ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ФАКУЛЬТЕТ МАТЕМАТИКИ И ЕСТЕСТВЕННОНАУЧНОГО ОБРАЗОВАНИЯ

КАФЕДРА ИНФОРМАТИКИ, ЕСТЕСТВЕННОНАУЧНЫХ ДИСЦИПЛИН И МЕТОДИК ПРЕПОДАВАНИЯ

РАЗРАБОТКА ПРИЛОЖЕНИЯ ДЛЯ ОБУЧЕНИЯ РЕКУРСИВНЫМ АЛГОРИТМАМ

Выпускная квалификационная работа
обучающегося по направлению подготовки 44.03.05 Педагогическое образование по профилю "Информатика и иностранный язык (английский)"
очной формы обучения, группы 02041205
Сулла Руслана Викторовича

Научный руководитель
к. ф.-м. н., доцент
Старовойтов А. С.

БЕЛГОРОД 2017

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

XXI век – век компьютеров и информации. С ростом количества информации увеличились объемы данных, хранимых на компьютерах. Для обработки таких огромных массивов данных нужны информационные системы, обладающие высокой скоростью работы и скоростью обработки информации [1]. Перед программистами стоит задача о том, как увеличить скорость работы алгоритмов, но при этом сохранить надежность и читаемость исходного кода, а также возможность последующих модификаций и относительно легкого сопровождения конечных программных продуктов. Рекурсивные алгоритмы помогают в реализации таких приложений, потому что данные алгоритмы доступны для использования практически во всех средах программирования [2].

Область применения рекурсии в настоящее время довольно широка, начиная от самых простых и примитивных программ нахождения наибольшего общего делителя или факториала числа, и заканчивая сложнейшими алгоритмами трансляции или численного анализа [2].

Важность рекурсии подчеркивали и известнейшие в науке информатики люди, такие как лауреаты премии Тьюринга американский специалист по системному программированию Дональд Кнут и английский ученый-теоретик Чарльз Хоар [3].

Актуальность данного исследования заключается в том, что рекурсивные алгоритмы являются одними из наиболее мощных, а также самых общих методов научного познания. Рекурсия не ограничивается использованием только лишь в сфере информатики, она используется во многих теоретических и прикладных естественнонаучных дисциплинах [4].

Объектом исследования являются рекурсивные алгоритмы.

Предмет исследования – разработка приложения для обучения рекурсивным алгоритмам.

Задачи:

1. Изучить понятие рекурсии, основные виды рекурсии;

2. Рассмотреть примеры реализации рекурсивных алгоритмов на языке программирования Pascal в среде программирования PascalABC.NET;

3. Разработать приложение для демонстрации работы рекурсивных алгоритмов;

4. Разработать модуль для среды PascalABC.NET, позволяющий изучить и протестировать программы, использующие рекурсию.

Структура работы: работа состоит из двух глав. В первой главе разбирается понятие рекурсии, ее основные характеристики, наиболее распространенные примеры. Во второй главе идет непосредственная разработка приложений для обучения рекурсивным алгоритмам. Помимо глав в работе представлены введение, заключение и список использованных источников. Вся работа занимает 59 листов.

1 Теоретический анализ рекурсии

1.1 Характеристика рекурсии

Рекурсия – это определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса, то есть ситуация, когда объект является частью самого себя [5]. Если брать конкретно программирование, то рекурсия – это процедура или функция, вызывающая сама себя. Другими словами, это такой метод определения функций (процедур), при котором описываемая функция (процедура) применена в теле своего же собственного определения. Сами же рекурсивные алгоритмы в сфере программирования реализуются в рекурсивных подпрограммах (процедурах или функциях) [5].

Рекурсия – очень мощный инструмент программирования. Широкое распространение она получила при программировании игр, а также переборных алгоритмов [5].

Характерной особенностью рекурсии является вычисление функции, постепенно упрощая ее аргумент, до тех пор пока не будет получен явный ответ, затем происходит обратный процесс, когда вычисление будет производиться для более сложных аргументов.

Описание рекурсивной функции состоит из двух частей: нерекурсивные ветви, рассматривающие простейшие случаи выдачи ответа и позволяющие завершить выполнение функции, и рекурсивные ветви, в которых записывается общее решение задачи, которое строится из более простых решений [5].

Бывают прямые и косвенные рекурсии. В прямой вызов рекурсивной процедуры осуществляется из той же функции. Например:

```
Procedure Rec();
begin
…
Rec();
```

...

```
End;
```

Косвенную рекурсию создают за счет вызова данной функции из какой-либо другой функции, которая сама вызывалась из данной функции. Данный алгоритм хорошо показан на рисунке 1.1.

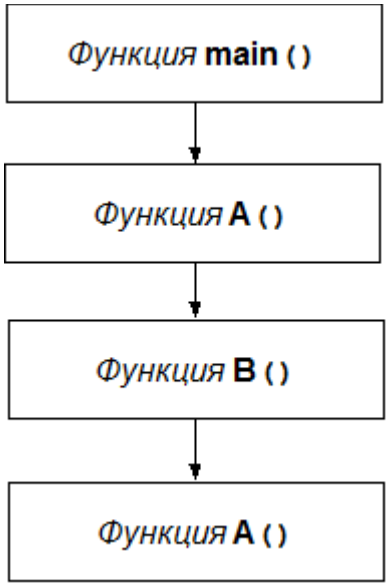

Рисунок 1.1 – Схема вызова косвенной рекурсии

Обязательным элементом рекурсивных подпрограмм должно быть условие, в зависимости от которого происходил бы выход из подпрограммы, чтобы избежать бесконечного выполнения подпрограммы, а в конечном счете аварийный выход [6].

Для рекурсии характерны такие понятия как рекурсивный спуск и рекурсивный подъем (см. рисунок 1.2). Рекурсивный спуск (или рекурсивное погружение) – это последовательный вызов подпрограммы самой себя. После многократного вызова рекурсивной подпрограммы нужно постепенно выходить из этого вызова, то есть производить рекурсивный подъем (или рекурсивный возврат, рекурсивное всплытие) [4].

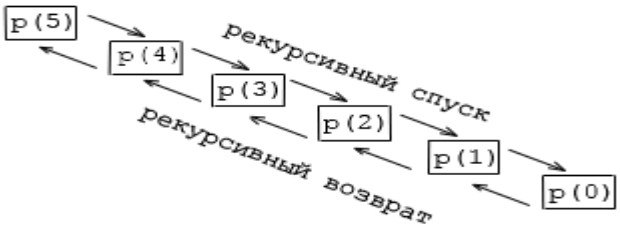

Рисунок 1.2 – Рекурсивный спуск и возврат

Количество вызовов процедуры или функции без осуществления возвратов называется глубиной рекурсии.

В современных системах программирования корректное функционирование рекурсивных подпрограмм обеспечивается с помощью абстрактного типа данных – стека [2]. Стек – это структура данных, реализованная по принципу LIFO (англ. Last In – First Out), т.е. «первым пришел – последним ушел». Хорошим примером иллюстрации стека является стопка тарелок: чтобы взять вторую сверху тарелку, нужно снять верхнюю, а чтобы снять последнюю, нужно снять все лежащие выше тарелки [2].

Стек – очень удобная структура данных для множества задач, в которых используются вложенные вызовы подпрограмм. Например, есть 3 процедуры А. Б и В. Процедура А вызывает процедуру Б, а та, в свою очередь, процедуру В. Выполняться данный алгоритм будет следующим образом: сначала будет произведен вызов А, когда выполнение А дойдет до вызова Б, А будет приостановлена и управление будет передано во входную точку Б, когда выполнение процедуры Б дойдет до вызова В, Б будет приостановлена и управление передастся в процедуру В, процедура В дойдет до своего полного завершения, произойдет возврат и возобновление Б, которая в свою очередь тоже достигнет своего завершения, выполнение вернется в А, которая теперь окончательно закончит свое выполнение. Без использования стека будет сложно правильно отследить все возвраты, а с использованием стека алгоритм будет следующим: когда процедура А вызывает процедуру Б, в стек заносится адрес возврата в А; когда Б вызывает В, в стек заносится адрес возврата в Б. Когда В заканчивается, адрес возврата выбирается из вершины стека - а это адрес

возврата в Б. Когда заканчивается Б, в вершине стека находится адрес возврата в А, и возврат из Б произойдет в А [3].

Рекурсия хотя и довольно проста в применении, но есть несколько трудностей в ее использовании. Например, есть дерево рекурсивных вызовов (см. рисунок 1.3), в котором для вычисления F(6) будут вызваны F(5) и F(4), которые в свою очередь вызовут F(4), F(3) и F(3), F(2).

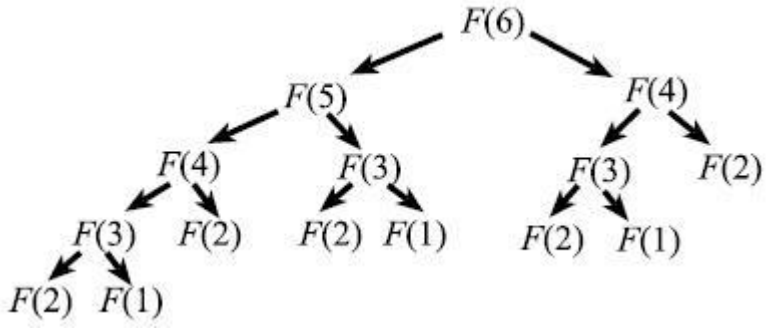

Рисунок 1.3 – Дерево рекурсивных вызовов

На схеме хорошо видно, что вызов подпрограммы F(3) будет производиться 3 раза, а F(2) и вовсе 5 раз. Несложно догадаться, что если вычислять, например, F(100), то повторные вызовы будут куда более многочисленны. Это главный и самый неприятный недостаток рекурсии – многочисленные повторные вызовы подпрограмм. К счастью, решение этой проблемы есть, и оно довольно простое – нужно просто запоминать найденные значения, чтобы не производить повторные вычисления (см. рисунок 1.4). Конечно, для таких операций придется расходовать память вычислительной машины, но для подавляющего большинства разрабатываемых программ это не будет являться проблемой [4].

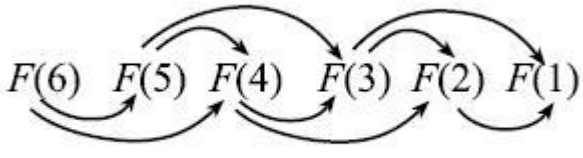

Рисунок 1.4 – Схема запоминания значений

Любую рекурсивную функцию можно описать и итеративно, то есть с помощью циклов, а также любую нерекурсивную функцию можно описать и рекурсивно [7].

Подводя итог вышесказанному, можно с уверенностью утверждать, что рекурсивные алгоритмы довольно мощные, быстрые, разнообразные, но использовать их тоже нужно правильно, чтобы не допустить аварийного завершения приложений, «зависания» компьютера, бесконечного выполнения подпрограмм.

1.2 Примеры использования рекурсивных алгоритмов

Рекурсивные алгоритмы встречаются очень часто, во многих программах, но есть несколько программ, с которыми сталкивается практически каждый, кто занимается программированием. В данном параграфе будут разобраны несколько программ, которые используют рекурсию в качестве метода решения задачи.

Задача 1: Поиск наибольшего общего делителя.

Наибольший общий делитель двух чисел – это то максимальное число, на которое оба заданных числа делятся без остатка. Например, для чисел 48 и 36 наибольшим общим делителем будет число 12. Убедиться в этом можно, разложив на простые сомножители.

48 = 2 * 2 * 2 * 2 * 3;

36 = 2 * 2 * 3 * 3.

Наибольшим общим делителем будет произведение одинаковых простых сомножителей, а это 2, 2 и 3, что в произведение и даст 12.

Есть и второй метод поиска НОД, который более распространен в программировании – алгоритм Евклида. Первым делом идет поиск остатка от деления первого числа на второе, если остаток не 0, то вместо первого числа подставляется второе, а вместо второго полученный остаток, так продолжать до

тех пор, пока остаток не будет равным нулю, тот делитель и будет наибольшим общим. Возвращаясь к примеру с числами 48 и 36, найдем НОД методом Евклида.

48 mod 36 = 12;

36 mod 12 = 0.

Остаток равен нулю, значит НОД = 12.

Теперь рассмотрим как будет выглядеть функция алгоритма Евклида на языке программирования Pascal.

```
Function NOD(A, B: Integer): Integer;
    Var OST: Integer;
    Begin
        OST := A Mod B;
        If OST = 0 Then
            NOD := B
        Else
            NOD := NOD (B, OST)
    End;
```

Функция NOD получает два числа A и B. Происходит вычисление остатка OST, который если равен 0, то получаем ответ, иначе вызываем снова функцию NOD, но с параметрами B и остатка OST. Рекурсивно вызывается процедура NOD [5].

Задача 2: Нахождение факториала числа.

Факториал – это произведение все натуральных чисел от 1 до N. Обозначается он как N! . Например, факториалом числа 6 является число 720.

