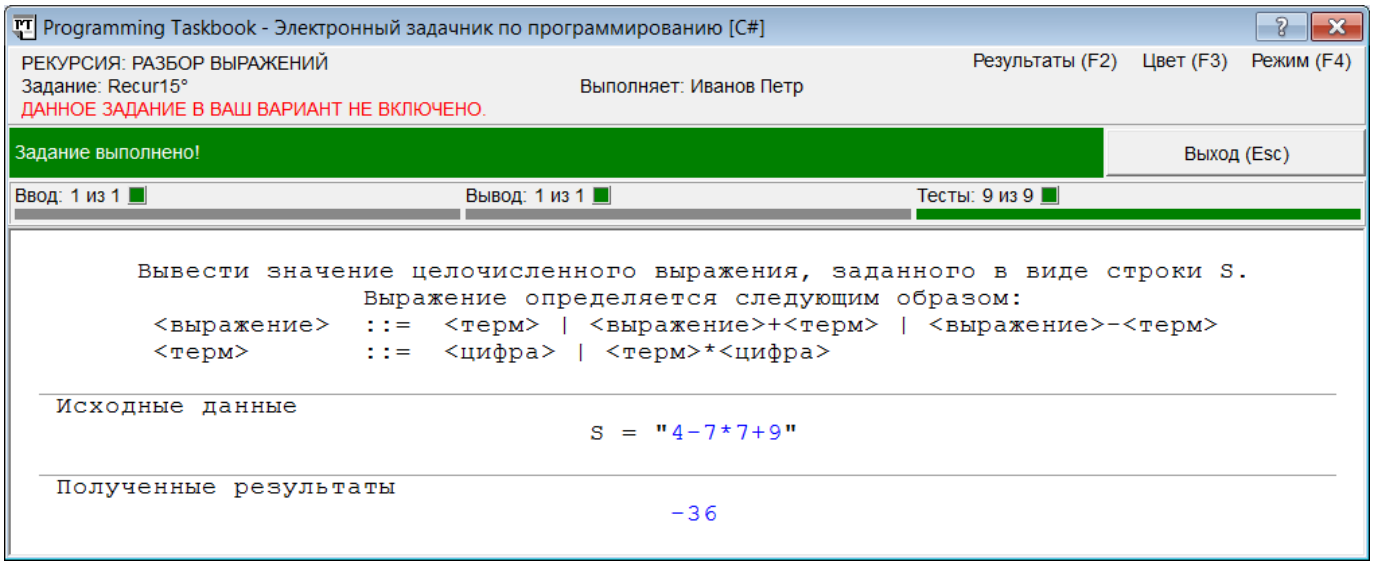
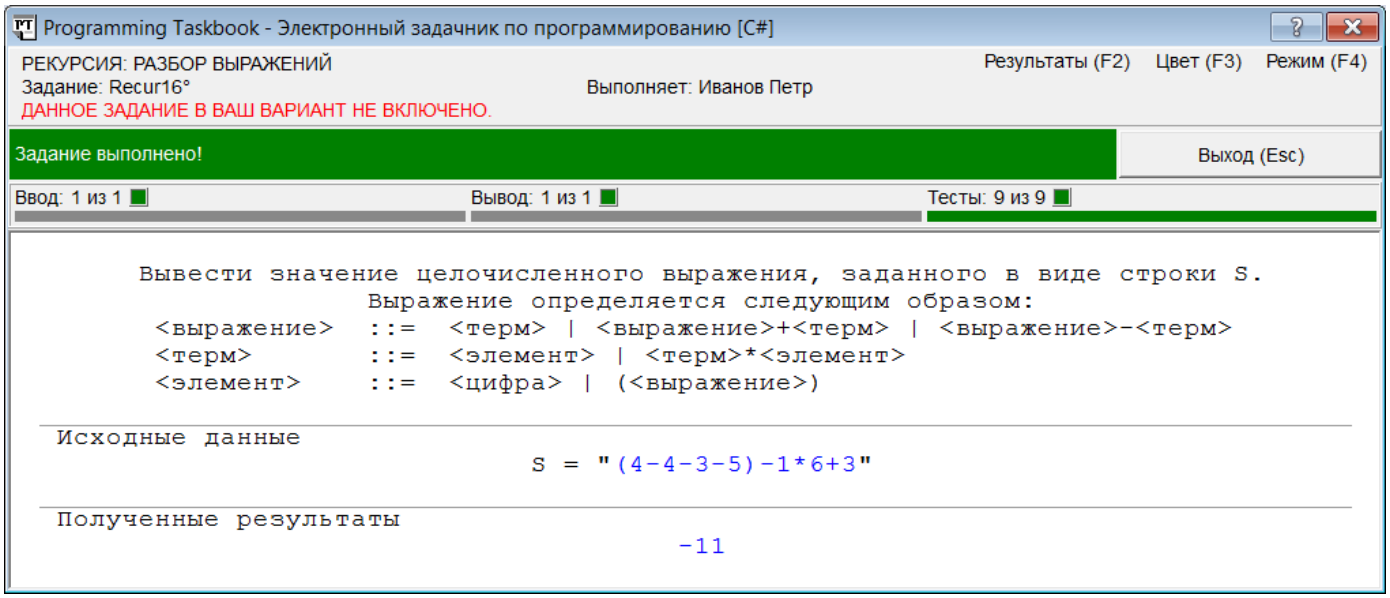
Разбор выражений




```
static string s;
static int k;

static char NextChar()
{
    if (++k < s.Length)
        return s[k];
    return '\0';
}

static int NextInt()
{
    return NextChar() - '0';
}

static void BackChar()
{
    k--;
}

static int Term()
{
    int i = NextInt();
    if (NextChar() == '*')
        return i * Term();
    BackChar();
    return i;
}
```

```
static int Expr()
{
    int i;
    switch (NextChar())
    {
        case '+':
            i = Term();
            return i + Expr();
        case '-':
            i = Term();
            return -i + Expr();
    }
    return 0;
}

public static int Rec15(string e)
{
    k = -1;
    s = e;
    int i = Term();
    return i + Expr();
}

public static void Solve()
{
    Task("Recur15");
    Put(Rec15(GetString()));
}
```




```
static string s;
static int k;

static int ToInt(char c)
{
    return c - '0';
}

static char NextChar()
{
    if (++k < s.Length)
        return s[k];
    return '\0';
}

static void BackChar()
{
    k--;
}

static int Elem()
{
    char c = NextChar();
    if (char.IsDigit(c))
        return ToInt(c);
    int i = Term();
    return i + Expr();
}
```

```
static int Term()
{
    int i = Elem();
    if (NextChar() == '*')
        return i * Term();
    BackChar();
    return i;
}

static int Expr()
{
    int i;
    switch (NextChar())
    {
        case '+':
            i = Term();
            return i + Expr();
        case '-':
            i = Term();
            return -i + Expr();
    }
    return 0;
}

static int Rec16(string e)
{
    k = -1;
    s = e;
    int i = Term();
    return i + Expr();
}

public static void Solve()
{
    Task("Recur16");
    Put(Rec16(GetString()));
}
```

