ЛАБОРАТОРНАЯ РАБОТА №4
ТЕМА: «РАЗРАБОТКА КОМАНДНОЙ СТРОКИ ДЛЯ УПРАВЛЕНИЯ ХОДОМ ВЫПОЛНЕНИЯ ПРОГРАММЫ»

ЦЕЛЬ РАБОТЫ

Изучение алгоритмов вычисления функциональных выражений в обратной польской записи и особенностей их программной реализации.

ХОД РАБОТЫ

1. Модифицированный алгоритм вычисления функциональных выражений в обратной польской записи

Как правило арифметические выражения удобно преобразовывать в обратную польскую запись (ОПЗ), чтобы избавиться от скобок, содержащихся в выражении. Известный алгоритм можно преобразовать для случая функциональных выражений, представив ее как n-местную операцию.

Предположим, у нас есть функция многих переменных $f(x, y, z)$, выполняющее определенное действие в программе. Предполагается, что записанная команда будет обрабатываться через командную строку.

В выражении $f(x, y, z)$ выделим следующие элементы, актуальные для дальнейшего анализа:

1. ‘x’ – переменная, хранящая значение;
2. ‘y’ – переменная, хранящая значение;
3. ‘z’ – переменная, хранящая значение;
4. ‘(’ – оператор «открывающая круглая скобка», после которой начинается перечисление параметров функции;
5. ‘)’ – оператор «закрывающая круглая скобка», которая завершает перечисление списка параметров функции;
6. ‘,’ – оператор, являющийся символом разделителем между аргументами функции;
7. ‘f’ – оператор, определяющий выполнение функции с тремя параметрами, записанными в аргументы.

Для обработки функциональных выражений определим два стека: **стек операндов**, где будут храниться значения, передаваемые в функцию, и **стек операторов**.

Стек – это линейная структура данных, в которую элементы добавляются и удаляются **только с одного конца**, называемого **вершиной** стека. Стек работает по принципу «элемент, помещенный в стек последним, извлечен будет первым». Иногда этот принцип обозначается сокращением **LIFO** (Last In – First Out, т.е. последним зашел – первым вышел).

Линейная структура данных «Стек» реализована в классе `Stack<T>` пространства имен `using System.Collections.Generic;`.

В классе `Stack` можно выделить два основных метода, которые позволяют управлять элементами:

1. `Push`: добавляет элемент в стек на первое место;
2. `Pop`: извлекает и возвращает первый элемент из стека;
3. `Peek`: просто возвращает первый элемент из стека без его удаления.

Алгоритм обработки входной строки, представляющей собой функциональное выражение, следующий:

1. Рассматриваем поочередно каждый символ. Если этот символ – число или символ, не являющийся оператором (например, символьный аргумент функции), то помещаем его в стек операндов;

2. Если текущий символ - знак операции, то помещаем его в стек операторов;

3. Если текущий символ – знак оператора-разделителя, то игнорируем его;

4. Если текущий символ - открывающая скобка, то помещаем ее в стек операций;

5. Если текущий символ - закрывающая скобка, то извлекаем символы из стека операций до тех пор, пока не встретим в стеке открывающую скобку,

которую следует просто уничтожить. Закрывающая скобка также уничтожается;

6. После обработки входной строки извлекаем элемент из стека операций и выполняем метод, соответствующий данному знаку операции. В параметры найденного метода передаем извлекаемые поочередно элементы из стека операций.

2. Добавление новых классов

Для реализации обработки функциональных выражений добавим следующие классы:

1. Класс `Operand` – представляет собой объект-операнд, в котором будет храниться значение для операнда из входной строки. Реализация класса представлена в листинге 4.1.

Листинг 4.1. Класс Operand

```
1  public class Operand
2  {
3  public object value;
4  public Operand(object NewValue)
5  {
6      this.value = NewValue;
7  }
8  }
```

Свойство `value` хранит значение текущего операнда из командной строки

2. Класс `OperatorMethod` – представляет собой сущность, хранящую в качестве свойств делегаты на методы с определенным количеством параметров. Реализация класса представлена в листинге 4.2.

