є те, що всі операції записуються безпосередньо в порядку їх виконання. Вона надзвичайно ефективна в тих випадках, коли для обчислень використовується стек.

Тетради є записом операцій у формі з чотирьох складових: *<операція>(<операнд1>,<операнд2>,<результат>)*. Тетради використовуються рідко, оскільки вимагають більше пам'яті для свого уявлення, чим тріади, не відображають взаємозв'язку операцій і, крім того, погано відображаються в команди асемблера і машинні коди, оскільки в наборах команд більшості сучасних машин не зустрічаються операції з трьома операндами.

Тріади є записом операцій у формі з трьох складових: *<операція>(<операнд1>,<операнд2>)*, при цьому один або обидва операнди можуть бути посиланнями на іншу тріаду в тому випадку, якщо як операнд даної тріади виступає результат виконання іншої тріади. Тому тріади при записі послідовно номерують для

зручності вказівки посилань одних тріад на інших. Наприклад, вираз *A := B*c + D - B*10*, записане у вигляді тріад матиме вигляд:

1) * (B, C)
2) + (^1, D)
3) * (B, 10)
4) - (^2, ^3)
5) := (A, ^4)

Тут операції позначені відповідним знаком (при цьому привласнення також є операцією), а знак ^ означає посилання операнда однієї тріади на результат іншої.

Команди асемблера зручні тим, що при їх використанні внутрішнє представлення програми повністю відповідає об'єктному коду і складні перетворення не потрібні. Проте використання команд асемблера вимагає додаткових структур для відображення їх взаємозв'язку. Крім того, внутрішнє представлення програми виходить залежним від результуючого кода, а це означає, що при орієнтації компілятора на інший результуючий код потрібно буде перебудовувати як само внутрішнє представлення програми, так і методи його обробки в алгоритмах оптимізації (при використанні тріад або тетрад цього не потрібно).

Для побудови внутрішнього представлення об'єктного коду (надалі - просто коди) по дереву виводу може використовуватися проста рекурсивна процедура. Ця процедура перш за все повинна визначити тип вузла дерева - він відповідає типу операції, символ якої знаходиться в листі дерева для поточного вузла. Цей лист є середнім листом вузла дерева для бінарних операцій і крайнім лівим листом - для унарних операцій. Після визначення типу процедура будує код для вузла дерева відповідно до типу операції. Якщо всі вузли наступного рівня для поточного вузла є листя дерева, то в код включаються операнди, відповідні цьому листю, і код, що вийшов, стає результатом виконання процедури. Інакше процедура повинна рекурсивно викликати сама себе для генерації коди вузлів дерева, що пролягають нижче, і результат виконання включити в свій створений код.

Тому для побудови внутрішнього представлення об'єктного коду по дереву виводу в першу чергу необхідно розробити форми представлення об'єктного коду для чотирьох випадків, відповідних видам поточного вузла дерева виводу:

- обидва вузли дерева, що пролягають нижче, - листя (термінальні символи граматики);
- тільки лівий вузол, що пролягає нижче, є листом дерева;
- тільки правий вузол, що пролягає нижче, є листом дерева:
- обидва вузли, що пролягають нижче, не є листям дерева.
- Розглянемо побудову двох видів внутрішнього уявлення по дереву виводу:
- побудова асемблерної коди по дереву виводу;
- побудова списку тріад по дереву виводу.

Побудова асемблерного коду по дереву виводу

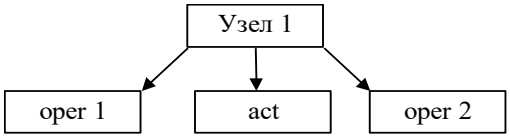
Як мова асемблера візьмемо мову асемблера процесорів типу Intel 80x86. При цьому вважатимемо, що операнди можуть бути поміщені в 16-розрядні регістри процесора і в коді результуючої об'єктної програми можуть використовуватися регістри AX (акумулятор) і DX (регістр даних), а також стік для зберігання проміжних результатів.