Recur18. Проверить правильность выражения, заданного в виде непустой строки *S*. Выражение определяется по тем же правилам, что и в задании Recur17:

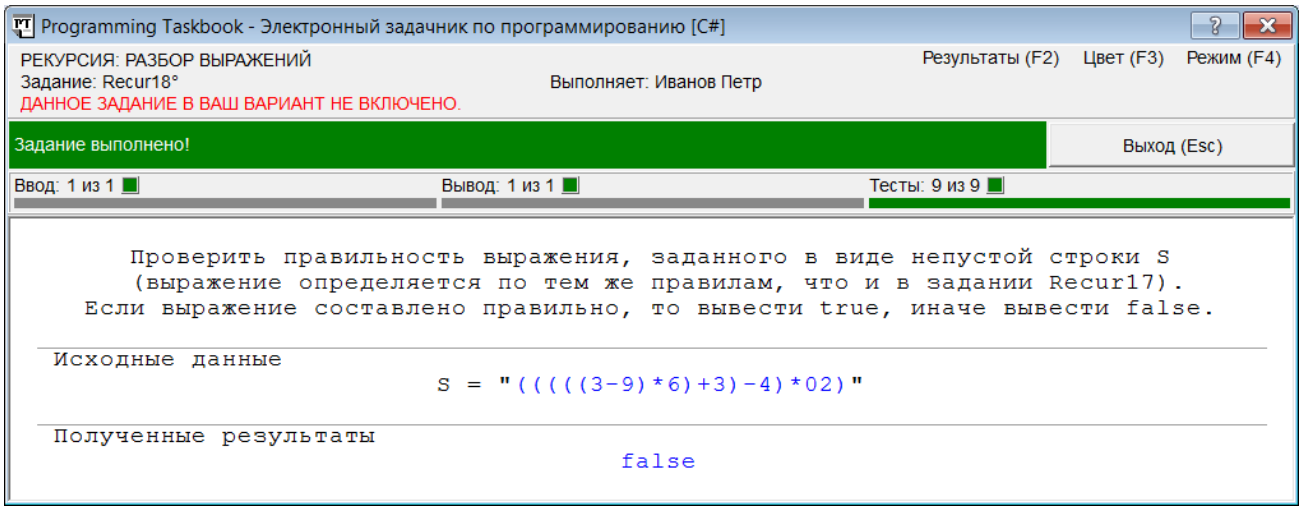
<выражение> ::= <цифра> | (<выражение><знак><выражение>)