6! = 1 * 2 * 3 * 4 * 5 * 6 = 720.

Из этого правила есть исключение: факториал нуля равен 1 [6].

На языке программирования Pascal функция нахождения факториала числа будет выглядеть следующим образом:

```
Function FACT(N: Integer) : Longint;
    Begin
```

```
    If N= 0 Then
    FACT := 1
    Else FACT := FACT(N-1) * N
End;
```

Функция FACT получает число N. Затем производится проверка: если N равно 0, то ответ будет равен 1 (по правилу). В противном случае будет выполняться действие: каждое последующее умножается на предыдущее. Рекурсивной функцией в данном примере является функция FACT, которая вызывает сама себя.

Задача 3: Построить ряд Фибоначчи.

Ряд Фибоначчи – это последовательность чисел, каждое последующее из которых равно сумме двух предыдущих, причем первые два числа ряда обязательно равны либо 0 и 1, либо 1 и 1.

Пример ряда Фибоначчи: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 и т.д.

Чтобы написать рекурсивную процедуру построения ряда Фибоначчи, нужно воспользоваться формулой $F_n=F_{n-1}+F_{n-2}$.

Функция на языке программирования Pascal выглядит следующим образом [7]:

```
Function FIB (N: Integer): Integer;
    Begin
        If (N <= 2) Then
            FIB := 1
        Else
            FIB := FIB (N-1) + FIB (N-2);
    End;
```

Сначала проверяется условие, что если N меньше либо равно 2, то ответ будет равен 1. Это следует из правила, когда первые два числа ряда Фибоначчи равны 1. Если N больше 2, то происходит рекурсивное обращение к функции FIB, которая по формуле вычисляет очередное число ряда [6].

Как уже было сказано выше, любую рекурсивную функцию можно записать и итеративно, то есть с помощью циклических алгоритмов. Значит,

любую из вышеописанных можно было решить с помощью циклов, но использование рекурсии значительно проще, а также код становится более компактным [7]. Да, рекурсивные функции потребляют больше памяти, но это касается более сложных задач, которые будут рассмотрены и реализованы в следующей главе.

2 Разработка приложений для обучения рекурсивным алгоритмам

2.1 Выбор средств реализации приложений

Для разработки задуманных приложений необходимо выбрать среду, в которой их программировать. Выбор был остановлен на среде программирования Object Pascal – Embarcadero RAD Studio XE3. На самом деле нет принципиальной разницы в версии, потому что те компоненты, которые нам понадобятся, имеются в любой версии среды. Почему именно Object Pascal? Здесь все очевидно – она очень проста в использовании, с ней нет никаких сложностей при установке, а также она нетребовательна к ресурсам [8].

Второе приложение – это обучающий модуль. Здесь выбор был остановлен на среде программирования PascalABC.NET. Во-первых, она абсолютно бесплатна, во-вторых, на ней производится обучение в школах и ВУЗах, в-третьих, библиотека NET позволяет создавать действительно серьезные приложения с большим функционалом.

2.2 Разработка интерактивной среды обучения рекурсивным алгоритмам

Количество задач, в которых могут использоваться рекурсивные алгоритмы, довольно велико. Это и различные вычисления факториала числа, и вычисление ряда Фибоначчи, и другие. Особый же интерес вызывают задачи, которые наглядно показывают то, как в них используется рекурсия. Эти задачи (а их будет 4) стоит поместить в одну программу, чтобы пользователь мог посмотреть поведение рекурсии. Треугольник Серпинского, ковер Серпинского, кривая дракона и фрактальное дерево – наиболее известные примеры таких программ. Их мы и возьмем за основу интерактивного приложения. Это очень просто сделать на языке программирования Object Pascal [8-14].

Создадим новый проект и к нему дополнительно 4 формы (см. рисунок 2.1).

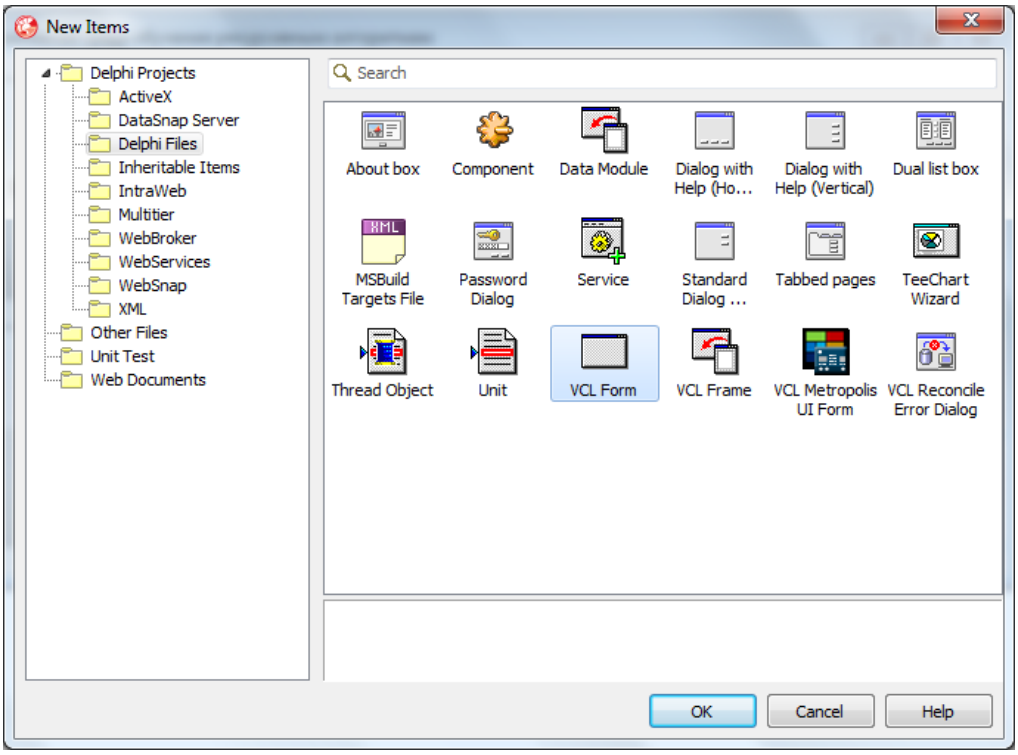

Рисунок 2.1 – Добавление дополнительных форм на проект

Их названия лучше оставить стандартными: Unit1, Unit2, Unit3, Unit4 и Unit5. В код Unit1, в раздел подключения модулей нужно добавить все остальные 4 формы. На главную форму Unit1 поместим 4 кнопки, дадим им соответственно «Фрактальное дерево», «Треугольник Серпинского», «Кривая Дракона» и «Ковер Серпинского» (см. рисунок 2.2).

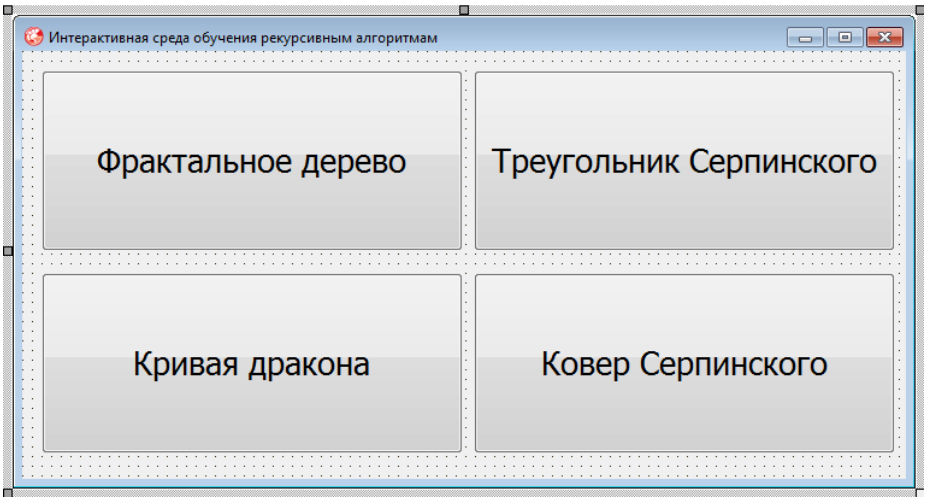

Рисунок 2.2 – Главное окно

Теперь двойным кликом по кнопке создадим обработчик события нажатия кнопки, чтобы по нажатии открывалось нужное окно с нужным примером.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.ShowModal;
end;
```

По аналогии сделать так с каждой кнопкой. Теперь если откомпилировать проект, то при нажатии на кнопку произойдет открытие другого окна. Эти окна пока пустые. Начнем их заполнять. Открываем код Unit2 и добавляем в него следующую процедуру:

```
procedure Line(x1, y1, x2, y2: real; Canv: TCanvas);
begin
 Canv.MoveTo(round(x1), round(y1));
 Canv.LineTo(round(x2), round(y2));
 end;
```

Данная процедура будет обязательной в каждом Unit, потому что она непосредственно занимается рисованием линий, которые показывают решение задачи.

2.2.1 Фрактальное дерево

Этот фрактал имеет следующие особенности: изначально рисуется ствол и две ветви, затем в процессе увеличение глубины рекурсии от каждой ветви будут отходить еще две ветви и так далее. Программа имеет такие параметры настроек как уровень, отклонение левой и правой осей, а также длину ветвей, поэтому на рабочую область необходимо поместить 4 компонента SpinEdit со вкладки Samples, одну кнопку со вкладки Standart и PaintBox со вкладки System.

Вычисление результата будет производить процедура Tree, имеющая следующий вид:

```
procedure Tree(Age, kx, ky, r: integer; Incl: real);
var
  sx, sy: integer;
begin
  r := round(r - 0.2 * r);
  Inc(Age);
  if Age = AgeF then
    begin
      Line(kx, ky, Round(kx + R * cos(Incl)),
Round(ky + R * sin(Incl)), Form3.PaintBox1.Canvas);
      sx:=round(kx + R * cos(Incl));
      sy:=round(ky + R * sin(Incl));
      Line(sx, sy, Round(sx + R * cos(Incl-angleL)),
Round(sy + R * sin(Incl - AngleL)),
Form3.PaintBox1.Canvas);
      Line(sx, sy, Round(sx + R * cos(Incl+angleR)),
Round(sy + R * sin(Incl + AngleR)),
Form3.PaintBox1.Canvas);
    end else begin
      sx:=round(kx + R * cos(incl));
      sy:=round(ky + R * sin(incl));
      Tree(Age, sx, sy, r + Random(Rnd), Incl -
AngleL);
      Tree(Age, sx, sy, r + Random(Rnd), Incl +
AngleR);
      Line(kx, ky, Round(kx + R * cos(Incl)),
Round(ky + R * sin(Incl)), Form3.PaintBox1.Canvas);
    end;
    end;
```

Затем в обработчике событий кнопки необходимо добавить следующий код:

```
procedure TForm3.Button1Click(Sender: TObject);
var
  X, Y, H:integer;
begin
  Coef := (Pi / 180);
  AngleL := SpinEdit2.Value * Coef;
  AngleR := SpinEdit3.Value * Coef;
  H := SpinEdit4.Value;
  AngleS := 3 / 2 * Pi;
  X := 320;
  Y := 480;
  PaintBox1.Canvas.CleanupInstance;
  PaintBox1.Canvas.Brush.Color := clWhite;
  PaintBox1.Canvas.Rectangle(0,  0,  PaintBox1.Width,
PaintBox1.Height);
  if (SpinEdit1.Value > 0) and (SpinEdit4.Value >= 0)
then
    begin
      AgeF := Spinedit1.Value;
      Tree(0, X, Y, H, AngleS);
    end;
end;
```

Это все, что необходимо для решение первой задачи. Запустив проект и нажав на кнопку, будет нарисовано фрактальное дерево (см. рисунок 2.3).

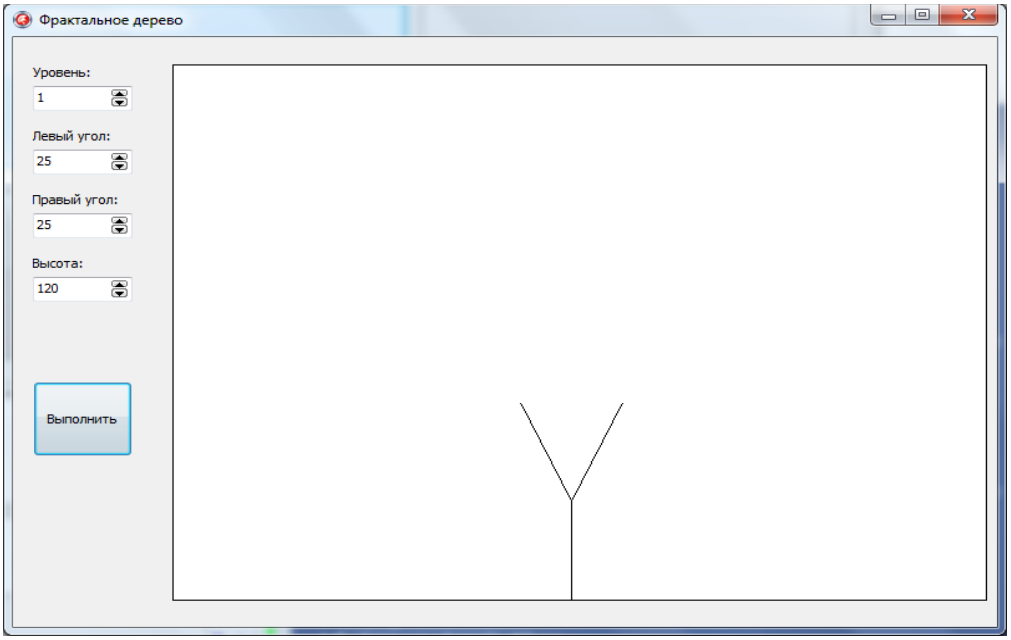

Рисунок 2.3 – Фрактальное дерево 1-го уровня

Меняя настройки, дерево будет также меняться в зависимости от выбранных параметров (см. рисунок 2.4).

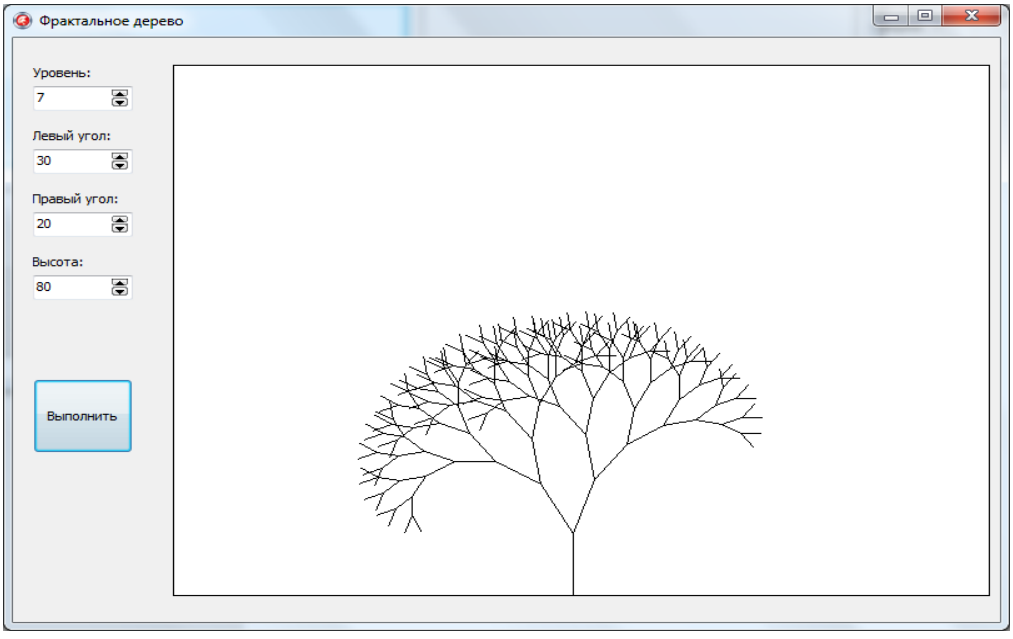

Рисунок 2.4 – Фрактальное дерево 7-го уровня

Меняя различные настройки, дерево будет «расти» по-другому.

2.2.2 Кривая дракона

Кривая дракона — это кривая без самопересечений, которая определяется рекурсивно. Для построения кривой берется отрезок, поворачивается на 90 градусов вокруг одной из вершин и полученный отрезок добавляется к исходному. Получается угол из двух отрезков. Далее данная процедура повторяется, линии будут становиться более сложными, и фигура все больше будет напоминать образ дракона, отсюда и название. Для данной программы достаточно трех компонентов: SpinEdit для настройки глубины, PaintBox для рисования кривой и Button для запуска процесса. Процедура вычисления очередного шага имеет следующий вид:

```
procedure Dragon(Age: integer; x1, y1, x2, y2: real;
n: real);
var
  dx, dy, ac, cx, cy: real;
begin
  Inc(Age);
  if Age = AgeF then Line(x1, y1, x2, y2,
Form4.PaintBox1.Canvas)
  else begin
    cx := (x2 + x1) / 2;
    cy := (y2 + y1) / 2;
    ac := sqrt(sqr(cx - x1) + sqr(cy - y1));
    dx := cx + ac * (cos(n + Pi / 2));
    dy := cy + ac * (sin(n + Pi / 2));
    Dragon(Age, x1, y1, dx, dy, n + 45 * f);
    Dragon(Age, x2, y2, dx, dy, n + 90 * f + 45 * f);
  end;
end;
```

При нажатии на кнопку должна выполниться процедура:

```
procedure TForm4.Button1Click(Sender: TObject);
begin
  x1 := 145;
```

```
  y1 := 160;
  x2 := 560;
  y2 := 160;
  f := (Pi / 180);
  AgeF := SpinEdit1.Value;
  PaintBox1.Canvas.Brush.Color := clWhite;
  PaintBox1.Canvas.Rectangle(0,  0,  PaintBox1.Width,
PaintBox1.Height);
  Dragon(0, x1, y1, x2, y2, 0);
end;
```

Запустив проект (см. рисунок 2.5), изначально будет нарисована только прямая линия, но при увеличении глубины рекурсии (увеличении значения SpinEdit) рисунок будет становиться все более сложным, приобретая образ похожий на дракона (см. рисунок 2.6).

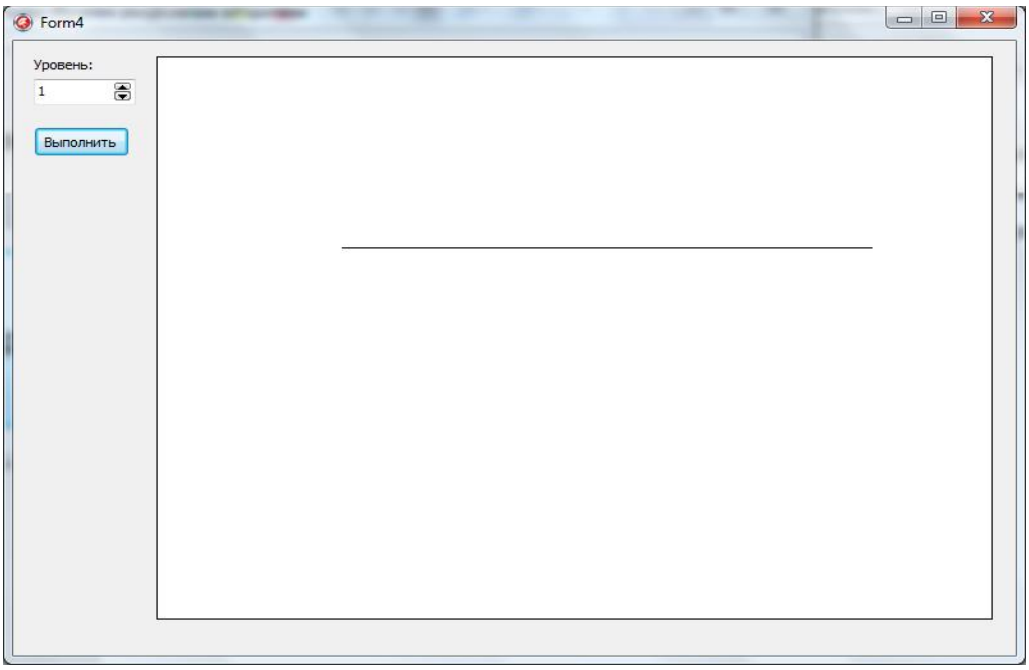

Рисунок 2.5 – Кривая дракона 1-го уровня

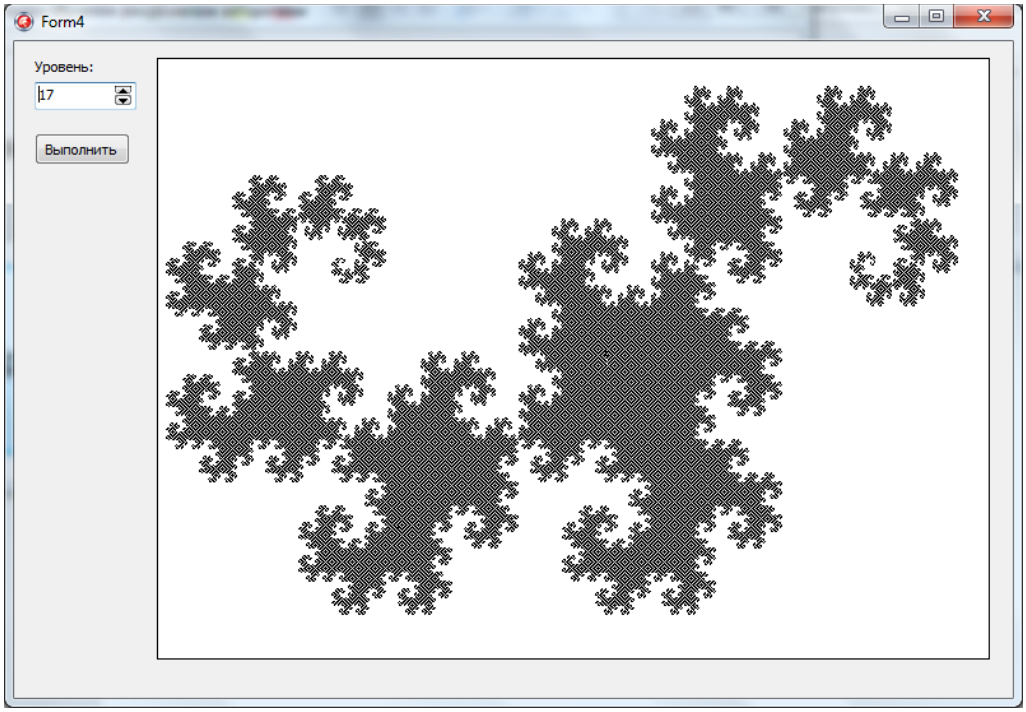

Рисунок 2.6 – Кривая дракона 17-го уровня

Уровни можно продолжать увеличивать, но на общем рисунке больших изменений не будет видно.

2.2.3 Треугольник Серпинского

Это фрактал, один из двумерных аналогов множества Кантора, предложенный польским математиком Вацлавом Серпинским в 1915 году. Данный треугольник имеет следующие особенности состоит из 3 одинаковых частей, коэффициент подобия 1 к 2; состоит из трёх своих копий, уменьшенных в два раза (это части треугольника Серпинского, содержащиеся в маленьких треугольниках, примыкающих к углам); имеет топологическую размерность 1; является замкнутым.

Также как и для кривой дракона, здесь понадобятся следующие компоненты на форме: SpinEdit, PaintBox и Button. Реализация на Object Pascal:

```
procedure Triangle(Age: integer; x1, y1, x2, y2, x3,
y3: real);
var
```

```
  xd, yd, xe, ye, xf, yf: real;
begin
  Inc(Age);
  if Age = AgeF then
    begin
      Line(x1, y1, x2, y2, Form2.PaintBox1.Canvas);
      Line(x2, y2, x3, y3, Form2.PaintBox1.Canvas);
      Line(x3, y3, x1, y1, Form2.PaintBox1.Canvas);
      Form2.PaintBox1.Canvas.Refresh;
    end else begin
      xd := round((x1 + x2) / 2);
      yd := round((y1 + y2) / 2);
      xe := round((x2 + x3) / 2);
      ye := round((y2 + y3) / 2);
      xf := round((x1 + x3) / 2);
      yf := round((y1 + y3) / 2);
      Triangle(age, x1, y1, xd, yd, xf, yf);
      Triangle(age, xd, yd, x2, y2, xe, ye);
      Triangle(age, xf, yf, xe, ye, x3, y3);
    end;
end;
```

Событие OnClick кнопки имеет следующий вид:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  PaintBox1.Canvas.Brush.Color := clWhite;
  PaintBox1.Canvas.Rectangle(0,  0,  PaintBox1.Width,
PaintBox1.Height);
  x1 := 10;
  y1 := 10;
  x2 := 320;
  y2 := 470;
```

```
    x3 := 630;
    y3 := 10;
    PaintBox1.Canvas.CleanupInstance;
    if SpinEdit1.Value > 0 then
    begin
      AgeF := Spinedit1.Value;
      Triangle(0, x1, y1, x2, y2, x3, y3);
    end;
end;
```

При запуске треугольник будет иметь следующий вид (см. рисунок 2.7):