Листинг 4.2. Класс OperatorMethod

```
1  public class OperatorMethod
2  {
3  public delegate void EmptyOperatorMethod();
4  public delegate void UnaryOperatorMethod(object
   operand);
5  public delegate void BinaryOperatorMethod(object
   operand1, object operand2);
```

```
6  public delegate void TrinaryOperatorMethod(object operand1, object operand2, object operand3);
7  }
```

В строке 3 хранится делегат, которому можно назначить метод без параметров, в строке 4 – с одним параметром, в строке 5 – с двумя параметрами, в строке 6 – с тремя параметрами.

3. Класс `Operator` – представляет собой объект-оператор. Данный класс будет хранить знак оператора и переменную каждого делегата, унаследованного от класса `OperatorMethod`. Реализация класса представлена в листинге 4.3.

Листинг 4.3. Класс Operator

```
1  public class Operator : OperatorMethod
2  {
3  public char symbolOperator;
4  public EmptyOperatorMethod operatorMethod = null;
5  public BinaryOperatorMethod binaryOperator = null;
6  public TrinaryOperatorMethod trinaryOperator = null;
7  public Operator(EmptyOperatorMethod operatorMethod, char symbolOperator)
8  {
9  this.operatorMethod = operatorMethod;
10 this.symbolOperator = symbolOperator;
11 }
12 public Operator(BinaryOperatorMethod binaryOperator, char symbolOperator)
13 {
14 this.binaryOperator = binaryOperator;
15 this.symbolOperator = symbolOperator;
16 }
17 public Operator(TrinaryOperatorMethod trinaryOperator, char symbolOperator)
18 {
19 this.trinaryOperator = trinaryOperator;
20 this.symbolOperator = symbolOperator;
21 }
22 public Operator(char symbolOperator)
23 {
24 this.symbolOperator = symbolOperator;
25 }
```

```
26 }
```

4. Класс `OperatorContainer` – представляет собой линейный список, хранящий объекты-операторы (экземпляры класса `Operator`). Реализация класса представлена в листинге 4.4.

Листинг 4.4. Класс OperatorContainer

```
1  public static class OperatorContainer
2  {
3  public static List<Operator> operators = new List<Operator>();
4  static OperatorContainer()
5  {
6  operators.Add(new Operator('S'));
7  operators.Add(new Operator('R'));
8  operators.Add(new Operator('E'));
9  operators.Add(new Operator('C'));
10 operators.Add(new Operator('M'));
11 operators.Add(new Operator(','));
12 operators.Add(new Operator('('));
13 operators.Add(new Operator(')'));
14 }
15 public static Operator FindOperator(char s)
16 {
17 foreach(Operator op in operators)
18 {
19 if(op.symbolOperator == s)
20 {
21 return op;
22 }
23 }
24 return null;
25 }
26 }
```

В строке 3 создается линейный список `operators`, который будет хранить объекты класса `Operator`. В статическом конструкторе (строка 4-14) происходит последовательное добавление в список `operators` новых объектов-операторов, которые будут использоваться в нашей программе. В строке 15 определяется метод `FindOperator` для поиска оператора по заданному символу `s` в контейнере `operators`. В случае удачного поиска

возвращается объект-оператор, знак которого совпал с искомым. В противном случае возвращается нулевой объект.

Класс `OperatorContainer` является статическим, так как все его свойства и методы являются статическим. Следовательно, к членам данного класса можно обращаться без создания его экземпляра.

3. Программирование модифицированного алгоритма

Для имитации работы командной строки разместим в главной форме элемент TextBox, который позволяет пользователю вводить текст команды для дальнейшей обработки. Переименуем в последующем описании данный элемент как `textBoxInputString`.

Определим два стека как свойства класса формы: стек операндов и стек операторов.

