МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

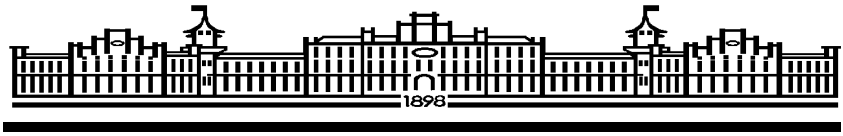
Національний технічний університет України

“Київський політехнічний інститут”

СИСТЕМНЕ ПРОГРАМУВАННЯ-2
РОЗРОБКА СИСТЕМНИХ ПРОГРАМ

Методичні вказівки до виконання
лабораторних робіт для студентів напрямку 6.050102
«Комп’ютерна інженерія»

Рекомендовано вченою радою факультету інформатики та обчислювальної техніки НТУУ «КПІ»



Київ 2012

МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

Національний технічний університет України

“Київський політехнічний інститут”

СИСТЕМНЕ ПРОГРАМУВАННЯ-2
РОЗРОБКА СИСТЕМНИХ ПРОГРАМ

Методичні вказівки до виконання
лабораторних робіт для студентів напрямку 6.050102
«Комп’ютерна інженерія»

Затверджено

на засіданні кафедри обчислювальної техніки

ФІОТ НТУУ «КПІ»

Протокол № від 12.02.12

Київ НТУУ «КПІ» 2012

УДК 681.3

ББК 0513-048р

П 797

Системне програмування – 2. Розробка системних програм: Методичні вказівки до виконання лабораторних робіт для студ. напрямку підготовки 6.050102 «Комп'ютерна інженерія» / В.І.Пустоваров О.В.Кузнєцов. – К.: НТУУ «КПІ», 2012. – 158 с.

Гриф надано вченою радою ФІОТ
(протокол № 4 від 28 листопада 2011 р.)

Методичні вказівки вміщують завдання до виконання лабораторних робіт по кредитному модулю «Системне програмування – 2. Розробка системних програм».

Наведені варіанти завдань. До кожної роботи надається необхідний теоретичний матеріал і перелік рекомендованої літератури.. Призначені для студентів напряму підготовки 6.050102 «Комп'ютерна інженерія».

Навчальне електронне видання

Укладачі: *Пустоваров Володимир Ілліч*, канд.техн.наук, доц.

Кузнєцов Олександр Вікторович, ст. викладач

Відповідальний
редактор: .*Стіренко С.Г. к.т.н., доцент*

Рецензент: *Марченко О.І*, канд. техн. наук, доц.

За редакцією авторів

ЗМІСТ

ВСТУП

Модуль «Системне програмування-2» є модулем нормативної дисципліни підготовки фахівців рівня бакалавр з напрямку 6.050102 «Комп'ютерна інженерія». Призначений для вивчення методів та засобів програмування для системних програм. Модуль вивчається на третьому курсі, тому вважається, що студенти вже засвоїли дисципліни «Програмування», «Системне програмування-1», «Інженерія програмного забезпечення» і в достатньої мірі володіють навичками створення програм за допомогою процедурно-орієнтованих мов програмування Pascal та C/C++.

Цикл лабораторних робіт складається з восьми робіт і призначений для покриття практичної частини другого кредитного модулю дисципліни «Системне програмування».

Роботи дозволяють отримати практичні навики написання елементів системних програм з використанням механізмів сучасних мов та бібліотек системного програмування.

Методичні вказівки включають для кожної роботи:

- теоретичний матеріал, необхідний для їх виконання;
- питання для самостійної перевірки;
- посилання на список рекомендованих джерел;
- варіанти завдань для виконання кожної лабораторної роботи з циклу.

1. МЕТОДИЧНІ ВКАЗІВКИ ЩО ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

1.1 Мета циклу лабораторних робіт

Метою проведення лабораторних занять з кредитного модулю "Системне програмування-2. Розробка системних програм" є закріплення теоретичних знань, умінь і навичок, пов'язаних з розробкою програмного забезпечення для системних програм (ПРСП), що базується на використанні механізмів роботи з найбільш поширеними структурами даних системних програм. Виконання робіт пов'язано з розробкою елементів системних програм, отриманні практичних навиків при роботі з мовами програмування типу Assembler та C/C++, а також використанні спеціальних бібліотек.

1.2 Зміст протоколу та оформлення лабораторних робіт

Протокол виконання кожної лабораторної роботи має містити:

- титульний лист;
- завдання на лабораторну роботу згідно варіанта;
- лістинг програми з коментаріями.
- результати експериментального виконання роботи;
- висновки по роботі (аналіз одержаних результатів).

Під час оформлення роботи слід звернути увагу на лістинг програми, в якому за допомогою відступів і коментарів відокремлювати основні частини програми, забезпечити легке читання лістинга. Програма повинна розпочинатися з заголовка, в якому треба вказати назву та варіант роботи, прізвище виконавця, дату, групу.

2. ЛАБОРАТОРНІ РОБОТИ

Цикл лабораторних робіт включає 8 робіт. Варіанти завдань видаються викладачем на підставі таблиць в описах робіт.

2.1 Лабораторна робота 2.1

Тема роботи: Модульне програмування в рамках базової системи проектування програм та його використання для побудови програм обробки таблиць

Мета роботи: Вивчення типів таблиць в системних програмах і конструкцій базової мови програмування для їх визначення. Пошук за прямою адресою. Основні типи залежностей та відношень, які реалізуються через пошук в таблицях системних програм. Лінійний та двійковий пошук. Вимоги до унікальності ключів.

Принципи побудови професійних системних програм

Щоб раціонально створювати програми і програмні комплекси, важливо використовувати принципи **модульного програмування**, тобто оформлення основи програми у вигляді комплексів процедур, функцій або інших підпрограм, які утворюють **модулі широкого використання** або **бібліотечні модулі мов програмування**. Базовий текст прикладних або системних програм також будується у вигляді окремого **управляючого модуля програми** або **модуля тестування**, які є головними модулями програми і включають набір контрольних прикладів або закінчену програму автономного тестування.

До того ж модульне програмування складає технологічну основу об'єктно-орієнтованого програмування та проектування об'єктів, коли модулі формуються з окремих об'єктів або груп взаємозв'язаних однотипних, успадкованих або близьких об'єктів. В таких об'єктно-орієнтованих модулях визначається як структура класу об'єкта, так і набори методів (процедур і функцій) для обробки примірників даних цього класу. В деяких випадках об'єктно-орієнтовані модулі реалізують через структури в рамках мов C/C++ та Асемблер або записи в рамках мови Pascal, наприклад, технологічні програми організації графічних інтерфейсів в середовищі UNIX, але такі засоби не мають вбудованих механізмів контролю за інкапсуляцією на відміну від примірників сучасних об'єктів та їх прототипів.

До організації раціонального комплексу програмних модулів, яка дозволяє багаторазове використання в системних програмах, висуваються наступні вимоги:

- функціональні елементи або методи повинні групуватись так, щоб кожна група включала і використовувала тільки або по 1, або по 2, або масив з заданою кількістю базових об'єктів;
- кожен функціональний елемент або метод повинен виконувати логічно закінчену дію і може включати будь-яку кількість викликів інших функціональних елементів;
- для загальності підпрограм більшість даних повинні передаватись до них через аргументи у вигляді власне даних, вказівників або посилань на складні агрегати даних;

- доцільно, щоб структури даних ієрархічно об’єднували дані, що стосуються єдиної фізичної сутності, що дозволяє зменшити кількість аргументів в підпрограмах (крім того, це складає основу формування об’єктно-орієнтованих даних в мові С без використання класів С++);
- кількість глобальних та статичних даних програм повинна бути мінімізовані тільки використанням лічильників об’єктів в групах та дійсно унікальних об’єктів програми;
- допоміжні підпрограми відображення для контролю результатів функціональних підпрограм повинні розміщуватися в окремому модулі відображення;
- контрольні приклади повинні розміщуватися в окремому тестовому модулі, де поряд з головною програмою або початковою функцією мови повинні знаходитися підпрограми генерації тестових послідовностей та зв’язок з діалоговими засобами формування даних контрольних прикладів.

Контрольні тестові проекти системних програм звичайно будуються для автономного налагодження окремих модулів, а випускні та інсталяційні редакції програм – для комплексного тестування програм і експлуатації для розв’язання задач користувача.

Короткі теоретичні відомості

Таблиці як двовимірні агрегати даних, які визначають функціональну залежність між окремими характеристиками, використовують в системних програмах та системах управління базами даних для відтворення даних при виконанні програм і маніпуляцій з базами даних.

Таблиці як базові структури і сховищ накопичуваної інформації системних програм

Основу кібернетичних методів складає абстрактна модель об’єкта, що змінюється в процесі розвитку або життєдіяльності системи. В залежності від типу об’єктів використовуються аналітичні (алгебраїчні), алгоритмічні (директивні), реляційні (табличні) та мережні моделі, а також їх різноманітні комбінації. В цій роботі основну увагу приділено вивченню техніки структурного проектування таблиць і побудові ефективних програм процедурними, об’єктно-орієнтованими та машинно-орієнтованими засобами програмування. При такому підході програми інформаційного пошуку розглядаються на загально системному рівні реалізації з абстрагуванням від прагматики конкретного застосування. Тому для прагматичних ілюстрацій візьмемо задачі пошуку при трансляції кодів з вхідних мов програмування та запитів.

В більшості системних програм головною метою пошуку є визначення характеристик, пов’язаних з символічними позначен-

нями елементів вхідної мови (ключових слів, імен, ідентифікаторів, роздільників, констант, тощо). Ці елементи розглядаються як аргументи пошуку або асоціативні ознаки інформації позначень. Функціями табличних перетворень є збереження або використання характеристик для обчислень елементів внутрішнього подання та вихідних кодів. Пошук за асоціаціями виконується як у вхідній оперативній інформації системної програми, так і в інформаційній базі (ІБ), де зберігаються систематизовані аргументи пошуку та їх впорядковані або систематизовані характеристики елементів кодів, що поєднуються в таблицю. Найбільш важливі задачі – це формування таблиць та швидкий оперативний пошук інформації в таблицях.

Особливості побудови сучасної ІБ демонструються типовими прикладами, які відрізняються роздільним зберіганням взірців характеристик об’єктів та їх структурних зв’язків. Такий підхід використовується для зберігання мультимедійних картин та сцен у вигляді кодів, узагальнених на рівні мови, призначеної для відтворення структур з керованими типами взірців.

Основу визначення функціональних залежностей в системних програмах та інформаційних системах складають однозначні та багатозначні табличні зв’язки між показниками або атрибутами об’єктів системних програм та інформаційних систем. Таблиці є основою так званої реляційної моделі в базах даних [1].

Реалізація таблиць як об’єктів в рамках мови C/C++ вимагає визначення структури для рядка таблиці, в якій повинні зберігатися ключові і функціональні поля. Саму ж таблицю варто визначати як

об’єкт класу, який включає рядки таблиці як динамічну складову у формі масиву структур. При додержанні вимог модульного програмування в проекті на мові C/C++ створюються файли заголовків, в яких зберігаються прототипи рядка таблиці, конструкторів, деструкторів та базових функцій або операцій, а також відповідний файл реалізації, приклади яких наведені в наступних параграфах. У відповідності з традиціями програмування роботи з таблицями реляційних баз даних [3] базовими операціями є: `select` – вибірка даних з таблиці; `insert` – додавання рядків даних до таблиці; `delete` – вилучення рядків даних з таблиці; `update` – заміна даних в рядку таблиці.

Найпростіший спосіб побудови ІБ полягає у визначенні структури окремих елементів, що вбудовуються в структуру таблиці. Рядки таблиці в загальному випадку складаються з полів двох типів елементів – аргументів та функціональних характеристик. Така класифікація умовна та визначається цільовими характеристиками предметної галузі та окремих задач. Як вже відзначалося, аргументом пошуку в загальному випадку можна використовувати декілька полів, але спочатку для спрощення задачі визначимо такі правила:

- аргумент пошуку подається одним полем структурного типу та пов’язаною з ним функцією порівняння, що дозволяє перекласти труднощі, викликані множиною значень та функціональними комбінаціями аргументів на методи порівняння;
- функціональна частина запису також зібрана в одному полі структурного типу;

- елемент, що аналізується або прототип пошуку, має таку ж або близьку структуру, як і елемент таблиці.

Ці припущення практично не знижують загальності аналізу та побудованих в результаті алгоритмів, але істотно спрощують програмування методів маніпуляції з таблицями. Кожний елемент звичайно зберігає декілька (m) характеристик і займає в пам'яті послідовність адресованих байтів. Якщо елемент займає k байтів і треба зберігати N елементів, то необхідно мати хоча б kN байтів пам'яті. Розмістити інформацію можна декількома способами [3]:

1. Всі елементи розмістити в kN послідовних байтах і побудувати таблицю з N елементів у вигляді масиву. Приклад такої таблиці наведено нижче, де елементи задаються структурою `struct` в мові С, записами `record` в мові Pascal, довжиною k байтів, що визначається сумою розмірів ключової та функціональної частини елемента таблиці.
2. Побудувати m таблиць у вигляді масивів, скажімо, T_1, T_2,... T_m, для кожної з m характеристик. При цьому i-й елемент таблиці буде розподілено по елементах масивів T_{1i}, T_{2i},..., T_{mi}. Цікаво відзначити, що при роботі коротких аргументів пошуку довжиною 1, 2 та 4 (для 32-розрядного режиму) байти процес обробки можна істотно прискорити, використовуючи машинні команди роботи з ланцюжками або рядками даних.
3. Розділити таблицю на l блоків, сегментів або підтаблиць. В граничному випадку можна одержати сукупність блоків, в яких зберігаються по одному елементу. В таку таблицю необхідно додати

покажчики на зв'язки у вигляді адрес пов'язаних елементів і, таким чином, організувати зв'язані списки або впорядковані деревоподібні структури.

При використанні мов високого рівня доцільно спочатку визначити структуру окремого запису або рядка таблиці за способами 1 і 3 як сукупності полів, для способу 2 – як контейнери окремих полів.

У форматі структур і класів можливо зберігати будь-які комплекси даних. У подальших прикладах цього параграфу визначаються тільки прототипи, для яких можна визначити різні реалізації методів залежно від обраних підходів та алгоритмів доступу даних та маніпуляцій даними. Формат структур доцільно використати для окремого групування функціональних і ключових полів, до яких здійснюється доступ за іменами. Структури ключових і функціональних полів таблиць в рамках мови C/C++ можуть бути спочатку оголошені, а потім детально визначені як структури або класи.

```
struct keyStr;
            // структура ключової частини таблиці
struct fStr;
          //структура функціональних полів таблиці
```

У форматі структур доцільно визначати ключові рядки та поля індексів, які використовуються як аргументи пошуку та впорядкування, та таблиці разом з відповідними методами, які реалізують типові функції для роботи зі структурними об'єктами. При визна-

ченні розширених структур більш високих рівнів до них включаються простіші структури, як в **контейнери**.

Серед широкої множини варіантів побудови таблиць оберемо для прикладів логічно найпростішу організацію записів реляційних таблиць у вигляді структур та додаткових змінних, які визначають поточні характеристики таблиці. Файл заголовків "tables.h" для створення таблиць в мові C/C++ може бути подібним до наступного прикладу, причому імена власних даних та їх типів програміст, як завжди, може зручно задавати за власними міркуваннями:

```
struct keyStr // ключова частина запису
{char* str;  // ключові поля
 int  nMod;};// (уточнюється за варіантом)
struct fStr;// функціональна частина запису
{long double _f;};
//f-поле (уточнюється за завданням)
struct recrd // структура рядка таблиці
{struct keyStr key;// примірник структури ключа
 struct fStr func;
// примірник функціональної частини
 char _del;};// ознака вилучення
// обробка таблиць за прямою адресою
// вибірка за прямою адресою
struct recrd* selNmb(struct recrd*, int nElm);
// включення за прямою адресою
struct recrd* insNmb(struct recrd*pElm,
       struct recrd*tb, int nElm, int*pQnElm);
// вилучення за прямою адресою
```

```
struct recrd* delNmb(struct recrd*, int nElm);
// корекція за прямою адресою
struct recrd* updNmb(struct recrd *pElm,
       struct recrd*tb, int nElm, int*pQnElm);
// порівняння рядків за відношенням порядку
int cmpStr(unsigned char* s1, unsigned char* s2);
// порівняння ключів за відношенням нерівності
int neqKey(struct recrd*, struct keyStr);
// порівняння ключів за відношенням порядку
int cmpKey(struct recrd*, struct keyStr);
// порівняння за відношенням схожості
int simKey(struct recrd*, struct keyStr);
// вибірка за лінійним пошуком
struct recrd* selLin(struct keyStr kArg,
        struct recrd*tb, int ln);
// вибірка за двійковим пошуком
struct recrd*selBin(struct keyStr kArg,
                    struct recrd*tb, int ln);
```

Прикладом функції пошуку за прямою адресою у файлі реалізації до файлів шаблону включено наступну функцію:

```
// вибірка за прямою адресою
struct recrd* selNmb(struct recrd* tb, int nElm)
{return &tb[nElm];
}
```

а прикладами функцій порівняння

```
// порівняння рядків за відношенням порядку
int neqKey(struct recrd* el, struct keyStr kArg)
// порівняння за відношенням нерівності
```

```
{return (strcmp(el->key.str, kArg.str)||
        el->key.nMod != kArg.nMod);
}
// порівняння структур за відношенням порядку
int cmpStr(unsigned char* s1, unsigned char* s2)
{unsigned n;
 while(s1[n]==s2[n]&&s1[n]!=0)n++;
 return s1[n]-s2[n];}
int cmpKey(struct recrd* el, struct keyStr kArg)
{ int i=cmpStr((unsigned char*)el->key.str,
             (unsigned char*)kArg.str);
 if(i)return i;
 return el->key.nMod - kArg.nMod;
}
// вибірка за лінійним пошуком
struct recrd*selLin(struct keyStr kArg,
                    struct recrd*tb,int ln)
{while(--ln>=0&&cmpKey(&tb[ln], kArg));
 if(ln<0)return 0;
 return &tb[ln];
}
// вибірка за двійковим пошуком
struct recrd*selBin
(struct keyStr kArg,// ключ аргументу пошуку
 struct recrd*tb,   // адреса початку таблиці
 int ln)     // кількість елементів таблиці
{int i, nD=-1, nU=ln, n=(nD+nU)>>1;
 while(i=cmpKey(&tb[n],kArg))
```

```
    {if(i>0)nU=n; else nD=n;
     n=(nD+nU)>>1;
     if(n==nD)return NULL;
    }
 return &tb[n];
}
// виведення рядка базової таблиці
void prRow(struct recrd* rw)
{if(rw==0)printf("is absent\n");
 else printf("%10s %3u %5.3f\n",
// Останній шаблон рядка формату і відповідне
// поле списку виведення даних необхідно
// узгодити з варіантом завдання
       rw->key.str,rw->key.nMod,rw->func._f);
}
```

При використанні об’єктно-орієнтованого програмування на базі класів C++ об’єкти можуть бути побудовані функціями передачі відповідних повідомлень ізоморфно до об’єднань структур. В цьому випадку відповідні функції включаються до класів як методи, зв’язані з базовим об’єктом, який стає досяжним без передачі параметрів, що спрощує описи методів-функцій. Базовий клас можна розширювати через успадковані класи, включаючи до нього додаткові ключові поля будь-яким чином. За необхідності використання класів доцільно задавати декларації класів з визначенням повних характеристик об’єкта таблиці з додаванням записів у вигляді примірників класу або запису.

Основні типи відношень в процедурах пошуку

Відношення рівності та нерівності

При пошуку за прямою адресою та лінійному пошуку в невпорядкованих таблицях перевіряються відношення рівності або нерівності для вибірки потрібних записів та елементів. Операції відношення мови C/C++ "=" та "!=" створюються за умовчанням для всіх визначених типів даних порівняння всіх відповідних пар полів основної структури. Однак для типів даних з динамічними складовими нерівність полів вказівників ще не свідчить про нерівність даних, що знаходяться за покажчиками. Тому операції "=" та "!=" для типів даних з динамічними складовими необхідно перевизначити. Стандартна функція ANSI C `strcmp` перевіряє саме відношення нерівності рядків і може успішно використовуватись в різних варіантах лінійного пошуку.

Відношення порядку

Впорядкування таблиць за ключовими полями стає можливим лише у випадку впорядкування кодів кожного з потрібних ключових полів. Відношення порядку встановлюються для даних з одновимірними множинами визначення значень (доменами). Всі числові і перенумеровані типи даних, в тому числі і полів, мають визначені відношення порядку. Тому набори операцій мови C/C++ "=" та "!=", які створюються за умовчанням, необхідно розширити додатковими операціями відношень мови C "<", "<=", ">=" та ">", в тому числі з урахуванням полів з вказівниками.

Для визначення відношення порядку багатокомпонентних типів необхідно, щоб кожний компонент мав відношення порядку, і щоб узагальнююче відношення будувалося за допомогою монотонних функцій. Відношення порядку завжди визначаються при побудові індексів в реляційних системах управління базами даних (СУБД), наприклад, в більшості SQL-серверів як зважене об'єднання полів, а в деяких системах, наприклад у FoxPro, припустимий індексний вираз, що визначає функцію полів.

Методи двійкового пошуку можна використовувати лише у впорядкованих таблицях або у відповідних індексах. Тобто таблиці для виконання такого пошуку повинні бути попередньо відсортовані і для роботи з ними треба створити функцію сортування.

Відношення близькості

При пошуку помилково підготовлених ключів в текстових редакторах та процесорах часто виникає потреба в визначенні схожості ключів пошуку. Такі дії часто виконуються в текстовому процесорі MS Word. Вони можуть будуватися на підрахунку кількості однакових n_e, схожих літер n_{si} за i-м типом схожості, а також літер, які не мають відповідника в іншому ключі і можуть спиратися на абсолютні і відносні формульні критерії схожості. Схожість літер може визначатися залежно від випадку аналізу за схожістю написання літер в різних алфавітах n_{s1}, за близькістю комп'ютерних кодів n_{s2} та за близькістю розташування на клавіатурі n_{s3}, а також з урахуванням кількості літер n_{s4}, які не мають відповідників в обох ключах.

При створенні програм порівняння за мірою близькості треба побудувати загальний критерій близькості як монотонну функцію $f(n_{s1}, n_{s2}, n_{s3}, n_{s4})$ в одному напрямку від n_{s1}, n_{s2} і n_{s3} та в іншому напрямку від n_{s4}. Крім того, попередньо необхідно організувати підрахунок n_{s1}, n_{s2}, n_{s3} і n_{s4}, при перегляді порівнюваних ключів. Результат пошуку за таким критерієм може бути неоднозначним, навіть за умови вимоги однозначності ключів. На алгоритм лінійного пошуку це практичного не впливає, а у випадку базового двійкового пошуку доцільно виконати додатковий пошук навколо найближчого ключа, знайденого за відношенням порядку.

Рекомендації з вибору алгоритму оцінки міри близькості за відношенням близькості полягають в тому, що найбільш повну і точну оцінку міри близькості можна одержати просуваючись за алгоритмами, в яких організуються вкладені цикли. В таких циклах симетрично визначається міра близькості між двома порівнюваними ключами, шляхом підрахунку однакових та/або різних символів і наступного підрахунку за формулами абсолютної або відносної міри близькості (відносно загальної довжини імен).

Вимога унікальності ключів і аргументів пошуку

В системних програмах звичайно додержуються вимог унікальності ключів, і порушення такої вимоги розглядається як помилка повторного опису. Таких помилок і порушень вимоги унікальності можна уникнути, додаючи до запису таблиці додаткове ключове поле позиції розміщення визначення ключа. Але в реляційних СУБД повторювані значення ключів часто використовують-

ся при зв'язуванні таблиць, і поля з однаковими значеннями ключів можна впорядковувати довільним чином. В цьому випадку важливо мати набір функцій, які дозволятимуть формувати: 1) перший ліпший результат пошуку; 2) черговий результат пошуку при черговому запиті; 3) всі можливі результати пошуку при одному виклику.

Структура програмного шаблона виконання роботи

Для спрощення розв'язання задачі слід використовувати прототип проекту spLb1.dsp, який зберігається в робочому просторі spLb1.dsw в папці spLb1, яка зберігає шаблони прототипів для всіх лабораторних робіт циклу «Системне програмування-2». До складу проекту консольної прикладної програми, орієнтованої на можливість використання в числі інших бібліотек стандартних функцій та комплексу об'єктів MFC (Microsoft Foundation Classes) входять модулі з наступними вхідними файлами заголовків і реалізацій:

- StdAfx.h і StdAfx.cpp – модуль для організації використання об'єктів класів MFC, створених поза стандартами ANSI;
- tables.h і tables.cpp – модуль для опису і організації використання структур або класів таблиць;
- vistab.h і vistab.cpp – модуль для відображення рядків або повних таблиць;
- spLb1.cpp – модуль контрольних прикладів для тестування функцій обробки таблиць.

При побудові проектів без функцій MFC та в інших системах, що дозволяють побудову проектів, файли StdAfx.h і StdAfx.cpp можуть бути опущені, а посилання на них в операторах `#include "StdAfx.h"` повинні бути виключені.

Розділ визначень і декларацій основної частини програми тестування в модулі spLb1.cpp повинен включати визначення таблиць, ініціалізацію їх елементів, спеціальні змінні, що зберігають довжину (тобто кількість елементів) таблиці. Виконувана частина модуля spLb1.cpp програми повинна включати циклічно повторювані виклики функцій для перевірки та відображення різних варіантів вхідних даних так, щоб перевірити роботу функцій при типових та кінцевих значення з множини припустимих значень аргументів.

Рекомендації з корекції модуля відображення

Для настройки виведення таблиць згідно з варіантом треба внести зміни до функції виведення рядка. Ігнорування необхідності таких змін може призвести до аварійного завершення програми. Головним для коректного відображення є додержання правил звертання до функції виведення `printf`, яка виконує в мові С форматне виведення на пристрій відображення або в стандартний файл виведення stdout в формі:

`printf(`*рядок формату, список аргументів виведення*`);`

Рядок формату задається або у вигляді константи-рядка, яка включає символи коду ASCII з відповідною національною таблицею кодування, замкнені в лапки, або покажчик на масив знаків або

символів. Кількість аргументів визначається кількістю специфікаторів перетворення даних виведення, наведених в таблиці 1.1, а тип даних, що виводиться, повинен відповідати типу специфікатора. При цьому відповідальність за кількість об'єктів виведення покладається на програміста, через що потрібно бути дуже уважним при написанні і модифікації подібних викликів функції.

