12. Метод часткових цілей

Метод часткових цілей застосовують для розв'язування складних нових задач, якщо нема відомих алгоритмів для схожих задач, і якщо врахувати одразу всі умови дуже важко. Цей метод передбачає зведення великої задачі до послідовності простіших. Звичайно ми сподіваємося, що простіші задачі легше розв'язати, ніж початкову, і що розв'язок початкової задачі можна отримати за допомогою розв'язування побудованих простіших задач.

Застосування методу часткових цілей на практиці є досить складним. Адже виокремлення простішої задачі справа скоріше інтуїції, ніж науки. Такому виокремленню можуть допомогти відповіді на такі запитання.

1. Чи можна розв'язати тільки частину задачі? Чи можна розв'язати всю задачу, ігноруючи деякі умови?
2. Чи можна розв'язати задачу для часткових випадків: створити алгоритм, що виконує усі вимоги до задачі тільки для деякої підмножини вхідних даних?
3. Чи добре ми зрозуміли постановку задачі? Чи не допоможе нам глибше розуміння деяких її особливостей?

12.1. Польський запис арифметичного виразу

Продемонструємо застосування методу часткових цілей на прикладі побудови *польського запису* заданого арифметичного виразу. Ми звикли записувати арифметичні вирази в *інфіксній* формі, коли знак операції записується *між* операндами (для бінарних операцій). Обчислення значеня виразу, записаного в такій формі, передбачає його багаторазовий перегляд: щоб обчислити всі частини виразу, записані в дужках, тоді виконати множення та ділення, і наприкінці – додавання та віднімання.

Багатократного перегляду виразу можна уникнути, якщо записати його у *суфіксній* або *префіксній* формах. Інверсним польським записом називають суфіксну (постфіксну) форму, за якою спочатку записують обидва операнди, а потім – знак операції. Такий спосіб дає змогу обходитись без дужок і обчислювати значення виразу за один перегляд його зліва-направо, а тому широко використовується різноманітними компіляторами.

Задача 48. *У стандартному вхідному потоці задано правильний запис деякого арифметичного виразу в інфіксній формі. (В арифметичних виразах використовують знаки чотирьох операцій: +, –, ×, /; круглі дужки для зміни порядку виконання операцій; операндами можуть бути цілі або дійсні числа.) Перетворити його до постфіксної форми.*

Як передбачити усі можливі варіанти розташування дужок та знаків операцій? Як розпізнати операнди, що можуть бути цілими чи дійсними числами? Задача не з простих. Розв'язати її одразу, мабуть, не вдасться, тому застосуємо метод часткових цілей і пошукаємо відповіді на сформульовані раніше запитання. Перше з них у пригоді нам не стане: важко собі уявити, як перетворити лише частину виразу, і як це може допомогти отримати розв'язок задачі. Третє запитання програмістові необхідно пам'ятати постійно. Воно буде корисним під час розв'язування будь-якої задачі, і не лише за допомогою методу часткових цілей.

Давайте скористаємось другим з перелічених запитань і полегшимо своє завдання, вилучивши з розгляду частину можливих варіантів вхідних даних. Спочатку побудуємо алгоритм переведення, що працюватиме для деяких простих виразів, а потім, поступово вдосконалюючи його, спробуємо отримати алгоритм переведення довільного арифметичного виразу.

Що можна спростити у вхідних даних? Розпізнавання операндів – багатоцифрових цілих чи дійсних – є дещо самостійною задачею. Її можна розглядати незалежно від побудови основного алгоритму. Тому обмежимося випадком, коли операнди є цілими

одноцифровими числами. Наявність у виразі дужок змінює пріоритети операцій, а це ускладнює алгоритм побудови польського запису. Тому наступне спрощення полягатиме в тому, щоб розглядати вирази без дужок. І, нарешті, домовимося, що у виразі є лише бінарні операції. Унарні «+» чи «–» розглянемо пізніше.

Отже, запис нашого «простого» арифметичного виразу може містити цифри і знаки «+»,«–», «*», «/» (у програмах замість знаку «×» для позначення множення використовують «*»), причому знаки завжди стоять між цифрами, наприклад: '3+7–2*4/5'. Польський запис цього виразу має вигляд '37+24*5/–'. Бачимо, що після перетворення знаки операцій пропустили поперед себе цифри-операнди. Знак «+» пропустив цифру «7», знак «*» – цифру «4», а знак «–» – взагалі усіх: і операнди, і знаки старших за пріоритетом операцій.