Листинг 4.5. Цикл обработки входной строки

```
1  private Stack<Operator> operators = new
   Stack<Operator>()
2  private Stack<Operand> operands = new
   Stack<Operand>()
```

Далее начинаем в цикле обрабатывать каждый элемент входной строки, записанной в поле `textBoxInputString`.

Листинг 4.6. Цикл обработки входной строки

```
1  for (int i = 0; i < textBoxInputString.Text.Length;
   i++)
2  {
```

Изначально проверяем текущий символ из входной строки на несоответствие знаку операции:

Листинг 4.7. Условие несоответствия знаку операции

```
1  if (IsNotOperation(textBoxInputString.Text[i]))
2  {
```

Реализация функции, проверяющей на несоответствие знаку операции, представлена в листинге 4.8.

Листинг 4.8. Функция проверки символов операций

```
1  private bool IsNotOperation(char item)
2  {
```

```
3  if (!(item == 'R' || item == 'M' || item == 'E' || item
   == 'C' || item == 'S' || item == ',' || item == '(' ||
   item == ')'))
4  {
5  return true;
6  }
7  else
8  {
9  return false;
10 }
11 }
```

Если условие из листинга 4.7 истинно, то выполняется проверка на несоответствие текущего символа цифровому значению (0, 1, 2, 3…9). Если условие истинно, то выполняется добавление текущего символа в стек операндов и переход к следующей итерации цикла (к следующему символу входной строки):

Листинг 4.4.

```
1  if (!(Char.IsDigit(textBoxInputString.Text[i])))
2  {
3  this.operands.Push(new
   Operand(textBoxInputString.Text[i]));
4  continue;
5  }
```

В листинге 4.9 метод `IsDigit` класса `Char` возвращает `true`, если параметр метода представляет собой символ-цифру (0-9), и возвращает `false` в противном случае. В строке 3 происходит добавление нового операнда в стек операндов `operands` с помощью метода `Push`. Обратите внимание, что для добавления объекта класса `Operand` вызывается его конструктор и передается в качестве параметра текущий символ, являющийся значением данного операнда.

Если условие из листинга 4.9 ложно, выполняется проверка условия соответствия текущего символа входной строки цифре (строка 1, листинг 4.10). Для учета многозначных чисел в параметрах функционального выражения выполняется дополнительная проверка соответствия следующего символа цифре (строка 3), при истинности которого выполняется добавление

сначала первой цифры многозначного числа (строка 5-8), а затем добавление следующей цифры из входной строки как младший разряд числа, помещенного в стек операндов (строка 9). В строке 13 проверяется условие соответствия следующего символа входной строки запятой или закрывающей скобке, который сигнализируют о завершении описания числовых параметров. Если оно истинно, то текущий символ добавляется в стек операндов.

Листинг 4.10.

```
1  else if (Char.IsDigit(textBoxInputString.Text[i]))
2  {
3  if (Char.IsDigit(textBoxInputString.Text[i + 1]))
4  {
5  if (flag)
6  {
7  this.operands.Push(new
   Operand(textBoxInputString.Text[i]));
8  }
9  this.operands.Push(new
   Operand(ConvertCharToInt(this.operands.Pop().value) *
   10 + ConvertCharToInt(textBoxInputString.Text[i +
   1])));
10 flag = false;
11 continue;
12 }
13 else if ((textBoxInputString.Text[i + 1] == ','
   || textBoxInputString.Text[i + 1] == ')')
   && !(Char.IsDigit(textBoxInputString.Text[i - 1])))
14 {
15 this.operands.Push(new Operand(ConvertCharToInt
   (textBoxInputString.Text[i])));
16 continue;
17 }
18 }
```

В листинге 4.11 представлено условие проверки равенства исходного символа входной строки символу оператора (здесь R - метод прорисовки прямоугольника).