В рядку формату спеціальні символи можуть задаватися після знака "\", а власне специфікатори перетворення починаються зі знака "%". Для включення в текст рядка одного знака "%" його необхідно повторити. Специфікатор перетворення в загальному випадку має синтаксис:

%*[вирівнювання][ширина][префікс формату]*
специфікатор формату

Ці префікси можуть використовуватися при роботі з рядками в форматі `%s` і `%p`, тому що звертання до рядків завжди реалізуються в формі вказівників.

При виконанні функції `printf` рядок формату передається в вихідний файл без змін до одержання першого символу `%`. Після чого виконується перетворення чергового аргументу у відповідності з поточним форматом, результат цього перетворення також розміщується в вихідному файлі.

***Таблиця* 1.1**
Формати для виведення даних

Специфікатор (остання літера)		Типи даних та відображення	Вирівнювання	Префікси формату
	Система числення | Знаки	**Цілі**		
c		Символ char	*[-]*	
s		Рядок *char	*[-] [+]* [ ]	*[l] [N] [F]*
		Цілі зі знаком		
d		int, long	[-] [#]	[l]
		Цілі без знака		
x		Шістнадцяткові,		
X		Те саме, великими		*[N] [F]*
o		Вісімкові		
u		Десяткові		
	Формат подання	**З плаваючою точкою**	*[-] [+] [#]* *[ ]*	*[.точність]* *[L]*
f		Фіксована точка		
e		Експоненційний		
E		Те ж з великим E		
g		Найкоротший f | e		
G		Найкоротший f | E		
	Вказівники			
p		У форматі адреси		*[N]* *[F]*

Умовні позначення:

Нахилені квадратні дужки *[]* – ознака факультативності або необов'язковості конструкції;

[-] – зміщення результату перетворення до лівої межі поля виведення;

[+] – виведення знака перед числом;

[ ] – пропуск замість плюса або виведення мінуса;

[#] – кодове подання з початковими або кінцевими нулями;

[l] – префікс довгого формату, який визначає подання даних, що перетворюються в більш довгих форматах;

ширина – визначає мінімальне число знаків або символів виведення.

точність – визначає число цифр після десяткової точки для форматів f, e і E або число значущих цифр для форматів g і G. Округлення виконується шляхом відкидання молодших розрядів. За умовчання приймається точність в шість десяткових цифр.

Рекомендації з модифікації і розширення функцій модулів

Для настройки виведення таблиць згідно з варіантом треба внести зміни до функції виведення рядка з потрібним форматом.

При побудові функцій вставки, вилучення і корекції таблиць для двійкового пошуку необхідне попереднє розпізнавання успішності пошуку. Для цього зручно використати функцію двійкового пошуку **struct** recrd*selBin(**struct** keyStr kArg, **struct** recrd *tb, **int** ln), яка повертає або адресу запису зі знайденим ключем, або стандартне значення невизначеного вказівника NULL. Для вилучення і корекції відповідні зміни можна виконати за адресою, яку повертає функція, а для вставки нового елемента треба змістити останні елементи таблиці до місця вставки в операторі циклу, а потім занести новий елемент до таблиці.

Завдання на роботу

Завдання на підготовку до роботи на комп'ютері

1. Визначити варіант завдання для основних і додаткової задачі за Табл.1.2.

2. Розробити структуру даних для елементу таблиці згідно з варіантом за Табл.1.2. Якщо за варіантом треба використати тип даних, який визначається програмістом (**enum, struct, union**), то цей тип необхідно визначити перед використанням. Перед початком заняття показати викладачу ***текст опису структури і необхідних типів даних функціональної частини поля***.

3. Ознайомитись з шаблоном програмного проекту spLb1. Настроїти відповідні дані в програмному проекті на мові С.

4. Настроїти функції пошуку за прямою адресою, а також лінійного та двійкового пошуку, відмітки про вилучення, упакування таблиці, впорядкування таблиці і вставку до таблиці за значенням ключових полів з використанням різних методів.

- для таблиці з доступом за прямою адресою налагодження виконати в файлах tables.h, table.cpp і vistab.cpp;
- для таблиці з доступом за ключовим полем або групою полів за методом лінійного пошуку – налагодження виконати в файлах tables.h, table.cpp і vistab.cpp;

5. Настроїти програми відображення з модуля vistab.cpp так, щоб вони давали можливість відтворювати потрібні типи даних. Перед початком заняття показати викладачу ***текст оператора виведення функціональних полів***.

6. Скласти алгоритми функцій вставки, вилучення і корекції в таблицях на базі методу двійкового пошуку і підготувати програмні модулі, які забезпечують вставку, вилучення і корекцію елемента таблиці за значенням ключового елемента: для таблиці з доступом за ключовим полем або групою полів за методом двійкового пошуку) – заготовки для функцій `insBin`, `delBin` і `updBin` в кінці модуля tables.cpp.

7. Підготувати програмні модулі для додаткової задачі: пошуку з порівнянням за мірою близькості за варіантом з таблиці 1.3.

8. Підготувати програмний модуль контрольної задачі, який виконує задані варіанти програм пошуку і вставки в таблицю за значенням ключового елемента, а також додаткових задач і дозволяє перевірити коректність виконання програм.

Завдання на роботу на комп'ютері

9. Побудувати програмний проект, ввівши програмні модулі у відповідні файли проекту і налагодити синтаксис.

10. Побудувати виконаваний модуль тестової програми і налагодити змістовне виконання програми.

11. Одержати результати виконання, проаналізувати їх і зробити висновки.

Порядок вибору варіанту:

За останньою цифрою номера залікової книжки або за порядковим номером студента в списку підгрупи визначте варіант завдання для задач за табл. 1.2 та додаткового завдання за табл.1.3.

Питання для самостійної перевірки

1. Як в мовах Асемблера та С можна реалізувати елементи таблиці та поєднати їх в повноцінні таблиці?
2. Яку роль грають ключові поля записів таблиць?
3. Яку роль грають функціональні поля записів таблиць?
4. Порівняйте основні методи пошуку за швидкістю?
5. В яких випадках доцільно використання пошуку за прямою адресою.
6. В яких випадках доцільно використання лінійного пошуку.
7. В яких випадках доцільно використання двійкового пошуку.

Таблиця **1.2**

Варіанти завдань для виконання пошуку

№ вар.	Тип ключа для прямої адреси	Тип ключа для інших видів пошуку	Тип функціонального поля	Тип вибірки
1	**unsigned char**	**char*_ unsigned short**	**float**	Перший
2	**unsigned char**	**char*_ unsigned int**	**double**	Всі
3	**unsigned char**	**char*_ unsigned long**	**struct**	Черговий
4	**unsigned char**	**char*_ unsigned char**	**union**	Перший
5	**unsigned char**	**char*_ unsigned short**	**enum**	Всі
6	**unsigned short**	**char*_ unsigned int**	**float**	Черговий
7	**unsigned short**	**char*_ unsigned long**	**double**	Перший
8	**unsigned short**	**char*_ unsigned char**	**struct**	Всі
9	**unsigned short**	**char*_ unsigned short**	**union**	Черговий
10	**unsigned short**	**char*_ unsigned int**	**enum**	Перший
11	**unsigned int**	**char*_ unsigned long**	**float**	Всі
12	**unsigned int**	**char*_ unsigned char**	**double**	Черговий
13	**unsigned int**	**char*_ unsigned short**	**struct**	Перший
14	**unsigned int**	**char*_ unsigned int**	**union**	Всі
15	**unsigned int**	**char*_ unsigned long**	**enum**	Черговий
16	**unsigned long**	**char*_ unsigned char**	**float**	Перший
17	**unsigned long**	**char*_ unsigned short**	**double**	Всі
18	**unsigned long**	**char*_ unsigned int**	**struct**	Перший
19	**unsigned long**	**char*_ unsigned long**	**union**	Всі
20	**unsigned long**	**char*_ unsigned char**	**enum**	Черговий

Варіанти типу вибірки означають:

- “Перший” – складання програми пошуку першого елементу поля з унікальними значеннями або з повтореннями значень;
- “Всі” – складання програми багатоваріантного пошуку всіх можливих результатів для ключа з повтореннями значень;
- “Черговий” – складання програми пошуку чергового варіанта результату для ключового поля з повтореннями значень.

***Таблиця* 1.3**

Варіанти завдань для виконання порівняння ключів з використанням міри схожості

№ вар.	Тип функції порівняння за мірою схожості
1	Співпадіння максимуму початкових літер у ключі.
2	Співпадіння максимальної кількості літер без урахування їх послідовності у ключі.
3	Співпадіння максимальної кількості літер з врахуванням послідовності їх входження у ключі.
4	Співпадіння максимуму початкових літер без врахування регістрів великих і маленьких літер.
5	Співпадіння максимальної кількості літер без врахування регістрів і їх послідовності в ключі.
6	Співпадіння максимальної кількості літер з врахуванням їх послідовності без врахування регістрів літер.
7	Співпадіння максимуму початкових літер, включаючи літери латинського і слов'янського алфавіту, що співпадають за написанням.
8	Співпадіння максимальної кількості літер, включаючи літери латинського і слов'янського алфавіту, що співпадають за написанням.
9	Співпадіння максимуму початкових літер, включаючи літери латинського і слов'янського алфавіту, що співпадають за написанням інваріантно до регістру літер.
10	Співпадіння максимальної кількості літер, включаючи літери латинського і слов'янського алфавіту, що співпадають за написанням інваріантно до регістру літер.

2.2 Лабораторна робота 2.2

Тема роботи: Побудова і використання об'єктів вузлів деревоподібних та ієрархічних графів

Мета роботи: вивчення методів створення та використання вузлів графів автоматів, а також деревоподібних та ієрархічних графів, організації доступу до інформації, реконструкції вхідного тексту та скорочення графів через вилучення повторних вузлів.

Короткі теоретичні відомості

Більшість видів граматичного аналізу та семантичної або змістовної обробки текстів програм або скриптів у форматах внутрішнього подання спираються на різні подання графів та їх деревоподібних підвидів. Автомати перетворення вхідного тексту на лексеми спираються на графи зміни станів автоматів, в яких важливими є стани та сигнали переключення автомату.

Графи, що використовуються для граматичного аналізу

В результаті синтаксичного аналізу формуються дерева розбору (parse tree), вузли яких відображують термінальні та нетермінальні позначення, розрізнені в процесі синтаксичного розбору шляхом використання правил підстановки [1]. Для подальшої семантичної обробки з використанням відношень передування або пріоритетів операцій вони перетворюються на графи підлеглості операцій та ключових слів або скорочені спрямовані ациклічні графи (directed acyclic graph – DAG) [7,8], які відображують розді-

льники і ключові слова як нетермінали, а імена та константи як термінали.

Реалізація автоматів з пам'яттю в системних програмах

Основу найпростішої програмної реалізації скінченного автомата складають коди стану автомата та так звана матриця переходів автомата. Коди стану, частіше за все, визначаються перенумерованим типом з іменованими значеннями станів. Приклад опису типу для подання автомата з 5-ма станами показано нижче на рис. 2.1. У вузлах, позначених кружальцями записано ім'я стану, а біля дуг, позначених стрілками записані імена сигналів, що переключають автомат до стану, показного в напрямку стрілки. Відсутність дуги, що виходить з вузла з відповідним номером сигналу, дає інформацію про те, що за цим сигналом відповідний стан не змінюється.

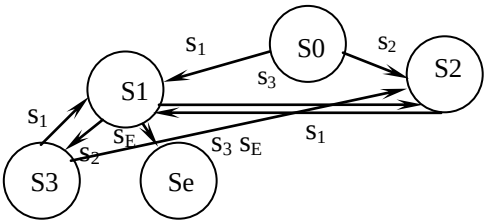


Рис. 2.1. Стани кінцевого автомата

```
enum autStat
{S0,   // S0 - Початковий стан
 S1,   // S1 - Перший стан
 S2,   // S2 - Другий стан
```

```
 S3,   // S3 - Третій стан
 Se    // Se - Кінцевий стан
};
```

Коди сигналів також зручно визначати іншим перенумерованим типом з іменованими значеннями кодів сигналів. Приклад опису типу для подання 5 типів сигналів, показаного на рис. 2.1, наведено нижче.

```
enum autSgn
{sg0,  // sg0 - Початковий сигнал
 sg1,  // sg1 - Перший сигнал
 sg2,  // sg2 - Другий сигнал
 sg3,  // sg3 - Третій сигнал
 sgE   // sgE - Кінцевий сигнал
};
```

Матриця переходів визначається двомірним масивом типу **enum** autStat, перший індекс якого визначає ціле число, яке відповідає попередньому стану автомата, а другий індекс – число, яке відповідає сигналу або класу сигналу для переведення автомата в наступний стан.

```
enum autStat nxtSts[Se+1][ sgE+1] =
{{S0,S1,S2,S0,S0},  //для S0
 {S1,S1,S1,S2,Se},  // S1
 {S2,S1,S2,S2,S2},  // S2
 {S3,S1,S3,S2,S3},  // S3
 {Se,Se,Se,Se,Se}}; // Se
```

Функція переходів визначена в лабораторній роботі наступним чином:

```
enum autStat nxtStat(enum autSgn sgn)
{static enum autStat s=S0;// поточний стан лексеми
 return s=nxtSts[s][sgn];} // новий стан лексеми
```

До виконуваної частини треба додати масив з послідовністю вхідних сигналів та цикл звертання до обробки з роздруком результатів моделювання роботи автомата:

```
enum autSgn ASgn[10]={sg1,sg2,sg1,sg0,...,sgE}; ...
for(n=0;n<10;n++)
      printf("%5d->%2d  ",ASgn[n],nxtStat(ASgn[n]));
```

Для того щоб надати можливість використання одноманітних структур даних на різних етапах обробки комп'ютерних мов і, таким чином, мінімізувати витрати на повторювані пересилання даних, важливо визначити базову структуру елемента внутрішнього подання або лексеми стандартним для всіх етапів виконання програми. Найпростіший спосіб побудови такої структури полягає у включенні до неї всіх можливих елементів, які створюються та використовуються на всіх етапах лексичного та синтаксичного аналізу, а також семантичної обробки вхідної мови. Тоді кожна лексема повинна визначатися структурою, яка зберігає дані лексичного аналізу: координати розміщення лексеми у вхідному файлі, код типу лексеми, який визначається за таблицями лексичного аналізу, а також зв'язки вузла лексеми в графі внутрішнього подання, які визначаються при синтаксичному аналізі з додатковими семантичними характеристики вузлів і підграфів.

```
struct lxNode//вузол дерева або САГ
{int ndOp;     //код операції або типу лексеми
```

```
unsigned stkLength;// номер модуля для терміналів
struct lxNode* prvNd, *pstNd;// до підлеглих вузлів
int dataType;   // код типу даних, які повертаються
unsigned resLength;  //довжина результату
int x, y, f;//координати розміщення у вхідному файлі
struct lxNode*prnNd;};//до батьківського вузла
```

Такі вузли з одного боку визначають масив лексем у послідовності їх надходження, в якому, як і в первинному графі підлеглості операцій, що утворюється при синтаксичному розборі, істотною є не послідовність полів, а визначальні зв'язки з іншими вузлами графа. Але при мінімізації вузли графа розглядаються як елементи таблиці, в якій ключами повинні бути однозначно подані вирази, для яких визначається відношення порядку в послідовності ієрархічного або рекурсивного відтворення рядків з ключових полів вузлів графа.

Визначення відношення порядку спирається на принципи відображення в деревах або графах префіксних форматів відтворення виразів. Як було показано в лабораторній роботі 2.1, відношення порядку ключових полів складає основу впорядкування таблиць та побудови впорядковуючих індексів для всіх видів підграфів. При використанні відношень порядку індекси для послідовностей записів (структур) вузлів лексем дозволяють підвищувати швидкість вибірки при розв'язанні різних задач семантичної обробки в системних програмах, насамперед, машинно-незалежної оптимізації. Приклади графів простих виразів і умовних операторів показані на рис 2.2, рис. 2.3, рис. 2.4 і рис. 2.5.

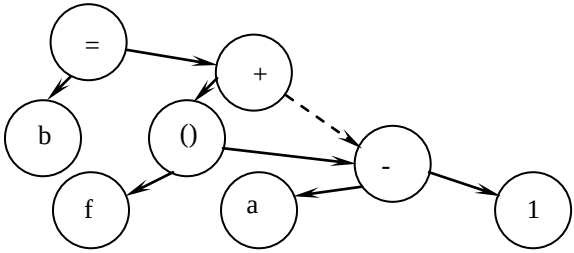


Рис. 2.2. Спрямований ациклічний граф оператора присвоювання `b=f(a-1)+a-1`

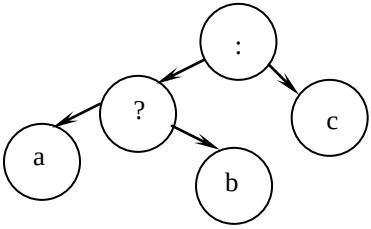


Рис. 2.3. Дерево підлеглості операцій для виразу `a?b:c`

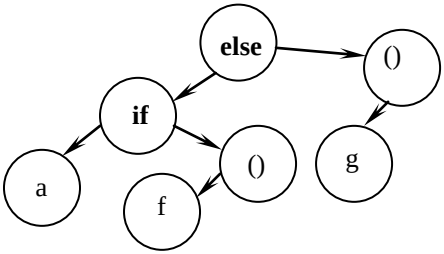


Рис.2.4. Дерево підлеглості для оператора **`if`**`(a)f();`**`else`** `g();`

В прикладах графів використані посилання на зв'язки на повторювані підвирази або підграфи показані *штриховою* лінією, а зворотні зв'язки для операторів **`break`** і **`continue`** показані на рис. 2.5 та рис. 2.7 *штрих-пунктирною* лінією. Ключові слова

endcase та **endloop** використовуються в деяких комп'ютерних мовах, щоб уникнути плутанини з закінченням відповідних блоків.

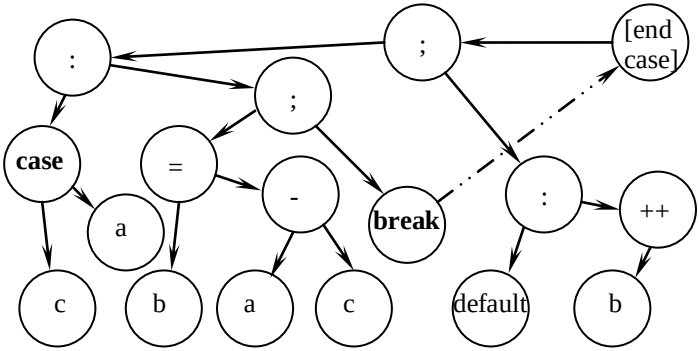


Рис. 2.5. Спрямований ациклічний граф для оператора:
switch(c){**case** a:b=a-c; **break; default:** b++;}

На рис. 2.6 показаний простий оператор циклу з післяумовою, а на рис. 2.7 – зв'язка операторів в циклі з передумовою, яка виконуються при двійковому пошуку.

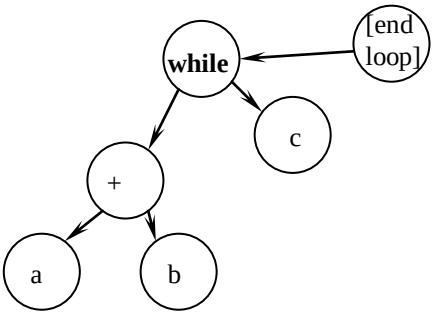


Рис. 2.6. Дерево підлеглості для оператора
do a+=b; **while**(c);

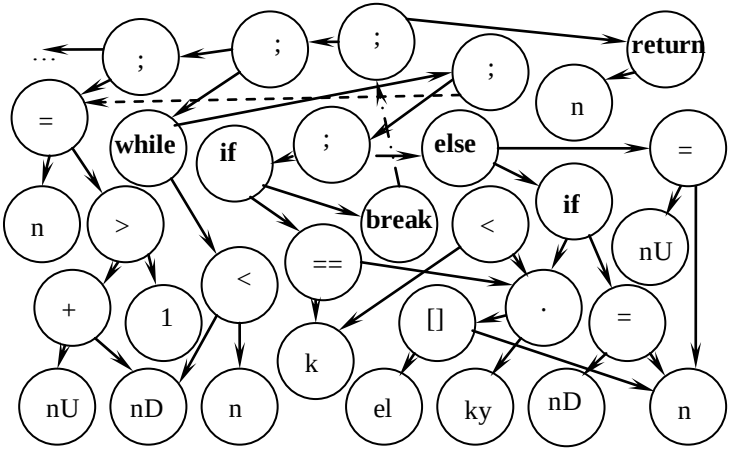


Рис. 2.7. Спрямований ациклічний граф для зв'язки операторів: …;

```
n=(nD+nU)>>1;while(nD<n){if(k==el[n].ky)break;
if(k<el[n].ky)nD=n; else nU=n;n=(nD+nU)/2;};
return n;
```

Різниця між поданнями операторів з передумовами і післяумовами полягає, насамперед, у приєднанні повторюваних операторів та умов до протилежних гілок графа. Крім того, вони повинні бути відображені вузлами семантично різних типів: `_whileP` та `_untilP` для передумов або `_whileN` та `_untilN` для післяумов. Цикли типу **`for`** також є циклами з передумовами і вузли їх заголовку розміщується за посиланням на перший (лівий) вказівник. Вузол [**`endloop`**] є взагалі необов'язковим і може використовуватися для відображення закінчення циклу в деяких мовах.

Для того щоб визначити відношення порядку і створити більш зручні умови для побудови індексів, що полегшують різні

види семантичної обробки, насамперед, машинно-незалежну оптимізацію важливо перегрупувати структуру **struct** lxNode до вигляду, де ключові елементи відношення порядку розташовані на початку структури:

```
struct lxNode //нетермінальний вузол дерева або САГ
{int ndOp;     //код типу лексеми
 struct lxNode* prvNd;// зв'язок з попередником
 struct lxNode* pstNd;// зв'язок з наступником
 int dataType;   // код типу даних, які повертаються
 unsigned resLength;  //довжина результату
 unsigned stkLength;//довжина стека обробки семантики
 int x, y, f;//координати розміщення у вхідному файлі
 struct lxNode* prnNd;//зв'язок з батьківським вузлом
};
```

Приклади подання графів простих виразів, показані на рис. 2.2, 2.3, 2.4, 2.5 і 2.6, будуються на основі масиву управління відтворенням символьних образів **char** *imgs[]. Нижче показано приклад масиву вузлів для рис. 2.2 з кореневим вузлом pic1[1].

```
char *imgs[]={"b","a","1","f","c","g","k","ky",
                                    "n","nD","nU","el"};
struct lxNode pic1[]=      // b=f(a-1)+a-1
{{_nam,(struct lxNode*)imgs[0], NULL},
 {_ass,&pic1[0], &pic1[2]},
 {_nam, (struct lxNode*)imgs[3], NULL},
 {_brkt,&pic1[2], &pic1[5] },
 {_nam,(struct lxNode*)imgs[1], NULL},
 {_sub, &pic1[4], &pic1[6]},
```

```
 {_srcn,(struct lxNode*)imgs[2], NULL},
 {_add, &pic1[3], &pic1[5]}
};
```

Наступний опис показує приклад масиву вузлів для рис. 2.3 з кореневим вузлом `pic2[1]`.

```
struct lxNode pic2[]=      // b=a?b:c
{{_nam, (struct lxNode*)imgs[0], NULL},
 {_ass, &pic2[0], &pic2[5]},
 {_nam, (struct lxNode*)imgs[1], NULL},
 {_qmrk,&pic2[1], &pic2[4]},
 {_nam, (struct lxNode*)imgs[0], },
 {_cln, &pic2[3], &pic2[6]},
 {_nam, (struct lxNode*)imgs[4], NULL}
};
```

Наступний опис показує приклад масиву вузлів для рис. 2.4 з кореневим вузлом `pic3[4]`.

```
  struct lxNode pic3[]=       // if(a)f();else g();
  {{_nam,(struct lxNode*)imgs[1], NULL},
   {_if, &pic3[0], &pic3[3]},
   {_nam,(struct lxNode*)imgs[3], NULL},
   {_brkt,&pic3[2], NULL, 0, 0, 0, 0, 0, NULL, 0},
   {_else,&pic3[1], &pic3[6] },
   {_nam,(struct lxNode*)imgs[5], NULL},
   {_brkt,&pic3[5], NULL},
  };
```

Наступний опис показує приклад масиву вузлів для рис. 2.5 з кореневим вузлом `pic4[16]`.

```
struct lxNode pic4[]= // switch(c)
            // {case a:b=a-c; break; default :b++;}
{{_nam, (struct lxNode*)imgs[4], NULL},
 {_case,&pic4[0], &pic4[2] },
 {_nam, (struct lxNode*)imgs[1], NULL},
 {_cln, &pic4[1], &pic4[9] },
 {_nam, (struct lxNode*)imgs[0], NULL},
 {_ass, &pic4[4], &pic4[7]},
 {_nam, (struct lxNode*)imgs[1], NULL},
 {_sub, &pic4[6], &pic4[8]},
 {_nam, (struct lxNode*)imgs[4], NULL},
 {_EOS, &pic4[5], &pic4[10]},
 {_break, NULL, NULL},
 {_EOS, &pic4[3], &pic4[13]},
 {_default, NULL, NULL},
 {_cln, &pic4[12],&pic4[15]},
 {_nam, (struct lxNode*)imgs[0], NULL},
 {_inr, &pic4[14], NULL},
 {_endcase, &pic4[11], NULL}
};
```

Наступний опис показує приклад масиву вузлів для рис. 2.6 з кореневим вузлом `pic5[0]`.

```
struct lxNode pic5[]=   // do a+=b; while(c);
{{_endloop,&pic5[4], NULL},
 {_nam,(struct lxNode*)imgs[1], NULL},
 {_asAdd,&pic5[1], &pic5[3] },
 {_nam,(struct lxNode*)imgs[0], NULL},
 {_whileN,&pic5[2],&pic5[5]},
```

```
 {_nam,(struct lxNode*)imgs[4], NULL}
};
```

Наступний опис показує приклад масиву вузлів для рис. 2.7 з кореневим вузлом `pic6[0]`.

```
struct lxNode pic6[]=   // n=(nD+nU)>>1;
while(nD<n){
 //  if(k==el[n].ky)break; if(k<el[n].ky)nD=n;
 //  else nU=n; n=(nD+nU)>>1;}return n;
{{_nam,(struct lxNode*)imgs[8], NULL},
 {_ass, &pic6[0], &pic6[5]},
 {_nam,(struct lxNode*)imgs[9]},
 {_add, &pic6[2], &pic6[4]},
 {_nam,(struct lxNode*)imgs[10], NULL},
 {_shRgt,&pic6[3], &pic6[6] },
 {_srcn,(struct lxNode*)imgs[2], NULL},
 {_EOS, &pic6[1], &pic6[8]},
 {_whileP,&pic6[10],&pic6[26]},
 {_nam,(struct lxNode*)imgs[9], NULL},
 {_lt, &pic6[9], &pic6[11]},
 {_nam,(struct lxNode*)imgs[8], NULL},
 {_if, &pic6[14], &pic6[19]},
 {_nam,(struct lxNode*)imgs[6]},
 {_eq, &pic6[13], &pic6[17]},
 {_nam,(struct lxNode*)imgs[11], NULL},
 {_ixbr,&pic6[15], &pic6[0] },
 {_fldDt,&pic6[16],&pic6[18] },
 {_nam,(struct lxNode*)imgs[7], NULL},
 {_break, NULL, NULL},
```

```
  {_EOS, &pic6[12],&pic6[24] },
  {_if, &pic6[22], &pic6[23]},
  {_lt, &pic6[13], &pic6[17]},
  {_ass, &pic6[2], &pic6[0]},
  {_else,&pic6[21],&pic6[25]},
  {_ass, &pic6[4], &pic6[0]},
  {_EOS, &pic6[20],&pic6[1]},
  {_return,&pic6[0], NULL},
  {_EOS, &pic6[7], &pic6[27]},
};
```

Алгоритм реконструкції як база семантичної обробки внутрішнього подання

Для побудови систем програмування важливо побудувати програми аналізу та реконфігурації на основі єдиних таблиць, зв'язаних з реалізованою мовою так, щоб для кожної нової мови будувався мінімум схожих таблиць на основі стандарту внутрішнього подання, прийнятого в багатомовній системі програмування. Гнучкість перебудови мов звичайно забезпечується наборами управляючих таблиць. Існує багато шляхів для визначення таких таблиць. Один з них розглянемо в цій роботі детально.

