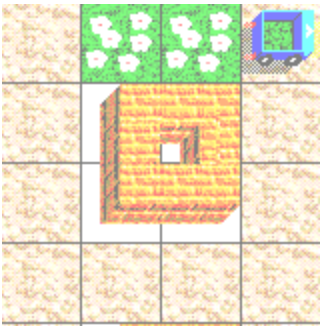
Федеральное агентство по образованию
ГОУВПО «Удмуртский государственный университет»
Кафедра высокопроизводительных вычислений
и параллельного программирования

# ОСНОВЫ АЛГОРИТМИКИ

Методическое пособие



Ижевск 2008

УДК 004.4(075)
ББК 32.973.26 – 018я7
Б 125

Б 125 Бабич О.В. Основы алгоритмики: Метод. пособие/ УдГУ. Ижевск, 2008. 60 с.

Пособие предназначено для студентов гуманитарных и других специальностей, для которых дисциплина «Информатика» не является профильной. Пособие содержит основные сведения курса алгоритмики, являющегося обязательной частью курса информатики.

УДК 004.4(075)
ББК 32.973.26 – 018я7

## Содержание

## Предисловие

*...Преобразование информации на основе формальных правил. Алгоритмизация как необходимое условие его автоматизации.*

*(из Государственного стандарта среднего общего образования)*

*Будем же учиться хорошо мыслить — вот основной принцип морали.*

*Б. Паскаль*

Для успешного изучения вузовского курса информатики необходимо иметь знания не ниже уровня средней школы.

Если учащимся частично утрачены знания по разделу алгоритмика школьного курса информатики, их можно восстановить, прочитав это пособие и выполнив задачи по каждому обязательному разделу курса алгоритмики. Раздел «Рекурсия и итерация» необязателен при первом чтении, его можно пропустить. Прежде чем приступать к изучению этого пособия рекомендуется найти и установить на компьютере свободно распространяемое программное обеспечение «Исполнители» © К. Поляков, 2000-2007, а также Turbo Pascal 5.5 фирмы Borland. Необходимо также найти учебное пособие К. Полякова «Алгоритмы и исполнители» (http://kpolyakov.narod.ru/download/algorithm.zip, дата посещения 17.03.2008) и какой-либо учебник по языку программирования Паскаль, чтобы пользоваться ими как справочниками по мере надобности. Желательно, чтобы под рукой были подготовленные в среде «Исполнители» лабиринты для всех задач и примеров.

# Введение

Требования к уровню знаний в высшем учебном заведении выше, чем в средней школе, что подтверждается, например, следующим фактом. Чтобы получить оценку не ниже тройки за ЕГЭ, правильных ответов должно быть не менее 40%. А чтобы получить оценку не ниже тройки за вузовский интернет-экзамен (ФЭПО), должно быть 50% правильных ответов по каждой теме. Средняя оценка за ЕГЭ по Российской Федерации составляет (на июнь 2008 г.) три балла. При этом для изучения любого из таких вузовских курсов, как математика, информатика, культура речи и др. необходимы знания в объеме школьного курса по этому предмету. Кроме того, ЕГЭ сдается не по всем предметам, а математика, информатика, культура речи и др. обязательно изучаются на любой специальности вуза.

По этим причинам часто оказывается, что знаний, полученных в среднем учебном заведении, недостаточно для усвоения вузовских курсов. В некоторых случаях быстро освоить необходимые навыки помогают специальные средства. К таким средствам относится среда «Исполнители», предназначенная для быстрого освоения курса алгоритмики, являющегося неотъемлемой частью курса информатики. Среда «Исполнители» имеет свой собственный язык программирования, позволяющий писать программы, не используя математических выражений и переменных, но используя наглядные изображения. Например, можно, написав программу управления роботом, наблюдать за его перемещениями по карте в процессе выполнения этой программы. Вместе со свободно распространяемой средой программирования «Исполнители» предоставляется учебник по ее использованию. В нем содержатся задачи, соответствующие школьному уровню образования. Для получения навыков, достаточных для продолжения изучения алгоритмики в вузе, необходимо получить навыки многократного выполнения типовых

программных конструкций. О том, как выполнять задания для получения таких навыков, рассказывается в этом пособии. В процессе получения навыков повышается алгоритмическая грамотность и компетентность в использовании среды программирования. В результате можно приступать к темам «Язык программирования Паскаль» и «Рекурсия».