Листинг 4.11.

```
1  else if (textBoxInputString.Text[i] == 'R')
```

```
2  {
3  if (this.operators.Count == 0)
4  {
5  this.operators.Push(OperatorContainer.FindOperator
   (textBoxInputString.Text[i]));
6  }
7  }
```

Символ-разделитель запятая в стек операторов не добавляется и игнорируется.

В листинге 4.12 представлены действия, выполняемые при равенстве исходного символа входной строки открывающей или закрывающей фигурной скобке.

Листинг 4.12.

```
1  else if (textBoxInputString.Text[i] == '(')
2  {
3  this.operators.Push(OperatorContainer.FindOperator
   (textBoxInputString.Text[i]));
4  }
5  else if (textBoxInputString.Text[i] == ')')
6  {
7  do
8  {
9  if (operators.Peek().symbolOperator == '(')
10 {
11 operators.Pop();
12 break;
13 }
14 if (operators.Count == 0)
15 {
16 break;
17 }
18 }
19 while (operators.Peek().symbolOperator != '(');
20 }
21 }
```

При равенстве текущего символа открывающей фигурной скобке происходит добавление символа в стек операторов (строка 3).

При равенстве текущего символа закрывающей фигурной скобке происходит извлечение элементов из стека операторов (строка 11) до тех пор, пока не будет достигнута открывающая фигурная скобка.

После выполнения вышеописанных действий извлекаем из стека операторов вершинный элемент, представляющий собой символ оператора и передаем его в метод обработки функциональных выражений `SelectingPerformingOperation` (листинг 4.13).

Листинг 4.13.

```
1  if (operators.Peek() != null)
2  {
3  this.SelectingPerformingOperation(operators.Peek());
4  }
5  else
6  {
7  MessageBox.Show("Введенной операции не существует");
8  }
```

Рассмотрим реализацию метода `SelectingPerformingOperation` (листинг 4.14).

Листинг 4.14.

```
1  private void SelectingPerformingOperation(Operator
   op)
2  {
3  if (op.symbolOperator == 'R')
4  {
5  this.figure = new Rectagle
   (Convert.ToInt32
   (Convert.ToString(operands.Pop().value)),
   Convert.ToInt32
   (Convert.ToString(operands.Pop().value)),
   Convert.ToInt32
   (Convert.ToString(operands.Pop().value)),
   Convert.ToInt32
   (Convert.ToString(operands.Pop().value)),
   Convert.ToString
   (operands.Pop().value));
6  op = new Operator(this.figure.Draw, 'R');
7  ShapeContainer.AddFigure(figure);
8  comboBox1.Items.Add(figure.name);
```

```
9   op.operatorMethod();
10  }
```

В качестве параметра метод принимает объект класса `Operator`, извлеченный из вершины стека операторов. В условии, представленном в строке 3, проверяем соответствие свойства `symbolOperator` оператора заранее определенному символу для операции прорисовки прямоугольника. В строке 5 создается объект класса «Прямоугольник», куда в качестве параметров передаются значения, поочередно извлекаемые из стека операндов с помощью метода `Pop` (ширина, длина, координаты базовой точки и имя фигуры). В строке 6 создается новый оператор (объект класса `Operator`), куда в качестве первого параметра передается ссылка на метод `Draw` ранее созданного объекта класса «Прямоугольник». В строке 7-8 созданная фигура добавляется в раскрывающийся список элементов и в контейнер, где хранятся все созданные фигуры. В строке 9 вызывается метод прорисовки `Draw` через делегат `operatorMethod`, определенный внутри класса `Operator`.

Для обработки нажатия на кнопку ENTER создаем событие `KeyDown` для элемента `textBoxInputString`. Реализация метода обработчика события представлен в листинге 4.15.

Листинг 4.15.

```
1   private void textBoxInputString_KeyDown(object sender,
    KeyEventArgs e)
2   {
3   if(e.KeyCode == Keys.Enter)
4   {
5   try
6   {
7   //выполняется обработка входной строки
8   }
9   catch
10  {
11  //добавляется информация о некорректной команде в
    историю команд
12  }
13  textBoxInputString.Text = "";
```

```
14  }
15  }
```

В строке 3 выполняется проверка условия нажатия клавиши ENTER.