Припустимо, що інфіксний запис заданого виразу зберігається у рядку *source*, а польський запис виразу необхідно побудувати у рядку *dest*. Щоб знак операції міг пропустити поперед себе інші літери, його потрібно десь тимчасово зберігати. Найкраще для цього використати стек. Якщо до виразу входять операції тільки однакового пріоритету, то до стека заносять знак операції тільки для того, щоб поміняти його місцями з другим операндом. Якщо ж у виразі є операції різних пріоритетів, то знак молодшої операції потрібно зберігати у стекові доти, доки до *dest* не буде перенесено операнди та знаки старших операцій.

Тепер сформулюємо алгоритм побудови рядка *dest*. Він полягає у послідовному перегляді літер рядка *source*. Залежно від значення чергової літери, її обробляють по-різному:

- якщо значенням є цифра, то її заносять у *dest*;
- якщо значенням є '+' чи '–', то аналізують стек: якщо він порожній, то просто заносять до нього літеру; а якщо непорожній, то вміст стека переносять до *dest* і заносять до стека літеру;
- якщо значенням є ' * ' чи '/', то аналізують стек: якщо він порожній, то заносять до нього літеру; якщо ні, то переносять знак операції з вершини стека до *dest*, якщо він є ' * ' чи '/', і записують літеру до стека.

Після аналізу усіх літери рядок *source* вміст стека переносять до *dest*.

Ми не будемо витрачати час на написання власного стека, а використаємо той, що входить до бібліотеки STL. Нагадаємо, що це шаблонний контейнер. Ми конкретизуватимемо його типом *char*, щоб можна було зберігати окремі літери рядка. Заносять елементи до *std::stack<char>* методом *push(char)*, вилучають – методом *pop()*, а читають значення з вершини – методом *top()*. Особливість отримання даних з такого стека в тому, що забирають елемент з вершини стека за два кроки: спочатку читають, а тоді вилучають.

Тепер описаний алгоритм перетворення виразу можна реалізувати такою програмою:

```
// Перетворює прості арифметичні вирази з інфіксної форми до постфіксної
// операндами можуть бути одноцифрові натуральні числа
// операціями - бінарні + - * /
void postfixLight()
{
    stack<char> Stack;
    string source, dest;
    cout << "Задайте арифметичний вираз: ";
    getline(cin, source);
    for (int i = 0; i < source.length(); ++i)
    { // аналізуємо кожну літеру заданого рядка
        switch (source[i])
        {
        case '0': case '1': case '2': case '3': case '4':
```

```
        case '5': case '6': case '7': case '8': case '9':
            dest += source[i]; break; // операнд -> результуючий рядок
        case '+': case '-':
            while (!Stack.empty())       // усі знаки операцій зі стека -> в dest
            {
                dest += Stack.top();
                Stack.pop();
            }
            Stack.push(source[i]); break; // новий знак в стек
        case '*': case '/':  // зі стека в dest операції рівного пріоритету
            while (!Stack.empty() && (Stack.top() == '*' || Stack.top() == '/'))
            {
                dest += Stack.top();
                Stack.pop();
            }
            Stack.push(source[i]); break; // новий знак в стек
        default:
            throw std::invalid_argument("Illegal symbol in the source");
        }
    }
    while (!Stack.empty()) // решта операцій до результуючого рядка
    {
        dest += Stack.top();
        Stack.pop();
    }
    cout << "Польський запис: " << dest << '\n';
}
```

Початок покладено! Тепер справа за розвитком і поступовим удосконаленням програми. Навчимо її опрацьовувати вирази з дужками.

Під час обчислення арифметичного виразу передусім необхідно виконати операції в дужках, тому поява у записі виразу літери «(» означає початок частини виразу з вищим пріоритетом. Усі знаки операцій, які на цей момент перебувають у стекові, мусять там залишатися доти, доки не закінчиться пріоритетна частина виразу. А закінчується вона літерою «)». Щоб досягти бажаного ефекту, можна заносити до стека дужку, яка відкривається, і тримати її там (а під нею і всі знаки операцій, що були в стеку на момент початку пріоритетної частини) аж доти, доки не буде опрацьовано дужку, яка закривається.

Це хороше рішення, проте воно впливає на спосіб виконання алгоритму, описаного раніше. Значення чергової літери '+' чи '–' означало у ньому перенесення вмісту непорожнього стека у *dest*. Те, що було правильним для виразу без дужок, не завжди спрацьовує для виразу з дужками. Літера '(' у стеку розмежовує знаки операцій звичайної та пріоритетної частин виразу. Тому, опрацьовуючи '+', необхідно вилучати літери зі стека до появи у його вершині '(' або доти, доки він не спорожніє. Так само удосконалюють опрацювання знаків інших операцій.

