В.Г. Левицький

БАЗОВІ КЛАСИ "СТЕК", "ХЕШ-ТАБЛИЦЯ" ТА "СКІНЧЕННИЙ АВТОМАТ" ДЛЯ ПРОГРАМНОГО КОМПЛЕКСУ "DSR OPEN LAB 1.0"

(Науковий керівник – кандидат технічних наук, доцент Колодницький М.М.)

Розглянуті вимоги до рівня базової бібліотеки структур даних. Представлена реалізація базових класів для структур даних, що використовувалися при побудові підсистеми компіляторів проблемно орієнтованих мов.

Вступ

Швидкий розвиток комп'ютерних інформаційних технологій (КІТ) формує нові вимоги до сучасного програмного забезпечення. Підтримка сучасних стандартів, врахування великої кількості машинних платформ та операційних систем, відкритість, що дозволяє миттєво реагувати на зміни технологій, стає неодмінною умовою успіху нових систем. В таких умовах лише максимальна машинна незалежність програмного коду, ефективне використання праці програміста дозволяє створювати конкурентноспроможне програмне забезпечення, що задовольняє всім вимогам сучасного рівня розвитку КІТ.

Прикладом можливого вирішення цієї проблеми є програмний комплекс (ПК) "DSR Open Lab 1.0", що розробляється на кафедрі ПЗОТ ЖІТІ [3, 4]. Необхідність оптимізації програмного коду за розміром та часом виконання, бажання уникнути багаторазової реалізації близьких задач і підвищити переносність програмного коду зумовило включення до ієрархії пакета рівень базової бібліотеки основних структур даних.

Бібліотека реалізована на мові C++ як сукупність базових шаблонів класів. При побудові переважної більшості структур даних як базові класи використовувалися шаблони бібліотеки STL (Standard Template Library), великого набору структур даних та алгоритмів, що є частиною ANSI/ISO Standard C++ Library.

Склад бібліотеки повністю визначається потребами пакета "DSR Open Lab 1.0" – до неї входять як структури даних загального вжитку, так і специфічні для деяких підсистем програмного комплексу. Далі розглянемо лише ті з них, що найбільш активно використовуються в підсистемі компіляторів ПК, а саме: стек, хеш-таблицю та скінченний автомат. Вимоги до цих структур даних зумовлюються тим фактом, що вони не лише повинні оптимізувати роботу підсистеми трансляторів, але й мати досить загальну структуру, яка дала б змогу використовувати їх в інших частинах програмного комплексу. Розглянемо особливості реалізації структур даних "стек", "хеш-таблиця" та "скінченний автомат".

Стек

Стек – це лінійний список, в якому будь-який доступ до елементів даних здійснюється лише в одному кінці списку [2]. Дана структура має досить широке застосування в різноманітних задачах і цілком задовільно реалізована в бібліотеці STL. Це дозволяє помістити до базової бібліотеки структур даних лише мінімальну надбудову стандартного класу "стек", яка б забезпечувала незалежність від інструментальних засобів розробки і оптимізувала б роботу компілятора за рахунок більш ефективної реалізації специфічних для процесу трансляції функцій (це, зокрема, операції одразу з декількома елементами стека, характерні для методу LR-розбору, який застосовується для синтаксичного аналізу описів математичних моделей у підсистемі компіляторів програмного комплексу "DSR Open Lab 1.0" [5]). Крім того, клас містить декілька функцій-синонімів і дозволяє використовувати будь-які з функцій-членів STL-класу стека, оскільки є його нащадком. Інтерфейс класу наведено нижче.

```
template <class T>
class CDSRStack : public stack<T> {
protected:
    void erase(deque<T>::iterator f, deque<T>::iterator l);
```

```
    void erase(deque<T>::iterator it);
public:
    void Push(const T& v);          // послати елемент до стека
    void Pop(void);                 // видалити елемент із стека
    void Pop(unsigned n);           // видалити n елементів із стека
    T& Top(void);                   // отримати верхній елемент
    const T& Top(void) const;       // отримати верхній елемент (константна реалізація)
    void Flush(void);               // очистити стек
};
```

Хеш-таблиця

Хеш-таблиця — це структура даних, в якій індекс елемента таблиці визначається видом цього елемента і отримується його хешуванням (виконанням над ним і, можливо, над його довжиною певних простих арифметичних чи логічних операцій) [1]. Простим прикладом хеш-таблиці можна вважати звичайний бібліотечний каталог — комірка таблиці визначається початковою літерою прізвища автора чи назви книги.

