1. ПРОГРАММИРОВАНИЕ РЕКУРСИВНЫХ АЛГОРИТМОВ

Цель: При освоении этого раздела студент должен познакомиться с основными понятиями и приемами рекурсивного программирования, получить навыки программирования рекурсивных процедур и функций на языке программирования C++.

Более подробно материал этого раздела представлен в [1, 8].

Рекурсивным называется объект, содержащий сам себя или определенный с помощью самого себя.

Мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания. Точно так же бесконечные вычисления можно описать с помощью конечной рекурсивной программы. Рекурсивные алгоритмы лучше всего использовать, когда решаемая задача, вычисляемая функция или обрабатываемая структура данных определены с помощью рекурсии.

Если процедура (функция) P содержит явное обращение к самой себе, она называется прямо рекурсивной. Если P содержит обращение к процедуре (функции) Q, которая содержит (прямо или косвенно) обращение к P, то P называется косвенно рекурсивной.

Многие известные функции могут быть определены рекурсивно. Например факториал, который присутствует практически во всех учебниках по программированию, а также наибольший общий делитель, числа Фибоначчи, степенная функция и др.

Рассмотрим использование рекурсии при программировании алгоритмов вычисления степенной функции.

1.1. Вычисление степенной функции

Степенная функция $f(a,n)=a^n$ (основание степени a – например, вещественное число, а показатель степени n – целое неотрицательное число):

$f(a,n) \equiv$ **if** $n = 0$ **then** 1 **else** $f(a,n-1)* a$,

(1.1)

или другой вариант

$f(a,n) \equiv$ **if** $n = 0$ **then** 1 **else** $(f^2(a, n$ **div** $2))*f(a, n$ **mod** $2)$

(1.2) Учитывая, что n **mod** 2 может принимать только значения 0 или 1, а $f(a,0) = 1$ и $f(a,1) = a$, получим еще один вариант.

$f(a,n) \equiv$ **if** $n =0$ **then** 1 **else if** Even(n) **then** $(f^2(a, n$ **div** $2))$ **else** $(f^2(a, n$ **div**

2)) *a, (1.3)*

где Even(*n*) – функция, возвращающая значение True, если n – четное.

Последнему определению (1.3) соответствует приведенная ниже программа, использующая рекурсивную функцию power.

```
//  Пример: рекурсивное вычисление степени a^n
//  вариант 1
#include <iostream>
#include <windows.h>
using namespace std ;
int main ( )
{       //для правильной кодировки русских букв:
        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);

        float power (float, unsigned int);
        float a, y;
        unsigned int n;
        cout << "Введите вещественное a:" << endl ;
        cin >> a;
        cout << "Введено a:" << a << "\n" ;
        cout << "Введите показатель степени n >= 0 :" << endl ;
        cin >> n;
        cout << "Введено n >= 0 :" << n << "\n" ;
//{n >= 0}
        y = power(a, n);
//(y = a^n)
        cout << "Конец рекурсивного вычисления -\n" ;
        cout << "Вычислено " << a << "^" << n << " = " << y << " \n" ;
        return 0 ;
}
float power ( float a, unsigned int n)
{       float p;
        if (n == 0) return 1;
        else    {
                        p = power (a, n/2);
                        p = p*p;
                        if ( n % 2) p = p*a;
                        return p;
        };
}
```

Возможны и другие варианты рекурсивной реализации функции power.

Вариант 2.

```
float power2 ( float a, unsigned int n)
{       float p;
        if (n == 0) return 1;
        else{
                p = power2 (a*a, n/2);
```

```
                if ( n % 2) p = p*a;
                return p;
        };
}
```

Вариант 3. С накопителем.

```
float power3 ( float a, unsigned int n, float b)
// accumulator b
{       float p;
        if (n == 0) return b;
        else{
                if ( n % 2) p = power3 (a, n-1, b*a);
                else p = power3 (a*a, n/2, b);
                return p;
        };
}
```

Параметр b хранит значение степени на предыдущем шаге.