Бачимо, що кількість перевірок катастрофічно зростає, внаслідок чого алгоритм поступово заплутується. Необхідно його спростити. З цією метою припишемо знакам операцій і дужкам числові значення їхніх пріоритетів. Використаємо також ще одну хитрість: щоб однаково опрацьовувати порожній та непорожній стек, занесемо у вершину порожнього стека деяку спеціальну літеру (наприклад, '#') і припишемо їй такий самий пріоритет, як і дужкам:

літера	пріоритет
'*', '/'	2
'+', '–'	1
'#', '(', ')'	0

Тепер у перевірках можна використовувати числові значення пріоритетів і сформулювати удосконалений алгоритм перетворення виразу. Щоб побудувати рядок *dest*, послідовно перебирають і аналізують літери рядка *source*. Залежно від значення чергову літеру обробляють по-різному:

- якщо значенням є цифра, то її заносять у *dest*;
- якщо значенням є знак операції, то вилучають зі стека і заносять до *dest* літери доти, доки їхній пріоритет більший чи дорівнює пріоритетові чергової літери; знак операції заносять до стека;
- якщо значенням є '(', то її заносять до стека;
- якщо значенням є ')', то вилучають зі стека і заносять до *dest* усі літери, які є у стекові над '('; вилучають зі стека '('.

Після завершення аналізу всіх літер рядка *source*, вміст стека (окрім спеціальної літери '#') переносять до *dest*. (Легко переконатися, що за цим алгоритмом ні знак операції, ні закриваюча дужка не «виштовхують» літеру '#' зі стека.)

Щоб спростити вигляд інструкції *switch*, для розпізнавання цифр використаємо стандартну функцію *isdigit*. «Розумніший» алгоритм реалізує нова програма.

```
// обчислення пріоритету операцій
int priority(char c)
{   switch (c) {
    case '*': case '/': return 2;
    case '+': case '-': return 1;
    case '#': case '(': case ')': return 0;
    default: return 10;
    }
}
// Перетворює арифметичні вирази з дужками. Операнди одноцифрові.
void postfix()
{   string source, dest;
    cout << "Задайте арифметичний вираз: ";
    getline(cin, source);
    stack<char> Stack; Stack.push('#');
    for (char c: source)
    {   // аналізуємо кожну літеру заданого рядка
        if (isdigit(c))
            dest += c; // операнд -> результуючий рядок
        else
        {   int p = priority(c);
            switch (c)
            {
            case '+': case '-': case '*': case '/':
                while (priority(Stack.top()) >= p)
                {           // старші операції полишають стек
                    dest += Stack.top(); Stack.pop();
                }
                Stack.push(c); break;
            case '(':       // початок пріоритетної частини
                Stack.push(c); break;
            case ')':       // закінчення пріоритетної частини
                while (Stack.top() != '(')
                {
                    dest += Stack.top(); Stack.pop();
                }           // дужку теж треба забрати
                Stack.pop(); break;
```

```
            } /* switch */
        } /* if */
    } /* for */
    while (Stack.top() != '#')
    {   // решта операцій до результуючого рядка
        dest += Stack.top(); Stack.pop();
    }
    cout << "Польський запис: " << dest << '\n';
}
```

Це вже досить «розумний» алгоритм. Він правильно перетворює вирази з довільними послідовностями арифметичних операцій і з довільним вкладенням дужок. Навчимо його тепер опрацьовувати знаки унарних операцій. З унарним знаком '+' все просто: його необхідно вилучити з рядка *source*, бо він нічого не змінює у виразі. Унарний мінус можна опрацювати двома різними способами: використати для його позначення спеціальний знак і приписати йому найвищий пріоритет (числове значення 3), або перетворити його в бінарний, вставивши перед ним в рядок *source* літеру '0'. Другий спосіб дещо простіший, його і реалізуємо.

Для повного розв'язання задачі 48 необхідно ще навчити алгоритм опрацьовувати багатоцифрові операнди. Це нескладно зробити, бо тепер просто доведеться пересилати з *source* до *dest* не одну цифру, а групу цифр. Деяке ускладнення пов'язане тільки з тим, що тепер треба буде якось розділяти операнди в рядку *dest*. Використаємо для цього літеру ' ', яку додаватимемо після кожного операнда заданого арифметичного виразу.