Спочатку визначимо іменування та кодування типів лексем за допомогою перенумерованого типу **`enum`** `tokType`, в якому доцільно визначитись з впорядкуванням кодів, яке буде корисним на більшості етапів розробки та настройки мови, включаючи всі види аналізу реконфігурацій і реконструкцій. Важливо забезпечити подання всіх змістовних синонімів та омонімів в окремих кодах

лексем. В наведеному описі нижче типу використане узагальнене семантичне іменування лексем з прив'язкою до їх позначень в мовах C/C++, Pascal та інших комп'ютерних мовах.

```
#define begOprtr 0x50
       // зміщення початку виконаваних операторів
enum tokType
{_nil, _nam,// зовнішнє подання
 _srcn,_cnst,
            // вхідне і внутрішнє кодування константи
 _if, _then, _else, _elseif,  // if then else elseif
 _case, _switch, _default, _endcase,
                          //case switch defualt endcase
 _break, _return, _whileP, _whileN,
                            // break return while while
 _continue, _repeat, _untilN, _endloop,
                            // continue repeat until do
 _for, _to, _downto, _step,// for to downto step
 _untilP, _loop, _with, _endif,
 _goto,_extern,_var,_const,
 _enum,_struct/*_record*/,_union,_register,//
 _unsigned,_signed,_char,_short,
 _int,_long,_sint64,_uint64,//
 _float,_double,_void,_auto,
 _static,_volatile,_typedef,_sizeof,//
 _real,_array,_set,_file,_object, _string, _label,
 _program,_function,_procedure /*task V*/,
 _macromodule,_primitive,_specify,_table, //V
 _generate,_config,_liblist,_library,   //V
```

```
_incdir,_include,_design,_defaultS,
_instance,_cell,_use,           //V
_automatic,_endmodule,_endfunction,_endtask,//V
_endprimitive,_endspecify,_endtable,
_endgenerate,_endconfig,        //V
_endcaseV,_casex,_casez,_wait,_forever,
_disable,_ifnone,            //V
_pulsestyle_onevent,_pulsestyle_ondetect,
_showcanceled,_noshowcanceled,      //V
_vectored,_scalared,_small,_medium,_large,//V
_genvar,_parameter,_localparam,_defparam,
_specparam,_PATHPULSE$,         //V
_inlineF,_forward,_interrupt,_exportF,_extrn,_asmb,
_input,_output,_inout,          //V|SQL+3
_objectP,_constructor,_desctructor,_property,
_resP,_abstract,                //P++9
_class,_public,_private,_protected,
_virtual,_friend,            //C++16
_new,_delete,_tryC,_catch,_throw/*raise*/,//C++20
_initial,_always,_assign,_deassign,
_force,_release,                //V+26
_reg,_time,_realtime,_event,_buf,_not, //V+32
_andG,_orG,_xorG,_nandG,_norG,_xnorG,  //V+38
_tran,_tranif0,_tranif1,_rtran,
_rtranif0,_rtranif1,            //V+44
_tri,_trior,_triand,_trireg,_tri0,_tri1,  //V+50
_wire,_wand,_wor,_wres,         //V+54
_supply0,_supply1,_highz0,_highz1,     //V+58
```

```
 _strong0,_strong1,_pull0,_pull1,
 _weak0,_weak1,                //V+64
 _pulldown,_pullup,_bufif0,_bufif1,
 _notif0,_notif1,                  //V+70
 _cmos,_rcmos,_nmos,_pmos,_rnmos,_rpmos,  //V+76
// відкриті і закриті дужки
 _fork,_join,// паралельних операторів
 _opbr,_ocbr,// послідовних операторів
 _ctbr,_fcbr,_ixbr,_scbr,// конкатенацій і індексу
 _brkt,_bckt, _tdkt,_tckt,// порядку, функцій і даних
 _eosP, eosS,// паралельні та послідовні
 _EOS=begOprtr, _comma, _cln, _qmrk,// ; , : ?
 _asOr, _asAnd, _asXor, _asAdd,  //|= =& =^ =+
 _asSub, _asMul, _asDiv, _asMod, // -= *= /= %=
 _asShr, _asShl, _ass, _dcr, _inr, // <<= >>= = -- ++
 _lt, _le, _eq, _ne, _ge, _gt,     // < <= == != >= >
 _add, _sub, _mul, _div,_fldDt,_fldPt,// + - * / . ->
 _pwr, _shLfa, _shRga,_eqB,_neB,// ** <<< >>> === !==
 _addU,_subU,_mulU, _andU,    // + - * & унарні
 _norB,_nandB,_nxorB,_xornB, _addr, //~| ~& ~^
 _mod, _orB, _andB, _xorB, // % (div mod) | & ^
 _shLft, _shRgt, _or, _and,   //<< >> || &&
                        // (or and xor shl shr or and)
 _xmrk, _invB,_divI, _in //_not,  _notB
};
```

Наведений тип включає імена кодів, згруповані за використанням в операторах різних мов, починаючи з виконауваних операторів, які є найбільш загальними в усіх комп’ютерних мовах, і

закінчуючи операторами описів і декларацій, які індивідуальними для різних комп'ютерних мов. Крім того, надається можливість включення кодів всіх ключових слів до цієї таблиці, навіть тих які виключаються при побудові внутрішньої форми кодів. Тобто такий тип можна використовувати, як при реконструкції вхідних текстів, так і при лексичному, синтаксичному та семантичному аналізі.

При реконструкції математичних і програмних текстів головною метою є коректне відтворення виразів математичних формул і програм з додержанням правил застосування мінімально потрібних дужок та інших допоміжних роздільників. Для цього треба, по-перше визначити стандартні і керовані шаблони відтворення, по-друге – автоматичне редагування табуляцій, обов'язкових знаків та переведень рядків текстів, по-третє – впорядкування конструкцій за шаблонами синтаксичного передування, а по-четверте блокування формування неявних ключових слів у вхідному тексті.

Базовим варіантом подання шаблону відтворення будемо вважати управляючі рядки видів `"xfx"`, `"xfy"` і `"yfx"` для інфіксних операцій, `"fx"` і `"fy"` для унарних префіксних операцій, `"xf"` і `"yf"` для унарних постфіксних операцій. Цей механізм використовується для відображення виразів та операторів в мові Prolog, яка має вбудовані засоби відтворення структур внутрішнього подання текстів на цій самій мові. В цих записах `y` означає автоматично асоціативний аргумент з операцією з пріоритетом `f`, а `x` означає неасоціативний аргумент з операцією `f`, тобто у випадку операнда `y` не треба брати підлеглу операцію того ж пріоритету в

дужки, а у випадку операнда `x` – треба [6]. Таким чином арифметичні операції і операції переліку можна описувати в процедурних мовах рядком `"yfx"`. Однак в більшості сучасних комп'ютерних мов, насамперед, мов запитів і процедурних мов існують більш різноманітні відношення між операціями і операндами, через що необхідно розширити набір шаблонів. Більше того, через використання літери `f` в багатьох ключових словах, що генеруються за шаблонами та бажаність використання символу табуляції для форматування тексту, краще замінити відтворювані літери `f`, `x` і `y` початковими літерами кодових таблиць коду ASCII або Unicode.

Програмна реалізація перевірки правил мінімального відтворення дужок та порожніх операторів

Основу реалізації правил складають шаблони, які визначають потреби дужок залежно від пріоритетів або передувань операцій та ключових слів операторів. Перевірка правил формування дужок виконується в процедурі **`void`** `prLxTxt(`**`struct`** `lxNode*rt)` і має вигляд продукційних правил, вбудованих в процедуру.

Типи дужок (порядку обробки виразів, операторних дужок та дужок визначення типів та значень даних) визначаються діапазоном значень, в якому знаходяться коди лексем. Таким чином, щоб скоротити програму формування дужок важливо впорядкувати значення типу **`enum`** `tokType`, для зменшення кількості діапазонів, які аналізуються при реконструкції.

Для того щоб відтворювати більш складні конструкції з двома підлеглими вузлами, треба визначити управляючі символи, віднісши решту символів до символів заповнення реконструйованого рядка. Управляючі символи повинні дозволяти відтворення будь-яких конструкцій будь-якої комп'ютерної мови, а також спрощувати еквівалентні перетворення істотно різного синтаксису семантично еквівалентних конструкцій для будь-якої пари комп'ютерних мов. Для реалізації цього потрібні, принаймні, символи управління конструкцій повтореннями синтаксичних елементів та символи управління умовно еквівалентними конструкціями різних мов:

- `\1` – для створення невідтворюваного еквівалента літери `f`,
- `\2(377)` – для встановлення прапорця або лічильника,
- `\3(376)` – для знищення прапорця або лічильника,
- `\4` – для відтворення аргументу без дужок,
- `\5` – для відтворення аргументу з дужками,
- `\6` – для повернення до номера попереднього аргументу, наприклад при відтворенні накопичувальних присвоювань та декларацій мови С в мові Pascal),
- `\7` – для створення нульового прапорця при першому вході,
- `\11=\t` – для керованої табуляції,
- `z` – ознаку пропуску першого аргументу.

Запропоновані вище коди управління можна замінити іншими кодами однозначної системи кодування. Для інших варіантів відтворення будується масив рядків `cpr[]`, в якому розміщуються шаблони реконструкції з літерами управління та іншими літерами,

які відтворюються при реконструкції без змін. Для мови С змістовна частина такого масиву має вигляд.

```
char *cprC[]={"", "", "", "", "\1\5y", "", "", "",
 "\7switch\5\n{\1y\377z\1 y", "", "", "\4;\n\376}",
 "","\1x","\1\5y","x\1\5","","\1x","x\1(!\5)","\1\4",
 "\1\5y", "", "", "", "\1(!\4)y", "", "", "",
 "", "", "", "", "\1\4\4", "", "", "",...
 "\4\1y","\4\1y","\4\1y","\4\1y","\4\1y","\4\1y",
 "\4\1y","\4\1y",   "\4\1y","\4\1y","\4\1y",
 "", "", "", "", "", "", "", "", "", "", "",...
};
```

Для відтворення базових функцій вузла будується масив рядків для заміни операції `f`, який для мови С має вигляд

```
  char *oprtrC[]={"", "", "", "",
  "if", "then", "else", "elseif",
  "case", "switch", "defualt", ""/*endcase*/,
  "break", "return", "while", "while", "continue",
  "do", "while", "do", "for", ";", ";", ";",
  "while", "do", "with", "endif",
  "goto", "extern", "var", "const",
  "enum", "struct", "union", "register",//
  "unsigned", "signed", "char", "short",
  "int","long","int64","int64",//
  "float", "double", "void", "auto",
  "static", "volatile", "typedef", "sizeof",//
  "real", "array", "set", "file",
  "object", "string", "label",
  "int main()","function", "procedure",
```

```
"","","","","","","","","",       //V+8
"","","","","","","","","","","","", //V+19
"","","","","","",             //V+24
"var","","","","","","","",       //V+31
"","","","","","","","","","",    //V+40
"","","","","","","",             //V+46
"inline", "forward", "interrupt", "export",
"extern", "_asm", "","","",  //Verilog|SQL+3
"object", "constructor", "destructor",
"property", "resP", "abstract",      //P+9
"class", "public", "private", "protected",
"virtual", "friend",           //C++15
"new","delete","try","catch","throw",//C++20
"\nfork", "join",
"\n{", "}", "{", "}", "[", "]", "(", ")",
",;\n",".;\n",";\n", ",", ":", "?",
"|=", "&=", "^=", "+=", "-=", "*=", "/=", "%=",
"<<=", ">>=", "=", "--", "++",
"<", "<=", "==", "!=", ">=", ">",
"+", "-", "*", "/",
".", "->", "**", "<<<", ">>>", "===", "!==",
"+", "-", "*", "&", "~|", "~&", "~^", "^~", "&",
"%", "|", "&", "^", "<<", ">>", "||", "&&",
"!","~","/"};
```

Крім того, передбачається використання різних варіантів реконструкції при наступних повтореннях вкладених конструкцій з використанням управляючих кодів спеціальних символів. Так при визначенні повторень фраз операторів мови на прикладі оператора

switch C/C++ при першій реконструкції внутрішній код _case відтворюється як switch + case, а при наступних реконструкціях просто як _case.

Для визначення мінімальної розстановки дужок, а також для аналізу передувань при синтаксичному аналізі використовуються ще два масиви функцій передувань fpr[] і gpr[], які за позиціями (номерами елементів) відповідають масивам oprtrC[] і cprC[], а за числовими пріоритетами – значенням з таблиці 3.1. З міркувань ідентичності відносних пріоритетів однакових операцій в різних мовах ці таблиці можуть бути єдиними для всіх мов, але, на жаль, цих міркувань додержуються не всі творці мов та систем трансляції.

Коди передувань набувають остаточних значень при проектуванні висхідного синтаксичного аналізатора в наступних лабораторних роботах. В базовій програмі замість кодів передувань, використані числа пріоритетів, які змінюються в зворотному порядку відносно кодів передувань з огляду на те, що більш пріоритетні операції передують, тобто виконуються раніше та мають менший порядковий номер передування ніж низькопріоритетні операції. Більше того для більш загальної обробки за таким поданням використовуються дві функції пріоритетів f_{prC} і g_{prC}. Частину елементи їх табличного визначення наведено нижче:

```
char fpr[]=
{0,0x4f,0x4f,0x4f,0x46,0x11,6,0x12,
 0x13,0x4e,0x12,0x1, 0x4e,0x4e,0x4e,0x4e,
```

```
 0x4e,0x4e,0x13,0x4e, 0x4e, 0x9, 0x9, 0x9, 0x4e,...
 0x4e,0x4e,0x4e,0x4e, 0x42, 0x2,
 0x42,2,0x43,3, 0x44,4,0x45,5,
 0x1, 0x1, 0x1, 0x2, 0x2, 0x43,
 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
 0x10, 0x10, 0x10, 0x3C,0x3C,
 0x1A, 0x1A, 0x1C, 0x1C, 0x1A, 0x1A,
 0x20, 0x20, 0x30, 0x30,
 0x3E,0x3E, 0x34,0x1E,0x1E,0x1C, 0x1C,
 0x3C,0x3C,0x3C,0x3C, 0x16,0x18,0x17,0x17, 0x4e,
 0x30,0x16,0x18, 0x17, 0x1E, 0x1E, 0x13, 0x15,
 0x4e,0x4e,0x30},
gpr[]=
{0, 0x4f, 0x4f, 0x4f, 6,0x12,6,0x12,
 0x13,0x4e,0x12,0x1, 0x12,0x1,0x4e,0x4e,
 0x11,0x1,0x1,0x11, 0x11, 0x9, 0x9, 0x9, 0x11,...
0x01,0x01,0x01,0x01, 0x2, 0x2,
 2,2, 3,3, 4,4, 5,5, 1,1, 1, 2,2, 0x43,
 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
 0x10, 0x10, 0x10, 0x3C,0x3C,
 0x1A, 0x1A, 0x1C, 0x1C, 0x1A, 0x1A,
 0x20, 0x20, 0x30, 0x30,
 0x3E,0x3E, 0x34,0x1E,0x1E,0x1C, 0x1C,
 0x3C,0x3C,0x3C,0x3C, 0x16,0x18,0x17,0x17, 0x4e,
 0x30,0x16,0x18,0x17, 0x1E, 0x1E, 0x13, 0x15,
 0x4e, 0x4e, 0x30};
```

Таблиця 2.1

Таблиця передувань операцій в поширених мовах програмування і моделювання

Характеристики операцій			Позначення в поширених мовах / передування		
					Verilog
Назва	**Категорія**	**Типи аргументів**	**Pascal**	**C**	**HDL**
Впорядкування обчислень	Зміна порядку	Будь-які	(...) /0	(...) /0	(...) /0
Функції, процедури, задачі	Звертання до кодів і даних	Будь-які аргументи	(..., ...) /0	(..., ...) /0	(..., ...) /0
Масиви		Цілі індекси	[*ціле*,...]/0	[*ціле*] /0	[*ціле*] /0
Конкатенації		Будь-які аргументи	-	-	{...,...}/0
Доступ до елементів структур		За іменем структури	.	. /0	. /0
		За покажчиком	^.	–> /0	-
Покажчики	Унарні	Будь-які	@ /1	& /1	-
Доступ за покажчиком	Унарні	Вказівники	^ /1	*= /1	-
Інверсія	Унарні	Бітові	not /1	~ /1	~ /1
Інверсія	Унарні	Логічні	not /1	! /1	! /1
Інкремент	Унарні арифметичні	Цілі та вказівники	-	++ /1	-
Декремент		Цілі та вказівники	-	— /1	-
Зміна знаку		Числові	– 1	– /1	– /1
Повтор знаку		Числові	+ 1	+ /1	+ /1
Піднесення до ступеню	Експонентні	Числові	^ /2	-	** /2
Множення	Бінарні мультиплікативні	Числові	* /2	*= /3	* /3
Ділення		Дійсні	/ /2	/= /3	/ /3
Ціле ділення		Цілі	div /2	/= /3	/ /3
Модуль		Цілі	mod /2	%= /3	% /3
Додавання	Бінарні адитивні	Числа та вказівники	+ /3	+= /4	+ /4
Віднімання		Числа та вказівники	– /3	–= /4	– /4
Входження	Бінарні	множинні	in /4	-	-
Зсув вліво	Бінарні бітові	Цілі або бітові	shl /2	<<= /5	<< /5
Зсув вправо		Цілі або бітові	shr /2	>>= /5	>> /5
Зсув вліво	Бінарні арифметичні	Цілі або бітові	-	-	<<< /5
Зсув вправо		Цілі або бітові	-	-	>>> /5
Менше	Відношення	Числа або рядки	< /4	< /6	< /6
Більше		Числа або рядки	> /4	> /6	> /6

Не менше	чисел і рядків	Числа або рядки	<= /4	<= /6	<= /6
Не більше		Числа або рядки	>= /4	>= /6	>= /6
Рівність	Відно-шення	Будь-які однотипні	= /4	== /7	== /7
Нерівність		Будь-які однотипні	<> /4	!= /7	!= /7
Рівність	Відно-шення	Чотиризначні бітові	-	-	=== /7
Нерівність		Чотиризначні бітові	-	-	!== /7
Кон'юнкція	Унарні згортки чотириз-начних бітових даних	Бітові або цілі	-	-	& /8
Інверсія кон'юнкції		Бітові або цілі	-	-	~& /8
Додавання за модулем 2		Бітові або цілі	-	-	^ /9
Рівнозначність		Бітові або цілі	-	-	~^ ^~ /9
Диз'юнкція		Бітові або цілі	-	-	| /10
Інверсія диз'юнкції		Бітові або цілі	-	-	~| /10
Кон'юнкція	Бінарні	Бітові і цілі	and /2	&= /8	& /8
Інверсія кон'юнкції*		Бітові і цілі	-	-	~& /8
Додавання за модулем 2		Бітові, цілі і логічні	xor /3	^= /9	^ /9
Рівнозначність*		Бітові, цілі і логічні	-	-	~^ ^~ /9
Диз'юнкція		Бітові і цілі	or 3	|= 10	| /10
Інверсія диз'юнкції*		Бітові і цілі	-	-	~| /10
Кон'юнкція		Логічні	and 2	&& 11	&& /11
Диз'юнкція		Логічні	or 3	|| 12	|| /12
Умовний вираз	Тернарна	будь-які? числові	-	?: 13	?: /13
Присвоювання	Бінарні	будь-які	:= 14	= /14	= /14
Перелік даних	Список	будь-які значення	-	, /15	, /15
Перелік дій	Список	будь-які врази і дії	-	; /16	; /16

Примітка: знак рівності в індексі характеристик операцій мови С позначає припустимість накопичуючого присвоювання, яке в мові Pascal визначається в формі: *змінна* := *змінна операція вираз*; і для яких в мові С припустима форма: *змінна операція* = *вираз*; зі значенням передування для комбінованої операції 14. Зірочкою * помічені логічно можливі операції.

Програмна реалізація відтворення текстів в довільній комп'ютерній мові

В різних комп'ютерних мовах семантично еквівалентні оператори можуть відтворюватися по-різному. Тому для уніфікованої реконструкції операторів різних мов треба визначити, яким уніфікованим позначенням з перенумерованого типу **enum** tokType відповідають рядки форматів відтворення з масиву, створеного для відповідної мови **char** *oprtrC[] для мови C/C++ або для мови Pascal **char** *oprtrP[], і що робити, коли однозначно еквівалентні засоби в мові відтворення відсутні. Тоді функцію відтворення можна створити у наступному вигляді.

```
void  prLxTxt(struct lxNode*rt) //корінь піддерева
{static int  // лічильники входжень
 struct lxNode* rt0; // робочий вказівник
 char n=0, c, bC=0, opCnt=0;
 if(rt->ndOp<=_cnst)
 {if(rt->ndOp!=_nil)// обробка термінального операнда
    {if(mode==1&&rt->ndOp<begOprtr-8)printf(" ");
     printf("%s",rt->prvNd);
     mode=1;
 }}// вихід з рекурсії
 else
 while((c=cpr[rt->ndOp][n])!=0)// перегляд шаблону
 {n++;    // просування по шаблону
  switch(c)      // аналіз літери шаблона
  {case 7:if(mdCnt==0)mdCnt=-1; else
    while(c!=-1)c=cpr[rt->ndOp][n++];
```

```
    c=0; break;
   case 6: opCnt--;// повернення до першого аргументу
   case -1: break;
   case -2: mdCnt=0; break;
   case 1:
    if(mode!=0&&rt->ndOp>=_if&&rt->ndOp<begOprtr-8)
      printf(" ");
    printf("%s",oprtr[rt->ndOp]);
    if(rt->ndOp>=begOprtr-8)mode=0;
    else mode=1; break;
   case 'x': case 'y': case 5:case 4:
    if(opCnt)rt0=rt->pstNd;// вибір аргументу
    else rt0=rt->prvNd;//перевірка потреби обрамлення
    if(c=='y')bC=fpr[rt->ndOp]>fpr[rt0->ndOp]&&rt0;
    else if(c==5)bC=1; else if(c==4)bC=0;
    else bC=fpr[rt->ndOp]>=fpr[rt0->ndOp]&&rt0;
    if(bC)prOpBr(rt0); // обрамлення дужками
    prLxTxt(rt0);  // рекурсивний виклик відтворення
    if(bC)prClBr(rt0); // обрамлення дужками
   case 'z': prLxTxt(rt->pstNd); break;
   default: printf("%c",c);
  }if(c==-1&&mdCnt!=0)break; //
 }}
}
```

Такі функції реконструкції складають основу програм ***конверторів***, що перетворюють тексти з однієї до іншої мови приблизно одного рівня. Для їх простої реалізації необхідна достатня схожість відповідних типів даних і наявність у форматі внутрішнього подан-

ня всіх операторів і операцій вихідної мови. Якщо вихідною мовою є мова Асемблера, то це практично означає трансляцію на символічний еквівалент машинного коду. Але для використання мови Асемблера треба визначити архітектуру запам'ятовуючої компоненти віртуальної машини, насамперед стеків і акумуляторів.

Структура програмного шаблона виконання роботи

Для спрощення розв'язання задачі слід використовувати прототип проекту spLb3.dsp, побудований на базі проекту spLb1.dsp і зберігається в робочому просторі spLb1.dsw в підпапці spLb3 папки spLb1, яка зберігає шаблони прототипів для всіх лабораторних робіт циклу «Системне програмування-2». До складу проекту консольної прикладної програми, орієнтованої на можливість використання в числі інших бібліотек стандартних функцій та об'єктів MFC (Microsoft Foundation Classes) входять модулі з наступними вхідними файлами заголовків і реалізацій:

- StdAfx.h і StdAfx.cpp – модуль для організації використання об'єктів класів MFC;
- token.h і token.cpp – модуль для опису кодування типу лексеми, структури вузла лексеми і організації використання структур або класів лексем в процесі лексичного і синтаксичного аналізу;
- visgrp.h і visgrp.cpp – модуль для відображення рядків або повних таблиць;
- parsP.cpp, parsC.cpp і parsV.cpp – модулі констант для відтворення лексем мов Pascal, C/C++ та Verilog HDL;

- automat.h і automat.cpp – модуль для визначення та управління автоматом;
- spLb3.cpp – модуль контрольних прикладів для тестування функцій обробки графів.

Розділ визначень і декларацій основної частини програми тестування в модулі spLb3.cpp повинен включати визначення послідовностей вузлів, що утворюють графи лексем. Розділ визначень модуля visgrp.cpp забезпечує визначення мов реконструкції за варіантом завдання. Виконувана частина модуля spLb3.cpp програми повинна включати циклічно повторювані виклики функцій реконструкції для різних варіантів операторів і мов.

Завдання на роботу

Завдання на підготовку до роботи на комп'ютері

1. Визначити варіант завдання для основних задач за таблицею 3.1. Визначити приклади лексем через константи в модулі тестування spLb3.cpp.

2. Ознайомитись з шаблоном програмного проекту spLb3.dsp. Настроїти відповідні дані символьних позначень та структур вузлів графів в програмному проекті на мові С.