Вариант 4. С использованием «хвостовой» рекурсии

```
float power4 ( float a, unsigned int n, float b)
// accumulator + tail recursion !
{
        if (n == 0) return b;
        else{
                if ( n % 2) {b = b*a; n = n-1;}
                else {a = a*a; n = n/2;}
                return power4 (a, n, b);
        };
}
```

Вызов power4 осуществляется на последнем шаге функции, при возвращении ее значения. Если в версии 3 использовались обращения к функции с разными значениями параметров в зависимости от четного или нечетного значения n на очередном шаге, то в версии 4 параметры вычисляются по-разному, но обращение к функции одинаковое. От этой версии уже 1 шаг до итеративной функции, приведенной ниже.

```
float power5 ( float a, unsigned int n) // not recursion
{       float b = 1;
        while(n!=0) {
                if (n % 2) {b = b*a; n = n-1;}
                else {a = a*a; n = n/2;}
        }
        return b;
}
```

1.2. Взаимно-рекурсивные функции и процедуры. Синтаксический анализатор понятия скобок

Рассмотрим пример взаимной рекурсии, когда P обращается к Q, а Q к P. Пусть требуется построить синтаксический анализатор понятия скобки:

скобки::=квадратные | *круглые*
квадратные::=[*круглые круглые*] | +
круглые::=(квадратные квадратные) | –

В этом рекурсивном определении последовательности символов, называемой скобки, присутствуют две взаимно-рекурсивные части: квадратные определяются через круглые, и наоборот, круглые – через квадратные. В простейшем случае квадратные есть символ «+», а круглые есть символ «–». Другие примеры последовательностей, порождаемых этим рекурсивным определением:

‘[– –]’, ‘(++)’, ‘[(++)([–(++)][– –])]’, ‘(+[(++)([–(++)][(+[– –])–])])’.

Синтаксическим анализатором будем называть программу, которая определяет, является ли заданная (входная) последовательность символов скобками или нет. В случае ответа «нет» сообщается место и причина ошибки.

Реализуем основную часть этой программы как булевскую функцию Bracket, которая вызывает две другие (парные) булевские функции Round и Square, определяющие, является ли текущая подпоследовательность частью круглые или квадратные соответственно. Каждая из функций Round и Square в свою очередь вызывает парную к себе (Square и Round соответственно). Входная последовательность читается из файла. Вспомогательные сообщения квалифицируют ошибки в записи последовательности скобки в том случае, когда результат функции Bracket есть False. Для формирования этих сообщений будет использована процедура Error.