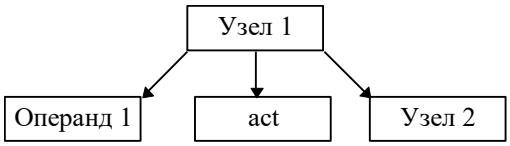
Тоді чотирьом формам поточного вузла дерева відповідатимуть наступні фрагменти коди на мові асемблера (табл. 5):

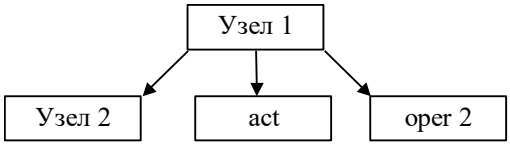
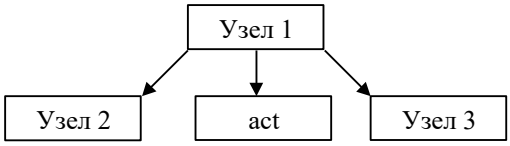
Таблиця 5.

Перетворення типових вузлів дерева виводу в код на мові асемблера

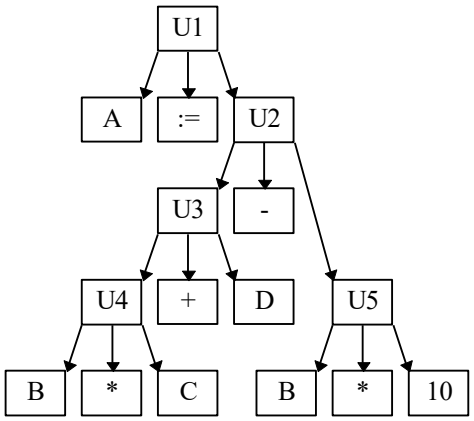
Вид вузла дерева	Результуючий код	Примітка

Вид вузла дерева	Результуючий код	Примітка
	mov ax,oper1 act ax,oper2	act - команда відповідної операції oper1,oper2 - операнди (листя дерева)

	Code(Узел 2)
	mov dx,ax
	mov ax,oper1
	act ax,dx
		Узел 2 - вузол (не лист!) дерева, що пролягає нижче
		Code(Узел 2) - код, що створюються процедурою для вузла, що пролягає нижче

	Code(Узел 2) act ax,oper2	Code(Узел 2) - код, що створюються процедурою для вузла, що пролягає нижче
	Code(Узел 2) push ax Code(Узел 3) mov dx,ax pop ax act ax,dx	Code(Узел 2) - код, що створюються процедурою для вузла, що пролягає нижче Code(Узел 3) - код, що створюються процедурою для вузла, що пролягає нижче push і pop - команди збереження результатів в стеку і витягання результатів із

Розглянемо приклад дерева виводу для виразу *A := B*c + D - B*10* на мал. 5 і відповідний йому фрагмент коди на мові асемблера, побудований по описаних вище правилах (звернете увагу, що для операції привласнення використовується окремий код, що не підпадає під загальні правила):



Мал. 5. Дерево виводу для арифметичного виразу.

Крок1:Code(U2)
mov A,ax ; операція привласнення
Крок 2: Code(U3)
push ax
Code(U5)
mov dx,ax
pop ax
sub ax,dx
mov A,ax ; операція привласнення
Крок 3: Code(U4)
add ax,D
push ax
Code(U5)
mov dx,ax
pop ax
sub ax,dx
mov A,ax ; операція привласнення
Крок 4: mov ax,B
mul ax,C
add ax,D
push ax
Code(U5)
mov dx,ax
pop ax
sub ax,dx
mov A,ax ; операція привласнення
Крок 5: mov ax,B
mul ax,C
add ax,D
push ax
mov ax,B
mul ax,10
mov dx,ax
pop ax
sub ax,dx
mov A,ax ; операція привласнення