## 1. Краткое описание среды «Исполнители»

При запуске программы «Исполнители» появляется окно, состоящее из нескольких частей : верхней, левой и еще двух, расположенных справа внизу. В верхней части располагаются меню и панель инструментов, в левой – окно редактора программ, а оставшееся место занимают поле исполнителя и консольное окно. Один из исполнителей – Робот. Он по заданной программе сажает цветы на прямоугольном поле, состоящем из клеток пяти видов : свободная клетка, грядка, клумба, стенка, база. Независимо от того, как расположены клетки разных видов, поле робота будем называть лабиринтом. Язык программирования Робота содержит команды поворота влево и вправо, а также перемещения робота на $n$ клеток вперед или назад : «направо», «налево», «вперед($n$)», «назад($n$)». Кроме этих команд есть такие алгоритмические конструкции как цикл, условный оператор и подпрограмма. Прежде чем приступать к чтению следующего раздела, следует освоить эти команды, выполнив упражнения из учебного пособия К. Полякова «Алгоритмы и исполнители».

## 2. Один из способов решения задачи обхода лабиринта с посадкой цветов

Предположим, что необходимо побывать во всех клетках лабиринта и посадить при этом цветы на всех грядках, встретившихся на пути, а затем вернуться на базу (рис.1).

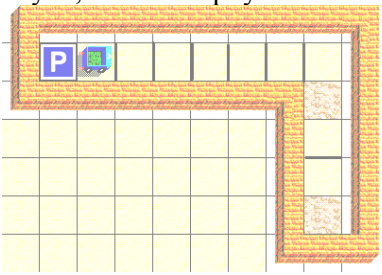


Рис.1. Коридор с поворотом

Дело в том, что по посаженным цветам робот ходить не может, поэтому порядок посадки цветов важен для успешного решения задачи.

Если робот начнет двигаться от базы до тупика и в порядке обнаружения грядок садить на них цветы, то из тупика робот не сможет вернуться назад на базу.

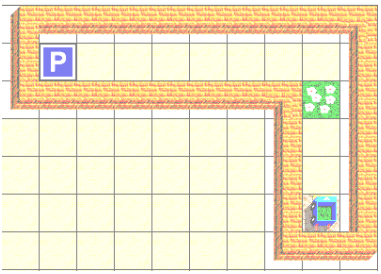


Рис.2. Ловушка

Он сам себе закроет выход из тупика, посадив перед выходом цветы и не имея возможности обойти их или пройти по ним (рис.2).

Поэтому цветы необходимо садить на обратном ходу, то есть при возвращении из тупика, когда уже не будет необходимости проходить через грядки, чтобы дойти до непосещенных клеток в тупике.

Программа прохождения такого лабиринта выглядит следующим образом :

```
ГлавнаяПодпрограмма
 {
        вперед(6);направо;вперед(4);кругом;
        посади;вперед(3);посади;вперед(1);
        налево;вперед(7);
 }
```

Если известно, что лабиринт представляет собой коридор с одним поворотом направо, но неизвестны длины прямых участков коридора (см. рис. 3), то невозможно обойтись без цикла по условию (т.е. без цикла «пока») :

```
ГлавнаяПодпрограмма
 {
  пока(впереди_свободно)
   вперед(1);
  направо;
  пока(впереди_свободно)
   вперед(1);
  кругом;
  посади;
  пока(впереди_свободно)
   вперед(1);
  назад(1);
  посади;
  вперед(1);
  налево;
  пока(впереди_свободно)
   вперед(1);
 }
```