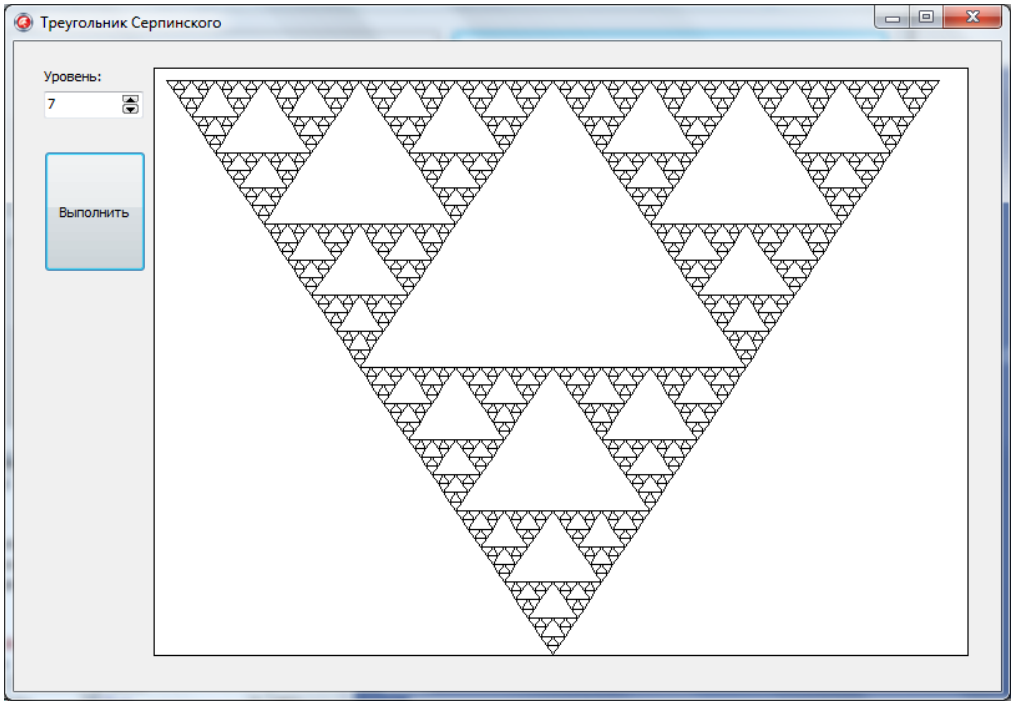

Рисунок 2.7 – Треугольник Серпинского 7-го уровня

Уровни вложенности не ограничены.

2.2.4 Ковер Серпинского

Данный фрактал похож на предыдущий, но вместо треугольника используется квадрат, а коэффициент подобия равен 1 к 3. Понадобятся те же

три компонента на форме (SpinEdit, PaintBox и Button). Процедура выглядит следующим образом:

```
procedure Cover(x1, y1, x2, y2: real; n: integer);
var
x11, y11, x22, y22: real;
r: TRect;
begin
if  n > 0  then
begin
x11 := 2 * x1 / 3 + x2 / 3;
x22 := x1 / 3 + 2 * x2 / 3;
y11 := 2 * y1 / 3 + y2 / 3;
y22 := y1 / 3 + 2 * y2 / 3;
r   :=   Rect(Round(x11),   Round(y11),   Round(x22),
Round(y22));
Form5.PaintBox1.Canvas.FillRect(r);
Cover(x1, y1, x11, y11, n-1);
Cover(x11, y1, x22, y11, n-1);
Cover(x22, y1, x2, y11, n-1);
Cover(x1, y11, x11, y22, n-1);
Cover(x22, y11, x2, y22, n-1);
Cover(x1, y22, x11, y2, n-1);
Cover(x11, y22, x22, y2, n-1);
Cover(x22, y22, x2, y2, n-1);
end;
end;
```

Событие OnClick для кнопки:

```
procedure TForm5.Button1Click(Sender: TObject);
var
r: TRect;
begin
```

```
Form5.Refresh;
x1 := 10;
y1 := 10;
x2 := 500;
y2 := 500;
Form5.PaintBox1.Canvas.Brush.Color := clWhite;
r := Rect(0, 0, PaintBox1.Width, PaintBox1.Height);
Form5.PaintBox1.Canvas.FillRect(r);
Form5.PaintBox1.Canvas.Brush.Color:= clBlack;
Cover(x1, y1, x2, y2, SpinEdit1.Value);
end;
```

Запустив проект, программа будет строить при помощи рекурсии следующий объект (см. рисунок 2.8):

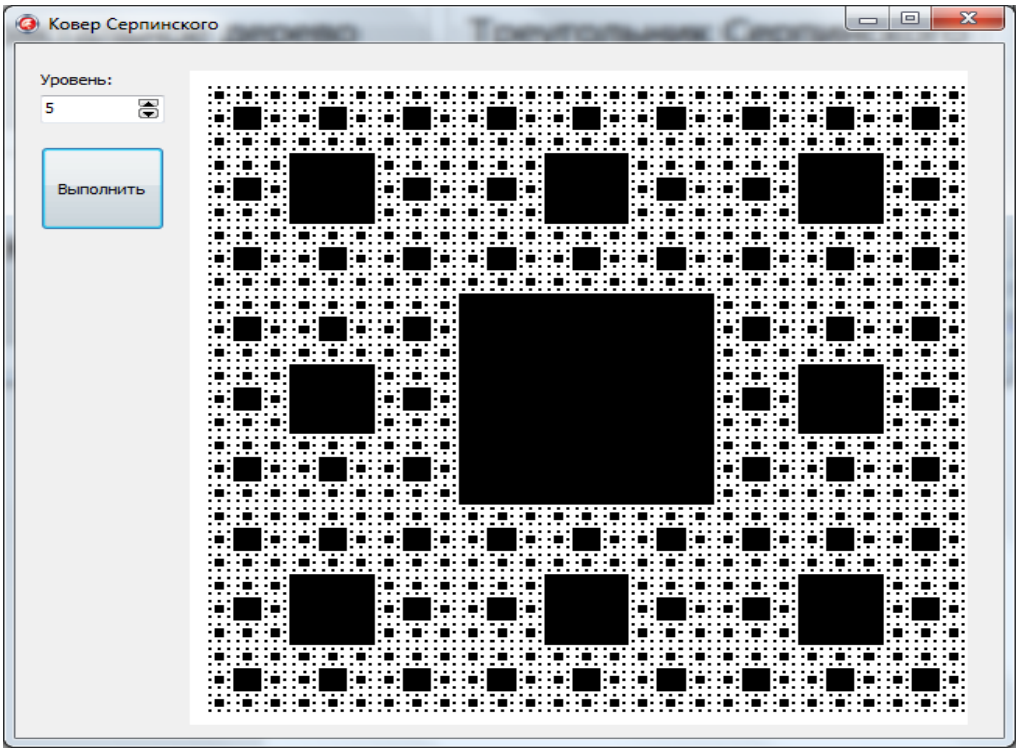

Рисунок 2.8 – Ковер Серпинского 5-го уровня

Это классические и самые распространенные примеры, демонстрирующие работу рекурсивных алгоритмов. Есть еще одна очень распространенная задача,

заключающаяся в поиске минимального пути в лабиринте, но она будет подробно расписана в следующем параграфе.

2.3 Разработка программы обучения рекурсии в виде модуля для PascalABC

Модуль на PascalABC - это автономно компилируемая программная единица, включающая в себя различные компоненты раздела описаний (типы, константы, переменные, процедуры и функции) и, возможно, некоторые исполняемые операторы инициирующей части. Говоря простым языком – это набор компонентов, позволяющий расширить функционал основной программы.

В языке программирования Pascal перед использованием компонентов модуля его нужно подключить в разделе Uses, указав имя подключаемого модуля.

В конечной программе нужно организовать удобный и красивый интерфейс. Сделать это можно, подключив встроенные библиотеки PascalABC.NET: FormsABC, GraphABC, ABCObjects. Библиотеки богаты на различные полезные функции, но использование одновременно этих трех компонентов довольно затруднительно, поэтому нужно найти компромиссный вариант, который позволит и создать удобный интерфейс, и использовать по максимуму всевозможные элементы: кнопки, формы, поля ввода, трекбары, чекбоксы и т.д. Отличным средством для решения подобной задачи является библиотека NET. Стоит отметить, что использовать данную библиотеку можно только в PascalABC.NET, в простом PascalABC подобной возможности нет.

Разработку модуля будем производить в 2 этапа: на первом разрабатывается конкретная программа, позволяющая протестировать на ошибки будущий продукт, а на втором этапе эта программа будет переведена

непосредственно в модуль, что позволит использовать ее для решения более широкого класса задач [15-30].

Как уже было рассмотрено ранее в предыдущей главе, существует множество программ, использующих рекурсивные алгоритмы. Наиболее интересным и функциональным на наш взгляд является рекурсивный алгоритм поиска кратчайшего пути в заданном лабиринте.

Напомним условие задачи. Дан лабиринт, состоящий из проходов, по которым можно передвигаться, и стен, сквозь которые передвигаться нельзя. Задаются точки входа и выхода, и нужно отыскать кратчайший путь из точки старта в точку финиша. Рекурсивным элементом в данной программе является движение по лабиринту с проверкой конкретных ячеек.

Создаем новый документ с названием Lab.pas. Инициализируем программу и консольный режим.

```
program Lab;
uses crt;
```

Далее идет раздел объявления переменных. Понадобятся как минимум следующий переменные: f1 и f2 – переменные текстовых файлов для ввода/вывода, a – символьный массив, представляющий матрицу лабиринта, x1 и y1– координаты начала, x2 и y2 – координаты конца, p и min – переменные длины пути и n – размерность лабиринта.

```
var
    f1, f2: text;
    a: array[0..41, 0..41] of char;
    x1, y1, x2, y2, p, min, n: integer;
```

После объявления переменных идут процедуры и функции. Достаточно будет трех процедур: ввести лабиринт из файла, графически показать нахождение пути и непосредственно процедура поиска пути. Ввод лабиринта производится из текстового файла с именем input.txt, вывод лабиринта – в output.txt. Лабиринт в текстовом файле будет представлен следующим образом: проход обозначен точками, стены знаком «#», точка входа «S» и точка выхода «F». Размера лабиринта 10*10 будет достаточно, точка входа имеет координаты

[1, 1], точка выхода [10, 10]. Важно также предусмотреть вероятность выхода за пределы лабиринта, исключить эту возможность можно, окружив лабиринт стенами. Таким образом, получится следующая процедура:

```
procedure input;
 var
  i, j: integer;
 begin
  clrscr;
  readln(f1, n);
  for i:=1 to n do
    for j:=1 to n do
     begin
      read(f1, a[i, j]);
      if a[i, j]='S' then begin x1:=i; y1:=j; end;
      if a[i, j]='F' then begin x1:=i; y1:=j; end;
     end;
  for i:=0 to n+1 do
    begin
     a[0, i]:='#';
     a[n+1, i]:='#';
     a[i, 0]:='#';
     a[i, n+1]:='#';
    end;
 end;
```

Для того, чтобы видеть как программа ищет кратчайший путь, нужно организовать консольный вывод. Делается это следующим образом:

```
procedure output(x, y: integer);
 var
  i, j: integer;
 begin
  clrscr;
```

```
 for i:=0 to n+1 do
  begin
   for j:=0 to n+1 do
    begin
     if (x=i) and (y=j) then write('@')
     else
        begin
         if a[i, j]='.' then write('.');
         if a[i, j]='#' then write('#');
         if a[i, j]='+' then write('+');
        end;
     end;
    writeln;
   end;
  readkey;
 end;
```

Осталось написать процедуру поиска пути. В ней нужно предусмотреть следующий моменты: при столкновении со стеной произвести поиск другого пути; не допустить прохождения по уже посещенным ячейкам; отсекать заведомо неподходящие решения, например, при помощи формулы p+abs(x-x2)+abs(y-y2); считать пройденный путь; при достижении финиша запоминать путь если он меньше найденного ранее.

Кроме того, как уже упоминалось ранее, в этой процедуре необходимо применить рекурсию. Суть ее будет заключаться в перемещении активной ячейки в соседнюю сверху, снизу, слева или справа. Получим следующую процедуру:

```
procedure move(x, y: integer);
 begin
  if p+abs(x-x2)+abs(y-y2)>=min then exit;
  if a[x, y]='#' then exit;
  if a[x, y]='+' then exit;
```

```
  if a[x, y]='F' then
   begin
    if p<min then min:=p;
    exit;
   end;
   output(x, y);
  a[x, y]:='+';
  inc(p);
  move(x, y+1);
  move(x-1, y);
  move(x, y-1);
  move(x+1, y);
  dec(p);
  a[x, y]:='.';
 end;
```

Осталось теперь написать основную программу, которая бы вызывала необходимые процедуры.

```
begin
 assign(f1, 'input.txt');
 reset(f1);
 input;
 min:=1600;
 p:=1;
 move(x1, y1);
 close(f1);
end.
```

У нас получилась готовая программа, которая ищет кратчайший путь. Запустив ее, появится лабиринт и в нем будут производиться необходимые манипуляции (см. рисунок 2.9).