Функции Round и Square реализованы так, что они читают очередной символ входной последовательности и далее действуют в прямом соответствии с рекурсивными определениями частей круглые и квадратные соответственно. При этом в функции Bracket приходится читать первый символ входной последовательности дважды. Можно было бы избежать этого, используя «заглядывание вперед», однако такая реализация менее прозрачна.

```
// Program SyntaxAnalysisOfBracket;
// вариант с синхронным выводом входной строки (до места ошибки включительно)
```

```
/*      Определения (синтаксис)
        Bracket = скобки, Round = кругл, Square = квадр
        скобки ::= квадр | кругл
        квадр ::= + | [кругл кругл]
        кругл ::= – | (квадр квадр)
*/
#include <iostream>
#include <fstream>
#include <iomanip>
#include <windows.h>
using namespace std ;

bool Bracket(ifstream &infile);
bool Round (ifstream &infile, char s);
bool Square (ifstream &infile, char s );
void Error (short k);

int main ( )
{
        setlocale (0,"Rus");                    // для MVC++ 2010

        ifstream infile ("in_seq5.txt");
        if (!infile) cout << "Входной файл не открыт!" << endl;

        cout << "Анализатор скобок:" << endl;
        bool b = Bracket (infile);

        cout << endl;
        if (b) cout << "ЭТО СКОБКИ!" << endl;
        else  cout <<"НЕТ, ЭТО НЕ СКОБКИ!" << endl;

        system("PAUSE");
        return 0;
}

bool Round (ifstream &infile, char s)
// кругл ::= – | (квадр квадр)
// s   — текущий символ входной строки
{       bool k;
        if (s == '-') { return true;}
        else if ( s == '(' )
                {       //кругл ::= (квадр квадр)
                        if (infile >> s)
                        {       cout << s;
                                k = Square (infile,s);
                                if (k)
                                {       if (infile >> s)
                                        {       cout << s;
                                                k = Square (infile,s);}
                                        else {Error (5); return false;} // квадр - пуст!
                                }
                                else return false; //первый квадр ошибочен
```

```
                                        if (k) // оба квадр правильны
                                                if (infile >> s)
                                                {       cout << s;
                                                        return (s == ')');
                                                }
                                                else {Error (6); return false;}
                                        else return false;
                                }
                                else { Error (5); return false;} // квадр   — пуст!
                        }
                        else  { Error(7); return false;} // не – и не ( }
}
// end of Round

bool Square (ifstream &infile, char s)
// квадр ::= + | [кругл кругл]
// s - текущий символ входной строки
{       bool k;
        if (s == '+') return true;
        else if ( s == '[' )
        {       //квадр ::= [кругл кругл]
                if (infile >> s)
                {       cout << s;
                        k = Round (infile,s);
                        if (k)
                        {       if (infile >> s)
                                {       cout << s;
                                        k = Round (infile,s);
                                }
                                else {Error (8); return false;} // кругл   — пуст!
                        }
                        else return false; //первый кругл ошибочен

                        if (k) // оба кругл правильны
                                if (infile >> s)
                                {       cout << s;
                                        return (s == ']');
                                }
                                else {Error (3); return false;}
                        else return false;
                }
                else { Error (8); return false;} // кругл  — пуст!
        }
        else  { Error(4); return false;} // не + и не [ }
}
// end of Square


bool Bracket(ifstream &infile)
{       char s;
        bool b;
```

```
        b = false;
        if (infile >> s)
        {       cout << s;
                if ((s == '+') || (s == '[')) b = Square (infile, s);
                else    if ((s == '-') || (s == '(')) b = Round (infile, s);
                else Error(2);   //недопустимый начальный символ
                infile >> s;
                if (b && !infile.eof())  Error(1);       // лишние символы
                b = (b && infile.eof());
        }
        else Error (0);           // пустая входная строка
return b;
}

void Error (short k)
{
        cout << endl << "err#" << k << endl;
 switch (k) {
        case 0: cout << "! - Пустая входная строка" << endl; break;
        //{Bracket}
        case 1: cout << "! - Лишние символы во входной строке" << endl; break;
        //{Bracket}
        case 2: cout << "! - Недопустимый начальный символ" << endl; break;
        //{Bracket}
        case 3: cout << "! - Отсутствует ']'." << endl;       break;
        //{Square}
        case 4: cout << "! - Отсутствует '+' или '['." << endl;       break;
        //{Square}
        case 5: cout << "! - Очередной квадр — пуст." << endl; break;
        //{Round}
        case 6: cout << "! - Отсутствует ')'." << endl;       break;
        //{Round}
        case 7: cout << "! - Отсутствует – или (." << endl; break;
        //{Round}
        case 8: cout << "! - Очередной кругл — пуст." << endl; break;
        //{Square}
        default : cout << "! - ...";break;
                // ?
 };
}
// end of Error
```

1.3. Требования и рекомендации к выполнению задания:

Прежде чем приступить к выполнению задания этого раздела, рекомендуется ознакомиться с разделом 2 учебного пособия [1].

Задания к разделу Рекурсия представлены в **Приложении 1**. При выполнении задания необходимо:

1. проанализировать полученное задание, выделив рекурсивно определяемые информационные объекты и (или) действия;
2. разработать программу, использующую рекурсию;
3. сопоставить рекурсивное решение с итеративным решением задачи;
4. сделать вывод о целесообразности и эффективности рекурсивного решения данной задачи.