Отриманий об'єктний код на мові асемблера, очевидно, може бути оптимізований, проте для його обробки потрібні спеціальні (орієнтовані саме на дану мову асемблера) методи і структури, що враховують взаємозв'язок операцій. Крім того, орієнтація на певну мову асемблера зводить нанівець універсальність методу. Так в приведеному прикладі використовується команда mul, яка в ранніх версіях процесорів фірми Intel має обмеження на типи операндів, а отже, в універсальному компіляторі не може бути використана так, як в даному прикладі. Це зажадає, щоб генерація коди для вузлів дерева йшла в залежності не тільки від операндів, але і від типу операції (навіть у приведеному прикладі таку залежність довелося встановити для операції привласнення).

Зазвичай такі проблеми вирішуються таким чином, що замість команд безпосередньо мови асемблера використовуються команди деякого близького до нього проміжного псевдокоду. Більшість цих команд один в один відображаються потім в команди мови асемблера, інші ж однозначно перетворяться у фіксовану послідовність команд.

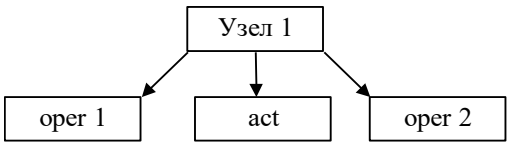
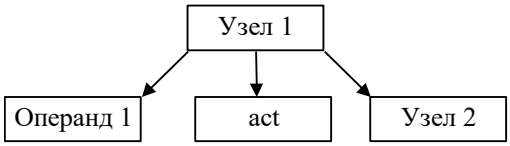
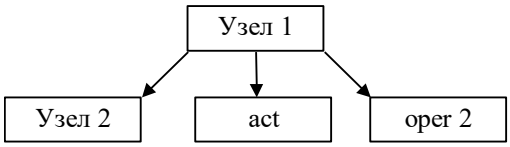
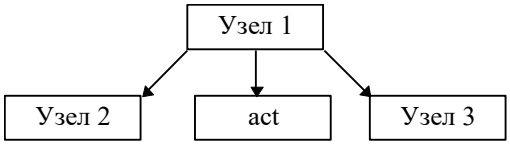
Побудова списку тріад по дереву виводу.

Тріади є універсальною, машинно-незалежною формою внутрішнього уявлення в компіляторі результуючої об'єктної програми, а тому не вимагають обмовки додаткових умов при генерації коду. Тріади взаємозв'язані між собою, тому для установки коректного взаємозв'язку процедура генерації коди повинна отримувати також поточний номер і чергової тріади.

Тоді чотирьом формам поточного вузла дерева відповідатимуть послідовності тріад об'єктної коди (табл. 6):

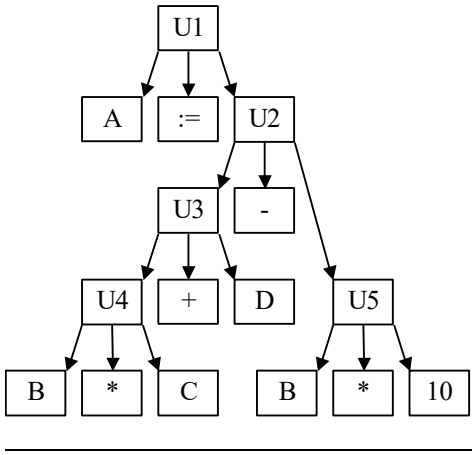
Таблиця 6.