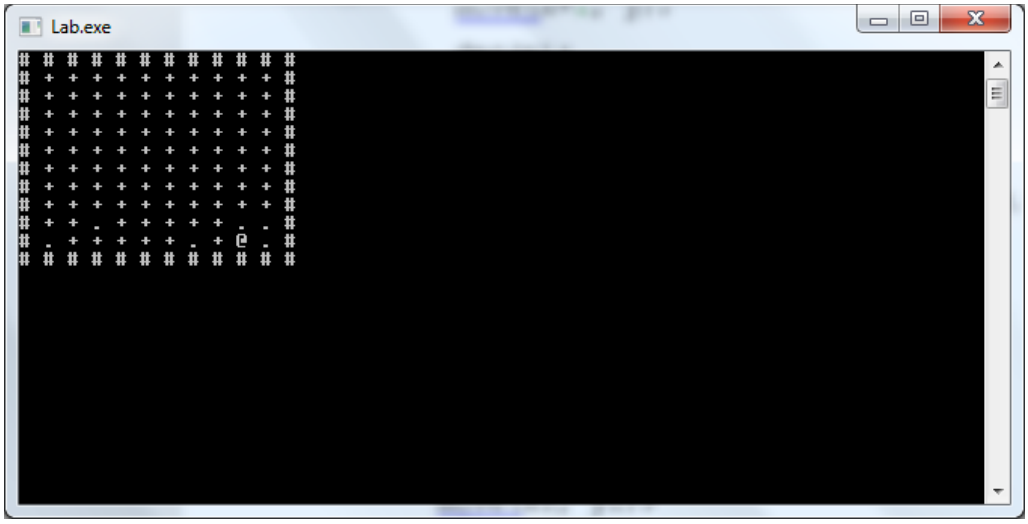

Рисунок 2.9 – Консольная версия программы поиска кратчайшего пути

Но данная программа хотя и выполняет необходимые действия, но она совсем простая, неудобная, не обладает большим функционалом и мы никак не можем повлиять на процесс поиска, кроме как нажатием любой клавиши для следующего хода.

Существенно расширить функционал поможет библиотека NET. Создадим новый документ под названием LabG.pas. Сперва нужно инициализировать необходимые компоненты, добавив в самом начале следующий код:

```
{$apptype windows}
{$reference 'System.Windows.Forms.dll'}
{$reference 'System.Drawing.dll'}
{$gendoc true}
```

Это позволит нам подключить пространства имен System и его подпространства. Делается это при помощи команды uses.

```
uses
  System,
  System.Drawing,
  System.Threading,
  System.Windows.Forms;
```

Далее идет описание переменных. Их теперь будет гораздо больше и типы будут куда более разнообразны. Помимо знакомых Integer, Boolean, String и Real будут использованы PictureBox для вывода изображений, Button для добавления кнопок, TrackBar для добавления трекбара (полосы с ползунком), Thread для организации нескольких потоков, RadioBox для добавления радиокнопок, OpenFileDialog для создания диалога загрузки файлов, Label для добавления меток (текста). Этот список можно продолжать до бесконечности, но обо всем по порядку. Первым же элементом, который нужно добавить – это сама форма. Форма – это основа, на которой будут располагаться все остальные компоненты. Назовем ее MainForm.

```
var
  MainForm: Form;
```

Объявления в разделе переменных для создания формы недостаточно, эту форму еще нужно создать. В основной части программы после Begin пишем:

```
MainForm := new Form;
```

Но при компиляции опять ничего не происходит. Объявить и создать форму тоже оказалось недостаточно, нужно ее еще запустить.

```
Application.Run(MainForm);
```

Теперь при компиляции появляется форма, но она пустая, без заголовков и маленького размера. Это нужно исправить, использовав методы Text, Width и Height для добавления заголовка, установления ширины и высоты соответственно.

Но так как задания могут быть разной размерности, то подобрать ширину и высоту проблематично. Логично будет сразу же при запуске разворачивать форму на полный экран при помощи команды:

```
MainForm.Text := 'Программа обучения рекурсии';
MainForm.WindowState := FormWindowState.Maximized;
```

Форма готова и теперь на нее можно добавлять компоненты. Все компоненты условно можно разделить на 2 группы: рабочая область и панель управления. Каждая группа будет располагаться на компоненте «Панель». Для этого в разделе описания переменных инициализируются компоненты.

```
CPanel, WPanel: Panel;
```

Прежде чем добавлять панели на форму, создадим отдельную процедуру, в которой будут находиться объявления всех элементов формы. Назовем эту процедуру InitControls. В этой процедуре создадим две панели, установим им размер и поместим на форму.

```
CPanel := new Panel;
CPanel.Width := 300;
CPanel.BorderStyle := BorderStyle.Fixed3D;
CPanel.Dock := DockStyle.Right;
MainForm.Controls.Add(CPanel);
WPanel := new Panel;
WPanel.BorderStyle := BorderStyle.Fixed3D;
WPanel.Dock := DockStyle.Fill;
Wpanel.SendToBack;
MainForm.Controls.Add(WPanel);
```

Не забыв вызвать процедуру InitControls в основной части программы, форма будет содержать две панели. Внешне форма будет иметь следующий вид (см. рисунок 2.10):

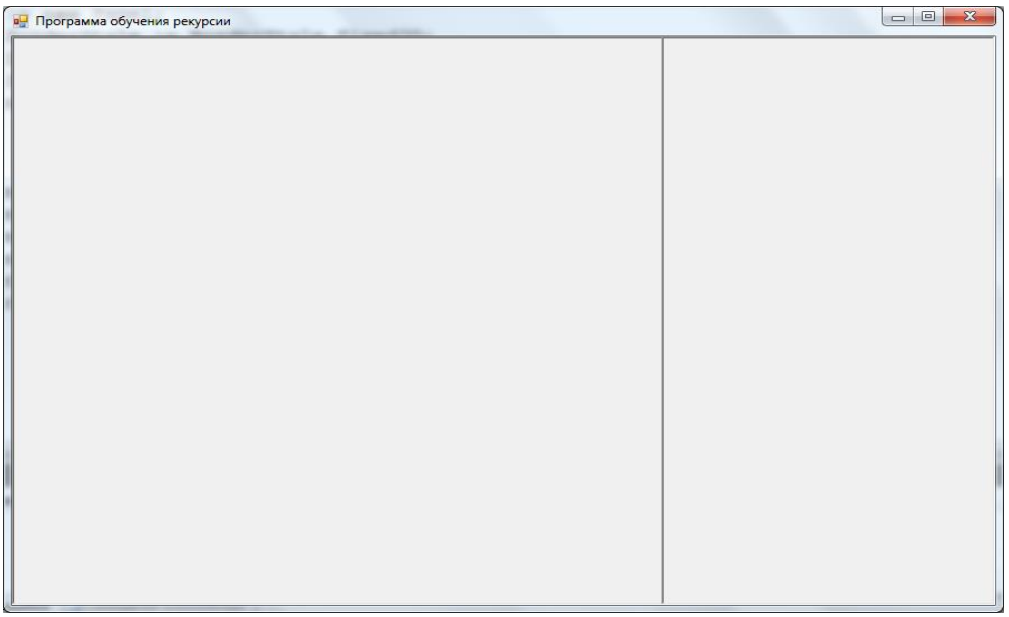

Рисунок 2.10 – Форма с панелями

Начнем заполнять панель управления. На нее необходимо поместить 5 кнопок (запуск, пауза, кнопка шага, загрузка файла и перезапуск окна), трекбар (регулировка скорости), 4 радиокнопки (переключение режимов), несколько меток (подписи элементов) и блок информации (вывод результата и промежуточных данных). В разделе описания переменных описываем эти объекты:

```
Start, Pause, Restart, Step, RestartLab: Button;
Speed: Trackbar;
Mode1, Mode2, Mode3, Mode4: RadioButton;
Info: RichTextBox;
SpeedName, Modes: system.Windows.Forms.Label;
```

В процедуре InitControls нужно создать все эти объекты при помощи конструктора New.

```
Start := new Button;
Pause := new Button;
Restart := new Button;
RestartLab := new Button;
Step := new Button;
Speed := new TrackBar;
Mode1 := new Radiobutton;
Mode2 := new Radiobutton;
Mode3 := new Radiobutton;
Mode4 := new Radiobutton;
SpeedName := new System.Windows.Forms.Label;
Modes := new System.Windows.Forms.Label;
Info := new RichTextBox;
```

Так как в среде PascalABC невозможно разместить компоненты при помощи мыши методом drag-and-drop, то придется заниматься размещением вручную, прописывая координаты. У каждого компонента имеется свойство Location, отвечающее за это. Вызвав это свойство, нужно указать координаты верхнего левого угла элемента. По просто прописать координаты не получится,

поэтому необходимо создать объект Point с нужными координатами. Важный нюанс: так как элементы управления находятся на панели, то и началом координат будет являться не верхняя левая точка формы, а верхняя левая точка панели. Разместим все элементы следующим образом:

```
Start.Location := new Point(10, 10);
Pause.Location := new Point(10, 10);
Restart.Location := new Point(10, 420);
RestartLab.Location := new Point(150, 420);
SpeedName.Location := new Point(10, 80);
Modes.Location := new Point(10, 190);
Step.Location := new Point(10, 370);
Speed.Location := new Point(10, 110);
Mode1.Location := new Point(10, 210);
Mode2.Location := new Point(10, 240);
Mode3.Location := new Point(10, 270);
Mode4.Location := new Point(10, 300);
Info.Location := new Point(0, 480);
```

Затем, отталкиваясь от позиций компонентов, нужно указать их размеры. Можно указывать отдельно ширину через свойство Width и высоту через Height, но это увеличит и объем кода, и затраченное время. Использование свойства Size в данном случае более рационально. При использовании этого свойства создается новый класс Size, в котором указывается ширина и высота. Были рассчитаны следующие размеры:

```
Start.Size := new Size(280, 50);
Pause.Size := new Size(280, 50);
Restart.Size := new Size(140, 50);
RestartLab.Size := new Size(140, 50);
SpeedName.Size := new Size(280, 30);
Modes.Size := new Size(280, 30);
Step.Size := new Size(280, 40);
Speed.Size := new Size(280, 80);
```

```
Mode1.Size := new Size(280, 40);
Mode2.Size := new Size(280, 40);
Mode3.Size := new Size(280, 40);
Mode4.Size := new Size(280, 40);
Info.Height := 80;
Info.Dock := DockStyle.Bottom;
```

Блок информации закреплен снизу, чтобы он не «плавал» при изменении размеров окна. Компоненты созданы, размещены на панели, отрегулированы размеры, но непонятно что каждый компонент значит. Решить эту проблему можно просто подписав их. Свойство Text отвечает как раз за это.

```
Start.Text := 'Старт';
Pause.Text := 'Стоп';
Restart.Text := 'Перезапустить программу';
RestartLab.Text := 'Загрузить файл';
SpeedName.Text := 'Скорость:';
Modes.Text := 'Режимы:';
Step.Text := 'Сделать шаг';
Mode1.Text := 'Обычный поиск';
Mode2.Text := 'Пошаговый поиск';
Mode3.Text := 'Поиск выхода с отображением пути';
Mode4.Text := 'Поиск выхода без отображения пути';
```

Запустив проект, никаких элементов управления не видно, а все потому, что они не были помещены на панель. Поместим их туда уже знакомой комбинацией.

```
CPanel.Controls.Add(Start);
CPanel.Controls.Add(Pause);
CPanel.Controls.Add(Restart);
CPanel.Controls.Add(RestartLab);
CPanel.Controls.Add(SpeedName);
CPanel.Controls.Add(Modes);
CPanel.Controls.Add(Step);
```

```
CPanel.Controls.Add(Speed);
CPanel.Controls.Add(Mode1);
CPanel.Controls.Add(Mode2);
CPanel.Controls.Add(Mode3);
CPanel.Controls.Add(Mode4);
CPanel.Controls.Add(Mode5);
CPanel.Controls.Add(Info);
```

Теперь, запустив проект, панель управления будет содержать объявленные компоненты (см. рисунок 2.11), ими можно оперировать, но никаких функций они не выполняют, пока что.