ПРИЛОЖЕНИЕ 1. ЗАДАНИЯ К РАЗДЕЛУ РЕКУРСИЯ

1. Для заданных неотрицательных целых n и m вычислить (рекурсивно) биномиальные коэффициенты, пользуясь их определением:

$$C_n^m = \begin{cases} 1, \text{если } m = 0, n > 0 \text{ или } m = n \geq 0, \\ 0, \text{если } m > n \geq 0, \\ C_{n-1}^{m-1} + C_{n-1}^{m} \text{ в остальных случаях.} \end{cases}$$

2. Задано конечное множество имен жителей некоторого города, причем для каждого из жителей перечислены имена его детей. Жители X и Y называются *родственниками*, если (а) либо X – ребенок Y, (б) либо Y – ребенок X, (в) либо существует некоторый Z, такой, что X является родственником Z, а Z является родственником Y. Перечислить все пары жителей города, которые являются родственниками.

3. Имеется n городов, пронумерованных от 1 до n. Некоторые пары городов соединены дорогами. Определить, можно ли попасть по этим дорогам из одного заданного города в другой заданный город. Входная информация о дорогах задается в виде последовательности пар чисел i и j ($i < j$ и $i, j \in 1..n$), указывающих, что i-й и j-й города соединены дорогами.

4. Напечатать все перестановки заданных n различных натуральных чисел (или символов).

5. Функция $f(n)$ определена для целых положительных чисел:

$$f(n) = \begin{cases} 1, & \text{если } n = 1, \\ \sum_{i=2}^{n} f(n \text{ div } i), & \text{если } n \geq 2. \end{cases}$$

Вычислить $f(k)$ для k= 15, 16,…, 30.

6. Построить синтаксический анализатор для понятия *простое выражение.*

простое_выражение::=простой_идентификатор |
(*простое_выражение знак_операции простое_выражение*)
простой_идентификатор::= буква
знак_операции:: = – | + | *

7. Построить синтаксический анализатор для понятия *вещественное число*.

вещественное_число::= *целое_число . целое_без_знака* |
*целое_число.целое_без_знака*E*целое число* |
*целое_число*E*целое_число*
целое_без_знака::=цифра | цифра целое_без_знака
целое_число::=целое_без_знака | + целое_без_знака | –целое_без_знака

8. Построить синтаксический анализатор для понятия *простое_логическое*.

простое_логическое::= TRUE | FALSE | *простой_идентификатор* |
NOT *простое_логическое* |
(*простое_логическое знак_операции простое_логическое*)
простой-идентификатор::=буква
знак-операции::= AND | OR

9. Разработать программу, которая по заданному *простому_логическому* выражению (определение понятия см. в предыдущей задаче), не содержащему вхождений простых идентификаторов, вычисляет значение этого выражения.

10. Построить синтаксический анализатор для определяемого далее понятия *константное_выражение.*

константное_выражение::=ряд_цифр|

константное_выражение знак_операции константное_выражение

знак_операции::=+ | – | *

ряд_цифр::=цифра | *цифра ряд_цифр*

11. Написать программу, которая по заданному (см. предыдущее задание) *константному_выражению* вычисляет его значение либо сообщает о переполнении (превышении заданного значения) в процессе вычислений.

12. Построить синтаксический анализатор для понятия *скобки*.

скобки::=квадратные | *круглые* | *фигурные*

квадратные::=[*круглые фигурные*] | +

круглые::=(*фигурные квадратные*) | –

фигурные::={*квадратные круглые*} | 0

13. Построить синтаксический анализатор для понятия *скобки.*

*скобки::=*A | *скобка скобки*

скобка::= (B *скобки*)

14. Построить синтаксический анализатор для понятия *скобки.*

*скобки::=*A | (B *скобки скобки*)

15. Построить синтаксический анализатор для понятия *скобки.*

*скобки::=*A | A (*ряд_скобок*)

ряд_скобок::= скобки | *скобки* ; *ряд_скобок*

16. Построить синтаксический анализатор для понятия *скобки.*

*скобки::=*A | B | (*скобки скобки*)

17. Функция Ф преобразования текста определяется следующим образом (аргумент функции – это текст, т. е. последовательность символов): Ф(γ)β, если α = β/γ и текст β не содержит вхождений символа «/»,

Ф(α)= α, если в α нет вхождений символа «/».