Перетворення типових вузлів дерева виводу в послідовність тріад

Вид вузла дерева	Результуючий код	Примітка
	i) act (oper1,oper2)	act - тип тріади oper1,oper2 - операнди (листя дерева виводу)
	i) Code(Узел 2,i) i+j) act(oper1,^i+j-1)	Узел 2 - вузол дерева виводу, що пролягає нижче Code(Узел 2,i) - послідовність тріад, що породжується для Узла2, починаючи з тріади з номером i j - кількість тріад, що породжуються для Узла2
	i) Code(Узел 2,i) i+j) act(^i+j-1,oper2)	Узел 2 - вузол дерева виводу, що пролягає нижче Code(Узел 2,i) - послідовність тріад, що породжується для Узла2, починаючи з тріади з номером i j - кількість тріад, що породжуються для Узла2
	i) Code(Узел 2,i) i+j) Code(Узел 3,i+j) i+j+k) act(^i+j-1,^i+j+k-1)	Узел 2, Узел 3 - вузли дерева виводу, що пролягають нижче Code(Узел 2,i) - послідовність тріад, що породжується для Узла2, починаючи з тріади з номером i j - кількість тріад, що породжуються для Узла2 Code(Узел 3,i+j) - послідовність тріад, що породжується для Узла3, починаючи з тріади з номером i+j

Вид вузла дерева	Результуючий код	Примітка
		k - кількість тріад, що породжуються для Узла3

Розглянемо той же приклад дерева виводу для виразу *A := B*c + D - B*10* на мал. 6 і відповідну йому послідовність тріад:



Мал. 6. Дерево виводу для арифметичного виразу.

соответствующую ему последовательность триад:

Крок 1: 1) Code(U2,1)
i) :=(A,^i-1)

Крок 2: 1) Code(U3,1)
j) Code(U5,j)
i-1) -(^j-1,^i-2)
i) :=(A,^i-1)

Крок 3: 1) Code(U4,1)
k) +(^k-1,D)
j) Code(U5,j)
i-1) -(^j-1,^i-2)
i) :=(A,^i-1)

Крок 4: 1) *(B,C)
2) +(^1,D)
3) Code(U5,3)
i-1) -(^j-1,^i-2)
i) :=(A,^i-1)

Крок 5: 1) *(B,C)
2) +(^1,D)
3) *(B,10)
4) -(^2,^3)
5) :=(A,^4)

Слід звернути увагу, що в даному алгоритмі послідовні номери тріад (а отже, і посилання на них) встановлюються не відразу. Це не має значення при рекурсивній організації процедури, але при іншому способі обходу дерева виводу в програмі генерації коди краще пов'язувати тріади між собою саме по посиланню (покажчику), а не по порядковому номеру.

Для тріад розроблені універсальні (машинно-незалежні) алгоритми оптимізації коди. Після їх виконання (оптимізації внутрішнього уявлення) тріади можуть бути перетворені в команди на мові асемблера.

Оптимізація об'єктної коди методом згортки

Згортка об'єктного коду - це виконання під час компіляції тих операцій початкової програми для яких значення операндів вже відомі. Тому немає необхідності багато разів виконувати їх в самій результуючій програмі - цілком достатньо один раз виконати їх при її компіляції.

Простий варіант згортки - виконання в компіляторі операцій, операндами яких є константи. Декілька більш складений процес визначення тих операцій, значення яких

можуть бути відомі в результаті виконання інших операцій. Для цього служить спеціальний алгоритм згортки.

Алгоритм згортки працює із спеціальною таблицею Т**,** яка містить пари <*переменная*>-<*константа*> для всіх змінних, значення яких вже відомі. Крім того, алгоритм згортки позначає ті операції у внутрішньому уявленні програми, для яких в результаті згортки вже не потрібна генерація коду. Оскільки при виконанні алгоритму згортки враховується взаємозв'язок операцій, то зручною формою уявлення для нього є тріади, оскільки в інших формах представлення операцій (таких як тетради або команди асемблера) потрібні додаткові структури, щоб відобразити зв'язок результатів одних операцій з операндами інших.

Алгоритм згортки тріад послідовно проглядає тріади лінійного списку і для кожної тріади робить наступне:

1. Якщо операнд є змінна, яка міститься в таблиці **Т**, то операнд замінюється на відповідне значення константи.