<знак> ::= + | – | *

Если выражение составлено правильно, то вывести True, иначе вывести False.

Виды ошибок.

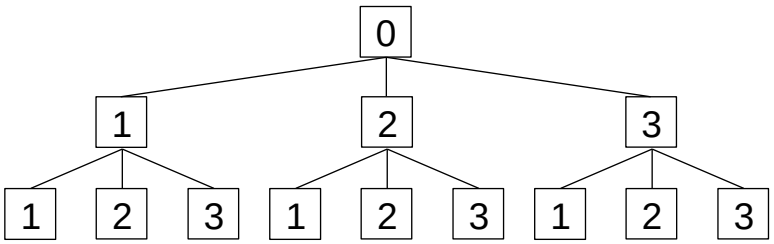
1. Ошибки в ситуации, когда выражение начинается с круглой скобки:
   (1.1) первый операнд не является допустимым или отсутствует;
   (1.2) знак операции не является допустимым или отсутствует;
   (1.3) второй операнд не является допустимым или отсутствует;
   (1.4) выражение не завершается круглой скобкой.
2. Ошибка в ситуации, когда выражение не содержит скобок:
   выражение не является цифрой или отсутствует.
3. Строка содержит лишние символы. Например, в строке «(2+4)–9» символы «–9» являются «лишними», так как разбор выражения будет закончен после просмотра текста «(2+4)» (для добавления второй операции в выражение необходимо, в силу приведенного выше определения, заключить ее в скобки: «((2+4)–9)»).

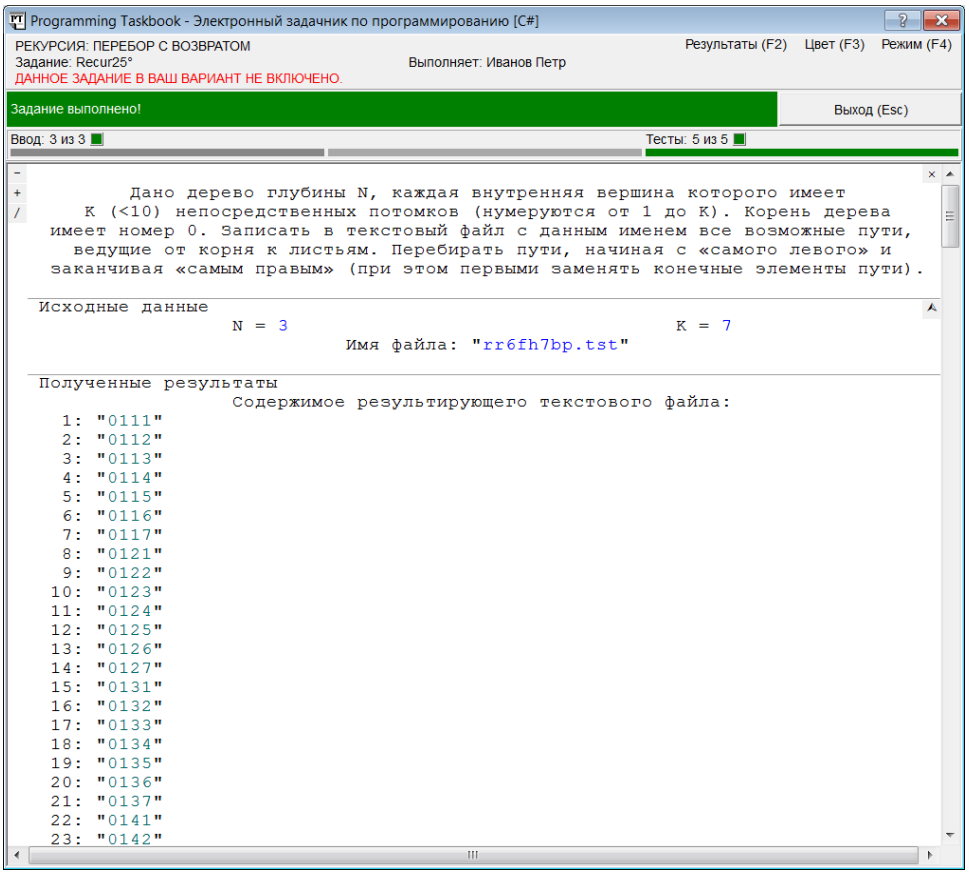



```
static string s;
static int k;

static char NextChar()
{
    if (++k < s.Length)
        return s[k];
    return '\0';
}

static bool Test()
{
    char c = NextChar();
    if (char.IsDigit(c))
        return true;
    if (c == '(')
    {
        if (!Test())
            return false; // (1.1)
        c = NextChar();
        if (c != '+' && c != '-'
                && c != '*')
            return false; // (1.2)
        if (!Test())
            return false; // (1.3)
        if (NextChar() != ')')
            return false; // (1.4)
        return true;
    }
    return false;         // (2)
}

static bool Rec18(string e)
{
    k = -1;
    s = e;
    return Test() &&
        k == s.Length - 1; // (3)
}

public static void Solve()
{
    Task("Recur18");
    Put(Rec18(GetString()));
}
```