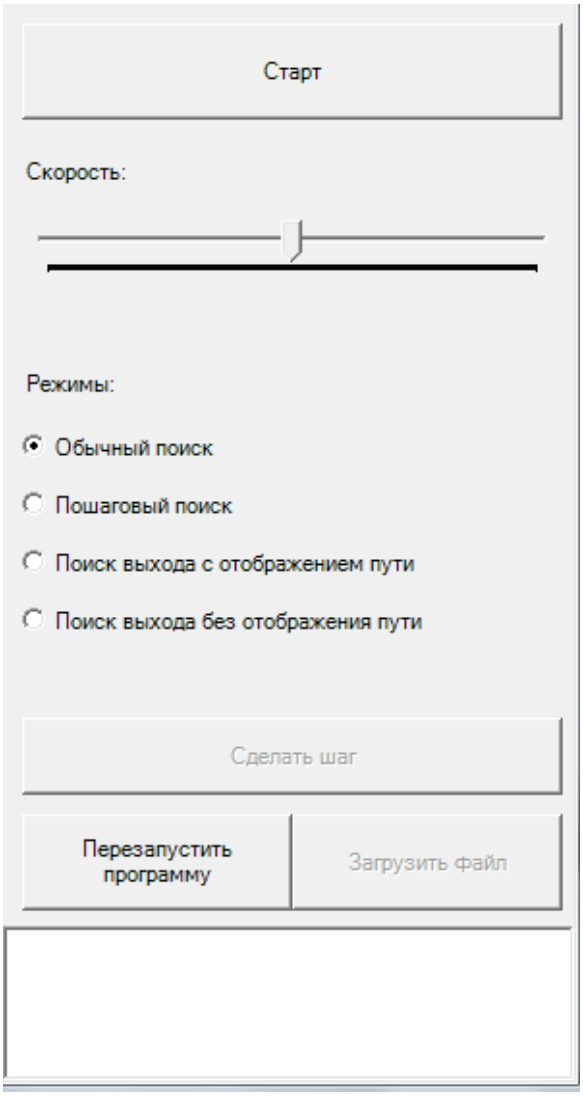

Рисунок 2.11 – Панель управления

Пока оставим панель управления в таком виде и перейдем к формированию лабиринта. Формироваться он будет из обычного текстового файла. В разделе описания переменных инициализируем переменную LabFromText типа textfile.

```
LabFromText: textfile;
```

Кроме того, понадобится двумерный массив и несколько целочисленных переменных: x1 и y1 (координаты входа), x2 и y2 (координаты выхода), n (размерность лабиринта), p (промежуточный путь), min (минимальный найденный путь), x и y (текущее положение маркера).

```
a: array[0..21, 0..21] of integer;
x1, y1, x2, y2, p, min, n, x, y: integer;
```

Вводить лабиринт будем в процедуре Input. Нам понадобится локальная переменная символьного типа. Далее в самой процедуре открываем файл на чтение, считываем двойным циклом считываем символы лабиринта и заносим их в массив, меняя символы на числовые значения для более удобной работы.

```
var
  c: char;
begin
  assign(LabFromText, fname);
  reset(LabFromText);
  readln(LabFromText, n);
  for j: integer := 1 to n do
  begin
    for i: integer := 1 to n do
    begin
      read(LabFromText, c);
      if c = '1' then a[i, j] := -1;
      if c = '0' then a[i, j] := 0;
      if c = 'S' then begin a[i, j] := 0;x1 := i;y1
:= j; end;
```

```
        if c = 'F' then begin a[i, j] := 0;x2 := i;y2
:= j; end;
      end;
      readln(LabFromText);
    end;
    close(LabFromText);
```

Чтобы предотвратить выходы за границу лабиринта, нужно сделать дополнительную рамку для сетки.

```
for i: integer := 0 to n + 1 do
  begin
    a[0, i] := -1;
    a[n + 1, i] := -1;
    a[i, 0] := -1;
    a[i, n + 1] := -1;
  end;
```

Далее нужно перевести символьные значения в изображения. Для начала необходимо найти или нарисовать самостоятельно следующие графические изображения: проход, стена, выход, персонаж и путь. Размеров 32x32 пикселей будет достаточно. Вывести изображения поможет класс PictureBox. Сначала он создается, указывается его размер, позиция, потом ему передается изображение из файла и затем он помещается на рабочую область, в нашем случае на панель WPanel.

```
for j: integer := 0 to n + 1 do
for i: integer := 0 to n + 1 do
begin
  Cell := new PictureBox;
  Cell.Size := new Size(32, 32);
  Cell.Location := new Point(i * 32, j * 32);
  if a[i, j] = -1 then begin
    Cell.BackgroundImage:=Image.FromFile('wall.png');
    WPanel.Controls.Add(Cell);
```

```
    end else begin
      Cell.BackgroundImage:=Image.FromFile('grass.png');
       WPanel.Controls.Add(Cell);
    end;
  end;
```

Чтобы проверить правильно ли формируется лабиринт, нужно эту процедуру вызвать. Вызывать ее нужно в обработчике событий нажатия кнопки LoadLab. Для этого создается еще одна процедура, например, LoadProc. В ней создается компонент OpenFileDialog – диалоговое окно открытия файла. При успешном открытии файла происходит запуск процедуры Input. Диалоговому окну можно добавить фильтр, чтобы не отображать никакие другие файлы, кроме текстовых. Полностью данная процедура будет выглядеть так:

```
procedure LoadProc(sender: Object; e: EventArgs);
begin
   WPanel.Controls.Clear;
   OpenLab := new OpenFileDialog;
   OpenLab.Filter := 'txt files|*.txt';
   if OpenLab.ShowDialog = DialogResult.OK then
    FName := OpenLab.FileName;
    Input;
    Restart.Enabled:=true;
    LoadLab.Enabled:=false;
    Start.Enabled:=true;
end;
```

В процедуре InitControls перед выводом кнопок нужно кнопке загрузки лабиринта LoadLab передать эту процедуру LoadProc следующим образом:

```
LoadLab.Click += LoadProc;
```

Теперь при запуске проекта и нажатии на кнопку загрузки файла будет открываться диалоговое окно, в котором нужно выбрать текстовый файл, из которого считывать лабиринт. После этих действий на форме появится красочный графический лабиринт (см. рисунок 2.12).

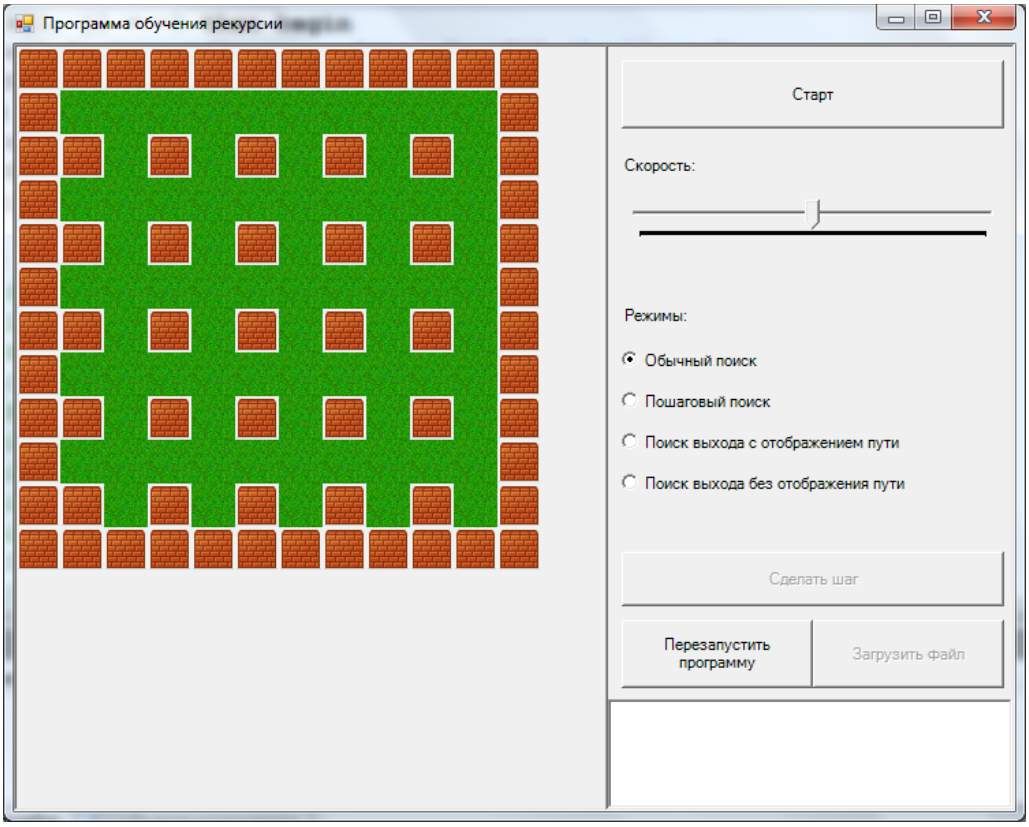

Рисунок 2.12 – Формирование лабиринта

Но внутри только проходы и стены. Непонятно где выход и вход в лабиринт. Возвращаемся в процедуру Input и добавляем туда эти элементы при помощи уже знакомого PictureBox.

```
Player := new PictureBox;
Player.Size := new Size(32, 32);
Player.Location := new Point(x1 * 32, y1 * 32);
Player.BackgroundImage := Image.FromFile('man.png');
WPanel.Controls.Add(Player);
Player.BringToFront();
CellFinish := new PictureBox;
CellFinish.Size := new Size(32, 32);
CellFinish.Location := new Point(x2 * 32, y2 * 32);
CellFinish.BackgroundImage:=Image.FromFile('target.pn
g');
WPanel.Controls.Add(CellFinish);
CellFinish.BringToFront();
```

Метод BringToFront() помещает элемент на передний план, чтобы его не перекрывали фоновые изображения. И последнее что необходимо сделать в процедуре считывания лабиринта – это вывести в окно информации координаты входа и выхода. Сделать это можно при помощи свойства AppendText компонента RichTextBox.

```
Info.AppendText('Координаты начала: X = ' + x1 + '
Y = ' + y1 + Environment.NewLine);
Info.AppendText('Координаты финиша: X = ' + x2 + '
Y = ' + y2 + Environment.NewLine);
```

Команда Environment.NewLine осуществляет перевод строки. Результат проделанных операций показан ниже (см. рисунок 2.13).

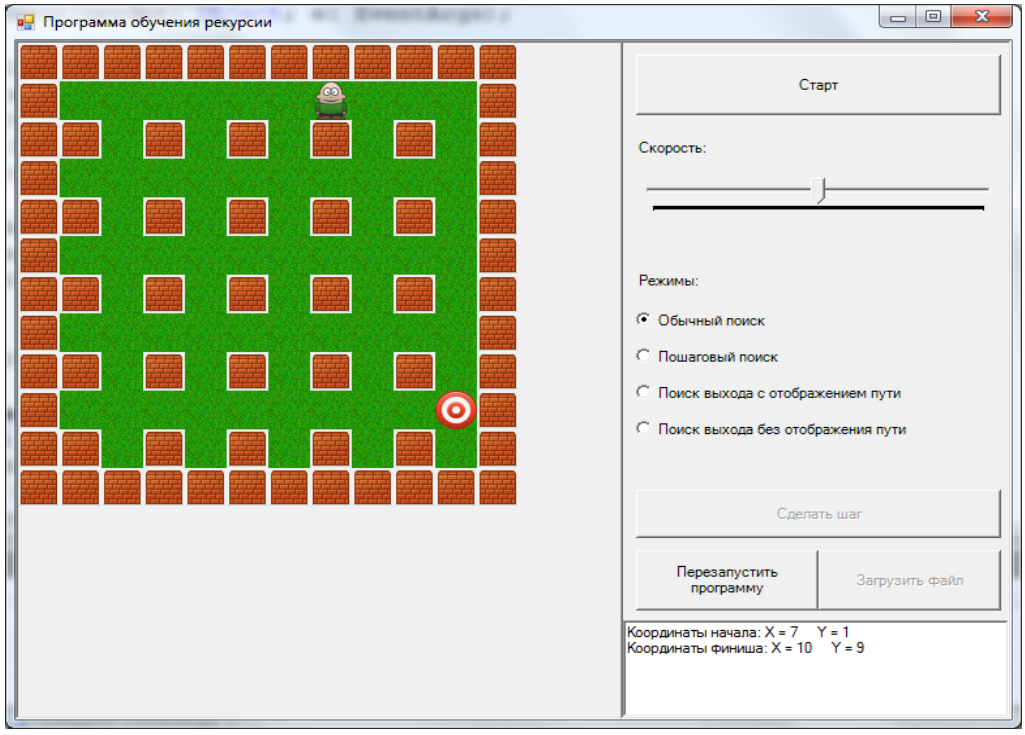

Рисунок 2.13 – Сгенерированный лабиринт

Готовый лабиринт есть, теперь нужно организовать поиск в нем. Процедуру поиска разделим на две части: первая выполняется до входа в рекурсию, вторая – после. Назовем их соответственно RecEnter и RecExit.