2. Якщо операнд є посилання на особливу тріаду типу *C(K,0)*, то операнд замінюється на значення константи *K*.

3. Якщо всі операнди тріади є константами, то тріада може бути згорнута. Тоді дана тріада виконується і замість неї поміщається особлива тріада виду *C(K,0)*, де *K* - константа, результат виконання згорнутої тріади. (При генерації коди для особливої тріади об'єктний код не породжується, а тому вона надалі може бути просто виключена).

4. Якщо тріада є привласненням типу A:=b, тоді:

- якщо B - константа, то A із значенням константи заноситься в таблицю T (якщо там вже було старе значення для A, то це старе значення виключається).
- якщо B - не константа, то A взагалі виключається з таблиці **Т**, якщо воно там є.

Розглянемо приклад виконання алгоритму.

Хай фрагмент початкової програми (записаною на мові типу Паскаль) має вигляд:

```
I := 1 + 1;
I := 3;
J := 6*I + I;
```

Її внутрішньо уявлення у формі тріад матиме вигляд:

```
1   + (1,1)
2   := (I ^1)
3   := (I, 3)
4   * (6, I)
5   + (^4, I)
6   := (J ^5)
```

Процес виконання алгоритму згортки можна відобразити в табл. 7:

Таблиця 7.

Приклад роботи алгоритму згортки

Тріада	**Крок 1**	**Крок 2**	**Крок 3**	**Крок 4**	**Крок 5**	**Крок 6**
1	C (2, 0)	C (2, 0)	C (2, 0)	C (2, 0)	C (2, 0)	C (2, 0)
2	:= (I, ^1)	:= (I, 2)	:= (I, 2)	:= (I, 2)	:= (I, 2)	:= (I, 2)
3	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)
4	* (6, I)	* (6, I)	* (6, I)	C (18, 0)	C (18, 0)	C (18, 0)
5	+ (^4, I)	+ (^4, I)	+ (^4, I)	+ (^4, I)	C (21, 0)	C (21, 0)
6	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, 21)
T	(,)	(I, 2)	(I, 3)	(I, 3)	(I, 3)	(I, 3) (J, 21)

Якщо виключити особливі тріади типу *C(K,0)* (які не породжують об'єктної коди), то в результаті виконання згортки отримаємо наступну послідовність тріад:

1) := (I, 2)
2) := (I, 3)
3) := (J, 21)

Оптимізація об'єктного коду методом виключення зайвих операцій

Определим понятие лишней операции. Операция линейного участка с порядковым номером i считается лишней, если существует более ранняя идентичная ей операция с порядковым номером j и никакая переменная, от которой зависит эта операция, не изменялась никакой третьей операций, имеющей порядковый номер между i и j.

Алгоритм исключения лишних операций просматривает операции в порядке их следования. Также как и алгоритму свертки, алгоритму исключения лишних операций проще всего работать с триадами, потому что они полностью отражают взаимосвязь операций.

Щоб стежити за внутрішньою залежністю змінних і тріад алгоритм привласнює їм деякі значення, звані числами залежності, по наступних правилах:

- спочатку для кожної змінної її число залежності рівне 0, оскільки на початку роботи програми значення змінної не залежить ні від якої тріади;
- після обробки i-ой тріади, в якій змінній A привласнюється деяке значення, число залежності A (dep(A)) отримує значення i, оскільки значення A тепер залежить від даної i-ой тріади;
- при обробці i-ой тріади її число залежності (dep(i)) приймається рівним значенню: 1+(максимальне з чисел залежності операндів).

Таким чином, при використанні чисел залежності тріад і змінних можна стверджувати, що якщо i-ая тріада ідентична j-ой тріаді (j<i), то i-ая тріада вважається зайвою в тому і лише у тому випадку, коли dep(i)=dep(j).

Алгоритм виключення зайвих операцій використовує в своїй роботі також особливого вигляду тріади *SAME(j,0)*, які означають, що деяка тріада i ідентична тріаді j.