Перебор с возвратом: Recur25

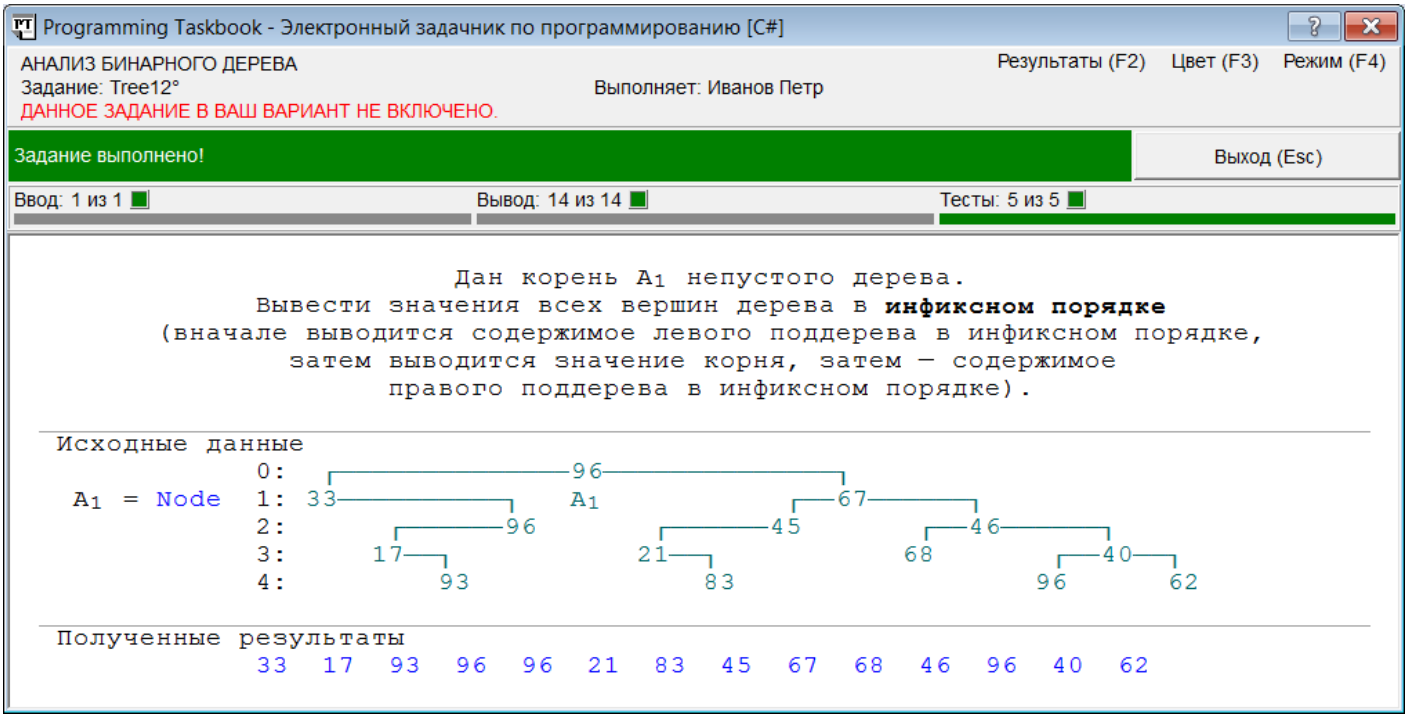
Recur25. Дано дерево глубины *N*, каждая внутренняя вершина которого имеет *K* (< 10) непосредственных потомков (нумеруются от 1 до *K*). Корень дерева имеет номер 0. Записать в текстовый файл с данным именем все возможные пути, ведущие от корня к листьям. Перебирать пути, начиная с «самого левого» и заканчивая «самым правым» (при этом первыми заменять конечные элементы пути).