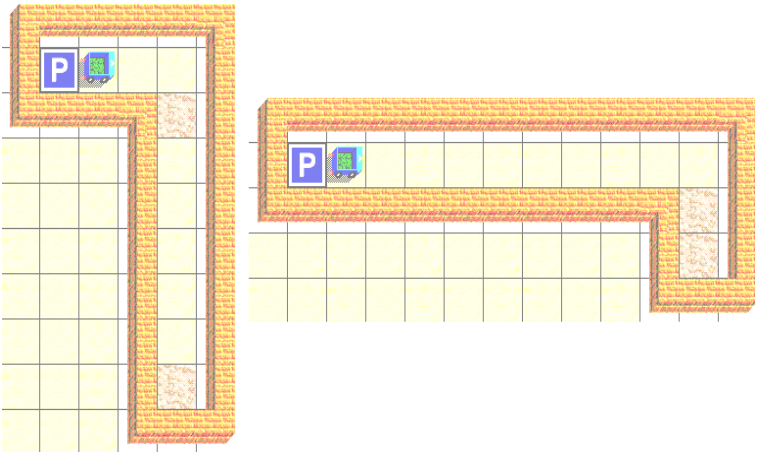


Рис.3. Коридоры с одним поворотом и разными длинами прямых участков

В этой программе многократно повторяется один и тот же вспомогательный набор действий :

```
пока(впереди_свободно)
    вперед(1);
```

В таком случае принято записывать этот набор действий в виде отдельной подпрограммы и вызывать его по имени этой подпрограммы.

```
/* описание главной подпрограммы */
ГлавнаяПодпрограмма
  {
   ДоСтены;/* вызов подпрограммы */
   направо;
   ДоСтены;/* вызов подпрограммы */
   кругом;
   посади;
   ДоСтены;/* вызов подпрограммы */
   назад(1);
   посади;
   вперед(1);
```

```
  налево;
  ДоСтены;/* вызов подпрограммы */
 }

/* описание вспомогательной подпрограммы :     */
ДоСтены /* название подпрограммы */
{                    /* начало действий    */
  пока(впереди_свободно)        /* действия  */
   вперед(1);                  /* действия  */
}                    /* окончание действий */
```

Предположим теперь, что в лабиринте точно так же, как раньше, известно количество и взаимное расположение поворотов (хотя поворотов уже больше одного), но неизвестны длины прямых участков пути (рис.4). А задача осталась той же : побывать во всех клетках и посадить цветы на все грядки, после чего вернуться на базу.

Поскольку в этом лабиринте есть развилки (а в предыдущих лабиринтах на рис. 1-3 их не было), то есть такие клетки, из которых можно двигаться не в двух, а уже в трех направлениях, то ограничиться одним условием (например, «впереди_свободно») теперь не получится. В зависимости от того, с какой стороны мы, двигаясь вперед, зайдем на развилку, свободными окажутся две клетки : или впереди и справа, или впереди и слева, или справа и слева. Кроме того, поскольку раньше (на рис. 1-3) было всего две грядки, причем находились они строго возле поворота и в тупике, то команда «посади» вызывалась только два раза. Теперь же количество грядок неизвестно, так как неизвестна длина пути. Поэтому в зависимости от того, возвращаемся мы или нет, приходится садить цветы или не садить. Так что необходимо написать две подпрограммы для каждого случая : когда надо садить цветы по дороге, а когда этого делать не нужно, чтобы не попасть в ловушку.

Разберем ситуации, которые могут неоднократно возникать при обходе лабиринта.