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Модифицировать программу, реализованную на предыдущей лабораторной работе «Создание и использование библиотеки классов для графических примитивов». Обновленная версия программы должна включать в себя следующие изменения:

1. Удаление всех элементов управления из формы (кнопок, лейблов, полей для ввода и прочих), кроме поля рисунка `PictureBox`, где будет размещаться битовая карта;
2. Добавление командной строки (для ее реализации можно использовать элемент `TextBox`), где будут указываться команды, которые должна будет выполнять программа (прорисовка, перемещение и удаление фигур);
3. Добавить историю команд, где будут размещаться выполненные и неудачные команды.

Команды должны выполняться при нажатии кнопки ENTER на клавиатуре.

На рисунке 4.1 представлен предполагаемый интерфейс программы для данной лабораторной работы.

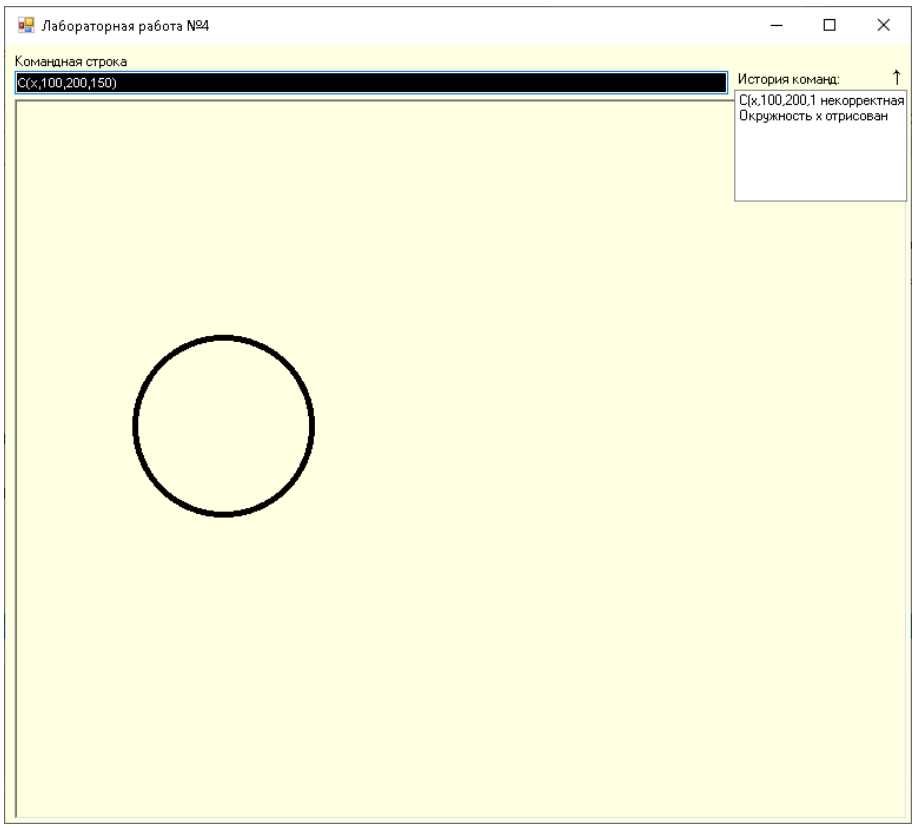

Рисунок 4.1 – Графический интерфейс программы с командной строкой

Список и формат записи команд, выполняемых через командную строку разрабатываемой программы, выбирается согласно варианту индивидуального задания.