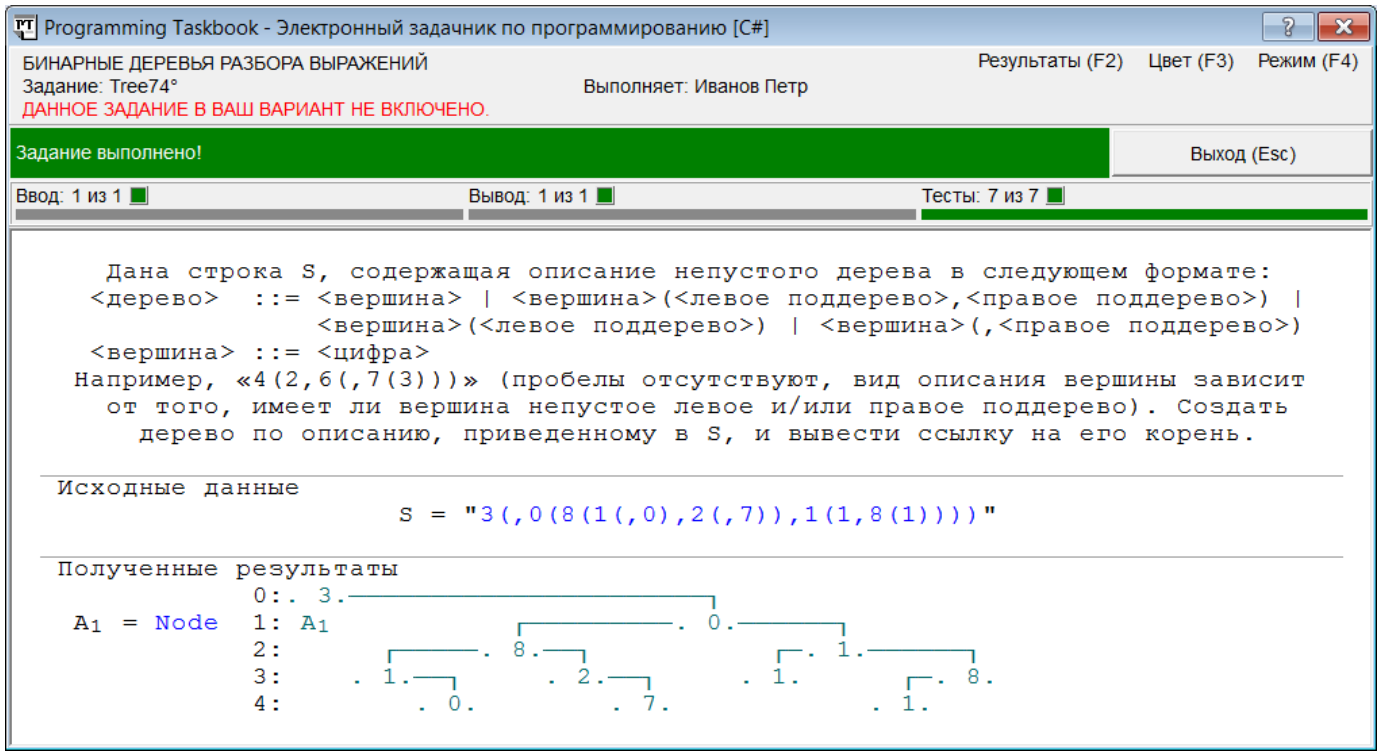



```
static int n, k;
static StreamWriter w;
static char[] c;
static void Step(int n0, int k0)
{
    c[n0] = (char)('0' + k0);
    if (n0 == n)
        w.WriteLine(c, 0, n + 1);
    else
        for (int i = 1; i <= k; i++)
            Step(n0 + 1, i);
}
public static void Solve()
{
    Task("Recur25");
    n = GetInt();
    k = GetInt();
    w = new StreamWriter(GetString());
    c = new char[n + 1];
    Step(0, 0);
    w.Close();
}
```