3. Підготовити настройку управляючих таблиць мов програмування `oprtrC[]` або `oprtrP[]`, `cprC[]` або `cprP[]`, `fpr[]` і `gpr[]`, а також згідно з варіантами, заданими в табл. 2.1. Також згідно з варіантами визначити масиви лексем **`struct`** `lxNode`

`token[]` і образів **`char`** `*imgs[]` для оператора, заданого у варіанті.

4. Використати масив матриці переходів **`char`** `nxtSts [Se+1][sgE+1]` з файлу automat.cpp шаблону програмного проекту spLb3 для формування станів автомату, які будуть пройдені за варіантом, заданим в таблиці 2.2.

5. Підготувати настройку програми реконструкції для еквівалентних конструкцій на альтернативній мові програмування.

6. Підготувати програмний модуль контрольної задачі, який виконує задані варіанти програм реконструкції і управління станом автомата і дозволяє перевірити коректність виконання програми і її окремих модулів.

Завдання на роботу на комп'ютері

7. Побудувати програмний проект, ввівши програмні модулі у відповідні файли проекту і налагодити синтаксис.

8. Побудувати виконуваний модуль тестової програми і налагодити змістовне виконання програми для перевірки результатів контрольних прикладів.

9. Одержати результати виконання, проаналізувати їх і зробити висновки.

Порядок вибору варіанту:

За останньою цифрою номера залікової книжки або за порядковим номером студента в списку підгрупи з доданим номером групи визначте за табл. 2.2 варіант оброблюваних даних та настройки програм за прикладом.

Таблиця 2.2

Варіанти завдань для виконання реконструкцій і роботи з графами

№ вар.	Вираз, який відтворюється в графі внутрішнього подання	Настроювання графа автомата з послідовними станами	Мова відтворення
1	**if**(c)b=(2*a +c/d)*2*a;	Стани 0..9; 3->7(*dlm*), 5->8(*cfr*)	C
2	**if** c<>0**then**b:=(2*a+c)*2*a;	Стани 1..7; 3->7(*dlm*), 5->2(*cfr*)	Pascal
3	**while**(n--)b+=a[n];	Стани 2..9; 3->8(*dlm*), 5->3(*ltr*)	C
4	**while** n<>0 **do begin** n:=n-1; b:=b+a[n] **end**	Стани 0..9; 3->7(*dlm*), 5->8(*cfr*), 3->2(*ltr*)	Pascal
5	**do** b+=a[--n]; **while**(n);	Стани 1..8; 3->8(*dlm*), 5->5(*ltr*)	C
6	**repeat begin** b:=b+a[n]; n:=n-1 **end until** n=0;	Стани 0..9; 3->7(*dlm*), 5->8(*cfr*), 3->3(*ltr*)	Pascal
7	**if**(c)b=sin(2*a);**else** b=2*a;	Стани 0..8; 3->3(*dlm*), 5->1(*ltr*)	C
8	**if** c **then** b:=sin(2*a) **else** b:=a;	Стани 0..8; 3->7(*dlm*), 5->5(*cfr*), 3->2(*ltr*), 6->8(*cfr*)	Pascal
9	b=c?d:2*a[n];	Стани 1..9; 3->3(*dlm*), 5->9(*ltr*)	C
10	**if** c!=0 **then** b:=d **else** b:=2*a[n];	Стани 1..9; 7->7(*dlm*), 5->3(*cfr*), 3->1(*ltr*)	Pascal
11	**switch**(c){**case** 0: b=2*a[n]; **break; default**: b=d**;}**	Стани 1..8; 3->7(*dlm*), 5->5(*cfr*), 4->8(*ltr*)	C
12	**case** c **begin** 0: b:=2*a[n]; **else**: b:=d **end**	Стани 2..9; 8->2(*dlm*), 5->7(*cfr*), 3->7(*ltr*)	Pascal
13	**for**(b=0;n;n--)b+=a[n];	Стани 0..6; 3->3(*dlm*), 5->2(*ltr*)	C
14	b:=0; **for** n:=n **downto** 0 **do** b:= b+a[n];	Стани 2..7; 3->7(*dlm*), 3->3(*cfr*), 1->4(*ltr*)	Pascal
15	**do{**--n; **if**(b==a[n]) **break; }while**(n);	Стани 1..7; 7->7(*dlm*), 3->3(*cfr*), 1->1(*ltr*)	C
16	**repeat** n:=n-1 **until** (n=0 **or** b!=a[n]);	Стани 2..9; 5->7(*dlm*), 3->3(*cfr*), 9->4(*ltr*)	Pascal
17	**do{**--n; **if**(b==a[n]) **return** n**;}while**(n); **return** 0**;**	Стани 0..7; 3->7(*dlm*), 5->3(*cfr*), 4->4(*ltr*)	C
18	**if** a-c=0 **then** b:=(a-c)*2*a;	Стани 2..9; 2->2(*dlm*), 5->9(*ltr*)	Pascal
19	**do**--n;**while**(n&&b==a[n]);	Стани 2..9; 2->2(*dlm*), 5->9(*ltr*)	C
20	**while**(--n&&b==a[n]);	Стани 1..7; 3->6(*dlm*), 5->5(*cfr*)	C

Питання для самоперевірки

1. Дайте визначення основних типів внутрішнього подання у вигляді графів даних, які визначають процес обробки виразів та операторів.
2. Якими структурами даних або об'єктами визначається вузол внутрішнього подання?
3. Як реалізуються зв'язки з підлеглими вузлами виразів в поданні графа?
4. Як відтворюються вузли роздільників і зв'язки з підлеглими вузлами в поданні графа?
5. Як відтворюються вузли ключових слів і зв'язки з підлеглими вузлами в поданні графа?
6. Як відтворюються вузли термінальних позначень і розміщуються дані в графі внутрішнього подання?
7. Які загальні альтернативи полям вказівникам вузлів графу можна використовувати при побудові внутрішнього подання?
8. Які типи даних використовуються для визначення станів і сигналів управління автоматів?
9. Якими масивами можна програмно визначити скінченний автомат?
10. Якими операторами мов програмування організуються переходи автомату з одного стану до іншого?

2.3 Лабораторна робота 2.3

Тема роботи: Створення та настроювання лексичних аналізаторів на основі автоматної граматики

Мета роботи: Вивчення схеми табличного подання автоматної граматики лексичного аналізу. Використання об'єктів стану графів автоматів для формування лексем у форматі внутрішнього подання вузлів графів розбору.

Короткі теоретичні відомості

Для побудови автомата лексичного аналізу потрібно визначити сигнали його переключення через таблиці класифікаторів літер з кодами, які зручно використовувати в подальшій обробці. Тоді кожний елемент таблиці класифікації повинен визначати код, істотний для аналізу лексем, наведений у формі коду відповідного типу сигналів автомата лексичного аналізу.

```
enum ltrType
{dgt,      //c0 десяткова цифра
 ltrexplt,//c1 літера-ознака експоненти
 ltrhxdgt,//c2 літера-шістнадцяткова цифра
 ltrtpcns,//c3 літера-визначник типу константи
 ltrnmelm,//c4 літери, які припустимі тільки в іменах
 ltrstrlm,//c5 літери для обмеження рядків і констант
 ltrtrnfm,
       //c6 літери початку перекодування літер рядків
 nc,       //c7 некласифіковані літери
```

```
 dldot,   //c8 точка як роздільник та літера констант
 ltrsign, //c9 знак числа або порядку
 dlmaux,  //c10 допоміжні роздільники типа пропусків
 dlmunop, //c11 поодинокі роздільники операцій
 dlmgrop, //c12 елемент початку групового роздільника
 dlmbrlst,//c13 роздільники елементів списків
 dlobrct, //c14 відкриті дужки
 dlcbrct, //c15 закриті дужки
 ltrcode=16//c16 ознака можливості кодування
};
```

Всі можливі стани для одного з можливих варіантів графу автомата лексичного аналізатора внесено до табл. 3.1.

Таблиця 3.1

Основні стани в типовому лексичному аналізаторі комп'ютерних мов і приклади їх кодування

Назва та зміст стану	**Позначення**	**Попередник**	**Ознака переходу до стану**	**Спосіб синтаксичної та семантичної обробки**	**Код стану**
Некласифікований об'єкт	E_u	Будь-який	Некласифікована вхідна літера	Накопичення некласифікованого терму та його нейтралізація	00000
Роздільник	S_0	Будь-який	Будь-який роздільник	Перехід до стану S_0	00001
Знак числа	S_{1g}	S_0	Знак "-" або "+"	Накопичення константи	00010
Ціле число	S_{1c}	S_0	Ознака початку константи або цифра	Накопичення константи	00011
Ціле число	S_{1c}	S_{1c}	Цифра відповідна основі	Накопичення константи	00011
Число з точкою	S_{2c}	$S_0 \vee S_{1c}$	Точка	Перехід до дробової частини	00100
Число з точкою	S_{2c}	S_{2c}	Цифра відповідна основі	Накопичення константи	00100

Ознака порядку	S_{1e}	$S_{2c} \vee S_{1c}$	Літера “e” або “E”	Перехід до підрахунку порядку	00101
Число з порядком	S_{1q}	S_{1e}	Знак “-” або “+”	Накопичення цифр порядку	00110
Число з порядком	S_{1p}	$S_{1e} \vee S_{1q}$	Десяткова цифра	Накопичення цифр порядку	00110
Ознака константи	S_{0c}	S_0	Цифра або літера ознаки	Накопичення елементів	00111
Ім’я	S_{1n}	S_0	Ознака початку імені	Накопичення імені	01000
Ім’я	S_{1n}	S_{1n}	Буква або цифра	Накопичення імені	01000
Символьні константи і рядки	S_{1s}	S_0	Літера початку рядка або символьної константи	Накопичення константи	01001
Символьні константи і рядки	S_{1s}	S_{1s}	Будь-яка літера	Накопичення константи	01001
Кодовані символьні рядки	S_{1t}	S_{1s}	Літера-ознака початку перекодування	Накопичення константи	01010
Кодовані символьні рядки	S_{1t}	S_{1t}	Будь-яка літера, включаючи ознаку закінчення рядка	Накопичення константи з перекодуванням	01010
Закінчена символьна константа	S_{2s}	$S_{1t} \vee S_{1s}$	Будь-яка літера, включаючи ознаку закінчення рядка	Накопичення константи з перекодуванням	01011
Місце основи	S_{0c}	S_0	Цифра 0	Фіксація місця основи	01100
Ознака основи	S_{0p}	S_{0c}	Літера ознаки або цифра	Фіксація основи	01101
Місце основи	S_{0c}	$S_0 \vee S_{1c}$	Ознака місця типу числа	Фіксація місця основи	01100
Ознака основи	S_{0p}	S_{0c} S_{0p}	Літера ознаки Додаткові пропуски	Фіксація основи	01101
Тіло числа	S_{3c}	S_{0p}	Цифра відповідна основі	Накопичення константи	01110

Проміжний роздільник групи	S_2	S_1	Початковий роздільник групи	Класифікація за таблицею	01101
Груповий роздільник	S_3	S_2	Останній роздільник з групи	Перевірка за правилом припустимості	01110
Аналіз роздільника	S_1	$S_0 \vee S_1$	Роздільник – не пропуск	Обробка роздільників і груп	01111
Помилки					
Неправильна константа	E_c	S_{1c}	Помилкова або некласифікована вхідна літера (нецифра або неознака)	Накопичення помилкової або некласифікованої константи для наступної корекції	10011
Неправильна константа з точкою або порядком	E_p E_q	S_{2c} S_q	Помилкова або некласифікована вхідна літера (нецифра або неознака)	Накопичення помилкової або некласифікованої константи для наступної корекції	10100 10010
Неправильне ім'я	E_n	S_{1n}	Помилкова або некласифікована вхідна літера	Накопичення помилкового або некласифікованого імені	10101
Неправильний терм	$E_x = E_c \vee E_n$	E_x	Нероздільник	Накопичення помилкового терму	10000
Неприпустиме сполучення операцій	E_o	S_{1o}	Помилковий роздільник в групі	Накопичення помилкового або некласифікованого роздільника та його нейтралізація	10001

Сама таблиця займає 256 байтів для перекодування кожного символу коду ASSCI з заданою таблицею кодування і має вигляд

```
enum ltrType ltCls[256] =
                        // Початок таблиці класифікаторів
{nc,nc,nc,nc,nc,nc,nc,nc,
 nc,dlmaux,dlmaux,nc,nc,dlmaux,nc,nc,
 nc,nc,nc,nc,nc,nc,nc,nc, nc,nc,nc,nc,nc,nc,nc,nc,
```

```
 dlmaux,dlmgrop,ltrstrlm,dlmunop,
 ltrnmelm,dlmunop,dlmgrop,ltrstrlm,
 dlobrct,dlcbrct,dlmunop,dlmgrop,
 dlmaux,dlmunop,dlmgrop,ltrstrlm,...
};
```

Для створення автомата, який визначає роботу лексичного аналізатора, необхідно визначити стан поточної лексеми, як стан автомата, а визначники чергових знаків розглядати як сигнали для зміни стану автомату. При побудові графа автомату слід враховувати можливість двох режимів: загального режиму виразів та спеціального режиму числових констант, які звичайно визначаються у синтаксисі будь-якої мови.

Нумерація кодів класів літер **enum** ltrType, визначена вище, придатна для реалізації та будь-яких видів обробки всіх мов програмування, запитів і моделювання. При реалізації різних видів аналізу природних мов цей тип повинен бути розширений додатковими класифікаторами літер.

Для програмної реалізації скінченного автомату лексичного аналізу треба визначити коди стану автомата, які відображують всі можливі стани закінчених та незакінчених лексем. Коди стану частіше за все визначаються перенумерованим типом з іменованими значеннями всіх можливих заключних і проміжних станів:

```
enum autStat
{Eu,   // Eu – Некласифікований об’єкт
 S0,   // S0 – Роздільник
 S1g,  // S1g – Знак числової константи
```

```
 S1c,  // S1c – Ціле число
 S2c,  // S2c – Число з точкою
 S1e,  // S1e - Літера "e" або "E"
 S1q,  // S1q - Знак "-" або "+"
 S1p,  // S1p - Десяткові цифри порядку
 S1n,  // S1n – Елементи імені
 S1s,  // S1s - Літери рядка або символьної константи
 S1t,  // S1t – Елементи констант, які перетворюються
 S2s,  // S2s – Ознака закінчення константи
 S2, // S2 – Початковий елемент групового роздільника
 S3,   // S3 – Наступний елемент групового роздільника
 Scr,  // Scr– Коментар-рядок
 Scl,  // Scl– Обмежений коментар
 Ec,   // Ec – Неправильна константа
 Ep,   // Ep – Неправильна константа з точкою
 Eq,   // Eq – Неправильна константа з порядком
 En,   // En – Неправильне ім'я
 Eo // Eo – Неприпустиме сполучення операцій
};
```

Так звана матриця переходів автомата визначається двомірним масивом типу **enum** autStat, перший індекс якого визначає ціле число, яке відповідає попередньому стану, а другий індекс – число, яке відповідає сигналу або класу сигналу для переведення в наступний стан.

```
enum autStat nxtSts[Eo+1][ltrcode+1] =
{{Eu,Eu,Eu,Eu,Eu,Eu,Eu,Eu, S0,S0,S0},   //для Eu
 {S1c,S1n,S1n,S1n,S1n,S1n,S1n,Eu,S2c,S0,S0},// S0
 {S1c,Ec,Ec,Ec,Ec,Ec,Ec,Eu, S2c,S0,S0}, //  S1g
```

```
 {S1c,S1e,Ec,Ec,Ec,Ec,Ec,Eu, S2c,S0,S0},   // S1c
 {S2c,S1e,Ec,Ec,Ec,Ec,Ec,Eu, Ec,S0,S0}, // S2c
 {S1p,Ec,Ec,Ec,Ec,Ec,Ec,Eu, Eu,S1q,S0}, // S1e
 {S1p,Ec,Ec,Ec,Ec,Ec,Ec,Eu, Eu,Eu,S0},  // S1q
 {S1p,Ec,Ec,Ec,Ec,Ec,Ec,Eu, Eu,S0,S0},  // S1p
 {S1n,S1n,S1n,S1n,S1n,S1n,S1n,En,S0,S0,S0},// S1n
 {S1s,S1s,S1s,S1s,S1s,S2s,S1t,Ec,S1s,S1s,S1s},//S1s
 {S1t,S1t,S1s,S1s,S1s,S2s,S1s,Ec,S1s,S1s,S1s},//S1t
 {S1s,Ec,S1s,S1s,S1s,S2s,S1t,Ec,S0,S0,S0},//  S2s
 {Ec,Ec,Ec,Ec,Ec,Ec,Ec,Ec, S0,S0,S0},   // Ec
 {Ep,Ep,Ep,Ep,Ep,Ep,Ec,Ec, S0,S0,S0},   // Ep
 {En,En,Eq,Eq,Eq,Eq,Eq,Ec, S0,S0,S0},...   // En
}
```

Функція формування лексем з вхідного тексту `lxAnlzr` накопичує символи лексеми з вхідного файлу в сегмент образів імен або констант (навіть краще до сегментів констант у внутрішньому поданні) аж до одержання чергового роздільника і розміщує вказівник на нього в елементі термінального вузла. Для цього вводячи з файлів послідовність літер в масив сегмента, заповнимо поля первинного заповнення вузла лексеми функцією `lxInit`.

Для аналізу та перетворення імен стандартних функцій, зарезервованих та ключових слів і групових роздільників, в шаблоні роботи побудована таблиця з елементами, символьні взірці яких взяті з масивів взірців для регенерації вхідних текстів з лабораторної роботи 2.2. Самі ж елементи **`struct`** `recrdKWD tablKWDC` таблиці для мови C/C++ впорядковані за зростанням значень клю-

чів і включають крім адреси ключа, внутрішній код та версію для використання більш загальних алгоритмів обробки.

```
struct recrdKWD tablKWDC[67]=
{{oprtrC[_ne],_ne,1},{oprtrC[_asMod],_asMod,1},
{oprtrC[_and],_and,1},{oprtrC[_asAnd],_asAnd,1},...
```

Для однолітерних роздільників, які відображують змістовні операції, побудована таблиця перекодування **char** dlCdsC на внутрішню форму через метод пошуку за прямою адресою.

До цих методів необхідно додати методи пошуку для перетворень в таблицях ключових слів, роздільників, констант та імен користувача, розроблені на базі функцій, налагоджених в лабораторних роботах 1.1 і 1.2. Ці методи дещо модифіковані відповідно змінам в структурах елементів таблиць.

Щоб уникнути повторень в елементах таблиці імен і ефективно використовувати структури термінальних вузлів як рядки таблиць імен і констант, доцільно побудувати індекс, який зможе організувати доступ до елементів таблиці, впорядкованих за ключами у вигляді імен разом з номерами модулів визначення вузлів або символічних значень констант.

Наведемо модифіковану структуру вузла індексу з лабораторної роботи 2.1, яка дозволятиме будувати структури двійкових дерев з невеликою надлишковістю.

```
struct indStrUS// структура вузла індексу у вигляді
                // двійкового дерева
{struct lxNode* pKyStr;//вказівник на вузол
 struct indStrUS* pLtPtr;//вказівник вліво
```

```
 struct indStrUS* pRtPtr;//вказівник вправо
 int dif; // робоче значення відхилення
};
```

Основні операції для роботи з таблицями, взяті з лабораторної роботи 2.1 і трохи промодифіковані. Вони включають функції вибірки `selBTr` і включення нового елемента `insBTr`.

```
// вибірка через пошук за двійковим деревом
struct indStrUS*selBTr (struct lxNode*kArg,
                        struct indStrUS*rtTb)
{int df;
 while(df=cmpTrm(kArg,rtTb->pKyStr))
if(df>0)       //просування вправо
      {if(rtTb->pRtPtr)rtTb=rtTb->pRtPtr;
       else break;}
else        //просування вліво
      {if(rtTb->pLtPtr)rtTb=rtTb->pLtPtr;
  else break;}
 rtTb->dif=df;
 return rtTb;
}
// включення через пошук за двійковим деревом
struct indStrUS*insBTr (struct lxNode*pElm,
                        struct indStrUS*rtTb)
{struct indStrUS*pInsNod;// вузол доданого елемента
 if(rtTb->pKyStr==NULL)
{rtTb->pKyStr=pElm;
 return rtTb;
}
```

```
   else{pInsNod=selBTr(pElm,rtTb);
     if(pInsNod->dif)
    {ndxNds[++nNdxNds]=nilNds;
     if(pInsNod->dif<0)
            pInsNod=pInsNod->pLtPtr=ndxNds+nNdxNds;
     else  pInsNod=pInsNod->pRtPtr=ndxNds+nNdxNds;
     ndxNds[nNdxNds].pKyStr=pElm;
   }}
    return pInsNod;
   }
```

З використанням перелічених основних операцій над таблицями та індексами термінальних елементів побудовано процедуру лексичного аналізу `lxAnlzr`, яка за один виклик формує вузол, який відповідає черговій лексемі і чий порядковий номер повертається цією функцією.

```
// функція лексичного аналізу чергової лексеми
int lxAnlzr(void)
{static int LexNmb = 0;
 static enum autStat s=S0, sP;
       // поточний та попередній стан лексеми
 char l;      // чергова літера
 enum ltrType c; // клас чергової літери
 lxInit();// заповнення позиції в тексті і таблицях
 do{sP=s;     // запам'ятовування стану
   l=ReadLtr();// читання літери
   c=ltCls[l];  // визначення класу літери
      s=nxtSts[s][c<dlmaux?c:dlmaux];// стан лексеми
   }while(s!=S0);   // перевірка кінця лексеми
```

```
 switch (sP)
 {case S1n:// пошук ключових слів та імен
    frmNam(sP, x);
       break;
  default:   // не дійшли до класифікованих помилок
  case Eu: Ec: Ep: Eq: En: Eo:// обробка помилок
       eNeut(lxNmb);     // фіксація помилки
  case S1c: S2c: S1p: S2s: // формування констант
       frmCns(sP, x);
       break;
  case S0:
       dGroup(lxNmb);// аналіз групових роздільників
 } return lxNmb++;
}
```

Використаємо структуру вузла лексеми **struct** lxNode, визначену в лабораторній роботі 1.3 для побудови таблиць термінальних позначень. Через те, що ці вузли є окремими елементами внутрішнього подання зв'язати їх в єдину таблицю краще за все можна за допомогою індексної надбудови.

```
  struct lxNode// вузол дерева, САГ або УСГ
  {int ndOp;  // код операції або типу лексеми
   unsigned stkLng; // номер модуля для терміналів
   struct lxNode* prvNd, pstNd;// до підлеглих
   int dataType;// код типу даних, які повертаються
   unsigned resLng; //довжина результату
   int x, y, f;//координати розміщення у файлі
   struct lxNode* prnNd;}; //до батьківського вузла
```

Фактично лексичний аналізатор використовує з цієї структури лише поля `ndOp` – для кодування типу лексеми; `stkLng` – для номера блоку визначення лексеми; `prvNd` – для зв'язку з елементами сегментів образів вхідного тексту, що фактично мають тип **char**`*`, та `x, y, f` – які являють собою координати розміщення лексеми у вхідному файлі. Інші поля використовуються на подальших етапах компіляції.

Для відтворення результатів лексичного аналізу використовується спеціальна процедура відображення.

Структура програмного шаблона виконання роботи

Для спрощення розв'язання задачі слід використовувати прототип проекту spLb4.dsp, побудований на базі проектів spLb1.dsp та spLb3.dsp, і зберігається в робочому просторі spLb1.dsw в підпапці spLb4 папки spLb1, яка зберігає шаблони прототипів для всіх лабораторних робіт курсу «Системне програмування-2». До складу проекту консольної прикладної програми, орієнтованої на можливість використання в числі інших бібліотек стандартних функцій та об'єктів MFC (Microsoft Foundation Classes) входять модулі з наступними вхідними файлами заголовків і реалізацій:

- StdAfx.h і StdAfx.cpp – модуль для організації використання об'єктів класів MFC;
- tables.h і tables.cpp – модуль для опису і організації використання структур або класів таблиць;

- token.h і token.cpp – модуль для опису кодування типу лексеми, вузла лексеми і організації використання структур або класів лексем в процесі лексичного і синтаксичного аналізу;
- visgrp.h і visgrp.cpp – модуль для відображення рядків графів або повних таблиць;
- parsP.cpp, parsC.cpp і parsV.cpp – модулі констант для лексичного аналізу та відтворення лексем мов Pascal, C/C++ та Verilog HDL, які використовують однойменні модулі проекту spLb3.dsp;
- spLb4.cpp – модуль контрольних прикладів для тестування функцій обробки графів.

Розділ визначень і декларацій основної частини програми тестування в модулі spLb4.cpp повинен включати визначення таблиць лексем, ініціалізацію їх елементів, спеціальні змінні, що зберігають довжину (тобто кількість елементів) таблиці. Виконувана частина модуля spLb4.cpp програми повинна включати циклічно повторювані виклики функцій для перевірки програми лексичного аналізу та відображення різних варіантів вхідних даних.

Завдання на роботу

Завдання на підготовку до роботи на комп'ютері

1. Визначити варіант завдання для основних задач за таблицею 4.2. Визначити приклади лексем через константи в модулі тестування.

2. Ознайомитись з шаблоном програмного проекту spLb4.dsp. Настроїти відповідні дані в програмному проекті на мові С.

3. Підготувати настройки таблиць лексичного аналізу **enum** `ltrType ltCls[256]` та **enum** `autStat nxtSts[Eo+1] [ltrcode+1]` для можливості реалізації оператора за заданим варіантом.

4. Використати структуру елементу **struct** `lxNode` з файлу index.h шаблону програмного проекту spLb4 для побудови елементу індексу таблиць лексем і визначити наступні підпрограми з методом впорядкування, заданим в таблиці 3.2.

- для вставки до таблиці з корекцією індексу;
- для вибірки з таблиці за індексом;
- для корекції таблиці з індексом.

5. Підготувати програмний модуль контрольної задачі, який виконує програму лексичного аналізу за варіантами вхідних текстів, заданими в таблиці 4.3, і дозволяє перевірити коректність виконання програм.

Завдання на роботу на комп'ютері

6. Побудувати програмний проект, ввівши програмні модулі у відповідні файли проекту, і налагодити синтаксис.

7. Побудувати виконуваний модуль тестової програми і налагодити змістовне виконання програми.

8. Одержати результати виконання, проаналізувати їх і зробити висновки.

Порядок вибору варіанту:

За останньою цифрою номера залікової книжки або за порядковим номером студента в списку підгрупи з доданим номером групи визначте варіант завдання для задач за табл. 3.3.