Алгоритм виключення зайвих операцій послідовно проглядає тріади лінійної ділянки. Він складається з наступних кроків, що виконуються для кожної тріади:

Якщо операнд посилається на особливу тріаду виду *SAME(j,0)*, то він замінюється на посилання на тріаду з номером j (^j).

2. Обчислюється число залежності поточної тріади з номером i, виходячи з чисел залежності її операндів.

3. Якщо існує ідентична j-ая тріада, причому j<i і dep(i)=dep(j), то поточна тріада і замінюється на тріаду особливого виду *SAME(j,0)*.

4. Якщо поточна тріада є привласнення, то обчислюється число залежності відповідної змінної.

Розглянемо роботу алгоритму на прикладі:

```
D := D + C*B;
A := D + C*B;
C := D + C*B;
```

Цьому фрагменту програми відповідатиме наступна послідовність тріад:

1) * (C, B)
2) + (D, ^1)
3) := (D, ^2)
4) * (C, B)
5) + (D, ^4)
6) := (A, ^5)
7) * (C, B)
8) + (D, ^7)

9) := (C, ^8)

Роботу алгоритму відобразимо в табл. 8.

Тепер, якщо виключити тріади особливого виду SAME(j,0), то в результаті виконання алгоритму отримаємо наступну послідовність тріад:

1) * (C, B)
2) + (D, ^1)
3) := (D, ^2)
4) + (D, ^1)
5) := (A, ^4)
6) := (C, ^4)

Зверніть увагу, що в підсумковій послідовності змінилася нумерація тріад і номера в посиланнях одних тріад на інших. Якщо в програмі компілятора як посилання використовувати не номери тріад, а безпосередньо покажчики на них, то зміну посилань в такому варіанті не буде потрібно.

Таблиця 8. Приклад роботи алгоритму виключення зайвих операцій.

Оброблювана тріада i	Числа залежності змінних A	B	C	D	Числа залежності тріад dep(i)	Тріади, отримані після виконання алгоритму
1) * (C, B)	0	0	0	0	1	1) * (C, B)
2) + (D, ^1)	0	0	0	0	2	2) + (D, ^1)
3) := (D, ^2)	0	0	0	3	3	3) := (D, ^2)
4) * (C, B)	0	0	0	3	1	4) SAME (1, 0)
5) + (D, ^4)	0	0	0	3	4	5) + (D, ^1)
6) := (A, ^5)	6	0	0	3	5	6) := (A, ^5)
7) * (C, B)	6	0	0	3	1	7) SAME (1, 0)
8) + (D, ^7)	6	0	0	3	4	8) SAME (5, 0)
9) := (C, ^8)	6	0	9	3	5	9) := (C, ^5)

Загальний алгоритм генерації і оптимізації об'єктного коду

Тепер розглянемо загальний варіант алгоритму генерації коди, яка отримує на вході дерево виводу (побудоване в результаті синтаксичного розбору) і створює по ньому фрагмент об'єктної коди лінійної ділянки результуючої програми.

Алгоритм повинен виконати наступну послідовність дій:

- побудувати послідовність тріад на основі дерева виводу;
- виконати оптимізацію коди методом згортки;
- виконати оптимізацію коди методом виключення зайвих операцій;

перетворити послідовність тріад в послідовність команд на мові асемблера (отримана послідовність команд і буде результатом виконання алгоритму).

Алгоритм перетворення тріад в команди мови асемблера - це єдина машинно-залежна частина загального алгоритму. При перетворенні компілятора для роботи з іншим результуючим об'єктним кодом потрібно буде змінити тільки цю частину, при цьому всі алгоритми оптимізації і внутрішнє представлення програми залишаться незмінними.