Ця структура даних традиційно застосовується в компіляторах, якщо під час трансляції необхідно зберігати інформацію про визначені функції користувача, змінні тощо. Така організація даних оптимізує роботу компілятора за часом, оскільки вдало реалізована хеш-таблиця дозволяє в разі необхідності знаходити необхідну інформацію практично за один крок, одразу визначаючи хешуванням індекс елемента.

Існує, однак, проблема колізій, тобто збігів результатів хешування двох різних елементів таблиці. І хоча вдало підібрана функція хешування, що враховує вид конкретної проблемно-орієнтованої мови та специфіку реалізації транслятора, значно зменшує ризик колізій, можливість несприятливого результату хешування потребує реалізації методів розрізнення і занесення до таблиці елементів з однаковими значеннями хеш-функцій.

Поставлену проблему можна вирішувати двома шляхами: рехешуванням та методом ланцюжків. У першому випадку при колізії намагаються визначити за певним правилом новий індекс для елемента, виконуючи над індексом, що виявив колізію, деякі арифметичні чи логічні дії. При лінійному рехешуванні, наприклад, новий елемент намагаються розташувати на початку таблиці; при рехешуванні додаванням — на позиції, кратній індексу, що викликав колізію; при випадковому рехешуванні — на позиції, яка однозначно залежить від попередньої, але випадковим чином розташована в таблиці.

Більш загальним методом вирішення колізій, який і застосовувався при побудові класу хеш-таблиці в базовій бібліотеці структур даних програмного комплексу "DSR Open Lab 1.0", є метод ланцюжків. Його суть полягає в тому, що в комірку хеш-таблиці заносять не сам елемент, а список елементів, що складається з елементів з однаковим значенням хеш-функції (рис. 1).

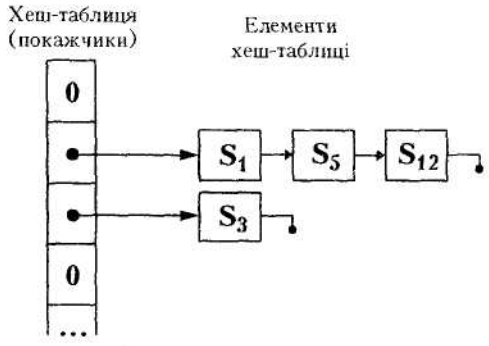

Рис. 1

Такий вибір методу вирішення колізій визначив реалізацію хеш-таблиці на основі шаблонів класів STL. Клас хеш-таблиці є вектором (vector), кожна комірка якого вказує на список (list) елементів чи містить нульовий покажчик, якщо елементи із відповідним значенням хеш-функції ще не з'являлись у таблиці чи були видалені з неї. Інтерфейс класу для хеш-таблиці наведено нижче.

```
template <class T>
class CDSRHashTable {
protected:
    vector<list<T>*>      data;
    unsigned long         table_size;          // розмір таблиці
    unsigned long         items_count;         // кількість елементів в таблиці
public:
    CDSRHashTable(unsigned long start_size = DSRDS_DEFAULT_HASH_TABLE_SIZE);
    unsigned long GetHashValue(const T& v) const;      // значення хеш-функції
    unsigned long TableSize(void) const;               // розмір таблиці
    unsigned long ItemsCount(void) const;                    //  кількість  елементів  в
таблиці
    void Add(const T& v);                              // помістити елемент до таблиці
    void Detach(const T& v);                           // видалити елемент
    T* Get(const T& v);                                // знайти елемент
    void Flush(void);                                  // очистити таблицю
};
```

Слід звернути увагу на реалізацію функцій `HashValue()` та `GetHashValue()`, що визначають індекс елемента в таблиці:

```
unsigned long HashValue(void*& v)    {      return (unsigned long)v;          }
template <class T>
unsigned long HashValue(const T& v)  {      return v.HashValue();             }
template <class T>
unsigned long CDSRHashTable<T>::GetHashValue(const T& v) const
                                     {      return HashValue(v) % TableSize(); }
```

Такий метод визначення індексу, очевидно, гарантує від виходу за рамки таблиці і дозволяє створювати хеш-таблиці як з елементів довільних класів, так і з покажчиків на змінні, використовуючи один і той самий набір функцій.