Таблиця 3.3

Варіанти завдань для виконання аналізу вхідного файлу з одержанням масиву лексем

№ вар.	Вираз, який відтворюється в графі внутрішнього подання	Мова відтворення
1	**int** main (**void**) {**float** a, c, d;…}	C
2	**if** c<>0**then**b:=(2*a+c)*2*a;	Pascal
3	**int** main (**void**) {**float** b, a[13]; **int** n;…}	C
4	**while** n<>0 **do begin** n:=n-1; b:=b+a[n] **end**	Pascal
5	**int** main (**void**) {**float** b, a[12]; **int** n; …}	C
6	**repeat begin** b:=b+a[n]; n:=n-1 **end until** n=0;	Pascal
7	**if**(c)b=sin(2*a);**else** b=2*a;	C
8	**if** c **then** b:=sin(2*a) **else** b:=a;	Pascal
9	b=c?d:2*a[n];	C
10	**if** c!=0 **then** b:=d **else** b:=2*a[n];	Pascal
11	**switch**(c){**case** 0: b=2*a[n]; **break; default**: b=d**;**}	C
12	**case** c **begin** 0: b:=2*a[n]; **else**: b:=d **end**	Pascal
13	**for**(b=0;n;n--)b+=a[n];	C
14	b:=0; **for** n:=n **downto** 0 **do** b:= b+a[n];	Pascal
15	**do**{--n; **if**(b==a[n]) **break;** }**while**(n);	C
16	**repeat** n:=n-1 **until** (n=0 **or** b!=a[n]);	Pascal
17	**do**{--n; **if**(b==a[n]) **return** n**;**}**while**(n); **return** 0**;**	C
18	**if** a-c=0 **then** b:=(a-c)*2*a;	Pascal
19	**do**--n; **while**(n&&b==a[n]);	C
20	**while**(--n&&b==a[n]);	C
21	**while**(n>0 **and** b=a[n]) **do** n:=n-1;	Pascal
22	**if**(c<a)b=sin(2*a);**else** b=2*a;	C

Питання для самоперевірки

1. Дайте визначення основних типів лексем для мов програмування.
2. Які стани автоматів лексичного аналізу необхідно визначити при побудові таблиць лексичного аналізатора?
3. Які типи даних використовуються для визначення автоматів?
4. Які об’єкти даних необхідно визначити для аналізу лексем за автоматною граматикою?
5. Яким чином можна використовувати поля структур для організації таблиць імен та констант?
6. Які таблиці треба побудувати для заповнення полів лексем термінальних позначень?
7. Які таблиці треба побудувати для заповнення полів лексем вузлів роздільників та ключових слів?
8. Як можна використовувати мову регулярних виразів для лексичного аналізу вхідних текстів?
9. Як доцільно кодувати типи лексем у їх внутрішньому поданні?
10. Навіщо у внутрішньому поданні лексем зберігаються координати вхідного тексту визначення лексем?

2.4 Лабораторна робота 2.4

Тема роботи: Створення і настроювання висхідних синтаксичних аналізаторів

Мета роботи: Одержання навичок настроювання таблиць висхідних синтаксичних аналізаторів та програм побудови графів та дерев розбору на етапі синтаксичного аналізу з запам'ятовуванням покажчиків на вхідні образи та внутрішні коди. Вивчення програм формування повідомлень про помилки та побудови графа підлеглості операцій та операторів в процесі висхідного синтаксичного аналізу.

Короткі теоретичні відомості

На основі попередньо визначеної і заповненої базової структури елемента внутрішнього подання лексеми (лабораторні роботи 2.3 та 2.4) треба організувати поступове доповнення полів покажчиками на всіх етапах обробки. Для того щоб надати можливість використання даних лексичного аналізу на наступних етапах необхідно розмістити результати лексичного аналізу в масиві лексем з частково заповненими полями.

Визначення значень і функцій передувань для ключових слів та операцій мови

Для побудови алгоритму синтаксичного аналізу простого операторного (або точніше операційного) передування будується стековий розпізнавач з послідовністю дій, що контролюють передування або пріоритети лексем, які розглядаються при синтаксич-

ному аналізі як термінальні символічні позначення. Як управляючі таблиці можуть бути використані вектори передувань або пріоритетів та матриці передування з правилами, що використовуються при перевірці передування операцій. На практиці граматика будь-якої мови програмування процедурного та об'єктно-орієнтованого типа породжує ланцюжки символічних позначень, які складаються з термінальних вузлів, що класифікуються як об'єкти, дії, операції (оператори) або операнди. Важливо відзначити, що термінальні операнди ніяким чином не впливають на послідовність виконання операцій, оскільки передують операціям. Тому при висхідному розборі можна приймати до уваги тільки самі оператори (операції), що дозволяє одержати спрощені варіанти алгоритмів.

Загальний алгоритм висхідного аналізу при синтаксичному розборі

Розглянемо застосування цього підходу для аналізу математичних виразів. Будемо використовувати структурований стек з елементами, що об'єднує в єдину структуру внутрішні подання чергової операції та операнда (операнд може бути порожнім або фіктивним). Наведемо на мові Асемблера найпростіший варіант структури, що припускає зберігання операндів тільки у формі подвійних слів цілого формату. Цей варіант структури фактично відповідає структурі, представленій на мові С в описах лабораторних робіт 2.2 і 2.3.

```
lxNode   STRUC; структура термінальних позначень
ndOp      dd  ?  ; тип вузла в коді enum tokType
prvNd  dd  ?  ; покажчик вузла-попередника
pstNd  dd  ?  ; покажчик вузла-наступника
```

```
dataType  dd  ?  ; тип результату
resLength dd  ?  ; довжина результату
x   dd  ?  ; номер позиції в рядку у файлі опису мови
y   dd  ?  ; номер рядка у файлі опису мови
f   dd  ?  ; номер файлу опису мови
prnNd  dd  ?  ; номер батьківського вузла
stkLength dd  ?   ; номер модуля визначення
lxNode   ENDS
```

В полі оператора (операції) розміщуються коди бінарних та унарних операцій, круглих дужок та роздільників, разом з характеристикою передування (пріоритетом), в тому числі і фіктивний оператор, яким починається та закінчується кожний програмний модуль. В полях операндів фіксується інформація про імена, ідентифікатори та значення констант в послідовності їх одержання. Порядок обробки визначається функцією передування або пріоритетом *p*. При визначенні кроку роботи загального алгоритму розбору операції будемо позначати таким чином: S_i – верхня операція в стеку, S_j – вхідна, тобто тільки-но введена операція.

Для визначення значень передувань або пріоритетів доцільно визначити перенумерований тип для кодування всіх варіантів значень. В шаблонному проекті цієї лабораторної роботи spLb5.dsp побудовано наступний перенумерований тип на мові С для кодів типів лексем. В цьому типі визначені іменовані значення пріоритетів для ключових слів так, щоб додержуватись відповідності зі значеннями передувань математичних операцій, показаних у табл. 3.1. Для дужок та ключових слів операторів значення функцій переду-

вань визначені так, щоб початкові елементи дужок і операторів мали великі пріоритети попередньої обробки $f(S_i)$ і маленькі пріоритети заключної оброки $g(S_i)$. В той самий час для проміжних та заключних елементів операторів та дужок пріоритети $f(S_i)$ і $g(S_i)$ повинні бути невеликими і можуть бути однаковими.

```
enum tokPrec // передування основних типів лексем
{nil,  // кінець файлу
 pend, // кінець модуля - '.' або 'endmodule'
 pclf,//закрита операторна - '}' або 'end' чи 'join'
 peos,// кінець оператора - ';'
 pekw,// початкове слово типу - 'enum', 'struct'
 pclb,// закритої дужки - ')'
 pmkw,// проміжного ключового слова - 'else', ...
 pskw,// початкового ключового слова - 'if', 'for'
 popf,//відкрита операторна - { або 'begin' чи 'fork'
 pcld,// закрита дужка даних - '}'
 pbkw,// початкового ключового слова конструкції –
                                      //'int', 'float'
 pdol,// роздільника списку –
                  //',' для параметрів ^ одну позицію
 pcls,// закритої дужки - ']'
 pass,// присвоювань - '=', '+=', ...
 psmc,// двокрапки - ':'
 pcnd,// умовної операції - '?'
 pacf,// доступу до полів - '.' та '->'
 porl,// логічної диз'юнкції - '||'
 panl,// логічної кон'юнкції - '&&'
 porw,// побітової диз'юнкції - '|' та '~|'
```

```
 pxrw,// додавання за модулем 2 - '^', '^~' та '~^'
 panw,// побітової кон'юнкції - '&' та '~&'
 pequ,//відношень рівності-нерівності –
                              //'==', '!=', '===','!=='
 prel,// відношень більше-менше '<', '<=', '>=' і '>'
 pshf,// зсувів - '<<', '>>', '<<<' та '>>>'
 padd,// додавання-віднімання - '+' та '-'
 pmul,// множення-ділення - '*', '/' та '%'
 ppwr,// піднесення до ступеню
 popd,// відкритої дужки даних '{'
 pops,// відкритої дужки індексу '['
 popb,// відкритої дужки функції або порядку дій '('
 puno,// унарних операцій
 ptrm,// терму: константи та імені змінної, функції,…
 pprg // кінцевого символу програми-модуля –
                              // 'program' або 'module'
};
```

В полях операндів фіксується інформація про імена, ідентифікатори та значення констант в послідовності їх одержання. Порядок обробки визначається значенням передування або пріоритету *p*.

1. Якщо $p(S_i) = p(S_j)$, помістити S_j до стеку операцій та перейти до обробки наступного вхідного символічного позначення.

2. Якщо $p(S_i) > p(S_j)$, викликати процедуру семантичної обробки, яка визначається позначенням S_i. Ця процедура виконує семантичну обробку та виключить зі стека операцію S_i, і, можливо, інші символи, а також змінить в стеку операнди, пов’язані з цими позначеннями, на результат виконання оператора S_i.

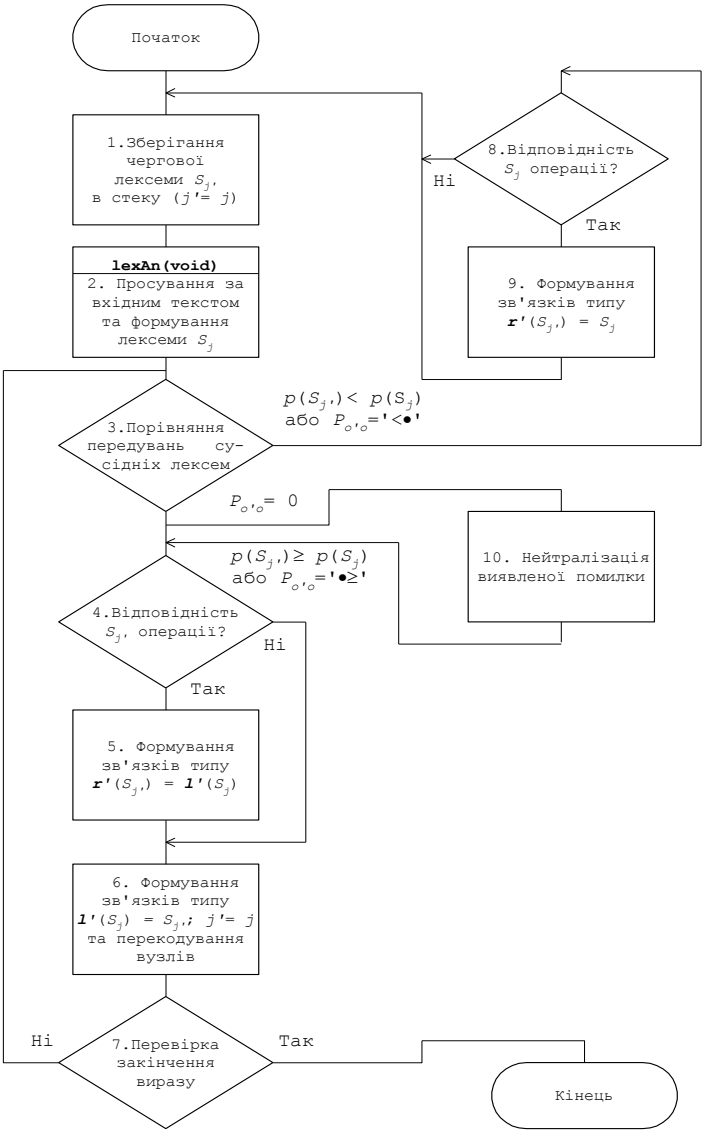


Рис. 5.1. Схема потоку управління для програми висхідного розбору

Значення характеристик передування (пріоритетів) $p(S_i)$ зберігаються в масивах, у відповідності з номерами кодів S_i, наприклад, в масиві `opPrFC` в файлі parpC.c для мови C, `opPrFP` в файлі parpP.c – для мови Pascal та `opPrFV` в файлі parpV.c – для мови Verilog.

В найбільш загальному випадку кількість рівнів пріоритетів не перевищує двох десятків, але в деяких системах передування передбачається значно більша кількість рівнів пріоритетів для потреб користувача. Версію, що використовує пріоритетну функцію передування, називають ще стековим алгоритмом висхідного розбору. Такий алгоритм в узагальненій структурованій формі може бути поданий наступною процедурою на мові Асемблера:

```
SyntUp PROC
      PUSH  BP   ; Збереження старого BP
      MOV   BP,SP   ; Визначення бази зони аргументів
; ES:DI використовується для перегляду ланцюжка
      XOR   AX,AX   ; Формування фіктивного операнда
      MOV   DX,AX   ; Формування фіктивної операції
LinsSt:PUSH DX   ; Запис операнда до стеку
      PUSH  AX    ; Запис фіктивної операції до стеку
      CALL  LexAn; Повертає в AL:DX операцію: операнд
      MOV   BP,SP
LcmpSt:CMP  AL,Prty[BP]  ; Порівняння пріоритетів
      JA    LinsSt
      CALL  SemProc;Повертає в DX результат і AL
      SUB   BP,4
      CMP   WORD PTR[BP],0;Контроль закінчення виразу
```

```
      MOV   SP,BP
      JNE   LcmpSt
      POP   BP
      RET
SyntUp ENDP
```

Тут ми звертаємося до процедури `LexAn`, що виконує аналіз пари ***операнд операція*** і повертає результати лексичного аналізу в парі DX:AX, і до процедури семантичної обробки, що повертає результат обчислень в AX.

Недоліком наведеної процедури `SyntUp` є неможливість природної обробки дужок та інших складених конструкцій внутрішніми засобами програми. Перший варіант розв'язання цієї проблеми полягає у віднесенні визначення пар функцій передування на етап семантичної обробки операції з високим значенням функції передування f так, щоб її значення в стеку дорівнювало g. Другий варіант полягає в узагальненні структури елемента стека включенням значення додаткової функції передування g, а також включити в програму додаткове порівняння поточного пріоритету з цим значенням передування після семантичної обробки.

Результатом аналізу буде так звана продукція синтаксичного аналізу, тобто доведене кінцеве позначення або метапозначення у будь-якій прийнятій для компілюючої системи формі. У зв'язку із зростанням обсягів оперативної пам'яті зміщується критерій ефективного використання пам'яті, що дозволяє змінити евристики раціонального програмування на зберігання основної частини програми у оперативній або віртуальній пам'яті. Це дозволяє використ-

товувати розширені подання продукцій, що можуть зберігати і результати семантичної обробки кожної конструкції у формі внутрішнього подання.

Але якщо за основу брати подання лексем у форматі `lexTerm`, то програму можна подати так, щоб уникнути повторних перетворень. Зв'язки між вузлами лексем будуть формуватись за алгоритмом, що відповідає процедурі `SyntUp`. Складемо набір правил для пар «операнд»-«операція», за якими встановлюються зв'язки між вузлами операцій та операндів дещо зміняться так, щоб відповідати поняттям синтаксичного дерева.

Тоді в стеку доцільно зберігати номери або вказівники на відкладені структури в форматі `LexTerm/dagNode`. Такий підхід в принципі зручний для реалізації виразів в мовах програмування з контекстно незалежною семантикою, тобто коли зміст окремих елементів виразів не залежить від їх взаємного розміщення. Більшість мов програмування спирається на аксіоми визначення способу перетворення типів для пар операндів. Як правило, ці аксіоми відповідають принципу мінімізації втрат інформації і тому операнди приводяться до більш загального типу.

Аналіз виконання програм висхідного граматичного аналізу

Для відтворення значень і функцій передувань в програмах висхідного розбору використовується спеціальні масиви, в яких встановлюється відповідність значень за номерами типів лексем.

Для кожного типу лексеми значення передування повинно бути занесене до масиву.

Використання значень і функцій передування

Значення характеристик передування (пріоритетів) $f(S_i) = p(S_i)$ і $g(S_i)$ зберігаються в масивах, у відповідності з кодами операцій S_i, наприклад, в масивах `opPrFC` та `opPrGC` в файлі parpC.c для мови C, `opPrFP` та `opPrGP` в файлі parpP.c – для мови Pascal та `opPrFV` та `opPrGV` в файлі parpV.c – для мови моделювання обладнання Verilog. Початок цих таблиць показано нижче:

```
enum tokPrec opPrFC[290]=
{nil,ptrm,//_nil, _nam,//0 зовнішнє подання
 ptrm,ptrm,//_srcn,_cnst,
           //2 вхідне і внутрішнє кодування константи
 pskw,pmkw,pmkw,nil,    //4 if then else elseif
 nil,pbkw,nil,nil,  //8 case switch defualt endcase
 pbkw,pskw,pskw,nil,    //12 break return while do
 pbkw,pbkw,nil,nil,     //16 continue repeat until
 pskw,pmkw,pmkw,pmkw,  // for to downto step
 nil,nil,pbkw,nil,  //_untilP, _loop, _with, _endif,
 nil,nil,nil,nil,nil,nil,nil,nil,
                 //_goto,_extern,//_var,_const,_enum,
               //_struct/*_record*/,_union,_register,
 nil,nil,nil,nil,nil,nil,nil,nil,//_unsigned,_signed,
           //_char,_short,_int,_long,_sint64,_uint64,
 nil,nil,nil,nil, nil,nil,nil,nil,  //_float,_double,
    //_void,_auto,_static,_volatile,_typedef,_sizeof,
 nil,...
};
enum tokPrec opPrGC[290]=
{nil,ptrm,    //_nil, _nam,// зовнішнє подання імені
 ptrm,ptrm,   //_srcn,  _cnst,
             // вхідне і внутрішнє кодування константи
```

```
 peos,peos,peos,nil,    //4 if then else elseif
 nil,nil,nil,nil,   //8 case switch defualt endcase
 pbkw,pskw,peos,nil,    // 12 break return while do
 pbkw,nil,nil,nil,      //16 continue repeat until
 peos,peos,nil,nil,     // for to downto step
 nil,nil,nil,nil,   //_untilP, _loop, _with, _endif,
 nil,nil,nil,nil, nil,nil,nil,nil,//_goto,_extern,
//_var,_const,_enum,_struct/_record/,_union,_register
 nil,nil,nil,nil, nil,nil,nil,nil,//_unsigned,
   //_signed,_char,_short,_int,_long,_sint64,_uint64,
 nil,nil,nil,nil, nil,nil,nil,nil, //_float,_double
  //,_void,_auto, _static,_volatile,_typedef,_sizeof,
 nil,...
};
```

Програмування різновидів алгоритму висхідного розбору

Для відтворення результатів синтаксичного аналізу використовується спеціальна процедура відображення **`void`** `prLxTxt(` **`struct`** `lxNode*rt)`, вперше використана в лабораторному занятті 2.28, яка виконує реконструкцію у відповідності зі зв'язками підлеглості.

Матриця передування може займати занадто великий обсяг пам'яті. Якщо для опису мови використано 120 термінальних та нетермінальних символічних позначень, знадобиться матриця, що складається з 120 елементів, кожний з яких має довжину не менше двох бітів. Однак в багатьох випадках інформація, яка закладена в матриці, може бути подана двома функціями f та g, такими, що з $R \bullet = S$ витікає $f(R) = g(S)$, з $R <\bullet S$ витікає $f(R) < g(S)$, а з $R \bullet > S$ випливає $f(R) > g(S)$ для всіх символів граматики. Це зветься лінеаризацією матриці. Якщо функції f та g ідентичні, то ігноруючи поря-

док і семантику виконання операторів, ми можемо користуватися лише однією функцією пріоритету p. Таким чином, обсяг потрібної матриці можна зменшити з n^2 до $2n$ або навіть n комірок.

Слід відзначити, що такі функції передування непоодинокі – якщо для даної матриці знайдеться хоча б одна пара функцій f та g, для тієї ж матриці існує нескінченна кількість таких функцій. Але є багато матриць передування, для яких ці функції не існують. Неможливо лініаризувати, наприклад, навіть таку просту матрицю передування розміром 2×2 для символів S_1 та S_2:

$$\begin{vmatrix} \bullet = \bullet > \\ \bullet = \bullet = \end{vmatrix}$$

Для подання матриці передування може бути визначений перенумерований тип виду

```
enum rlPrec // передування пар лексем
{_pEq,// однакове передування
 _pLt,// перша передує другій
 _pGt,// друга передує першій
 _err};  // неприпустима комбінація
```

Тоді опис самої матриця буде мати вигляд

```
enum rlPrec mPrc[nTrm+nNtrm][nTrm+nNtrm];
```

де `nTrm` – кількість термінальних позначень, а `nNtrm` – кількість нетермінальних позначень.

Структура програмного шаблона виконання роботи

Для спрощення розв'язання задачі слід використовувати прототип проекту цього практичного заняття spLb5.dsp, який зберігається в робочому просторі spLb1.dsw в папці spLb1, яка зберігає шаблони прототипів для всіх практичних занять циклу «Системне програмування-2». До складу проекту консольної прикладної програми, орієнтованої на можливість використання в числі інших бібліотек стандартних функцій, входять модулі з наступними вхідними файлами заголовків і реалізацій:

- StdAfx.h і StdAfx.cpp – модуль для організації використання об'єктів класів MFC;
- langio.cpp – модуль для введення-виведення даних транслятора.
- tables.h і tables.cpp – модуль для опису і організації використання структур або класів таблиць;
- index.h і index.cpp – модуль для побудови впорядковуючого індексу для образів імен і констант;
- lexan.h і lexan.cpp – модуль для виконання лексичного аналізу;
- token.h і token.cpp – модуль для опису кодування типу лексеми, структури вузла лексеми і організації використання структур або класів лексем в процесі лексичного і синтаксичного аналізу;
- visgrp.h і visgrp.cpp – модуль для відображення рядків або повних таблиць;

- parsP.cpp, parsC.cpp і parsV.cpp – модулі констант для відтворення лексем мов Pascal, C/C++ та Verilog HDL, які використовують однойменні модулі проекту spLb3.dsp;
- spLb5.cpp – модуль контрольних прикладів для тестування функцій обробки графів.

Розділ визначень і декларацій основної частини програми тестування в модулі spLb5.cpp повинен включати виклик функцій ініціалізації таблиць для потрібної мови, їх елементів, спеціальні змінні, що зберігають довжину (тобто кількість елементів) таблиці. Розділ визначень модуля visgrp.cpp забезпечує визначення мов реконструкції за варіантом завдання. Виконувана частина модуля spLb5.cpp програми повинна включати циклічно повторювані виклики функцій для перевірки програми лексичного аналізу та відображення різних варіантів вхідних даних.

Завдання на роботу

Завдання на підготовку до роботи на комп'ютері:

1. Визначити варіант завдання для основних задач за таблицею 5.1. Визначити приклади лексем через файл в папці spLb5 модуля тестування spLb5.cpp.

2. Відповісти на контрольні запитання.

3. Підготувати настройки вхідної мови програмування.

4. Використати структуру елементу **`struct`** `lxNode` шаблону програмного проекту spLb5 для побудови елементу індексу таблиць лексем і визначити початковий стан індексу.

5. Підготувати програмний модуль контрольної задачі, який виконує заданий варіант з таблиці 4.1 і дозволяє перевірити коректність виконання програм.

Завдання на роботу на комп'ютері

6. Побудувати програмний проект, ввівши програмні модулі у відповідні файли проекту і налагодити синтаксис.

7. Побудувати виконуваний модуль тестової програми і налагодити змістовне виконання програми для перевірки результатів контрольних прикладів.

8. Одержати результати виконання програми висхідного розбору, зробивши наступні дії:

- вставити контрольну точку в модулі spLb5.cpp перед викликом функції `prLxTxt(nodes+nr);`
- проглянути в режимі налагодження масив лексем **`struct`** `lxNode nodes [200]` і проаналізувати в режимі налагодження поля типів лексем **`enum`** `tokType ndOp`, координат лексем у вхідному тесті **`int`** `x, y, f` та полів покажчиків **`struct`** `lxNode* prvNd` і **`struct`** `lxNode* pstNd` для базових лексем вхідного тексту;
- проглянути реконструкцію тексту викликом функції `prLxTxt(nodes+ nr);`
- зробити висновки про роботу програми та настройки таблиць.

9. Продемонструвати результати викладачам

Порядок вибору варіанту:

За останньою цифрою номера залікової книжки або за порядковим номером студента в списку підгрупи з доданим номером групи визначте за табл. 4.1 варіант оброблюваних даних та настройки програм за прикладом.

Таблиця 4.1

Варіанти завдань для виконання аналізу вхідного файлу з одержанням масиву лексем

№ вар.	Вираз, який відтворюється в графі внутрішнього подання	Мова відтворення
1	b=(2*a +c/d)*2*a;	C
2	b:=(2*a+c)*2*a;	Pascal
3	b+=a[n];	C
4	n:=n-1; b:=b+a[n] **;**	Pascal
5	b+=a[--n];	C
6	b:=b+a[n]; n:=n-1 ; n=0;	Pascal
7	b=sin(2*a); b=2*a;	C
8	b:=sin(2*a); b:=a;	Pascal
9	b=c?d:2*a[n];	C
10	b:=d!=0**;** b:=2*a[n];	Pascal
11	b=2*a[n]; b=d**;**	C
12	b:=2*a[n]; b:=d;	Pascal
13	b=0;b+=a[n];	C
14	b:=0; n:=n; b:= b+a[n];	Pascal
15	--n; b=c==a[n]**;**	C
16	n:=n-1; b:=0; b:=n!=a[n]);	Pascal
17	--n; b=n==a[n];	C
18	b:=a-c=0; c:=(a-c)*2*a;	Pascal
19	--n; b=n&&b==a[n];	C
20	b=--n&&b==a[n]);	C
21	b:=n>0 **and** b=a[n]; n:=n-1;	Pascal
22	b=c<a; b=sin(2*a); b=2*a;	C

Контрольні запитання про особливості висхідного синтаксичного розбору

1. Дайте визначення основних типів виразів в мовах програмування.