Например: Ф(«ла/ска»)=«скала», Ф(«б/ру/с»)=«сруб», Ф(«ца/ри/ца»)= «царица», Ф(«ум/ри/ва/к/а»)= «аквариум». Реализовать функцию Ф рекурсивно.

18. Пусть определена функция Ф преобразования целочисленного вектора α:

$$\Phi(\alpha)=\begin{cases}\alpha, \text{ если } \|\alpha\|=1,\\ ab, \text{ если } \|\alpha\|=2,\ \alpha=ab \text{ и } a\le b,\\ ba, \text{ если } \|\alpha\|=2,\ \alpha=ab \text{ и } b<a,\\ \Phi(\beta)\Phi(\gamma), \text{ если } \|\alpha\|>2,\ \alpha=\beta\gamma, \text{ где } \|\beta\|=\|\gamma\| \text{ или } \|\beta\|=\|\gamma\|+1.\end{cases}$$

Например: Ф(1,2,3,4,5) = 1,2,3,4,5; Ф(4,3,2,1) = 3,4,1,2; Ф(4,3,2) = 3,4,2. Отметим, что функция Ф преобразует вектор, не меняя его длину. Реализовать функцию Ф рекурсивно.

19. Функция Ф преобразования целочисленного вектора α определена следующим образом:

$$\Phi(\alpha) = \begin{cases} \alpha, \text{если } \|\alpha\| \le 2, \\ \Phi(\beta)\Phi(\gamma), \text{если } \alpha = \beta\gamma, \|\beta\| = \|\gamma\|, \|\alpha\| > 2, \\ \Phi(\beta\, a)\ \Phi\ (a\ \gamma), \text{ если } \alpha = \beta a\gamma, \|\beta\| = \|\gamma\|, \ \|\alpha\| > 2, \ \|a\| = 1. \end{cases}$$

Например: Ф(1,2,3,4,5) = 1,2,2,3,3,4,4,5; Ф(1,2,3,4,5,6) = 1,2,2,3,4,5,5,6; Ф(1,2,3,4,5,6,7) = 1,2,3,4,4,5,6,7; Ф(1,2,3,4,5,6,7,8) = 1,2,3,4,5,6,7,8. Отметим, что функция Ф может удлинять вектор. Реализовать функцию Ф рекурсивно.

20. Построить синтаксический анализатор понятия *список_параметров.*

список_параметров::= параметр | *параметр* , *список_параметров*

параметр::= имя=цифра цифра | *имя*=(*список_параметров*)

имя::= буква буква буква

21. Построить синтаксический анализатор для понятия *скобки*.

скобки::=квадратные | *круглые*

квадратные:: = [[*квадратные*] (*круглые*)] | B

круглые::=((*круглые*) [*квадратные*]) | A

22. Построить синтаксический анализатор для определенного далее понятия *логическое_выражение*.

логическое_выражение ::= TRUE | FALSE |*идентификатор* |
NOT (*операнд*) |*операция* (*операнды*)

идентификатор::=буква

операция::= AND | OR

операнды::= операнд |*операнд, операнды*

операнд::= логическое_выражение

23. Разработать программу, которая, имея на входе заданное (см. предыдущее задание) *логическое_выражение*, не содержащее вхождений идентификаторов, вычисляет значение этого выражения и печатает само выражение и его значение.

24. Построить синтаксический анализатор для понятия *текст_со_скобками.*

текст_со_скобками::=элемент | *элемент текст_со_скобками*

*элемент::=*A |B | (*текст_со_скобками*) | [*текст_со_скобками*] |
{ *текст_со_скобками* }

25. Построить синтаксический анализатор для параметризованного понятия *скобки(T)*, где *T* – заданное конечное множество, а круглые скобки «(» и «)» не являются терминальными символами, а отражают зависимость определяемого понятия от параметра *T*.

скобки(T)::=элемент(T) | *список(скобки(T))*

*список(E)::=*N | [*ряд(E)*]

ряд(E)::=элемент(E) | *элемент(E) ряд(E)*