Метод ланцюжків вирішення колізій був вибраний для реалізації, оскільки він гарантує те, що до таблиці будь-якого розміру можна занести довільну кількість елементів, обмежену лише ресурсами конкретного комп'ютера. Слід зауважити, однак, що вибір розміру хеш-таблиці є досить важливим етапом при побудові цієї структури даних — ефективність пошуку в невеликих хеш-таблицях різко падає із зростанням кількості елементів, оскільки часті колізії призводять до лінійної часової складності пошуку. Під час створення підсистеми компіляторів проблемно-орієнтованих мов опису математичних моделей для програмного комплексу "DSR Open Lab 1.0" була вироблена така методика використання хеш-таблиць: розмір таблиць, склад яких змінюється динамічно, під час трансляції, слід вибирати окремо для кожного виду елементів, оцінивши можливу їх кількість, виходячи із специфіки конкретної проблемно-орієнтованої мови (бажано, щоб розмір таблиці був простим числом). Розмір таблиць, всі елементи яких відомі заздалегідь, визначається на основі аналізу всього спектра результатів хеш-функцій, що виключає можливість колізій і оптимізує транслятор за часом компіляції.

Скінченний автомат

Реалізація даної математичної структури має певні особливості у порівнянні з іншими структурами даних, оскільки в залежності від області застосовування означення скінченого автомата можуть трохи варіюватися. Вимоги до базового класу автомата у програмному комплексі "DSR Open Lab 1.0" формуються, виходячи з потреб аналізу цієї математичної структури в розділі "Дискретні логічні моделі" і автоматизації побудови компіляторів за рахунок конструювання сканера у вигляді сукупності скінченних автоматів [6]. Можна було б обмежитись одним лише загальним шаблоном класу детермінованого скінченного автомата, однак, зважаючи на одну з головних вимог трансляції — оптимальність обробки вхідних описів за часом, було прийнято рішення реалізувати в базовій бібліотеці структур даних два різних класи, один з яких відповідав би поняттю автомата, що використовується в математичній лінгвістиці та теорії компіляторів, а інший мав би найбільш загальні риси даної математичної структури (тобто, був би сукупністю множин вхідних та вихідних змінних, внутрішніх змінних станів автомата, функцій переходу та виходу). Таке вирішення проблеми загальності і оптимальності за часом програмного коду дозволило, з одного боку, зберегти можливість створення довільних модифікацій базового класу в рамках означення скінченного

автомата як множини п'яти підмножин, а з іншого — врахувати специфіку процесу трансляції.

Далі наведено інтерфейс найбільш загальної реалізації базового шаблона класу скінченного автомата:

```
template <class T_x, class T_y>
class CDSRAutomaton {
private:
    T_x             x_eof;          // ознака кінця послідовності значень вхідних змінних
protected:
    unsigned long   x_count,        // потужність множини вхідних змінних
                    y_count,        // потужність множини вихідних змінних
                    s_count,        // загальна кількість станів автомата
                    StateIst,       // початковий стан автомата
                    CurrentState,   // поточний стан автомата
                    PrevState;      // попередній стан автомата
public:
    CDSRAutomaton(unsigned long arg_xc, unsigned long arg_yc,
                  unsigned long arg_sc, unsigned long arg_1st = 1);
virtual unsigned long     GetJmp(T_x x, unsigned long s) = 0;  //     функція     переходу
автомата із
                                                               // одного стану в інший
virtual T_yGetOut(T_x x, unsigned long s) = 0;                 // функція виходу
    unsigned long   Step(T_x in, T_y& out);       // один такт роботи автомата
    int             EOF(T_x x);                   // перевірка на кінець послідовності
    int             EOF(void);                    // ознака кінця послідовності
                                                  // значень вхідних змінних
    int             Analisys(T_x *in_str,         // робочий цикл аналізу послідовності
                             T_y *out_str = 0);   // значень вхідних змінних
    void            SetEOF(T_x x);          // встановити нове значення ознаки
                                            // кінця послідовності значень вхідних змінних
};
```