2. Яким чином обробляються імена та ключові слова у висхідному синтаксичному аналізаторі?

3. Яка попередня обробка повинна виконуватися в синтаксичному аналізаторі для операцій мови?

4. Які поля структур лексем термінальних вузлів заповнюються на етапі синтаксичного аналізу?

5. Які поля структур лексем вузлів роздільників та ключових слів заповнюються на етапі синтаксичного аналізу?

6. Яким чином порівнюються передування для лексем при висхідному розборі?

7. Як може організовуватися накопичення стеку високопріоритетних операцій?

8. Яким чином визначаються передування при висхідному синтаксичному аналізі?

9. Які вузли можуть вважатися надлишковими в структурі графу або дерева розбору?

10. Які синтаксичні помилки можуть розпізнаватися при висхідному синтаксичному розборі?

Лабораторна робота 2.5

Тема роботи: Створення і настроювання низхідних синтаксичних аналізаторів на базі використання метамов Бекуса

Мета роботи: Одержання навичок створення механізмів синтаксичного розбору методом рекурсивного спуска і створення обробника синтаксичних помилок вхідного тексту.

Короткі теоретичні відомості

Результатом виконання цієї роботи повинен бути синтаксичний аналізатор, що працює за принципом рекурсивного спуску. Створена програма буде навчальним виконанням часткового аналізатора для мови, \подобної процедурним мовам Pascal або C. Результатом роботи аналізатора є дерево розбору, яке перетворюється на дерево підлеглості операцій і операторів, з яким може працювати семантичний аналізатор. Так само реалізовано виявлення синтаксичних помилок (через особливості метода рекурсивного спуска вхідні дані обробляються до першої помилки).

Структура вхідних даних

Аналізатор побудовано на основі шаблонів лабораторних робіт курсу «Системне програмування» (а саме: на базі лабораторної роботи 2.2). Безпосередніми вхідними даними є

масив лексем структурного типу `struct lxNode` з лабораторної роботи 2.3.

Використання формальних граматик низхідного розбору

Визначення формальної граматики включає:

- множину (алфавіт) термінальних символів **T**;
- множина (алфавіт) нетермінальних символів **N**;
- початковий нетермінальний символ **Z** з множини **N**;
- множина породжуючих правил підстановки виду **A ::= B**, де **B** - ланцюжок з будь-яких символів граматики **(N ∪ T)**, **A** - ланцюжок, що включає хоча б один нетермінальний символ.

Зрозуміло, що будь-яка задача перетворення ланцюжків символів в рамках будь-якої граматики може бути розв'язана шляхом повного перебору всіх можливих варіантів підстановок, що викликає за собою експоненціальну трудомісткість розв'язання задачі, не прийнятну в реальних умовах. Будь-яка людина, яка використовує транслятор, в праві чекати, що останній не надто зменшує швидкість роботи при рості обсягу програми, яка транслюється (тобто трудомісткість близька до лінійної). Для цього насамперед потрібно відмовитися від «тупого» перебору варіантів підстановки з поверненнями до проміжних ланцюжків і забезпечити на кожному кроці вибір єдино правильного напрямку руху з декількох можливих (так званий **жадібний алгоритм**).

Синтаксис будь-якої мови програмування визначається контекстно-довільною формальною граматикою - системою терміна-

льних і нетермінальних символів і множини правил. Програма, що аналізується, представляється в такій граматиці реченням мови. Задача синтаксичного аналізу - визначити, чи є це речення правильним, і побудувати для нього послідовність безпосередніх виведень з початкового символу **Z,** або синтаксичне дерево.

В синтаксичному аналізі, аналогічно, відомий метод **рекурсивного спуска,** оснований на «зашиванні» правил граматики безпосередньо в управляючі конструкції розпізнавача. Ідеї низхідного розбору, прийняті в LL(1)-граматиках, в ньому повністю зберігаються. Дві букви L в LL(1) означають, що рядки розбираються зліва направо (Left) і використовуються самі ліві виведення (Left), а цифра 1 – що варіанти породжуючих правил обираються за допомогою одного попередньо переглянутого символу. Таким чином, граматику називають **LL(1)-граматикою**, якщо для кожного нетерміналу, що з'являється в лівій частині більше одного породжуючого правила, множини направляючих символів, що відповідають правим частинам альтернативних породжуючих правил – такі, що не пересікаються. Всі LL(1)-граматики можна розбирати детерміновано згори до низу.

Ідеї низхідного розбору, прийняті в LL(1)-граматиках повністю зберігаються при рекурсивному спуску:

- відбувається послідовний перегляд вхідного рядка зліва-направо;

- черговий символ вхідного рядка є основою для вибору однієї з правих частин правил групи при заміні поточного нетерміналу;
- термінальні символи вхідного рядка і правої части правила «взаємно знищуються»;
- віднайдення нетерміналу в правій части рекурсивно повторює цей самий процес.

В методі рекурсивного спуска над даними відбуваються такі зміни:

- кожному нетерміналу відповідає окрема **процедура (функція),** що розпізнає (обирає і «закриває») одну з правих частин правила, яке має в лівій частині цей нетермінал (тобто для кожної групи правил пишеться свій розпізнавач);
- у вхідному рядку наявний покажчик (індекс) на поточний символ, що «закривається правилами». Цей символ і є основою для вибору необхідної правої части правила. Сам вибір «зашитий» в розпізнавачі у вигляді конструкцій **if** або **switch.** Правила вибору базуються на побудові **множин вибору символів,** як це прийнято в LL(1)-граматиці;
- перегляд обраної частини реалізується в тексті процедури-розпізнавача шляхом порівняння очікуваного символу правої частини і поточного символу вхідного строки;
- якщо в правій части очікується термінальний символ і він співпадає з черговим символом вхідного рядка, то символ у вхід-

ному рядку пропускається, а розпізнавач переходить до наступного символу правої частини;

- неспівпадіння термінального символу правої частини і чергового символу вхідного рядка свідчить про синтаксичну помилку;
- якщо в правій частині зустрічається нетермінальний символ, то для нього необхідно викликати аналогічну процедуру (функцію) розпізнавання.

Вузол дерева синтаксичного розбору

Результатом роботи аналізатора є деревоподібна структура, вузлами якої є примірники класу synNode, що має таку структуру:

```
enum synType {
    _prgm, _block, _compound_statement,
    _if_node, _if_without_else, _if_with_else,
    _for_node, _statement, _statement_body,
    _assignment, _bool_expression, _bool_factor,
    _expression, _term, _signed_factor,
    _unsigned_factor,   _terminal
    };
class synNode
{
    public:
        synNode::synNode(enum synType code);
        synNode::synNode();
        void toString(int space);
               //метод виведення вмісту вузла за рядком
        void addChild(synNode* child);
```

```
                                        //додавання нащадків
    enum synType ndOp;      //код типа лексеми
    struct lxNode* termin;
                //термінальний символ даного вузла
    synNode* prvNd;         //попередник
    synNode* child [100];   //нащадки
private:
    int currentSize;        ////кількость нащадків
};
```

Аналізатори нетерміналів граматики

Для кожного з нетерміналів граматики був написаний метод, який обробляв відповідний потік лексем. За поточною лексемою ми визначаємо, яке з правил нам треба обробляти, після чого починаємо обробку. Структура лексем не підкоряється правилам граматики то генерується повідомлення про помилку.

```
synNode* trace_Terminal(lxNode* tok)
{
    synNode* temp = new synNode(_terminal);
    temp->termin = tok;
    return temp;
};
```

Список аналізаторів:

```
synNode* synAnalysis(struct lxNode* nodes);
synNode* trace_Statement();
synNode* trace_Bool_Expression();
synNode* trace_If();
synNode* trace_For();
```

```
synNode* trace_While();
synNode* trace_Statement();
synNode* trace_Compound();
synNode* trace_Block();
synNode* trace_Program();
synNode* trace_Expression();
synNode* trace_Signed_Factor();
synNode* trace_Term();
synNode* trace_Unsigned_Factor();
synNode* trace_Statement_Body();
synNode* trace_Assignment();
```

Приклад реалізації правил розбору граматики

Обробник правил:

_if_statement::= if_with_else | if_without_else

if_without_else ::= "if" bool_expression "then" block ";"

if_with_else ::= "if" bool_expression block "else" block ";"

має вигляд

```
synNode* trace_If()
{
    thread->matchNext(_if);
    synNode* temp = new synNode(_if_node);
    temp->addChild(trace_Bool_Expression());
     thread->matchNext(_then);
    temp->addChild(trace_Block());
    if (thread->lookNext() == _else)
    {
        thread->matchNext(_else);
```

```
        temp->addChild(trace_Block());
    }
    thread->matchNext(_EOS);
    return temp;
};
```

Структура програмного шаблона виконання роботи

Для спрощення розв'язання задачі слід використовувати прототип проекту spLb6.dsp, що зберігається в робочому просторі spLb1.dsw в папці spLb1, яка зберігає шаблони прототипів для всіх лабораторних робіт циклу «Системне програмування». В состав проекту консольної прикладної програми, орієнтованої на можливість використання в числі інших бібліотек стандартних функцій і комплексу об'єктів MFC (Microsoft Foundation Classes) входять модулі з наступними вхідними файлами заголовків і реалізацій:

- StdAfx.h і StdAfx.cpp – модуль для організації використання об'єктів класів MFC, створених поза стандартами ANSI;
- tables.h і tables.cpp – модуль для опису і організації використання структур або класів таблиць;
- spLb6.cpp – модуль контрольних прикладів для тестування побудови дерева синтаксичного аналізу
- synan.h і synan.cpp – основні структури даних і методи для реалізації синтаксичного розбору методом рекурсивного спуска
- інші модулі реалізують функціональність попередніх лабораторних робіт (лексичний аналіз, пошук в таблицях)

При побудові проектів без функцій MFC и в других системах, які дозволяють побудову проектів, файли StdAfx.h і StdAfx.cpp мо-

жуть бути опущені, а посилання на них в операторах #include “StdAfx.h” повинні бути виключені.

Розділ визначень і декларацій основної частини програми тестування в модулі spLb1.cpp повинен включати визначення таблиць, ініціалізацію їх елементів, спеціальні змінні, що зберігають довжину (тобто кількість елементів) таблиці. Виконувана частина модуля spLb1.cpp програми повинна включать циклічно повторювані виклики функцій для перевірки і відображення різних варіантів вхідних даних так, щоб перевірити роботу функцій при типових і граничних значеннях з множини припустимих значений аргументів.

Інструкція по роботі з програмою

Для роботи програмі необхідний файл з первинним кодом, на мові, заданій граматикою, яка наведена вище (розділ «Формування граматики»). Для того щоб получить текстове представлення роботи синтаксичного аналізатора слід виконати такі кроки:

1) Створити файл з вхідним кодом, який буде розбиратися. Його треба зберегти в файлі з назвою в форматі *.h;

2) Запустити програму spLb6.exe з командного рядки або ж запустити програму з середовища розробки;

3) На запит «Please enter file Name:» треба ввести ім’я файлу (без розширення);

4) Якщо вхідний код не включає помилок, то програма повинна вивести результат розбору, а саме дерево розбору. В дереві всі елементи (одна рядок – один елемент) мають свій ступінь вкла-

деності, яка відображується відступом (чим глибше елемент знаходиться в дереві – тем більше відступ). На одному рівні відступів знаходяться нащадки одного елемента:

```
Елемент1
    Нащадок1
        Нащадок2Рівня
            …
            Покажчик на вузол, що включає термінал
    Нащадок2
…
НащадокN ;
```

5) Якщо у вхідному коді є помилки, то буде виведена діагностика виду:

```
Error on <Номер помилкової лексеми> lexem:
 have to be "<код лексеми, яка очікується>" but we have "<код лексеми яка є в коді>"
```

Після знаходження першої помилки програма закінчує роботу.

Для подальшого використання результатами розбору найважливішим є кореневий вузол mainNode, з якого починається все дерево розбору. Його структура указана вище (розділ «Вузол дерева синтаксичного розбору»).

Завдання на лабораторну роботу

1) Підготовка програми

а) Проаналізувати конструкції (задані згідно з варіантом): визначити які саме правила граматики треба довизначити

а) Реалізувати тіла процедур граматичного розбору аналогічно тому, як це показано в розділі «Приклади реалізації правил розбору граматики»

2) Створити тестовий приклад (згідно з варіантом)

3) Налагодити і запустити програму. Получить дерево граматичного розбору. Скопіювати результати роботи програми і вставити їх в протокол.

4) Внести помилку в код, що аналізується, і розглянути реакцію програми на неї. Результати внести до протоколу.

5) Зробити висновки по роботі

Таблиця 5.1

Варіанти завдань для виконання аналізу вхідного файлу

№ вар.	Вираз, який відтворюється в дереві розбору
1	`If (a>b) then begin end;`
2	`If (a>b) then begin For i:= 1 to n do begin a:=b; end; end;`
3	`If (a<c) then begin For i:= 1 to n do begin end; end;`
4	`For i:= 1 to n do begin If (a>b) then begin end; end;`
5	`If (a>b) then begin a:= b; end;`
6	`For i:= 1 to n do begin a:=b; end;`
7	`If (a<c) then begin For i:= 1 to n do begin a:= b; end; end;`
8	`For i:= 1 to n do begin If (a>b) then begin a:= b; end; end;`
9	`If (a>b) then begin end else begin end;`
10	`If (a>b) then begin a:=b; end else begin a:=b; end;`
11	`For a:= 1 to n begin for b:=1 to n do begin a:=m; end; end;`
12	`for i :=1 to n+m do begin d:=j; if (n<m) then begin end; end;`

13	`If (a>j) then begin if (p<n) then begin k:=t; end; end;`
14	`For i:= 1 to n do begin end;`
15	`If (a>b) then begin a:=b; end else begin end;`
16	`If (a<b) then begin For t:= 1 to p do begin a:=b; end; end;`
17	`For a:= 1 to n begin for b:=1 to n do begin a:=m; end; end;`
18	`If (a>b) then begin a:=b; end else begin a:=b; end;`
19	`If (a<c) then begin For i:= 1 to n do begin a:= b; end; end;`
20	`For a:= 4 to y begin for g:=1 to s do begin p:=m; end; end;`

Контрольні запитання про особливості низхідного синтаксичного розбору

1. Дайте визначення основних типів операторів в мовах програмування.

2. Яким чином обробляються імена та ключові слова у низхідному синтаксичному аналізаторі?

3. Яка попередня обробка повинна виконуватися в синтаксичному аналізаторі для операторів комп'ютерної мови?

4. Які синтаксичні помилки можуть розпізнаватися при низхідному синтаксичному розборі?

5. Які поля структур лексем вузлів роздільників та ключових слів заповнюються на етапі синтаксичного аналізу?

6. Яким чином враховуються передування для лексем при низхідному розборі?

7. Як може організовуватися накопичення стеку оброблюваних операторів?

2.6 Лабораторна робота 2.6

Тема роботи: Побудова, настроювання та використання семантичного аналізатора в трансляторах

Мета роботи: Одержання навичок настройки таблиць для семантичних аналізаторів для обробки для визначення описів даних програм і дерев підлеглості з запам'ятовуванням типів даних для результатів кожного графа внутрішнього подання програми, в тому числі таблиць відповідності операндів і операцій. Вивчення процедур розпізнавання типів і формування внутрішнього подання констант у відповідній машинній формі інструментальної машини.

Короткі теоретичні відомості

На основі попередньо визначених полів базової структури елемента внутрішнього подання лексеми, заповнених на етапі лексичного аналізу і уточнених на етапі синтаксичного аналізу, треба організувати поступове заповнення полів типів і довжини даних на етапі семантичного аналізу. Це надасть можливість використання даних попередніх етапів трансляції на наступних етапах семантичної обробки.

Заповнення таблиць і правил для семантичного аналізу

Семантичний аналіз, який є попереднім етапом для всіх видів семантичної обробки, призначений для визначення змістовної суперечності або коректності тексту на комп'ютерній мові. Вхідними даними семантичного аналізу є дерева підлеглості операцій з полями для запам'ятовування типів даних для результатів кожного під-

дерева внутрішнього подання програми. Управляючими даними для визначення типів результатів є таблиці відповідності операндів та операцій. Результатом семантичного аналізу є також визначення характеристик типів результатів операцій та операторів на синтаксичному дереві розбору або спрямованому графі внутрішнього подання.

Гнучкість семантичного аналізу забезпечується як табличною організацією обробки складних типів і типів, що визначаються користувачами, так і табличною організацією аналізу відповідності операндів і результатів використаним операціям і операторам мови. Для виконання семантичного аналізу важливо визначити раціональний підхід для кодування типів даних. З одного боку потрібно визначити нумерацію системних типів складених з декількох ключових слів, що визначається перенумерованим типом **enum** `datType`, з другого треба визначити лічильник рівня вкладеності покажчиків, за одиницю якого обираємо константу `cdPtr`, з третього – треба визначити поточне значення модифікатора для заданого типу з набору визначень модифікаторів.

Вимоги до кодування внутрішнього подання типів:

- однозначний код для будь-яких типів з різними варіантами модифікаторів;
- легкість розбиття елементів коду на окремі поля;
- легкість визначення або підрахунку номера типу користувача та типу і рівня покажчика та забезпечення загальної кількості типів.

Приклад реалізації кодування внутрішнього подання типів для занять з комп'ютерного практикуму включає базовий перенумерований тип, який фактично задає розбиття на три поля кодування.

Номер рівня покажчика (12 бітів)	Ознака константи (1 біт)	Тип пам'яті (3 біти)	Ознака масиву (1 біт)	Код типу користувача (3 біти)	Номер типу користувача (12 бітів)

```
enum datType//кодування типів даних в семантичному
аналізі
{_v,                  // порожній тип даних
 _uc=4,_us,_ui,_ui64,// стандартні цілі без знака
 _sc=8,_ss,_si,_si64,  // стандартні цілі зі знаком
 _f,_d,_ld,_rel,     // дані з плаваючою точкою
 _lbl,           // мітки
// інші стандартні типи
 _geq = 0x0ffe,      // загальний тип для рівності
 _gen = 0x0fff,      // загальний (довільний) тип
 _enm = 0x1000,      // перенумеровані типи enum
 _str = 0x2000,      // структурні типи /*_record*/,
 _unn = 0x3000,      // типи об'єднань union
 _cls = 0x4000,      // типи класів
 _obj = 0x5000,      // типи об'єктів
 _fun = 0x6000,      // функціональні типи
 _ctp = 0x7000,      // умовні типи мови Pascal
 _fl, _tp, _vl, _vr,   //
};
// модифікатори кодів типів мови C/C++
#define cdPtr 0x100000// код покажчика 1-го рівня
```

```
#define cdCns 0x080000// код константного типу даних
#define cdArr 0x108000// код даних типу масиву
#define cdCna 0x188000// код константного масиву
#define cdReg 0x010000// код регістрового типу даних
#define cdExt 0x020000// код зовнішнього типу даних
#define cdStt 0x030000// код статичного типу даних
#define cdAut 0x040000// код автоматичного типу даних
#define cdVlt 0x070000// код примусового типу даних
```

Таке визначення базових типів, покажчиків, масивів та модифікаторів кодів дозволяє гнучко визначати практично будь-який тип в рамках однієї системи кодування і визначати при цьому до 2^{12} варіантів типів користувача.

Таблиці семантичного аналізу включають механізми визначення кодів типів даних за допомогою перенумерованого типу **enum** datType та констант модифікаторів типів. Коди типів даних визначаються в операторах обробки декларацій і використовують такі основні таблиці: визначення типів з модифікаціями; управління семантичною обробкою операцій.

Елементи основних таблиць визначаються наступними структурами:

```
// Елемент таблиці модифікованих типів
struct recrdTPD  // структура рядка таблиці
                         // модифікованих типів
{enum tokType kTp[3];//примірник структури ключа
 unsigned  dTp;//примірник функціональної частини
 unsigned  ln;   // довжина даних типу
};
```

```
struct recrdSMA
              // структура рядка таблиці припустимості
                        // типів для операцій
{enum tokType oprtn;// код операції
 int oprd1,ln1;//код типу та довжина 1-го арг.
 int oprd2,ln2;//код типу та довжина 2-го арг.
 int res, lnRes;//код типу та довжина результату
 _fop*pintf; // покажчик на функцію інтерпретації
 char  *assCd;
};
```

Базові типи, що визначаються ключовими словами, показані в таблиці характеристик типів відносно коду першого ключового слова, що визначає тип. Елементи цієї таблиці мають структуру.

```
struct recrdTMD// структура рядка таблиці базо-
вих типів
{enum datType tpLx;// примірник структури ключа
 unsigned  md;   // модифікатор
 unsigned  ln;   // базова або гранична довжина даних
типу
};
```

А сама таблиця для мови C/C++ має вигляд.

```
struct recrdTMD tpLxMd[]=
           // масив кодів та ознак ключових слів типів
{{_v, 0, 0},     //0 _void
 {_v, 0, 0},     //1 _extern
 {_v, 0, 0},     //2 _var
 {_v,cdCns,0},   //3 _const
 {_enm, 0,32},   //4 _enum
```

```
 {_str, 0, 0},   //5 _struct/*_record*/
 {_unn, 0, 0},   //6 _union
 {_v,cdReg,0},   //7 _register
 {_ui,0,32},     //8 _unsigned
 {_si,0,32},     //9 _signed
 {_si,0,8},      //10 _char
 {_si,0,16},     //11 _short
 {_si,0,32},     //12 _int
 {_si,0,32},     //13 _long
 {_si,0,64},     //14 _sint64
 {_ui,0,64},     //15 _uint64
 {_f,0,32},      //16 _float
 {_d,0,64},      //17 _double
};
```

Таблиця типів, що визначаються декількома словами для мови C/C++, має наступний вигляд.

```
struct recrdTPD tpTbl[]= //  таблиця  модифікованих
типів
{{{_void,_void,_void},_v,0},
 {{_enum,_void,_void},_enm,32},
 {{_struct,_void,_void},_str,0},
 {{_union,_void,_void},_unn,0},
 {{_unsigned,_void,_void},_ui,32},
 {{_signed,_void,_void},_si,32},
 {{_char,_unsigned,_void},_uc,8},
 {{_char,_signed,_void},_sc,8},//4
 {{_char,_void,_void},_sc,8},
 {{_short,_void,_void},_si,16},
```

```
 {{_short,_unsigned,_void},_ui,16},
 {{_short,_signed,_void},_si,16},
 {{_int,_void,_void},_si,32},//9
 {{_int,_unsigned,_void},_ui,32},
 {{_int,_signed,_void},_si,32},
 {{_int,_long,_void},_si,32},
 {{_long,_void,_void},_si,32},
 {{_float,_void,_void},_f,32},//14
 {{_double,_void,_void},_d,64},
 {{_double,_long,_void},_ld,80},
 {{_class,_void,_void},_cls,0},
};
```

Якщо таблицю типів розширити константними, регістровими та зовнішніми типами, то в кожному з їх елементів в останньому елементі ключа додається відображення першого ключового слова-модифікатора з потроєнням загального обсягу таблиці.

Частина таблиці припустимості для типів операндів для типових комбінацій вузлів графів для оператора `if`, закодованого значенням `_if`, та операції накопичувального присвоювання +=, закодованої значенням `_asAdd`, має вигляд.

```
struct recrdSMA  ftTbl[]=
          // таблиця припустимості типів для операцій
{{_if,_ui,32,_v,0,_v,0},
 {_if,_ui,32,_ui,32,_v,0},
 {_if,_ui,32,_si,32,_v,0},
 {_if,_ui,32,_f,32,_v,0},
 {_if,_ui,32,_d,64,_v,0},
```

```
{_if,_si,32,_v,0,_v,0},
{_if,_si,32,_ui,32,_v,0},
{_if,_si,32,_si,32,_v,0},
{_if,_si,32,_f,32,_v,0},
{_if,_si,32,_d,64,_v,0},
{_if,_f,32,_v,0,_v,0},
{_if,_f,32,_ui,32,_v,0},
{_if,_f,32,_si,32,_v,0},
{_if,_f,32,_f,32,_v,0},
{_if,_f,32,_d,64,_v,0},
{_if,_d,64,_v,0,_v,0},
{_if,_d,64,_ui,32,_v,0},
{_if,_d,64,_si,32,_v,0},
{_if,_d,64,_f,32,_v,0},
{_if,_d,64,_d,64,_v,0},
{_if,_ui|cdPtr,32,_v,0,_v,0},
{_if,_ui|cdPtr,32,_ui,32,_v,0},
{_if,_ui|cdPtr,32,_si,32,_v,0},
{_if,_ui|cdPtr,32,_f,32,_v,0},
{_if,_ui|cdPtr,32,_d,32,_v,0},
{_if,_si|cdPtr,32,_v,0,_v,0},
{_if,_si|cdPtr,32,_ui,32,_v,0},
{_if,_si|cdPtr,32,_si,32,_v,0},
{_if,_si|cdPtr,32,_f,32,_v,0},
{_if,_si|cdPtr,32,_d,32,_v,0},
{_if,_ui|cdPtr|cdArr,32,_v,0,_v,0},
{_if,_ui|cdPtr|cdArr,32,_ui,32,_v,0},
{_if,_ui|cdPtr|cdArr,32,_si,32,_v,0},
```

```
{_if,_ui|cdPtr|cdArr,32,_f,32,_v,0},
{_if,_ui|cdPtr|cdArr,32,_d,32,_v,0},
{_if,_si|cdPtr|cdArr,32,_v,0,_v,0},
{_if,_si|cdPtr|cdArr,32,_ui,32,_v,0},
{_if,_si|cdPtr|cdArr,32,_si,32,_v,0},
{_if,_si|cdPtr|cdArr,32,_f,32,_v,0},
{_if,_si|cdPtr|cdArr,32,_d,32,_v,0},
{_if,_f|cdPtr,32,_v,0,_v,0},
{_if,_f|cdPtr,32,_ui,32,_v,0},
{_if,_f|cdPtr,32,_si,32,_v,0},
{_if,_f|cdPtr,32,_f,32,_v,0},
{_if,_f|cdPtr,32,_d,64,_v,0},
{_if,_d|cdPtr,64,_v,0,_v,0},
{_if,_d|cdPtr,64,_ui,32,_v,0},
{_if,_d|cdPtr,64,_si,32,_v,0},
{_if,_d|cdPtr,64,_f,32,_v,0},
{_if,_d|cdPtr,64,_d,64,_v,0},
{_else,_v,0,_ui,32,_v,0},
    . . .
{_asXor,_si,32,_si,32,_si,32},
{_asAdd,_ui,32,_ui,32,_ui,32},
{_asAdd,_ui,32,_si,32,_ui,32},
{_asAdd,_ui,32,_f,32,_ui,32},
{_asAdd,_ui,32,_d,32,_ui,32},
{_asAdd,_si,32,_ui,32,_si,32},
{_asAdd,_si,32,_si,32,_si,32},
{_asAdd,_si,32,_f,32,_si,32},
{_asAdd,_si,32,_d,32,_si,32},
```

```
{_asAdd,_f,32,_ui,32,_f,32},
{_asAdd,_f,32,_si,32,_f,32},
{_asAdd,_f,32,_f,32,_f,32},
{_asAdd,_f,32,_d,64,_f,32},
{_asAdd,_d,64,_ui,32,_d,32},
{_asAdd,_d,64,_si,32,_d,64},
{_asAdd,_d,64,_f,32,_d,64},
{_asAdd,_d,64,_d,64,_d,64},
{_asAdd,_ui+cdPtr,32,_ui,32,_ui+cdPtr,32},
{_asAdd,_ui+cdPtr,32,_si,32,_ui+cdPtr,32},
{_asAdd,_si+cdPtr,32,_ui,32,_si+cdPtr,32},
{_asAdd,_si+cdPtr,32,_si,32,_si+cdPtr,32},
{_asAdd,_f+cdPtr,32,_ui,32,_f+cdPtr,32},
{_asAdd,_f+cdPtr,32,_si,32,_f+cdPtr,32},
{_asAdd,_d+cdPtr,32,_ui,32,_d+cdPtr,32},
{_asAdd,_d+cdPtr,32,_si,32,_d+cdPtr,32},
{_asSub,_ui,32,_ui,32,_ui,32},
    . . .
```

Елементи наведеної частини таблиці визначають відповідності аргументів і результатів для всіх числових типів даних та покажчиків на дані цих типів.