У даній роботі алгоритм перетворення тріад в команди мови асемблера пропонується розробити самостійно. У тривіальному вигляді такий алгоритм замінює кожну тріаду на послідовність відповідних команд, а результат її виконання запам'ятовується в тимчасовій змінній з деяким ім'ям (наприклад, TMPi, де i - номер тріади). Тоді замість посилання на цю тріаду в іншій тріаді буде підставлено значення цієї змінної. Проте алгоритм може передбачати і оптимізацію тимчасових змінних.

ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Отримати варіант завдання у викладача.
2. Вивчити алгоритм генерації об'єктного коду по дереву синтаксичного розбору.
3. Розробити фрагменти об'єктного коду, що реалізовують на мові асемблера прості операції в заданій граматиці.
4. Виконати генерацію об'єктного коду уручну для вибраного простого прикладу. Перевірити коректність результату.
5. Вивчити алгоритми оптимізації результуючого коду методом згортки і методом виключення зайвих операцій.
6. Підготувати і захистити звіт.
7. Написати і відладити програму на ЕОМ.
8. Здати працюючу програму викладачеві.

ВИМОГИ ДО ОФОРМЛЕННЯ ЗВІТУ

- Завдання по лабораторній роботі.
- Короткий виклад меті роботи.
- Запис заданої граматики вхідної мови у формі Бекуса-наура.
- Фрагменти об'єктної коди на мові асемблера для операцій заданої граматики.
- Простий приклад генерації коду по дереву синтаксичного розбору.

Текст програми (оформляється після виконання програми на ЕОМ)

ОСНОВНІ КОНТРОЛЬНІ ПИТАННЯ

1 Що таке транслятор, компілятор і інтерпретатор? Розкажіть про загальну структуру компілятора.
2 Як будується дерево виводу (синтаксичного розбору)? Які початкові дані необхідні для його побудови ?
3 Поясните роботу алгоритму генерації об'єктного коду по дереву синтаксичного розбору.
4 Розкажіть, що таке синтаксично керований переклад.
5 За рахунок чого забезпечується можливість генерації коду на різних об'єктних мовах поодинці і тому ж дереву?
6 Яку роль виконує генерація об'єктного коду в процесі компіляції?
7 Які дані необхідні компілятору для генерації об'єктного коду? Які дії виконує компілятор перед генерацією?
8 Дайте визначення поняттю оптимізації програми. Для чого використовується оптимізація?
9 Поясните, чому генерацію програми доводиться проводити в два етапи: генерація і оптимізація.
10 Які існують методи оптимізації об'єктного коду?
11 Що таке тріади і для чого вони використовуються? Які ще існують методи для представлення об'єктних команд?
12 Поясните роботу алгоритму згортки.
13 Що таке зайва операція? Що таке “число залежності”?
14 Поясните роботу алгоритму виключення зайвих операцій.

ВАРІАНТИ ЗАВДАНЬ

Варіанти завдань відповідають варіанту завдань для роботи №7.

Для виконання роботи рекомендується використовувати результати, отримані в ході виконання робіт №7 і №8.

ЛІТЕРАТУРА

1. Молчанов а.Ю. Системне програмне забезпечення. Лабораторний практикум. – Спб.: Пітер, 2005 – 284 з.

2. Гордєєв а.В., Молчанов а.Ю. Системне програмне забезпечення – Спб.: Пітер, 2001 (2002, 2003) - 736 з.

3. Ахо Альфред В., Мережі Раві, Ульман Джеффрі Д. Компілятори: принципи, технології і інструменти – М.: Видавничий дім «Вільямс», 2003 – 768 з.

4. Робін Хантер Основні концепції компіляторів – М.: Видавничий дім «Вільямс», 2002 – 256 з.

5. Бржезовський а.В., Корсакова н.В., Фільчаков в.В. Лексичний і синтаксичний аналіз. Формальні мови і граматики - Л.: ЛІАП, 1990.

6. Люіс Ф. і ін. Теоретичні основи побудови компіляторів - М.: Мир, 1979.

7. Ахо А., Ульман Дж. Теорія синтаксичного аналізу, перекладу і компіляції - М.: Мир, 1978, т.1.