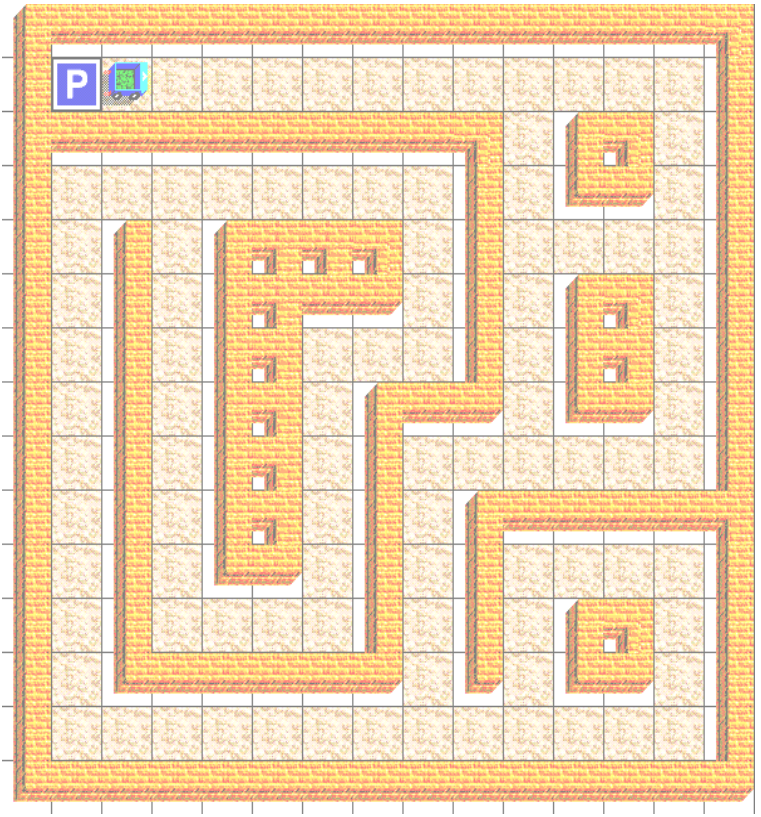


Рис.4. Лабиринт с множеством поворотов

1) Возможно придется пройти прямо до первого поворота направо (рис.5).

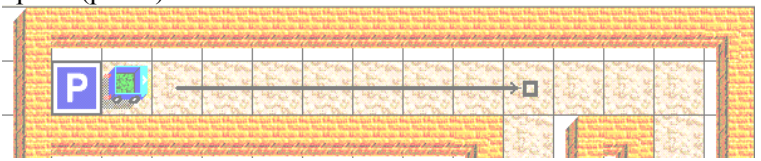


Рис.5. Первый поворот направо

Подпрограмма для перехода может быть записана таким образом :

```
П /* Прогулка до поворота */
{
  пока(впереди_свободно и
    не (справа_свободно или
      слева_свободно))
    вперед(1);
}
```

Чтобы не писать еще одну подпрограмму для поворота налево, здесь проверяется возможность обоих поворотов : хоть налево, хоть направо, а движение осуществляется, если впереди нет препятствия.

Если похожие действия будут выполняться на обратном пути с одновременной посадкой цветов, то необходимо добавить команду «посади».

```
О /* Озеленение */
{
  пока(впереди_свободно и
    не (справа_свободно или
      слева_свободно))
  {
    посади;
    вперед(1);
  }
}
```

Итак, приступим к написанию главной подпрограммы. Сначала выполним переход, как показано на рис 5.

```
ГлавнаяПодпрограмма
  {
   П;
  }
```

Мы окажемся в левом верхнем углу замкнутого пути.

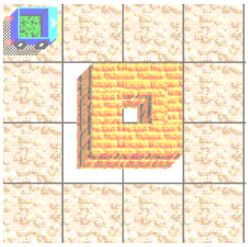


Рис. 6. Цикл (замкнутый путь)

Если теперь сажать цветы по правому «полукругу», то часть задачи будет решена, а у робота останется возможность посетить и другие клетки. Для начала посадим цветы до следующего поворота направо. Для этого изменим главную подпрограмму.

```
ГлавнаяПодпрограмма
 {
  П;вперед(1);О;
 }
```

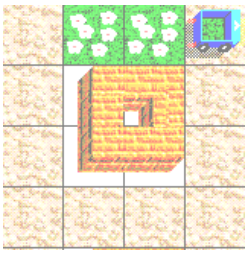


Рис. 7. Начало озеленения дуги цикла

Теперь надо повернуться направо и посадить цветы до ближайшего поворота (рис. 8).

```
ГлавнаяПодпрограмма
 {
  П;вперед(1);О;направо;О;
 }
```

Теперь робот находится на повороте другого цикла, поэтому в таком положении подпрограмма «О» не поможет. Но после

посадки цветов на текущую грядку и выхода на прямой участок мы сможем продолжить озеленение.