Структура програмного шаблона виконання роботи

Для спрощення розв'язання задачі слід використовувати прототип проекту цього практичного заняття spLb7.dsp, який зберігається в робочому просторі spLb1.dsw в папці spLb1, яка зберігає шаблони прототипів для всіх лабораторних занять циклу «Системне програмування-2». До складу проекту консольної прикладної

програми, орієнтованої на можливість використання в числі інших бібліотек стандартних функцій входять модулі з наступними вхідними файлами заголовків і реалізацій:

- StdAfx.h і StdAfx.cpp – модуль для організації використання об'єктів класів MFC;
- langio.cpp – модуль для введення-виведення даних транслятора.
- tables.h і tables.cpp – модуль для опису і організації використання структур або класів таблиць;
- index.h і index.cpp – модуль для побудови впорядковуючого індексу для образів імен і констант;
- lexan.h і lexan.cpp – модуль для виконання лексичного аналізу;
- token.h і token.cpp – модуль для опису кодування типу лексеми, структури вузла лексеми і організації використання структур або класів лексем в процесі лексичного і синтаксичного аналізу;
- visgrp.h і visgrp.cpp – модуль для відображення рядків або повних таблиць;
- parsP.cpp, parsC.cpp і parsV.cpp – модулі констант для відтворення лексем мов Pascal, C/C++ та Verilog HDL, які використовують однойменні модулі проекту spLb3.dsp;
- spLb7.cpp – модуль контрольних прикладів для тестування функцій обробки графів.

Розділ визначень і декларацій основної частини програми тестування в модулі spLb7.cpp повинен включати виклик функцій ініціалізації таблиць для потрібної мови їх елементів, спеціальні змінні, що зберігають довжину (тобто кількість елементів) таблиці. Розділ визначень модуля visgrp.cpp забезпечує визначення мов реконструкції за варіантом завдання. Виконувана частина модуля spLb5.cpp програми повинна включати циклічно повторювані виклики функцій для перевірки програми лексичного аналізу та відображення різних варіантів вхідних даних.

Зверніть увагу, що таблиця визначення типів результатів `struct recrdSMAftTbl[179]` в модулі реалізації таблиць semanT.cpp в проекті spLb7.dsp заповнена частково для можливості додавання елементів, потрібних за варіантом.

Завдання на роботу

Завдання на підготовку до роботи на комп'ютері:

1. Визначити варіант завдання для основних задач за таблицею 7.1. Визначити приклади лексем через файл в папці spLb7 модуля тестування spLb7.cpp.

2. Відповісти на контрольні запитання.

3. Підготувати настройки вхідної мови програмування.

4. Використати структуру елементу **`struct`** `lxNode` з файлу index.h шаблону програмного проекту spLb7 для побудови елементу індексу таблиць лексем і визначити поля, що заповнюються при семантичному аналізі.

5. Підготувати програмний модуль контрольної задачі, який виконує заданий варіант з таблиці 6.1 і дозволяє перевірити коректність виконання програм. Для цього доповнити таблиці відповідності типів результатів типам операндів **`struct`** `recrdSMA ftTbl[179]` в модулі реалізації таблиць semanT.cpp для семантичного аналізу, доповнивши її потрібними для варіанта елементами.

Порядок вибору варіанту:

За останньою цифрою номера залікової книжки або за порядковим номером студента в списку підгрупи з доданим номером групи визначте за табл. 6.1 варіант оброблюваних даних та настройки програм за прикладом.

Завдання на роботу на комп'ютері

6. Побудувати програмний проект, ввівши програмні модулі у відповідні файли проекту і налагодити синтаксис.

7. Побудувати виконуваний модуль тестової програми і налагодити змістовне виконання програми для перевірки результатів контрольних прикладів, заданих в варіантах з таблиці 6.1, що включають помилкові покажчики в описах даних.

8. Одержати результати виконання, проаналізувати виникнення діагностичних повідомлень в них в режимі налагодження і зробити висновки.

9. Одержати результати виконання, проаналізувати коректність сформованих типів результатів в режимі налагодження і зробити висновки.

10. Продемонструвати результати викладачам

Таблиця 6.1

Варіанти завдань для виконання аналізу файлу лексем

№ вар.	Вираз, який відтворюється в графі внутрішнього подання	Мова відтворення
1	**float** b, a, c, *d; b=(2*a +c/d)*2*a;	C
2	**double** *b, a[5]; **int** n; b:=(2*a+c)*2*a;	Pascal
3	**float** *b, a[4]; **int** n; b+=a[n];	C
4	**float** *b, a[3]; **int** n; n:=n-1; b:=b+a[n] **;**	Pascal
5	**float** *b, a[5]; **char** n; b+=a[--n];	C
6	**double** *b, a[4]; **char** n; b:=b+a[n]; n:=n-1 ; n=0;	Pascal
7	**double** *a; **int** b; b=sin(2*a); b=2*a;	C
8	**double** b; **unsigned** *a; b:=sin(2*a); b:=a;	Pascal
9	**float** b, *d, a[5]; **int** c; b=c?d:2*a[n];	C
10	**double** b, a[4]; **unsigned** n,*d; b:=d!=0**;** b:=2*a[n];	Pascal
11	**double** b, a[4]; **short** *n,d; b=2*a[n]; b=d**;**	C
12	**float** b, a[3]; **unsigned** n,*d; b:=2*a[n]; b:=d;	Pascal
13	**float** *b, a[3]; **short** n,d; b=0;b+=a[n];	C
14	**float** b, a[5]; **short** *n; b:=0; n:=n; b:= b+a[n];	Pascal
15	**float** a[3] ,*c; **int** b, n; --n; b=c==a[n]**;**	C
16	**float** b, a[3]; **long** n; n:=n-1; b:=0; b:=n!=a[n]);	Pascal
17	**double** b, a[3]; **long** n; --n; b=n==a[n];	C
18	**double** b, a[3]; **long** *n; b:=a-c=0; c:=(a-c)*2*a;	Pascal
19	**double** b, a[6]; **char** *n; --n; b=n&&b==a[n];	C
20	**double** *b, a[3]; **short** n; b=--n&&b==a[n]);	C
21	**float** *b, a[2]; **long** n; b:=n>0 **and** b=a[n]; n:=n-1;	Pascal
22	**double** b, a[3]; **long** *n; b=c<a; b=sin(2*a); b=2*a;	C

Контрольні запитання про особливості семантичного аналізу

1. Як визначають в компіляторах основні типи даних?

2. Яким чином забезпечується відповідність типів даних операндів та результатів операцій при семантичному аналізі?

3. Які поля повинні входити до таблиць семантичного аналізатора при реалізації комп'ютерної мови?

4. Які поля структур лексем термінальних вузлів заповнюються на етапі семантичного аналізу?

5. Які поля структур лексем вузлів роздільників та ключових слів заповнюються на етапі семантичного аналізу?

6. Які дані включаються до таблиць семантичної відповідності операндів?

7. Як організується рекурсивний алгоритм узагальненої семантичної обробки у застосуванні до семантичного аналізу?

8. Які особливості необхідно визначити для семантичного аналізу операцій присвоювання?

9. Які таблиці треба використовувати для визначення типів даних в операторах описів типів і примірників даних?

10. Які помилки можуть розпізнаватися при семантичному аналізі?

2.7 Лабораторна робота 2.7

Тема роботи: **Побудова, настроювання та використання інтерпретатора цільової мови в програмах трансляції**

Мета роботи: Одержання навичок використання вставок на мові Асемблера для побудови та оптимізації абстрактної машини інтерпретації комп'ютерної мови. Вивчення угод про зв'язки для створення процедур і функцій інтерпретації операцій і операторів комп'ютерних мов і звертання до них за допомогою операторів мови С з використанням функціонального типа даних.

Короткі теоретичні відомості

На основі попередньо визначених, заповнених на етапі лексичного аналізу і уточнених на етапах синтаксичного та семантичного аналізу полів базової структури елементів внутрішнього подання лексем можна організувати інтерпретацію шляхом виконання операторів програми у внутрішньому поданні комп'ютерної мови крок за кроком. Для цього необхідно визначити архітектуру віртуальної машини цільової мови та розподілити пам'ять для змінних програми в процесі обробки операторів визначення або декларації змінних. Це надає можливість виконання інтерпретації фрагментів програм з константними даними в процесі трансляції, а також організувати виконуваний етап інтерпретації комп'ютерної мови разом з налагодженням на рівні обробки внутрішнього подання з константними аргументами.

Заповнення таблиць і правил для функціональної інтерпретації

Інтерпретація константних виразів, яка є попереднім етапом для оптимізації та генерації кодів, призначений для виконання операцій або тексту операторів на комп'ютерній мові при константних або введених даних. Вхідними даними інтерпретації є ініціалізовані або присвоєні значення змінних. Управляючими даними для визначення результатів інтерпретації є будь-яка внутрішня форма подання програми, в тому числі дерева та графи підлеглості операцій та операторів. Результатом інтерпретації є попереднє або обумовлене етапом виконання визначення значень результатів операцій та операторів на будь-якій формі внутрішнього подання.

Гнучкість інтерпретації забезпечується як табличною організацією вибору підпрограм інтерпретації окремих операцій для даних всіх наперед визначених типів і типів, що визначаються користувачами, так і табличною організацією доступу до операндів і результатів операцій і операторів комп'ютерної мови. Для інтерпретації операцій і операторів важливо визначити раціональний підхід для кодування типів даних. З одного боку потрібно визначити операції для системних типів, що визначаються функціями найбільш загальних типів аргументів, з іншого – треба визначити механізм створення операцій за умовчанням в типах, визначених користувачем. Вимоги до кодування внутрішнього подання типів залишаються такими самими, як для семантичного аналізу.

Для успішної інтерпретації виконуваних операцій та операторів в кодах на мові Асемблера необхідно розширити таблиці операцій до формату, придатного для виконання кодів, заданих на інструментальній мові програмування або в форматі текстових вставок на мові Асемблера. Формат структури елемента таблиці для семантичного аналізу та інтерпретації внутрішніх кодів розширюється покажчиками на функції інтерпретації типу `_fop*`, заданими в модулях інтерпретатора.

Щоб узагальнити схему звертання до підпрограм інтерпретації системних (визначених наперед) типів створено об'єднання всіх стандартних типів даних мови С та типи даних фірмових розширень розміром до 8-ми байтів:

```
union pGnDat//тип покажчика на сховище даних
{int      *_id;  // на 4-байтні цілі дані
 short *_sd;  // на 2-байтні цілі дані
 char  *_cd;  // на 1-байтні цілі дані
 float *_fd;  // на 4-байтні дані з плаваючою точкою
 double   *_dd;// на 8-байтні дані з плаваючою точкою
 void  *_vd;  // на дані типів enum struct union class
 union pGnDat *_pd;// на дані покажчиків різних типів
 long double*_td;
                //на 10-байтні дані з плаваючою точкою
 __int64  *_i8;  // на 8-байтні цілі дані
};
```

Для більш повної реалізації комп'ютерних мов до цього об'єднання слід додати механізм опису всіх інших системних типів

мови або замінити відповідні типи покажчиками на дані цих типів. Для типів, що визначаються користувачем, слід створити механізм підрахунку обсягу одного примірника даних і резервувати відповідний обсяг для акумуляторів та елементів стеку. В цьому випадку доцільно уточнити визначення типу вузлів, використане в Лабораторній роботі 3, з врахуванням об'єднання покажчиків на дані арифметичних типів та типів, визначених користувачем, додавши додаткове об'єднання та замінивши поле покажчика **struct** lxNode*pstNd на поле об'єднання **union** pGnNode pstNd. Визначення структури вузла набуває вигляду:

```
struct lxNode;//вузол дерева або САГ
union pGnNode//тип покажчика на сховище даних
{union pGnDat pNmb;//тип покажчика на стандартні дані
 struct lxNode*pstNd;//покажчик вузла дерева або САГ
};
struct lxNode//вузол дерева або САГ
{enum tokType ndOp; //код типу лексеми
 struct lxNode*prvNd;// зв'язок з вузлом-попередником
 union pGnNode pstNd;// зв'язок з вузлом-наступником
 int dataType;   // код типу даних, які повертаються
 unsigned resLength; //довжина результату
 int x, y, f;//координати розміщення у вхідному файлі
 struct lxNode* prnNd;//зв'язок з батьківським вузлом
 unsigned stkLength;//довжина стека обробки семантики
};
```

Якщо тип _fop визначити як функціональний з аргументами і результатом, що об'єднують покажчики:

```
typedef union pGnDat _fop(union pGnDat,union pGnDat);
```
то структура рядка таблиці операцій прийме вигляд:
```
struct recrdSMA  // структура рядка таблиці операцій
{enum tokType oprtn;// код операції
unsigned oprd1,ln1;//код типу та довжина першого ар-
гументу
unsigned oprd2,ln2;//код типу та довжина другого ар-
гументу
unsigned res,lnRes;//код типу та довжина результату
_fop*pintf;  // покажчик на функцію інтерпретації
};
```

Для такого визначення можна навести приклади визначення функцій додавання та відповідних елементів таблиць:
```
union pGnDat _fadd(union pGnDat p1,union pGnDat p2)
{ *(pAccD32._pd._fd)=*(p1._fd)+*(p2._fd);
 return pAccD32._pd;
}
union pGnDat _dadd(union pGnDat p1,union pGnDat p2)
{ *(pAccD32._pd._dd)=*(p1._dd)+*(p2._dd);
 return pAccD32._pd;
}
```

Покажчик на підпрограми бінарних операцій в проекті, який включає підготовку операндів, наведено нижче:
```
    _fop *_paddf=&_fadd;
```
а таблиця інтерпретації операцій буде включати такі варіанти подання елементів:
```
struct recrdSMA ftTbl[]=//таблиця припустимості типів
```

```
    ...
{_add,_f,32,_f,32,_f,32,_paddf},
                                  //елемент для додавання
{_add,_f,32,_d,64,_d,64,&_dadd},
                                  //елемент для додавання
    ...
```

Крім того, необхідно визначити перетворення функції числових перетворень, в тому числі перетворення до формату **float** в мові C/C++:

```
float cnvTo_f(int td,union pGnDat p1,union pGnDat p2)
{switch(td)
 {case _d:
    *(pAccD32._pd._fd)=*(p1._fd)=*(p2._dd); break;
  case _f:
    *(pAccD32._pd._fd)=*(p1._fd)=*(p2._fd); break;
  default:
    *(pAccD32._pd._fd)=*(p1._fd)=*(p2._id);
 }
 return *(pAcc32._pd._fd);
}
```

Організація віртуальної машини інтерпретації

Для коректної інтерпретації внутрішнього подання мови важливо раціонально визначити архітектуру віртуальної або абстрактної машини мови. Основними елементами такої машини повинні бути:

1. **Сховище управляючих даних або програми машини.** В нашому випадку ми будемо використовувати для цієї мети раніше визначене дерево підлеглості операторів і операцій. Деякі сучасні мови програмування, такі як Java, мають стандарт віртуальної машини, який необхідно додержувати при будь-якій реалізації.

2. **Сховища інформаційних даних програми в мові, що інтерпретується.** Ці сховища повинні раціонально реалізовувати збереження даних в програмно доступних регістрах і оперативній пам'яті інструментального комп'ютера. Деякі комп'ютерні мови, в тому числі мови моделювання обладнання VHDL та Verilog HDL мають стандарти на деякі вбудовані типи даних, яких також необхідно додержуватись. Ці елементи архітектури повинна включати принаймні наступні частини:

а) **акумулятори**, як правило різні для різних типів даних і такі, що за можливістю відповідають архітектурі інструментального процесора;

б) **стекова пам'ять** для збереження проміжних даних, аргументів підпрограм та локальних даних у вигляді одного або декількох стеків безпосередньо даних та покажчиків на них (на стеки можуть впливати також особливості мовної парадигми);

в) **адресована пам'ять** для доступу до глобальних та статичних даних за іменами.

Зрозуміло, що будь-який нестандартизований елемент архітектури можна реалізувати різними способами, і тому може існувати

безліч реалізацій віртуальних машин. Спроектуємо архітектуру з наступних міркувань:

- використання найбільшої раціональної розрядності акумуляторів для основних типів даних, що відображується 32-бітовим фізичним акумулятором `eax` цільового процесора для збереження цілих даних або покажчиків, та 80-бітовою верхівкою стека регістрів `st` або `st[0]` для даних з плаваючою точкою;

- використання блоків пам'яті відповідних розмірів для збереження іменованих та проміжних даних в пам'яті і стеку.

Внутрішні службові операції віртуальної машини повинні забезпечувати копіювання даних до стеку та зі стеку з раціональним використанням виділеного обсягу стеку. Для зручності обробки та економного використання серед різних варіантів організації стека в шаблоні проекту оберемо такий, в якому окремо будемо зберігати дані потрібної довжини, але блоками довжиною не менше 4-байтів. Крім того, для регулярного доступу до проміжних даних створимо об'єднаний стек типів проміжних даних і покажчиків на їх розміщення у стеку даних.

Для другого стеку слід визначити тип даних окремих елементів.

```
struct stkPEl
          // елемент стека типів і покажчиків на дані
{enum datType _dt;  // коди типів даних в стеку
 union pGnDat _pd;  // покажчик на дані різних типів
};
```

Тоді виділення пам'яті під два потрібні стеки для інтерпретації операцій та операторів можна виконати наступними операторами:

```
#define STKLEN  32 // визначення граничного розміру стека
unsigned   sdtPtr=STKLEN*2, // покажчик стека даних
        stpPtr=STKLEN;   // покажчик стека покажчиків
unsigned dtStk[STKLEN];// резервування стека даних
union stkPEl ptStk[STKLEN];
                        //резервування стека покажчиків
```

Тоді функції запису і читання для двох потрібних стеків, що зберігають і відновлюють проміжні дані, можна задати наступними операторами:

```
void push(union pGnDat d)
{ptStk[stpPtr]._dt=pd;
 ptStk[stpPtr--]._pd=pd;
}
void pop(union pGnDat *pd)
{*pd=ptStk[++stpPtr]._dt;
 *pd=ptStk[stpPtr]._pd;
}
```

В інтерпретаторі переходи можуть визначатися внутрішніми кодами програми або у функціях шляхом зміни попереднього покажчика покажчиком на поточний вузол графа.

Основні типи реалізації інтерпретаторів

В основу структури інтерпретатора можуть бути покладені дві базові схеми:

1) схема управління через розгалуження оператором `switch`;

2) схема управління операціями через індексований перехід або виклик функції.

При роботі за кожною зі схем спочатку для кожного нетермінального вузла графа підлеглості рекурсивно інтерпретуються підлеглі операції по відгалуженням на вузол-попередника і вузол-наступника, як показано нижче або в іншій послідовності та більш складним чином в залежності від семантики оператора або операції.

```
vp1=SmIntrp(nd->prvNd,1);
vp2=SmIntrp(nd->pstNd.pstNd,1);
```

Потім визначається необхідність перетворення типів, і виконуються необхідні перетворення. (Ця необхідність може бути усунута розширенням набора інтерпретуючих функцій). Фрагмент реалізації інтерпретатора з розгалуженням за оператором **`switch`** показано нижче.

```
switch(nd->ndOp)
{case _asAdd:
   if(nd->prvNd)*(pAcc32._fd)=*(vp1._fd)+=*(vp2._fd);
        break;
 case _lt:
   if(nd->prvNd)*(pAcc32._id)=*(vp1._fd)<*(vp2._fd);
         break;
```

```
case _gt:
    if(nd-prvNd)*(pAcc32._id)=*(vp1._fd)>*(vp2._fd);
        break;
case _add:
   if(nd->prvNd)*(pAcc32._fd)= *(vp1._fd)+*(vp2._fd);
        break;...
```

Приклад використання переходу на функцію інтерпретації операцій в проекті наведено нижче:

```
ftImp.oprd1=nd->dataType; ftImp.ln1=nd->resLength;
ftImp.oprd2=nd->dataType; ftImp.ln2=nd->resLength;
ftImp.oprtn=nd->ndOp;
struct recrdSMA*pftImp = selBin(&ftImp, ftTbl, 361);
if(pftImp)*(pAcc32._fd)=*(pftImp-
>pintf(vp1,vp2)._fd);
```

В інтерпретаторі з розгалуженнями переходи можуть визначатися внутрішніми кодами програми або у функціях шляхом зміни попереднього покажчика покажчиком на поточний вузол графа.

Приклад звертання до підпрограм бінарних операцій за покажчиком в проекті в формі дизасемблювання наведено нижче:

```
push ebp;Запис до стеку старої бази аргументів
```

Далі наведено приклад прологу підпрограм бінарних операцій в проекті нижче:

```
push ebp;Запис до стека старої бази аргументів
mov  ebp,esp; Формування нової бази
sub  esp,5Ch; Резервування локальних даних
push ebx   ; Збереження робочих регістрів
push esi
```

```
push edi
lea  edi,[ebp-5Ch]; Початок локальних даних
mov  ecx,17h  ; Довжина блоку локальних даних
mov  eax,0CCCCCCCCh;Код неініціалізованих даних
rep stos   dword ptr [edi]  ; Заповнення блоку
```

Такий самий блок для 16-бітового режиму мав би вигляд:

```
push      bp ; Запис до стеку старої бази
аргументів
mov  bp,sp; Формування нової бази
sub  sp,5Ch;Резервування локальних даних
push   bx  ; Збереження робочих регістрів
push   si
push   di
lea  di,[bp-5Ch];Адреса початку локальних даних
mov  cx,17h   ;Довжина блоку локальних даних
mov  ax,0CCCCCCCCh; Код неініціалізованих даних
rep stos   dword ptr [di]   ; Заповнення блоку
```

Блок епілогу функції для 32-бітового налагоджувального режиму має вигляд

```
mov    eax,offset acc32;Завантаження покажчика
                                  ; на результат
pop    edi ; Відновлення робочих регістрів
pop    esi
pop    ebx
add    esp,5Ch; Звільнення стеку від аргументів
cmp    ebp,esp; Контроль коректності виконання
call   __chkesp ; процедури за стеком
mov    esp,ebp
```

```
pop     ebp; Відновлення старої бази стеку
ret
```

і забезпечує повернення покажчика на результат типу **union** gnDat* в акумуляторі eax. А блок епілогу функції для 16-бітового режиму може мати вигляд

```
mov     ax,offset acc32;Завантаження покажчика
                                ; на результат
pop     di  ; Відновлення робочих регістрів
pop     si
pop     bx
add     sp,5Ch ; Звільнення стеку від аргументів
cmp     bp,sp  ; Контроль коректності виконання
call    __chkesp ; процедури за стеком
mov     sp,bp
pop     bp  ; Відновлення старої бази стеку
ret
```

Тоді код виклику vp1=SmIntrp(nd->prvNd,vp2->_id) в 32-бітовому режимі налагодження матиме вигляд:

```
mov ecx,dword ptr [ebp-8]
mov  edx,dword ptr [ecx]
push  edx   ; Підготовка 1-го аргументу
mov  eax,dword ptr [ebp+8]
mov  ecx,dword ptr [eax+4]
push ecx   ; Підготовка 2-го аргументу
call SmIntrp
add  esp,8;Компенсація адрес аргументів в стеку
mov  dword ptr[ebp-4],eax;Збереження результату
```

А код такого самого виклику в 16-бітовому режимі з 32-бітовими адресами може мати вигляд:

```
mov   bx,dword ptr [bp-8]
mov   edx,dword ptr [bx]
push  dx   ; Підготовка 1-го аргументу
mov   bx,dword ptr [bp+8]
mov   cx,dword ptr [bx+4]
push  cx   ; Підготовка 2-го аргументу
call  SmIntrp
add   sp,8;Компенсація адрес аргументів в стеку
mov   dword ptr [bp-4],ax;Збереження результату
```

Таблиці інтерпретації команд абстрактної машини машинними командами повинні бути розширеними покажчиками на інтерпретуючі функції або текстом інтерпретації окремих операторів та операцій.

В тих випадках, коли машинні команди інструментального процесора мають більшу потужність, ніж команди, які генеруються інструментальними трансляторами доцільно використовувати вставки на мові Асемблера.

Так, наприклад, для даних, описаних як глобальна змінна та локальні змінні `*vp1, *vp2` типу

```
union    gnDat
{int     _id;      // цілі 4 байти
 short   _sd;      // цілі 2 байти
 char    _cd;      // цілі 1 байт
 float   _fd;      // дійсні 4 байти
 double  _dd;      // дійсні 8 байтів
```

```
    __int64 _i8;   // цілі 8 байтів
};
union gnDat acc32;
union gnDat *vp1, *vp2;
```

Оператор мови С/С++

```
acc32._id=vp1->_id*vp2->_id;
```

можна замінити вставкою на мові Асемблера

```
_asm{ mov    ecx,dword ptr [ebp-4]
      mov    edx,dword ptr [ebp-8]
      mov    eax,dword ptr [ecx]
      imul      eax,dword ptr [edx]
      mov    dword ptr [acc32],eax
      mov    dword ptr [acc32+4],edx
      }
```

де результат займає 64 біти.