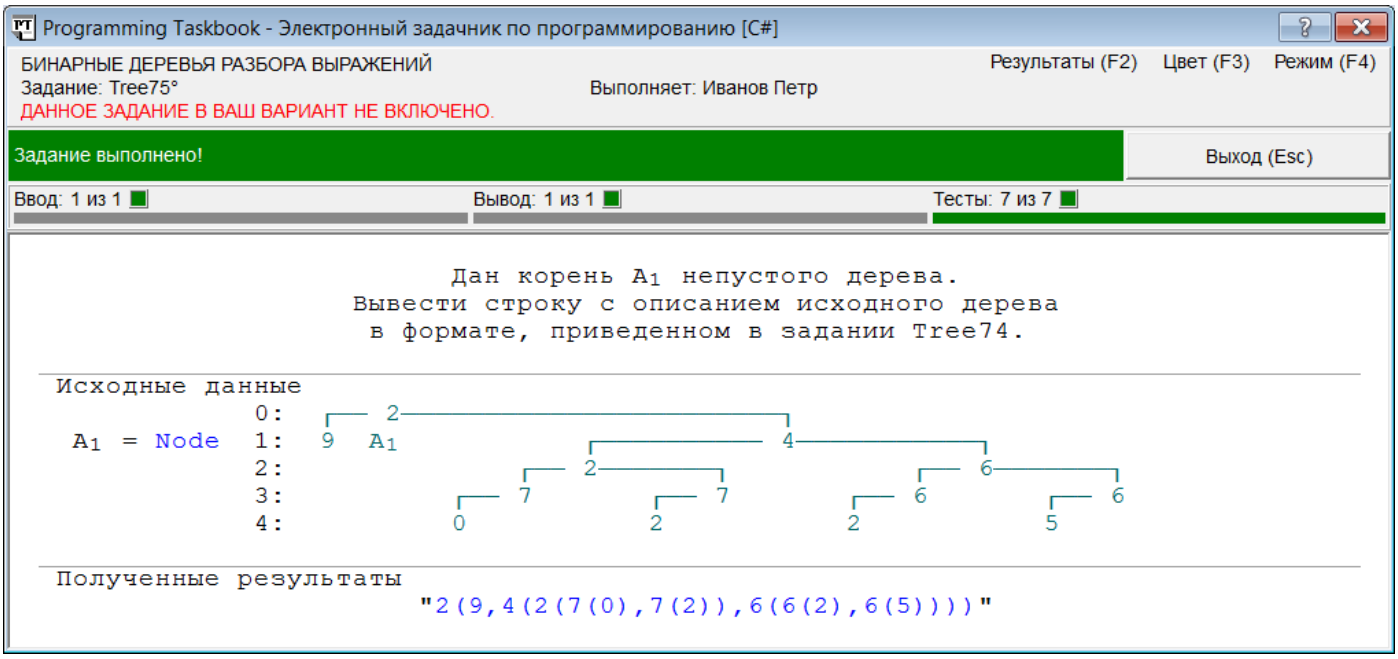
Деревья




```
static void Infix(Node a)
{
    if (a == null)
        return;
    Infix(a.Left);
    Put(a.Data);
    Infix(a.Right);
}

public static void Solve()
{
    Task("Tree12");
    Infix(GetNode());
}
```




```
static string s;
static int k;

static char NextChar()
{
    if (++k < s.Length)
        return s[k];
    return '\0';
}

static void BackChar()
{
    k--;
}

static Node CreateNode(char c)
{
    Node a = new Node(c - '0');
    if (NextChar() != '(')
    {
        BackChar();
        return a;
    }
    c = NextChar();
    if (c != ',')
    {
        a.Left = CreateNode(c);
        c = NextChar();
    }
    if (c == ',')
    {
        a.Right = CreateNode(NextChar());
        NextChar();
    }
    return a;
}

static Node Tree74(string s0)
{
    k = -1;
    s = s0;
    return CreateNode(NextChar());
}

public static void Solve()
{
    Task("Tree74");
    Put(Tree74(GetString()));
}
```




```
static string TreeToStr(Node a)
{
    string s = a.Data + "";
    if (a.Left == null && a.Right == null)
        return s;
    s += "(";
    if (a.Left != null)
        s += TreeToStr(a.Left);
    if (a.Right != null)
        s += "," + TreeToStr(a.Right);
    return s + ")";
}

public static void Solve()
{
    Task("Tree75");
    Put(TreeToStr(GetNode()));
}
```