Кожне дописування літери до рядка-результату спричиняє його перебудову. Це досить затратно. Простіше буде використати для побудови перетвореного виразу потік виведення в рядок. Отримаємо програму *postfixExt*:

```
// перевіряє, чи належить символ до запису числа
bool isDigital(char c)
{   return '0' <= c && c <= '9' || toupper(c) == 'E' || c == '.';
}
// Перетворює арифметичні вирази з багатоцифровими операндами
string translate(string source)
{   // перетворимо знаки унарних операцій у рядку source
    //  спочатку - на початку рядка:
    if (source[0] == '-') source = '0' + source;  // мінус став бінарним
    else if (source[0] == '+') source.erase(0, 1);// вилучили унарний плюс
    //  потім - після кожної дужки (; перетворення такі ж
    int i = 1;
    while (i < source.length())
    {   if (source[i - 1] == '(')
        {   if (source[i] == '-')
            {   source.insert(i, "0"); ++i;   }
            else if (source[i] == '+') source.erase(i, 1);
        } ++i;
    }
    // приготували місце для перетвореного виразу
    std::ostringstream dest;
    stack<char> Stack; Stack.push('#');
    // аналізуємо кожну літеру заданого рядка
    for (int i = 0; i < source.length(); )
    {   if (isDigital(source[i]))
        {   while (i < source.length() && isDigital(source[i]))
                dest << source[i++];
            dest << ' ';  // завершили операнд
        }
```

```
        else
        {   char c = source[i];
            switch (c)
            {
            case '+': case '-': case '*': case '/':
            {   int p = priority(c);
                while (priority(Stack.top()) >= p)
                {           // старші операції полишають стек
                    dest << Stack.top();
                    Stack.pop();
                } Stack.push(c); break;
            }
            case '(':       // початок пріоритетної частини
                Stack.push(c); break;
            case ')':       // закінчення пріоритетної частини
                while (Stack.top() != '(')
                {   dest << Stack.top();
                    Stack.pop();
                }           // дужку теж треба забрати
                Stack.pop(); break;
            default:
                throw std::invalid_argument("Illegal symbol in the source");
            } /* switch */ ++i;
        } /* if */
    } /* for */
    while (Stack.top() != '#')
    {   // решта операцій до результуючого рядка
        dest << Stack.top();
        Stack.pop();
    }
    return dest.str();
}
void postfixExt()
{   string source;
    cout << "Задайте арифметичний вираз: ";
    getline(cin, source);
    try { cout << "Польський запис: " << translate(source) << '\n'; }
    catch (std::invalid_argument& exc)
    {   cout << exc.what() << '\n';   }
}
```

Задачу розв'язано: програма *postfixExt* виконує всі сформульовані в умові вимоги. Проте межі досконалості немає! Програма *postfixAdvanced* могла б виявляти помилки у записі виразу, опрацьовувати не тільки арифметичні, але й алгебричні вирази, до запису яких належать піднесення до степеня, виклики стандартних математичних функцій тощо. Побудову такої програми, сподіваємося, виконають уже наші читачі.

12.2. Обчислення значення постфіксного виразу

Було б неправильно не показати, як відбувається обчислення арифметичного виразу, записаного в постфіксній формі. Під час перетворення інфіксного запису в постфіксний знаки операцій «переховуалися» в стеку, щоб пропустити поперед себе операнди. Тепер операції і операнди поміняються ролями: до стеку потрібно заносити операнди. Справді, у постфіксному записі бінарна операція має вигляд *лівий_операнд правий_операнд знак_операції*. Алгоритм її обчислення можна сформулювати так:

1) *лівий_операнд* занести в стек;
2) *правий_операнд* занести в стек;
3) розпізнати *знак_операції*;
4) дістати операнди зі стека, виконати операцію;
5) результат занести в стек.

Наприклад, вираз «2+3*4» у постфіксному записі матиме вигляд «234*+». Щоб його обчислити потрібно виконати таку послідовність кроків:

- 2 – операнд? так – в стек;
- 3 – операнд? так – в стек;
- 4 – операнд? так – в стек;
- * – операнд? ні:
  - дістати 4, дістати 3, виконати 3*4=12;
  - 12 – в стек;
- + – операнд? ні:
  - дістати 7, дістати 2, виконати 2+12=14;
  - 14 – в стек.

Результат обчислення виразу зберігатиметься у вершині стека.

Алгоритм обчислення реалізує функція *Calculate*. Для простоти вважатимемо, що операнди – одноцифрові цілі.

```
int Calculate(string expr)
{
	stack<int> Stack;
	for (char c: expr)
	{
		if (isdigit(c))
		{
			Stack.push(c - '0');
		}
		else
		{
			int right = Stack.top(); Stack.pop();
			int left = Stack.top(); Stack.pop();
			switch (c)
			{
			case '+': Stack.push(left + right); break;
			case '-': Stack.push(left - right); break;
			case '*': Stack.push(left * right); break;
			case '/': Stack.push(left / right); break;
			}
		}
	}
	return Stack.top();
}
```

Наприклад, обчислення значення полінома $x^4 + 3x^3 - 5x^2 + x - 7$ в точці $x = 2$ можна записати виразом «(((2+3)*2–5)*2+1)*2–7». Випробуємо на ньому нашу програму.

```
Задайте арифметичний вираз: (((2+3)*2–5)*2+1)*2–7
Польський запис: 23+2*5-2*1+2*7-
Значення = 15
```