А оператор мови С/С++

```
acc32._fd=vp1->_fd*vp2->_fd;
```

можна замінити вставкою на мові Асемблера

```
_asm{  mov ecx,dword ptr [ebp-4]; адреса 1-го множника
       mov edx,dword ptr [ebp-8]; адреса 2-го множника
       fld  dword ptr [ecx]; копія 1-го множника
       fmul dword ptr [edx]; множення на 2-й множник
       fstp dword ptr [acc32];молодші біти результату
    }
```

де результат займає 32 біти.

Структура програмного шаблона виконання роботи

Для спрощення розв'язання задачі слід використовувати прототип проекту цього практичного заняття spLb8.dsp, який зберігається в робочому просторі spLb1.dsw в папці spLb1, яка зберігає шаблони прототипів для всіх лабораторних занять циклу «Системне програмування-2». До складу проекту консольної прикладної програми, орієнтованої на можливість використання в числі інших бібліотек стандартних функцій входять модулі з наступними вхідними файлами заголовків і реалізацій:

- StdAfx.h і StdAfx.cpp – модуль для організації використання об'єктів класів MFC;
- langio.cpp – модуль для введення-виведення даних транслятора.
- tables.h і tables.cpp – модуль для опису і організації використання структур або класів таблиць;
- index.h і index.cpp – модуль для побудови впорядковуючого індексу для образів імен і констант;
- lexan.h і lexan.cpp – модуль для виконання лексичного аналізу;
- token.h і token.cpp – модуль для опису кодування типу лексеми, структури вузла лексеми і організації використання структур або класів лексем в процесі лексичного і синтаксичного аналізу;

- visgrp.h і visgrp.cpp – модуль для відображення рядків або повних таблиць;
- parsP.cpp, parsC.cpp і parsV.cpp – модулі констант для відтворення лексем мов Pascal, C/C++ та Verilog HDL, які використовують однойменні модулі проекту spLb3.dsp;
- spLb8.cpp – модуль контрольних прикладів для тестування функцій обробки графів.

Розділ визначень і декларацій основної частини програми тестування в модулі spLb8.cpp повинен включати виклик функцій ініціалізації елементів таблиць для потрібної мови, спеціальні змінні, що зберігають довжину (тобто кількість елементів) таблиці. Розділ визначень модуля visgrp.cpp забезпечує визначення мов реконструкції за варіантом завдання. Виконувана частина модуля spLb5.cpp програми повинна включати циклічно повторювані виклики функцій для перевірки програми лексичного аналізу та відображення різних варіантів вхідних даних.

Завдання на роботу

Завдання на підготовку до роботи на комп'ютері:

1. Визначити варіант завдання для основних задач за таблицею 8.1. Визначити приклади лексем через файл в папці spLb8 модуля тестування spLb8.cpp.

2. Відповісти на контрольні запитання.

3. Підготувати настройки вхідної мови програмування.

4. Використати структуру елементу **`struct`** `lxNode` з файлу index.h шаблону програмного проекту spLb8 для побудови елементу індексу таблиць лексем для прискорення пошуку і визначити функції доступу до імен за індексом.

5. Підготувати програмний модуль контрольної задачі, який виконує заданий варіант з таблиці 7.1 і дозволяє перевірити коректність виконання програм. Для цього організувати пошук в таблиці відповідності типів результатів типам операндів для семантичного аналізу.

Завдання на роботу на комп'ютері

6. Побудувати програмний проект, ввівши програмні модулі у відповідні файли проекту і налагодити синтаксис.

7. Побудувати виконуваний модуль тестової програми і налагодити змістовне виконання програми для перевірки результатів контрольних прикладів.

8. Одержати результати виконання, проаналізувати їх в режимі налагодження і зробити висновки.

9. Продивившись в режимі дизасемблювання фрагменти операцій, що виконуються в інтерпретаторі, замінити оператори мови C/C++ операторами асемблерних вставок, приклад яких наведено в інтерпретації множення з фіксованою точкою.

10. Продемонструвати результати викладачам.

Таблиця 7.1

Варіанти завдань для обробки вхідного файлу з інтерпретацією графа підлеглості

№ вар.	Вираз, який відтворюється в графі внутрішнього подання	Мова відтворення
1	**float** b, a, c, d; b=(2*a +c/d)*2*a;	C
2	**double** b, a[5]; **int** n; b:=(2*a+c)*2*a;	Pascal
3	**float** b, a[4]; **int** n; b+=a[n];	C
4	**float** b, a[3]; **int** n; n:=n-1; b:=b+a[n] **;**	Pascal
5	**float** b, a[5]; **char** n; b+=a[--n];	C
6	**double** b, a[4]; **char** n; b:=b+a[n]; n:=n-1 ; n=0;	Pascal
7	**double** a; **int** b; b=sin(2*a); b=2*a;	C
8	**double** b; **unsigned** a; b:=sin(2*a); b:=a;	Pascal
9	**float** b, d, a[5]; **int** c; b=c?d:2*a[n];	C
10	**double** b, a[4]; **unsigned** n,d; b:=d!=0**;** b:=2*a[n];	Pascal
11	**double** b, a[4]; **short** n,d; b=2*a[n]; b=d**;**	C
12	**float** b, a[3]; **unsigned** n,d; b:=2*a[n]; b:=d;	Pascal
13	**float** b, a[3]; **short** n,d; b=0;b+=a[n];	C
14	**float** b, a[5]; **short** n; b:=0; n:=n; b:= b+a[n];	Pascal
15	**float** a[3] ,c; **int** b, n; --n; b=c==a[n]**;**	C
16	**float** b, a[3]; **long** n; n:=n-1; b:=0; b:=n!=a[n];	Pascal
17	**double** b, a[3]; **long** n; --n; b=n==a[n];	C
18	double a[3]; Boolean b, integer n; b:=a-c=0; c:=(a-c)*2*a;	Pascal
19	**double** b, a[6]; **char** n; --n; b=n&&b==a[n];	C
20	**double** b, a[3]; **short** n; b=--n&&b==a[n]);	C
21	**float** b, a[2]; **long** n; b:=n>0 **and** b=a[n]; n:=n-1;	Pascal
22	**double** b, a[3]; **long** n; b=c<a; b=sin(2*a); b=2*a;	C

Контрольні запитання про особливості інтерпретації

1. Які особливості обробки основних типів даних виникають в інтерпретаторах?

2. Яким чином забезпечується відповідність типів даних операндів та результатів операцій при інтерпретації?

3. Які поля повинні входити до таблиць інтерпретації при реалізації комп'ютерної мови?

4. Які дані результатів формуються на етапі інтерпретації?

5. Які поля структур лексем вузлів роздільників та ключових слів заповнюються на етапі інтерпретації?

6. Які дані включаються до таблиць інтерпретації?

7. Як організується рекурсивний алгоритм узагальненої семантичної обробки у застосуванні до інтерпретації?

8. Які особливості необхідно визначити для інтерпретації операцій присвоювання?

9. Які поля таблиць треба використовувати для визначення типів даних в операторах описів типів і примірників даних?

10. Які помилки можуть розпізнаватися при інтерпретації?

2.8 Лабораторна робота 2.8

Тема роботи: **Організація доступу до архітектури процесора та машинних команд при генерації кодів**

Мета роботи: Одержання навичок побудови програм автоматичної генерації програмних модулів об'єктних кодів для компіляторів з використанням угод базової мови програмування про між модульні зв'язки для генерації кодів вставок на мові Асемблера.

Короткі теоретичні відомості

На основі попередньо визначених, заповнених на етапі лексичного аналізу і уточнених на етапах синтаксичного та семантичного аналізу полів базової структури елемента внутрішнього подання лексем можна послідовно згенерувати блоки виконуваних кодів на мові Асемблера. Для цього треба організувати стандартний розподіл пам'яті для змінних програми відповідно угодам про зв'язки цільової системи програмування в процесі обробки операторів визначення або декларації змінних. Організація сучасних систем програмування надає можливості програмування різних модулів програмних проектів на різних мовах програмування, включаючи мови Асемблера, мови високого рівня та вставки на мові Асемблера.

Формування таблиць і правил для генерації виконуваних кодів

Генерація кодів, яка є заключнім етапом роботи компіляторів, призначеним для підготовки виконуваних кодів для наступного виконання операцій та операторів комп'ютерної мови, що реалізує компілятор. Вхідні, проміжні та результуючі дані для згенерованих кодів здебільшого розміщуються у відповідних файлах операційних систем. Управляючими даними для формування результатів роботи згенерованого коду є програма у двійковому форматі або на мові Асемблера цільового комп'ютера з використанням зручних макросів для формування кодів складних операторів. Результатом генерації кодів є об'єктні файли, що включають дані та команди виконаваних операторів і призначені для наступного поєднання з бібліотечними модулями з можливістю контролю виконання програм у режимі налагодження.

Гнучкість генерації кодів забезпечується як табличною організацією вибору макросів або фрагментів двійкових кодів згенерованих фрагментів програм, що реалізують окремі операції та фрагменти операторів для даних всіх наперед визначених типів і типів, визначених користувачами, так і табличною організацією доступу до операндів і результатів операцій і операторів комп'ютерної мови. Для генерації кодів операцій і операторів так само, як і для інтерпретації важливо визначити раціональний підхід для внутрішнього кодування даних різних типів. З одного боку операції для системних типів даних визначаються функціями для найбільш за-

гальних типів аргументів, а з іншого – вони визначають механізм створення операцій за умовчанням для типів, визначених користувачем. Вимоги до кодування внутрішнього подання даних залишаються такими ж, як і для інших видів семантичної обробки: семантичного аналізу, інтерпретації та машинно-незалежної оптимізації.

Для успішного поєднання генерації виконуваних кодів операцій та операторів в кодах на мові Асемблера з іншими етапами семантичної обробка необхідно розширити таблиці операцій додатковим полем у форматі, придатного для генерації кодів в текстовому форматі Асемблера. Формат структури елемента таблиці для семантичного аналізу та інтерпретації внутрішніх кодів доповнюється в нашій реалізації покажчиками на текстові рядки макросів типу **char** *assCd у форматі, що відповідає форматному рядку функцій стандартних функцій ANSI форматного введення-виведення мови C/C++.

Для успішної трансляції виконуваних операцій та операторів в коди на мові Асемблера необхідно розширити таблиці операцій до формату, придатного для генерації кодів в текстовому форматі Асемблера або безпосередньо у форматі машинних команд. Формат структури елемента таблиці для семантичного аналізу, інтерпретації та генерації кодів через макроси виведення функцій форматного виведення:

```
typedef union gnDat _fop(union gnDat*,union gnDat*);
struct recrdSMA  // структура рядка таблиці операцій
{enum tokType oprtn;// код операції
unsigned oprd1,ln1;//код типу та довжина п1-го арг.
```

```
unsigned oprd2,ln2;//код типу та довжина 2-го арг.
unsigned res,lnRes;//код типу та довжина результату
_fop   *pintf;   // покажчик на функцію інтерпретації
char   *assCd;   // покажчик на макрос або двійковий
код
};
```

Двійкові дані форматі можуть заповнювати рядок у форматі **char*** як байти кодовані у вісімковій системі у форматі кодованих знаків \ooo. Формування таких байтів здійснюється через визначення часткових полів, що потребує додаткових детальних таблиць заповнення часткових полів команди. Тому в шаблоні виконання цієї лабораторної роботи використані текстові макроси, визначені у вигляді вставок на мові Асемблера.

Типові форматні рядки макросів крім традиційних кодів переводу рядків \n та табуляції \t включають елементи форматної підстановки %s, які при генерації функцією sprintf заміщуються іменами або образами значень даних та формують вихідний об'єктний файл на мові С з використанням вставок на мові Асемблера. Структура такого файлу включає заголовок з визначенням глобальних даних, функцій та їх внутрішніх даних та деяких числових констант, використаних у вхідному тексті, які неможливо відтворити як безпосередні аргументи команд (Це дуже поширене для констант в командах операцій з плаваючою точкою).

Для полегшення створення таблиць слід визначити декілька класів макросів у відповідності з основними типами операцій та

окремими вузлами операторів, що використовується у класі графів підлеглості у форматі внутрішнього подання:

- клас операцій присвоювання з двома аргументами: перший – приймач результату або ім'я змінної приймача результату, або так зване ліве значення у формі покажчика на приймач результату; другий – джерело результату: регістр зі значенням результату, змінна-носій результату чи умовчання для результатів обчислень з плаваючою точкою;
- клас операцій присвоювання з трьома аргументами: перший – приймач результату або ім'я змінної приймача результату, або так зване ліве значення у формі покажчика на приймач результату; другий – джерело результату: регістр зі значенням результату, змінна-носій результату чи умовчання для результатів обчислень з плаваючою точкою; третій – носій індексу: індексний регістр, що за іменем відрізняється від робочих регістрів, використаних при обчисленнях (цей формат менш загальний, але трохи більш ефективний при реалізації в системі команд процесорів сімейства Pentium);
- класи операцій накопичувального присвоювання, аналогічні двом попереднім з формуванням результатів для цілих даних та покажчиків в головній пам'яті, а для плаваючої точки через верхівку стека відповідних регістрів;
- клас бінарних операцій з двома аргументами: перший з яких може бути акумулятором або верхівкою стеку, а другий – акумулятором чи простою змінною;

- класи префіксних і постфікних унарних операцій з одним аргументом, який може бути результатом операцій в акумуляторі або простою змінною;
- клас вузлів умовних операторів, операторів циклів та переходів – більшість таких вузлів, крім двох або одного звичайних аргументів мають ще і додаткові, що визначають початкові мітки та переходи на них; базові аргументи можуть формуватися як вкладені або обрамлятися текстом з міток та переходів на них (останнє використане в шаблоні spLb9.dsp).

Приклад початку таблиці в spLb9.dsp проекті наведено нижче:

```
struct recrdSMA ftTbl[]=//таблиця припустимості типів
// приклади умовних операторів
{{_if,_ui,32,_v,0,_v,0,"\n\tcmp\teax,0\n\tjz\t%s"},
 {_if,_ui,32,_ui,32,_v,0,"\n\tcmp\teax,0\n\tjz\t%s"},
{_if,_ui,32,_si,32,_v,0,"\n\tcmp\teax,0\n\tjz\t%s"},
...
// приклади присвоювань
 {_ass,_d,64,_d,64,_d,64,"\n\tfstp%s"},
 {_ass,_ui|cdPtr,32,_ui|cdPtr,32,_ui|cdPtr,32,
                                    "\n\tmov\t%s,%s"},
 {_ass,_ui|cdPtr,32,_si|cdPtr,32,_ui|cdPtr,32,
                                  "\n\tmov\t%s,%s"},...
// приклади операцій відношень
 {_eq,_d,64,_f,32,_ui,32,
     "\n\tfcomp\t%s%s\n\tfstsw\tax\n\tsahf\n\tsete\ta
                                 l\n \tmovsx\teax,al"},
```

```
 {_eq,_d,64,_d,64,_ui,32,
     "\n\tfcomp\t%s%s\n\tfstsw\tax\n\tsahf\n\tsete\ta
                                  l\n\tmovsx\teax,al"},
 {_ne,_ui,32,_ui,32,_ui,32,
   "\n\tcmp\t%s%s\n\tsetne\tal\n\tmovsx\teax,al"},...
// приклади арифметичних операцій
 {_add,_ui,32,_d,64,_d,64,"\n\tfadd\t%s%s"},
 {_add,_ui,32,_ui|cdPtr,32,_ui|cdPtr,32,
        "\n\tmov\tcl,2\n\tshl\t%scl\n\tadd\teax,%s"},
 {_add,_ui,32,_si|cdPtr,32,_si|cdPtr,32,
     "\n\tmov\tcl,2\n\tshl\t%scl\n\tadd\teax,%s"},...
// приклади операцій індексації
 {_ixbz,_si|cdPtr|cdArr,32,_si,32,_si,32,
            "\n\tmov\tesi,%s\n\tmov\teax,%s[esi*4]"},
 {_ixbz,_f|cdPtr|cdArr,32,_ui,32,_f,32,
                "\n\tmov\tesi,%s\n\tfld\t%s[esi*4]"},
 {_ixbz,_f|cdPtr|cdArr,32,_si,32,_f,32,
                "\n\tmov\tesi,%s\n\tfld\t%s[esi*4]"},
 {_ixbz,_d|cdPtr|cdArr,32,_ui,32,_d,64,
             "\n\tmov\tesi,%s\n\tfld\t%s[esi*8]"},...
```

Для шаблона проекту цього заняття spLb9.dsp макроси повинні бути оформлені як рядки форматів з одним або декількома шаблонами заміщення `%s` у відповідності до використаного режиму обробки. Рядки задаються в декількох форматах.

```
"\n\tcmp\teax,0\n\tjz\t%s" // Переходи за цілими
"\n\tftst\n\tfstsw\tax\n\tsahf\n\tfincstp\n\tjz\t%s"
                // Умовні переходи за плаваючою точкою
"\n\tcmp\teax,0\n\tjz\t%s\n%s:"
```

```
   // Умовні переходи за цілими даними в постумовамах
"\n\tftst\n\tfstsw\tax\n\tsahf\n\tfincstp\n\tjz\t%s\n
%s:" // Умовні переходи за даними з плаваючою точкою
"\n\tadd\t%s,%s"  // Присвоювання цілих з додаванням
"\n\tfild\t%s\n\tfadd\t%s\n\tfistp\t%s"
 // Присвоювання даних з плаваючою точкою з додаванням
```

Шаблони макросів генерації кодів доцільно формувати у відповідності з командами абстрактної машини, побудованими для взаємодії з елементами архітектури абстрактної машини.

.

Використання уніфікованого рекурсивного алгоритму для генерації виконаваних кодів

Коди виклику `vp1=SmCdGen(nd->prvNd,1)` та `vp2=SmCdGen(nd-> pstNd,1)` використовуються в різних місцях рекурсивної процедури `SmCdGen`.

В тих випадках, коли оброблюються операції або оператори присвоювання, спочатку виконується генерація кодів правого аргументу операції `vp2`, а в інших випадках спочатку генеруються лівого аргументу `vp1`.

Так наприклад, для даних описаних як глобальна змінна `acc32` та локальні змінні `*vp1, *vp2` типу

```
union     gnDat
{int      _id;       // цілі 4 байти
 short _sd;       // цілі 2 байти
 char     _cd;       // цілі 1 байт
 float _fd;       // дійсні 4 байти
```

```
 void     *_pd;//покажчики на дані всіх типів 4 байти
 double   _dd;       // дійсні 8 байтів
 __int64  _i8;   // цілі 8 байтів
};
union gnDat acc32;
union gnDat *vp1, *vp2;
```

Для генерації кодів структура віртуальної машини повинна бути модифікована так, щоб раціонально використовувати spLb9.dsp, який зберігається в робочому просторі spLb1.dsw в папці spLb1, яка зберігає шаблони прототипів для всіх практичних занять змістовного модуля «Системне програмування-2».

Структура фрагментів генерації кодів машинних операцій

Основні типи арифметико-логічних операцій визначаються розміщенням аргументів та результату:

1) обидва аргументи в пам'яті;
2) перший аргумент в акумуляторі, другий – в пам'яті;
3) перший - в стеку , другий –в акумуляторі.

Структура програмного шаблона виконання роботи

Для спрощення розв'язання задачі слід використовувати прототип проекту цього практичного заняття spLb9.dsp, який зберігається в робочому просторі spLb1.dsw в папці spLb1, яка зберігає шаблони прототипів для всіх практичних занять циклу «Системне програмування і операційні системи». До складу проекту консольної прикладної програми, орієнтованої на можливість використан-

ня в числі інших бібліотек стандартних функцій входять модулі з наступними вхідними файлами заголовків і реалізацій:

- StdAfx.h і StdAfx.cpp – модуль для організації використання об'єктів класів MFC;
- langio.cpp – модуль для введення-виведення даних транслятора.
- tables.h і tables.cpp – модуль для опису і організації використання структур або класів таблиць;
- index.h і index.cpp – модуль для побудови впорядковуючого індексу для образів імен і констант;
- lexan.h і lexan.cpp – модуль для виконання лексичного аналізу;
- token.h і token.cpp – модуль для опису кодування типу лексеми, структури вузла лексеми і організації використання структур або класів лексем в процесі лексичного і синтаксичного аналізу;
- visgrp.h і visgrp.cpp – модуль для відображення рядків або повних таблиць;
- parsP.cpp, parsC.cpp і parsV.cpp – модулі констант для відтворення лексем мов Pascal, C/C++ та Verilog HDL, які використовують однойменні модулі проекту spLb3.dsp;
- spLb9.cpp – модуль контрольних прикладів для тестування функцій обробки графів.

Розділ визначень і декларацій основної частини програми тестування в модулі spLb9.cpp повинен включати виклик функцій

ініціалізації таблиць для потрібної мови їх елементів, спеціальні змінні, що зберігають довжину (тобто кількість елементів) таблиці. Розділ визначень модуля visgrp.cpp забезпечує визначення мов реконструкції за варіантом завдання. Виконувана частина модуля spLb9.cpp програми повинна включати циклічно повторювані виклики функцій для перевірки програми лексичного аналізу та відображення різних варіантів вхідних даних.

Завдання на роботу

Завдання на підготовку до роботи на комп'ютері:

1. Визначити варіант завдання для основних задач за таблицею 9.1. Визначити приклади лексем через файл в папці spLb9 модуля тестування spLb9.cpp.

2. Відповісти на контрольні запитання.

3. Підготувати настройки вхідної мови програмування.

4. Використати структуру елементу **`struct`** `lxNode` з файлу index.h шаблону програмного проекту spLb9 для побудови елементу індексу таблиць лексем і визначити.

5. Підготувати програмний модуль контрольної задачі, який виконує заданий варіант з таблиці 9.1 і дозволяє перевірити коректність виконання програм. Для цього організувати пошук в таблиці відповідності типів результатів типам операндів для семантичного аналізу.

Таблиця 9.1

Варіанти завдань для виконання аналізу з генерацію кодів

№ вар.	Вираз, який відтворюється в графі внутрішнього подання	Мова відтворення
1	**float** b, a, c, d; b=(2*a +c/d)*2*a;	C
2	**double** b, a[5]; **int** n; b:=(2*a+c)*2*a;	Pascal
3	**float** b, a[4]; **int** n; b+=a[n];	C
4	**float** b, a[3]; **int** n; n:=n-1; b:=b+a[n] **;**	Pascal
5	**float** b, a[5]; **char** n; b+=a[--n];	C
6	**double** b, a[4]; **char** n; b:=b+a[n]; n:=n-1 ; n=0;	Pascal
7	**double** a; **int** b; b=sin(2*a); b=2*a;	C
8	**double** b; **unsigned** a; b:=sin(2*a); b:=a;	Pascal
9	**float** b, d, a[5]; **int** c; b=c?d:2*a[n];	C
10	**double** b, a[4]; **unsigned** n,d; b:=d!=0**;** b:=2*a[n];	Pascal
11	**double** b, a[4]; **short** n,d; b=2*a[n]; b=d**;**	C
12	**float** b, a[3]; **unsigned** n,d; b:=2*a[n]; b:=d;	Pascal
13	**float** b, a[3]; **short** n,d; b=0;b+=a[n];	C
14	**float** b, a[5]; **short** n; b:=0; n:=n; b:= b+a[n];	Pascal
15	**float** a[3]={1.1, 3.14, -5} ,c; **int** b, n; --n; b=c==a[n]**;**	C
16	**float** b, a[3]; **long** n; n:=n-1; b:=0; b:=n!=a[n];	Pascal
17	**double** b, a[3]; **long** n; --n; b=n==a[n];	C
18	double a[3]; Boolean b, int n; b:=a-c=0; c:=(a-c)*2*a;	Pascal
19	**double** b, a[6]; **char** n; --n; b=n&&b==a[n];	C
20	**double** b, a[3]; **short** n; b=--n&&b==a[n]);	C
21	**float** b, a[2]; **long** n; b:=n>0 **and** b=a[n]; n:=n-1;	Pascal
22	**double** b, a[3]; **long** n; b=c<a; b=sin(2*a); b=2*a;	C

- .

Завдання на роботу на комп'ютері

6. Побудувати програмний проект, ввівши програмні модулі у відповідні файли проекту і налагодити синтаксис.

7. Побудувати виконуваний модуль тестової програми і налагодити змістовне виконання програми для перевірки результатів контрольних прикладів.

8. Одержати результати виконання у формі вихідного об’єктного файлу spLb9c.cpp в проекті spLb9c.dsp, проаналізувати виконання цього файлу в режимі налагодження і зробити висновки.

9. Продивившись в режимі дизасемблювання фрагменти операцій, що виконуються в замінити оператори мови C/C++ операторами вставок на мові Асемблера, приклад яких наведено в інтерпретації множення з фіксованою точкою.

10. Продемонструвати результати викладачам.

.

Контрольні запитання про особливості генерації кодів

1. Які особливості обробки основних типів даних виникають в генераторах кодів?

2. Яким чином забезпечується відповідність типів даних операндів та результатів операцій при генерації кодів?

3. Які поля повинні входити до таблиць генерації кодів при реалізації комп'ютерної мови?

4. Які дані результатів формуються на етапі генерації кодів?

5. Які поля структур лексем вузлів роздільників та ключових слів заповнюються на етапі генерації кодів?

6. Які дані включаються до таблиць генерації кодів?

7. Як організується рекурсивний алгоритм узагальненої семантичної обробки у застосуванні до генерації кодів?

8. Які особливості необхідно визначити для генерації кодів операцій присвоювання?

9. Які поля таблиць треба використовувати для визначення типів даних в операторах описів типів і примірників даних?

10. Які помилки можуть розпізнаватися при генерації кодів?

.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Бек Л.* Введение в системное программирование: Пер. с англ.- М.: Мир, 1988. - 448 с., ил.

2. *Проценко В.С., Чаленко П.Й., Ставровський А.Б.* Техніка програмування мовою Сі: Навчальний посібник. – К.: Либідь, 1993, 224 с.

3. *Пустоваров В.И.* Ассемблер: программирование и анализ корректности машинных программ. – К: BHV, 2000, 480 с.

4. *Пустоваров В.И.* Язык ассемблера в программировании информационных и управляющих систем. М:. "Энтроп", К: "Век", 1996, 304 с. – К.: Юниор, 1997. – 304 с.

5. *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии, инструменты: Пер. с англ. – М.: Издательский дом Вильямс, 2001. – 768 с.

6. *Стобо Дж.* Язык программирования Пролог: Пер. с англ. – М.: Радио и связь, 1993. – 386 с.

7. Metzger R.C., Zhaofang W. Automatic algorithm recognition and replacement: a new approach to program optimization / The MIT Press, Cambridge, 2000. 219 p.

8. Muchnick S.S. Advanced compiler design and implementation – San Francisco, Morgan Kaufmann Publishers, 1997. – 856 p.