В первой части мы выводим то, как производится поиск, то есть перемещение иконки персонажа и установка отметок на тех клетках, где он был.

```
Player.Location := new Point(x * 32, y * 32);
if (a[x, y] = 2) then begin
    PathShow := new PictureBox;
    PathShow.Size := new Size(32, 32);
    PathShow.Location := new Point(x * 32, y * 32);
    PathShow.BackgroundImage
    :=Image.FromFile('step.png');
    WPanel.Invoke(Path);
    PathShow.BringToFront();
    Player.BringToFront;
        end;
```

Во второй части, когда происходит рекурсивный подъем, нужно наоборот убирать отметки о пути, чтобы произвести поиск по другому маршруту.

```
PathShow := new PictureBox;
PathShow.Size := new Size(32, 32);
PathShow.Location := new Point(x * 32, y * 32);
PathShow.BackgroundImage
:=Image.FromFile('grass.png');
WPanel.Invoke(Path);
PathShow.BringToFront();
Player.BringToFront();
CellFinish.BringToFront();
```

В этих двух частях встречается такой метод как Invoke. Он необходим для организации многопоточности. Использование нескольких потоков в программе значительно ускоряет ее работу и позволяет параллельно производить несколько действий. В данном случае это позволит управлять поиском прямо во время его выполнения: ускорить или замедлить, остановить или снова запустить, переключить режим поиска. Поток, также как и другие компоненты

нужно объявить, создать и запустить. В разделе описания переменных поток описывается следующим образом:

```
LabThread: System.Threading.Thread;
```

Затем поток создается:

```
LabThread := new Thread(StartSearch);
```

И запускается. Запускать его логично при нажатии на кнопку «Старт», но при повторном нажатии на данную кнопку программа попытается запустить уже активный поток и это приведет к ошибке. Поэтому нужно проследить, чтобы поток был запущен всего один раз. Самым простым способом сделать это является использование логической переменной. Если эта переменная, например, ложна, то поток запускается и меняет значение переменной на истину. Таким образом, поток будет запущен всего однажды. Создадим процедуру StartThread() для запуска потока и процесса поиска, и сделаем запуск этой процедуры по нажатию кнопки «Старт».

```
Start.Click += StartThread;
```

Теперь при запуске программы и нажатии на кнопку начнется поиск по лабиринту (см. рисунок 2.14).

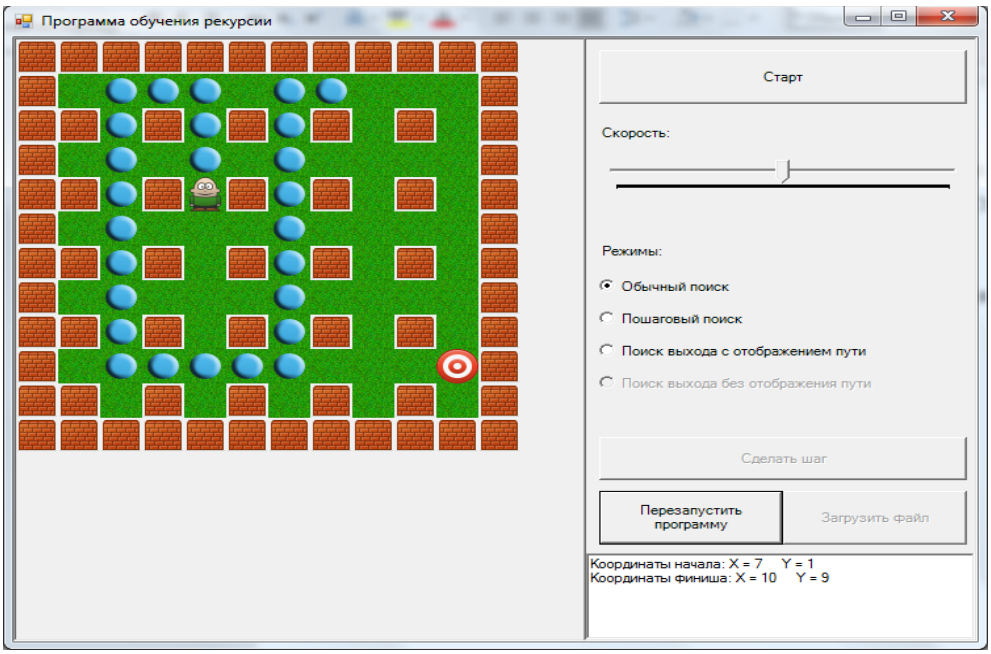

Рисунок 2.14 – Поиск пути

Кнопки «Старт» и «Загрузить файл» - единственные компоненты на панели управления, которые функционируют. Сделаем возможность ставить поиск на паузу и менять скорость поиска. Но сперва сделаем следующее: при запуске поиска прятать кнопку «Старт» и показывать кнопку «Стоп» и наоборот. В процедуру StartThread добавим строчки:

```
Press := false;
Start.Visible:=false;
Pause.Visible:=true;
```

А для кнопки «Стоп» создадим новую процедуру, например, StopSearch и внесем в нее следующий код:

```
Press := true;
Pause.Visible:=false;
Start.Visible:=true;
```

В InitControls добавим обработчик событий по аналогии с предыдущими кнопками:

```
Pause.Click += StopSearch;
```

И теперь в процедуру RecEnter добавим такие строки:

```
if Press = false then Sleep(Speed.Value) else
    begin
    repeat sleep(100) until Press = false; end;
```

Уже не первый раз встречается переменная Press, она хранит логическое значение нажата ли кнопка паузы. То есть, исходя из кода выше, если пауза не нажата, то скорость регулируется трекбаром, иначе – процесс поиска стоит на паузе пока не будет снова нажата кнопка «Старт». При запуске программы теперь работает и постановка на паузу и регулировка скорости.

Прежде чем перейти к настройке режимов поиска, нужно сделать функционирующей последнюю кнопку – перезапуск программы. Все делается по аналогии: создается новая процедура, а затем производится ее вызов при нажатии на кнопку. Процедура будет содержать всего одну строку:

```
Application.Restart();
```

Теперь перейдем к программированию режимов. Пока работает только самый первый режим – обычный поиск до тех пор, пока не будет найден кратчайший путь. Необходимо реализовать еще 2 режима: пошаговый поиск и поиск до нахождения выхода. Последний содержит две вариации: с показом процесса поиска и с показом только результата.

Начнем по порядку. Кнопка очередного шага должна быть неактивна при обычном поиске и активна при включении других режимов.

```
if (Mode2.Checked=true) or (Mode3.Checked=true) or
(Mode4.Checked=true) then Step.Enabled:=true
else Step.Enabled:=false;
```

Персонаж не должен менять свое положение если включен 4-й режим – поиска без отображения пути.

```
if (Mode4.Checked=false) then Player.Location := new
Point(x * 32, y * 32);
```

Пошаговый режим должен останавливать поиск при выполнении очередного шага.

```
if Mode2.Checked=true then begin
    ShowStep := true;
        repeat sleep(100) until (ShowStep = false)
    or (Mode2.Checked = false);
end;
```

Если включен 3-й или 4-й режимы (поиск до выхода), нужно автоматически остановить поиск при достижении точки выхода.

```
if (x = x2) and (y = y2) and (Mode3.Checked = true)
then begin
    ShowStep := true;
repeat sleep(100) until (ShowStep = false) or
(Mode3.Checked = false);
end;
```

Если запустить программу, то можно менять режимы и процесс поиска будет меняться при смене режима. Но есть один недостаток: когда

минимальный путь найден, то он не отображается из-за рекурсивного всплытия (см. рисунок 2.15).

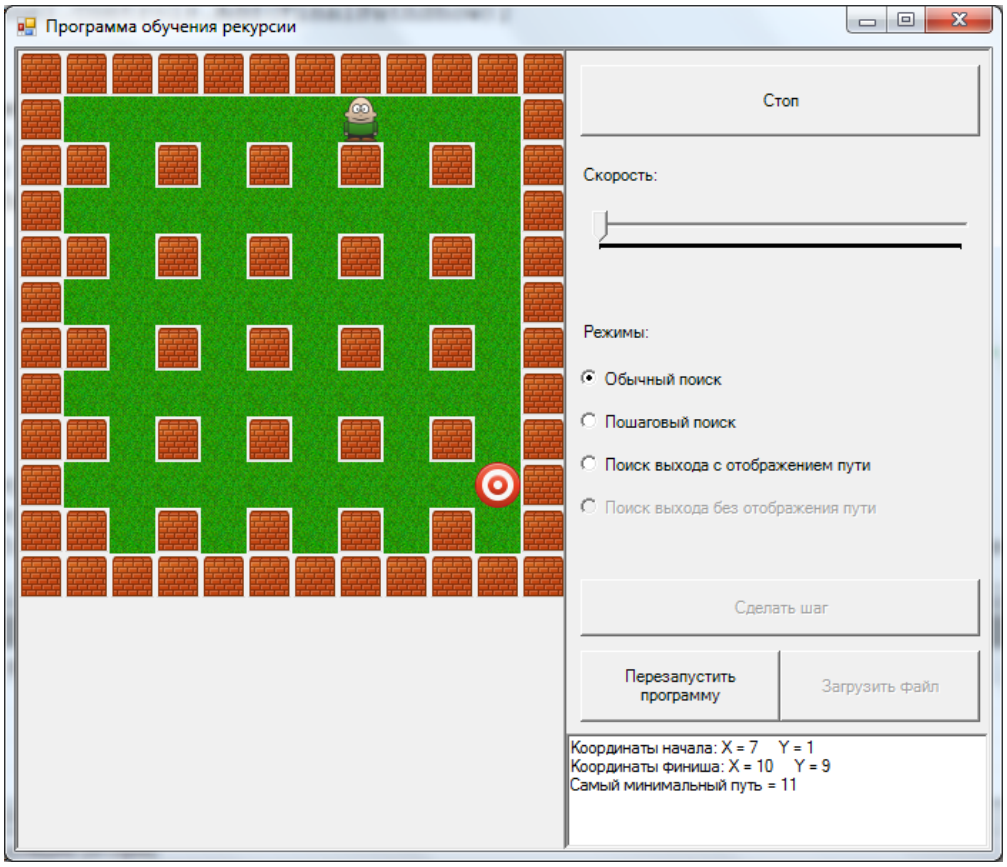

Рисунок 2.15 – Минимальный путь

Решить эту проблему можно разными способами, но наиболее простым является использование дополнительного массива, в котором будет храниться координаты последнего найденного пути. В RecEnter нужно добавить следующий код:

```
if (x = x2) and (y = y2) then begin
    for i: integer := 1 to n do
      for j: integer := 1 to n do b[i, j] := 0;
    for i: integer := 1 to n do
      for j: integer := 1 to n do if a[i, j] = 2 then
b[i, j] := 2;
  end;
```

Суть вышеприведенного кода в следующем: при достижении точки выхода сначала производится удаление предыдущего найденного пути (если он был) и затем запоминается новый более короткий путь.

Теперь нужно после завершения процедуры поиска вывести содержимое дополнительного массива. Удобнее это сделать в отдельной процедуре, а также отметками другого цвета. Процедура будет иметь следующий вид:

```
procedure StartSearchAnswer;
begin
  for i: integer := 1 to n do
    for j: integer := 1 to n do
      if b[i, j] = 2 then begin
        FinalPathShow := new PictureBox;
        FinalPathShow.Size := new Size(32, 32);
        FinalPathShow.Location := new Point(i * 32, j
* 32);
        FinalPathShow.BackgroundImage
:=Image.FromFile('path.png');
        WPanel.Invoke(FinalPath);
        FinalPathShow.BringToFront();
      end;
  Player.Location := new Point(x1 * 32, y1 * 32);
  Player.BringToFront;
  Info.AppendText('Самый минимальный путь = ' + min +
Environment.NewLine);
end;
```

Вызывать ее нужно сразу же после процедуры поиска пути. Теперь после завершения поиска лабиринт покажет не только длину минимального пути, но и сам этот путь (см. рисунок 2.16).

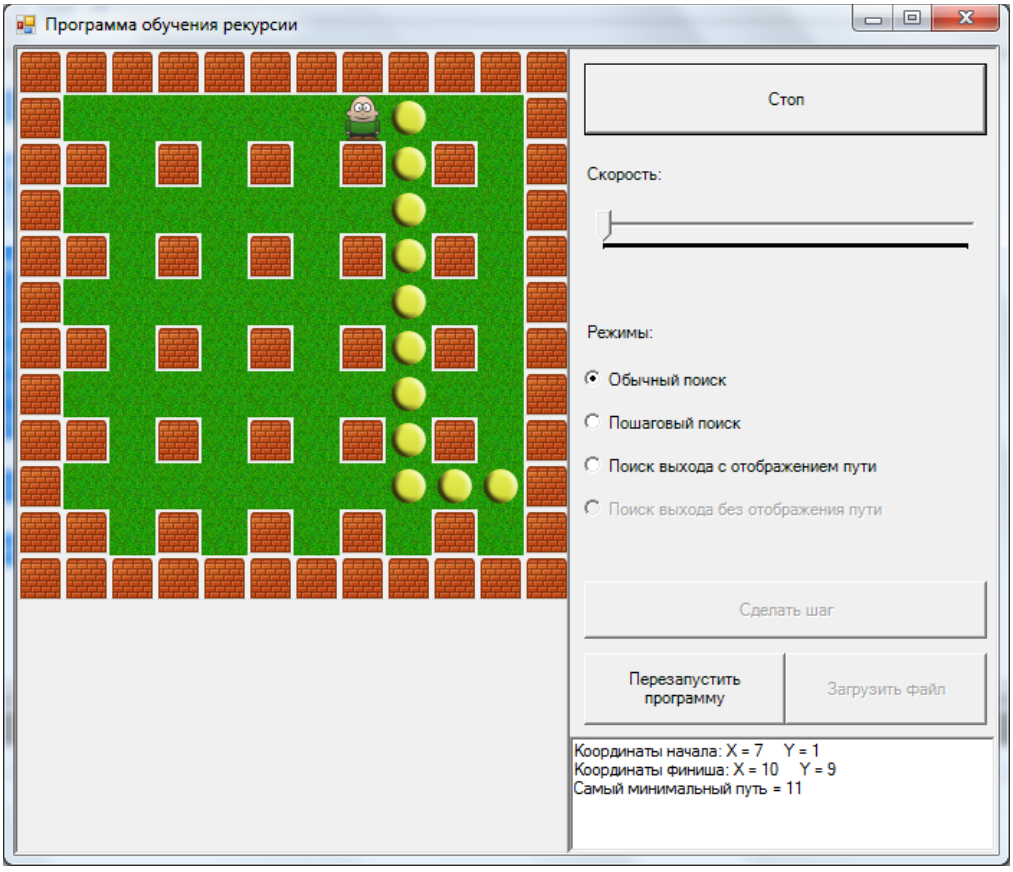

Рисунок 2.16 – Отображение минимального пути

Программа работает исправно, но изначально целью было не создание отдельное программы, а создание модуля, который бы «перехватывал» данные пользователя, рисовал лабиринт и помогал произвести отладку программы пользователя. Переделать программу в модуль совсем несложно. Сперва нужно в самом начале добавить команду, которая указывает, что это модуль.

```
unit RecMod_Stable;
```