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ

№	Набор команд	Формат команд
1	Создание прямоугольника	R(name,x,y,w,h)
	Перемещение прямоугольника	M(name,dx,dy)
	Удаление прямоугольника	D(name)
2	Создание квадрата	S(name,x,y,a)
	Перемещение квадрата	M(name,dx,dy)
	Удаление квадрата	D(name)
3	Создание эллипса	E(name,x,y,w,h)
	Перемещение эллипса	M(name,dx,dy)
	Удаление эллипса	D(name)
4	Создание окружности	C(name,x,y,r)
	Перемещение окружности	M(name,dx,dy)
	Удаление окружности	D(name)
5	Создание прямоугольника	R[name;x;y;w;h]
	Перемещение прямоугольника	M[name;dx;dy]
	Удаление прямоугольника	D[name]
6	Создание квадрата	S[name;x;y;a]
	Перемещение квадрата	M[name;dx;dy]
	Удаление квадрата	D[name]
7	Создание эллипса	E[name;x;y;w;h]
	Перемещение эллипса	M[name;dx;dy]
	Удаление эллипса	D[name]
8	Создание окружности	C[name;x;y;r]
	Перемещение окружности	M[name;dx;dy]
	Удаление окружности	D[name]
9	Создание прямоугольника	R(name,x,y,w,h)
	Перемещение прямоугольника	M(name,dx,dy)
	Удаление прямоугольника	D(name)

№	Набор команд	Формат команд
	Изменение размеров прямоугольника	I(name,cx,cy)
10	Создание квадрата	S(name,x,y,a)
	Перемещение квадрата	M(name,dx,dy)
	Удаление квадрата	D(name)
	Изменение размеров квадрата	I(name,ca)
11	Создание эллипса	E(name,x,y,w,h)
	Перемещение эллипса	M(name,dx,dy)
	Удаление эллипса	D(name)
	Изменение размеров эллипса	I(name,cx,cy)
12	Создание окружности	C(name,x,y,r)
	Перемещение окружности	M(name,dx,dy)
	Удаление окружности	D(name)
	Изменение радиуса окружности	I(name,cr)
13	Создание массива точек	A(nameA, i, x_1, y_1, …, x_i, y_i)
	Создание многоугольника	P(name, nameA)
	Перемещение многоугольника	M(name,dx,dy)
	Удаление многоугольника	D(name)
14	Создание массива точек	A[nameA; i; x_1; y_1; …; x_i; y_i]
	Создание многоугольника	P[name; nameA]
	Перемещение многоугольника	M[name;dx;dy]
	Удаление многоугольника	D[name]
15	Создание массива точек	A(nameA, x_1, y_1, x_2, y_2, x_3, y_3)
	Создание треугольника	T(name, nameA)
	Перемещение треугольника	M(name,dx,dy)
	Удаление треугольника	D(name)
16	Создание массива точек	A[nameA, x_1, y_1, x_2, y_2, x_3, y_3]
	Создание треугольника	T[name, nameA]

№	Набор команд	Формат команд
	Перемещение треугольника	M[name,dx,dy]
	Удаление треугольника	D[name]
17	Создание сложной фигуры	O(name, x, y, w, h)
	Перемещение сложной фигуры	M(name, x, y, w, h)
	Удаление сложной фигуры	D(name)
18	Создание сложной фигуры	O[name; x; y; w; h]
	Перемещение сложной фигуры	M[name; x; y; w; h]
	Удаление сложной фигуры	D[name]

Пояснение условных обозначений:

name – имя фигуры

x – координата базовой точки фигуры по оси X

y – координата базовой точки фигуры по оси Y

w – длина фигуры

h – ширина фигуры

a – сторона квадрата

r – радиус окружности

dx – смещение фигуры по оси X

dy – смещение фигуры по оси Y

cx – изменение значения длины фигуры

cy – изменение значения ширины фигуры

ca – изменение значения стороны квадрата

cr – изменение значения радиуса окружности

nameA – имя массива точек

i – количество точек в массиве nameA

$x_1, y_1, \ldots, x_i, y_i$ – координаты по оси X-Y точек из массива nameA

R(x1,100,50,180,120) – создание прямоугольника x1 с координатами базовой точки (100,50), длиной равной 180 и шириной 120.