Розглянемо детальніше базовий клас скінченного автомата, що використовується при лексичному аналізі текстів. Для нього заздалегідь відома множина допустимих значень вхідних змінних (символи з набору ASCII для даної реалізації транслятора), а функції вихідних змінних виконує віртуальна функція, визначена користувачем. З використанням бібліотечного класу скінченного автомата процес побудови сканера зводиться до виділення з всієї множини термінальних символів набору лексем порівняно складної структури і породження від базового класу автомата, що приймає дані ланцюжки символів. Така методика дозволила формалізувати побудову лексичного аналізатора як сукупності скінченних автоматів, що послідовно намагаються прийняти наступну вхідну лексему [5]. В процесі конструювання підсистеми компіляторів програмного комплексу "DSR Open Lab 1.0" були побудовані класи автоматів, що приймають дійсну та цілу беззнакову константу (автомат К1), ідентифікатор змінної спеціального виду, загальний оператор і строкову константу в стилі мови С (автомат К2).

1. Скінченний автомат для беззнакових констант дійсного та цілого типу (К1).

Лексеми «ціла константа» та «дійсна константа» визначаються такими правилами у формі Бекуса-Наура (БНФ):

<дійсне число> ::= <мантиса> E <порядок> | <мантиса> e <порядок> | <мантиса>

<мантиса> ::= <ціле число>.<ціле число> | <ціле число>. | .<ціле число> | <ціле число>

<порядок> ::= <ціле число> | + <ціле число> | −<ціле число>

<ціле число> ::= <цифра> | <ціле число> <цифра>

<цифра> ::= 0|1|...|9

Скінченний автомат К1 задається множинами S, Σ, F, δ:

- множина станів S = {S1, ..., S8};
- вхідний алфавіт $\sum = \{d, p, ., +, -\}$, де d – цифра; p – E або e;
- множина кінцевих станів F = {S2, S4, S8}, де S2 відповідає цілому числу, S4 – дійсному числу з фіксованою комою, S8 – дійсному числу з плаваючою комою;
- множина переходів δ представлена схематично на рис. 2 (кінцеві стани заштриховані);
- початковий стан – S1.

2. *Скінченний автомат для строкових констант* (К2).

Лексема «строкова константа» визначається такими правилами у формі БНФ (LF – перевід строки; CR – повернення каретки, all – довільні графічні символи ASCII):

<строкова константа> ::= " <символи>

<символи> ::= " | all <символи> | \ all <символи> |
\ LF CR <символи> | " LF CR " <символи>

Скінченний автомат К2 задається множинами S, Σ, F, δ:

- множина станів S = {S1, ..., S7};
- вхідний алфавіт $\Sigma = \{", \backslash, LF, CR, all\}$, де *LF* – перевід строки; *CR* – повернення каретки, *all* – довільні графічні символи ASCII крім «"» та «\»;
- множина кінцевих станів F = {S7};
- множина переходів δ представлена схематично на рис. 3 (кінцеві стани заштриховані);
- початковий стан – S1.