Перед подключение библиотек идет служебное слово interface. А после подключения библиотек идет перечень всех процедур, которые используются в модуле. В данном случае список будет следующим:

```
procedure Input;
procedure Path;
procedure FinalPath;
procedure StartThread(sender: Object; e: EventArgs);
procedure StartSearchAnswer;
procedure StopSearch(sender: Object; e: EventArgs);
```

```
procedure CloseWindow(sender: Object; e: EventArgs);
procedure RestartWindow(sender: Object; e:
EventArgs);
procedure NextStepPath(sender: Object; e: EventArgs);
procedure InitControls;
procedure StartApp;
procedure RecEnter(x, y: integer);
procedure RecExit(x, y: integer);
procedure LoadProc(sender: Object; e: EventArgs);
procedure Mode4Seek;
```

После описания переменных идет еще одно служебное слово – implementation, затем все процедуры и, наконец, основная программа. Но в модуле не может быть кода, находящегося за рамками процедур, то есть создание формы, вызов InitControls и запуск формы нужно также поместить в процедуру, например, с именем StartApp.

Теперь модуль готов к работе. Разумеется, его использование не ограничивается данной задачей поиска минимального пути, а он предназначен для всего класса подобных задач.

Ниже представлены результаты тестирования другого лабиринта (см. рисунок 2.17) и нахождение его минимального пути до выхода (см. рисунок 2.18).

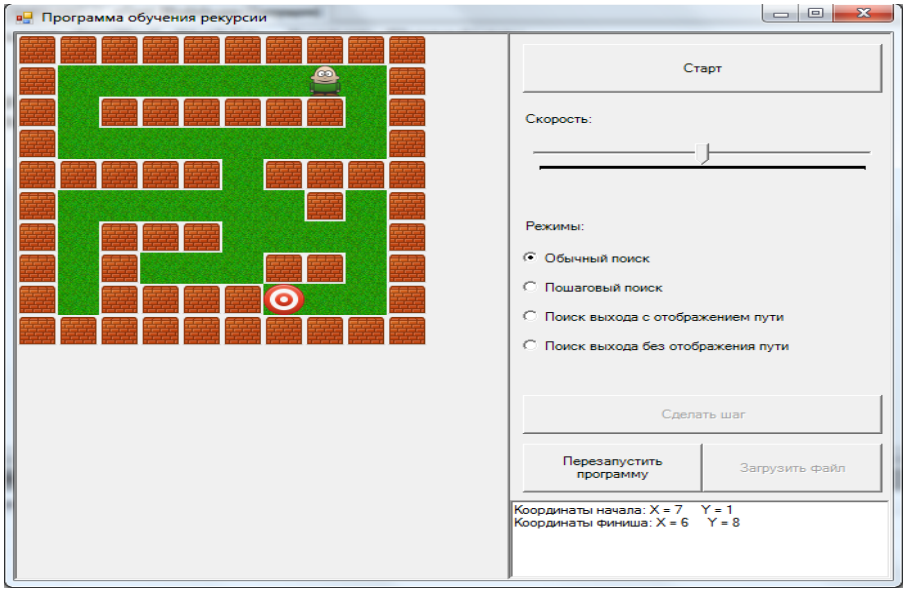

Рисунок 2.17 – Лабиринт №2

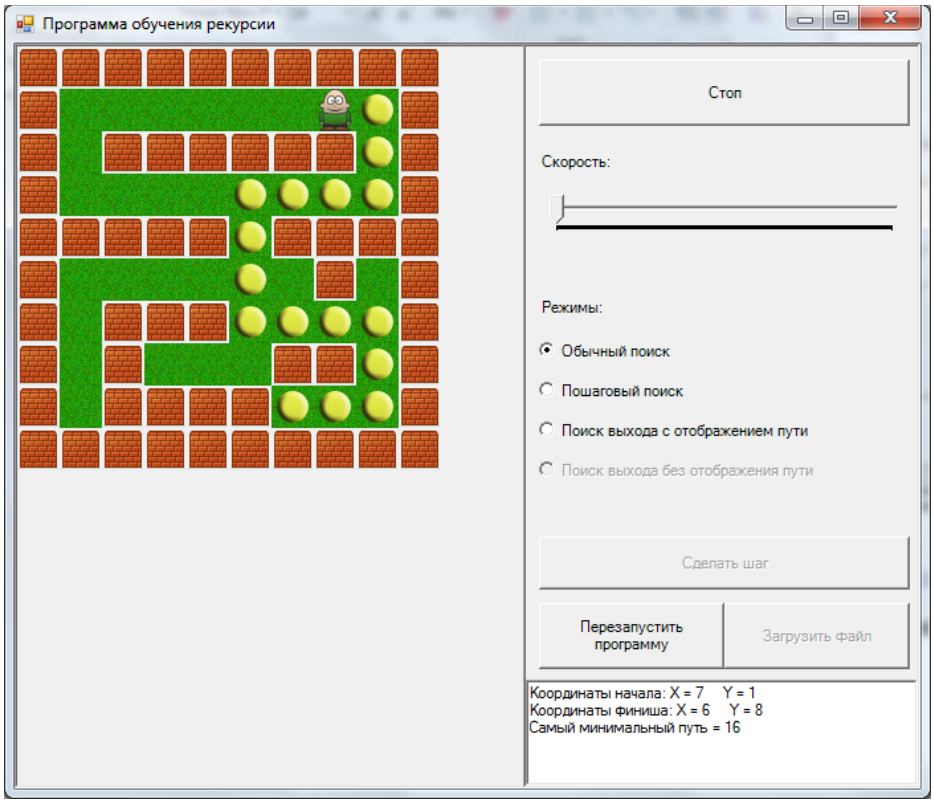

Рисунок 2.18 – Кратчайший путь из лабиринта №2

2.4 Инструкция пользования модулем

Модуль готов. С его помощью можно решить большой класс задач с использованием рекурсивных алгоритмов, но, чтобы модуль работал исправно,

конечный пользователь должен придерживаться следующего шаблона в своей программе:

```
uses RecMod_Stable, System.Threading;
procedure move(x, y:integer);
begin
  //тело программы
  RecEnter(x, y);
  //рекурсивный вызов
  RecExit(x, y);
end;
procedure StartSearch;
begin
  Move(x1, y1);
  StartSearchAnswer;
end;
begin
LabThread := new Thread(StartSearch);
StartApp;
end.
```

Подобная структура необходима из-за сложной организации многопоточности. Если отказаться от использования нескольких потоков, то программа будет работать в разы медленнее, а также одновременно нельзя будет наблюдать за процессом поиска и управлять им. Потоки же это делать позволяют.

ЗАКЛЮЧЕНИЕ

Представленная выпускная квалификационная работа призвана помочь при обучении рекурсивным алгоритмам. Были представлены несколько интерактивных демонстрационных примеров, которые наглядно и пошагово показывают очередное действие. Также был разработан модуль для среды программирования PascalABC.NET, которая является полностью бесплатной и используется в большинстве учебных заведений. Данный модуль позволит пользователям собственноручно попробовать написать программу с использованием рекурсивных алгоритмов, а управляющие элементы модуля помогут произвести отладку, протестировать и проанализировать полученные результаты.

Рекурсивные алгоритмы используются в программировании повсеместно, поэтому потребность в их изучении очень велика. Олимпиадные задания, задания из экзаменационных билетов – везде может встретиться рекурсия. Изучение же рекурсии по наглядным графическим интерактивным примерам позволит не только повысить эффективность усвоения основных ее особенностей, но и способствовать привлечению внимания со стороны пользователей.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Хмыров С.Б. Трудовая подготовка и профориентация сельских школьников. – М., 2005. – 112 с.

2. Игошин В.И. Математическая логика и теория алгоритмов. – М.: Академия, 2008. – 448 с.

3. Эббинхауз Г.-Д., Якобс К., Ман Ф.К., Хермес Г. Машины Тьюринга и вычислимые функции. – М.: Мир, 2002. – 264 с.

4. Ильиных А.П. Теория алгоритмов. Учебное пособие. – Екатеринбург, 2006. – 149 с.

5. Успенский В.А. Лекции о вычислимых функциях. – М.: Физматлит, 2000. – 491 с.

6. Мальцев А.И. Алгоритмы и рекурсивные функции. – М.: Наука, 2006. – 234 с.

7. Петер Р. Рекурсивные функции. – М.: Наука, 2004. – 171 с.

8. Культин Н. Основы программирования в Delphi 7 - СПб.: БХВ-Петербург, 2003. – 393 с.

9. Белов В.В., Чистякова В.И. Программирование в Delphi. Процедурное, объектно-ориентированное, визуальное программирование: учебное пособие / В.В. Белов, В.И. Чистякова – М.: Горячая Линия – Телеком, 2014. – 240 с.

10. Фаронов В.В. Delphi. Программирование на языке высокого уровня. – М.: 2003. – 640 с.

11. Ревич, Ю. Нестандартные приемы программирования на Delphi / Ю. Ревич. - М.: БХВ-Петербург, 2016. - 560 с.

12. Санников, Е. В. Курс практического программирования в Delphi. Объектно-ориентированное программирование / Е.В. Санников. - М.: Солон-Пресс, 2013. - 188 с.

13. Фленов, М.Е. DirectX и Delphi. Искусство программирования (+ CD-ROM) / М.Е. Фленов. - М.: БХВ-Петербург, 2010. - 482 с.

14. Климова, Л. М. Delphi 7. Основы программирования. Решение типовых задач. Самоучитель / Л.М. Климова. - М.: КУДИЦ-Образ, 2017. - 480 с.

15. Водолазов Н.Н., Михалкович С.С., Ткачук А.В. Опыт разработки учебного языка программирования для платформы .NET. — Ростов-на-Дону: Изд-во «ЦВВР», 2007. — 312 с.

16. Абрамян М.Э., Михалкович С.С. Основы программирования на языке Паскаль: Скалярные типы данных, управляющие операторы, процедуры и функции. — Ростов-на-Дону: ООО «ЦВВР», 2004. — 198 с.

17. Златопольский Д.М. Сборник задач по программированию / Д.М. Златопольский– СПб.: БХВ - Петербург, 2011. – 304 с.

18. Йенсен К., Вирт Н. Паскаль — руководство для пользователей и описание языка. — М.: Мир, 2002. – 319 с.

19. Богомолова О.Б. Информатика. Полный справочник для подготовки к ЕГЭ / Богомолова О.Б. – М.: АСТ, 2013. – 416 с.

20. Самылкина Н.Н. ЕГЭ-2016. Информатика. Тематические тренировочные задания / Н.Н. Самылкина, Синицкая И.В., Соболева В.В. – М.: Эксмо, 2015. – 176 с.

21. Абрамов В.Г., Трифонов Н.П., Трифонова Г.Н. Введение в язык Паскаль. — М.: Наука, 1988. – 256 с.

22. Окулов С.М. Программирование в алгоритмах. – М.:Бином. Лаборатория знаний, 2013. – 384 с.

23. Касаткин В. Н. Информация. Алгоритмы. ЭВМ. — М.: Просвещение, 2001. – 271 с.

24. Бондарев В.М., Рублинецкий В.И., Качко Е.Г. Основы программирования. —Харьков: Фолио, Ростов н/Д: Феникс, 2007. – 413 с.

25. Михалкович С.С. Учебная система программирования PascalABC: опыт разработки и использования. — М., 2006. — 412 с.

26. Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989. – 237 с.

27. Гладков В. П. Задачи по информатике на вступительном экзамене в вуз и их решения: Учебное пособие. — Пермь: Перм. техн. ун-т, 1994. – 102 с.

28. Электронный ресурс http://pascalabc.net/.
29. Электронный ресурс http://www.coderun.com/ide/.
30. Электронный ресурс http://ideone.com/.

ПРИЛОЖЕНИЕ А

Программный комплекс

Приложение находится на диске и прилагается к работе.