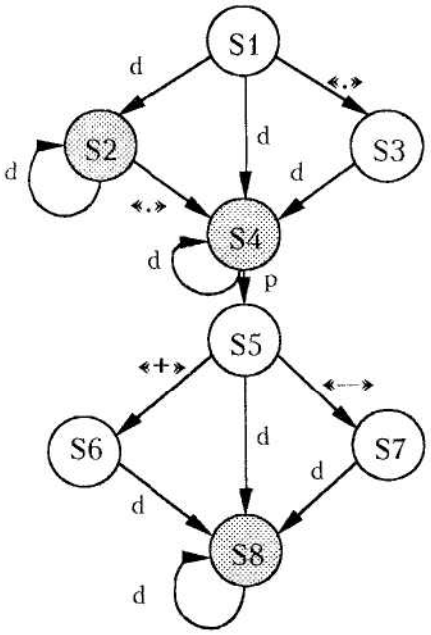

Рис. 2

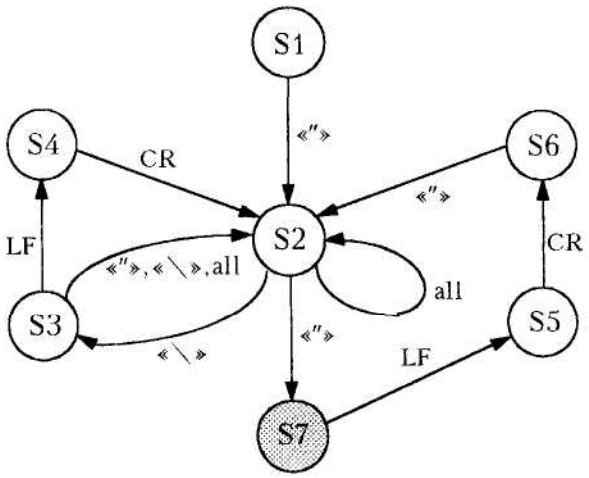

Рис. 3

Інтерфейс базового класу скінченного автомата для транслятора наведено нижче.

```
class CDSKLexAuto {
protected:
    unsigned        CurrentState,  // поточний стан автомата
                    PrevState,     // попередній стан автомата
                    nStates,       // загальна кількість станів автомата
                    nAlphabet      // кількість символів вхідного алфавіту
    unsigned        *Jmp,          // таблиця переходів
                    *EndStates;    // вектор обробки кінцевих станів автомата
    unsigned        read;          // кількість символів, що були прийняті
                                   // останнього разу
    char            character;     // поточний вхідний символ

  virtual    void          Deeds(void);                  // виконання довільних дій в
залежності
                                                        // від виду вхідного опису
  virtual    unsigned      What(char nextS) = 0;         // функція перекодування символа
вхідного
                                                        // алфавіту в індекс таблиці переходів
  public :
    CDSRLexAuto(unsigned anSt, unsigned anAB, unsigned *aJmp, unsigned *aEndStates );
    int             Analisys(char *aText);          // робочий цикл аналізу вхідного тексту
    unsigned        GetRead(void);                  // кількість символів, що були прийняті
                                                    // останнього разу
};
```

Переваги реалізації базової бібліотеки структур даних

Досвід створення програмного комплексу "DSR Open Lab 1.0" довів безсумнівну користь реалізації рівня базової бібліотеки структур даних у загальній ієрархії пакета. Відкритість та універсальність бібліотеки шаблонів класів гарантують простоту збереження відповідності сучасним технологіям і стандартам. Машинна незалежність та підвищення ефективності використання праці програміста за рахунок повторного використання програмного коду дають особливо високий виграш у часі розробки і забезпеченні надійності програм саме у разі конструювання великих пакетів, звільняючи розробників від необхідності дублювання схожих алгоритмів і трудомісткої модифікації програм при зміні вимог до програмного комплексу.

ЛІТЕРАТУРА:

1. *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин: Пер. с англ. – М.: Мир, 1975. – 544 с.
2. *Кнут Д.* Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы. – М.: Мир, 1976. – 736 с.
3. *Колодницкий Н.М.* Пакет программ "Dynamical systems research (DSR)" // Праці Житомирського філіалу КПІ. Серія А. Техніка. – Вип. 1. – Житомир: ЖФ КПІ, 1993. – С. 81–94.
4. *Колодницький М.М., Рожик О.А., Левицький В.Г., Шкаленко С.П., Гладишев А.В.* Програмний комплекс для моделювання динамічних систем "DSR LAB. 1.0" // Сучасні технології в аерокосмічному комплексі. Матеріали III Міжнародної наук.-практ. конференції, 9–11 вересня 1997 року. – Житомир: ЖІТІ, 1997. – С. 97–99.
5. *Колодницький М.М., Левицький В.Г., Рожик О.А.* Автоматизація побудови компілятора мови опису математичних моделей динамічних систем // Вісник ЖІТІ, 1997. – № 4. – С. 138–152.
6. *Оллонгрен А.* Определение языков программирования интерпретирующими автоматами: Пер. с англ. – М.: Мир, 1979. – 288 с.
7. *Хантер Р.* Проектирование и конструирование компиляторов: Пер. с англ. – М.: Финансы и статистика, 1984. – 232 с.

ЛЕВИЦЬКИЙ В'ячеслав Георгійович – студент 5-го курсу Житомирського інженерно-технологічного інституту.

Наукові інтереси:
- комп'ютерні інформаційні технології;
- моделювання і розв'язок задач за допомогою обчислювальної техніки;
- побудова компіляторів.