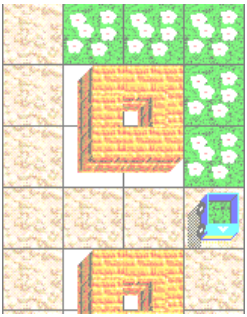


Рис. 8. Продолжение озеленения дуги цикла

```
ГлавнаяПодпрограмма
 {
  П;вперед(1);О;направо;О;
  направо;посади;вперед(1);О;
 }
```

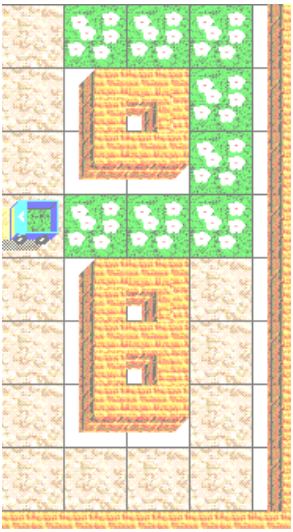


Рис. 9. Превращение нижнего цикла в тупик (разделение замкнутого пути клумбами)

Для продолжения озеленения зайдем в тупик, образовавшийся из нижнего цикла (рис. 9), и посадим там цветы.

```
ГлавнаяПодпрограмма
 {
  П;вперед(1);О;направо;О;
  направо;посади;вперед(1);О;
  налево;вперед(1);П;
  налево;вперед(1);П;
  налево;вперед(1);П;
  кругом;
 }
```

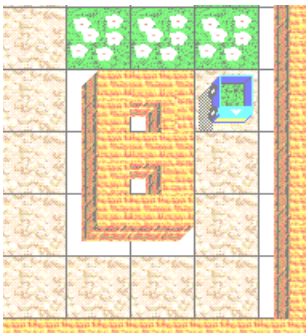


Рис. 10. Готовность к посадке цветов на обратном пути

Прежде чем приступить к посадке цветов, заменим трижды повторяющееся действие *налево;вперед(1);П;* циклом «*повтори*», чтобы программу было легче читать.

```
ГлавнаяПодпрограмма
 {
  П;вперед(1);О;направо;О;
  направо;посади;вперед(1);О;
  повтори (3)
   {налево;вперед(1);П;}
  кругом;
 }
```

Теперь можно сажать цветы.

```
ГлавнаяПодпрограмма
  {
   П;вперед(1);О;направо;О;
   направо;посади;вперед(1);О;
   повтори (3)
    {налево;вперед(1);П;}
   кругом;
   О;направо;О;
  }
```

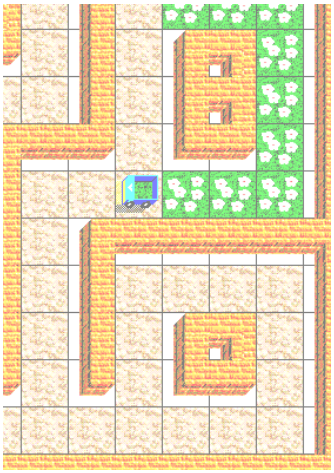


Рис.11. Путь к следующему циклу

Теперь ситуация повторяется. Хотя мы находимся совершенно в другой клетке лабиринта, а идти вниз прямо из этой клетки невозможно, тем не менее, сделав три поворота налево с переходами, мы окажемся в левом нижнем углу следующего цикла.

```
ГлавнаяПодпрограмма
  {
   П;вперед(1);О;направо;О;
   направо;посади;вперед(1);О;
   повтори (3)
```

```
  {налево;вперед(1);П;}
 кругом;
 О;направо;О;вперед(1);
 повтори (3)
  {П;налево;вперед(1);}
}
```

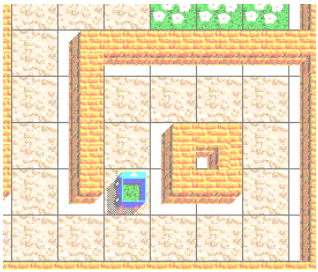


Рис.12. Готовность к озеленению нового цикла

Теперь можно три раза подряд вызвать подпрограмму озеленения, не забывая каждый раз поворачивать направо. А потом оставшийся от цикла тупик тоже превратить в цветник.

```
ГлавнаяПодпрограмма
 {
 П;вперед(1);О;направо;О;
 направо;посади;вперед(1);О;
 повтори (3)
  {налево;вперед(1);П;}
 кругом;
 О;направо;О;вперед(1);
 повтори (3)
  {П;налево;вперед(1);}
 повтори (3)
  {О;направо;}
 О;
}
```

Теперь нужно пройти к последнему циклу в этом лабиринте. Путь к нему указан на рис. 13 стрелкой, а сам цикл помечен

замкнутой пунктирной линией. Здесь нам тоже пригодится цикл «*повтори*».

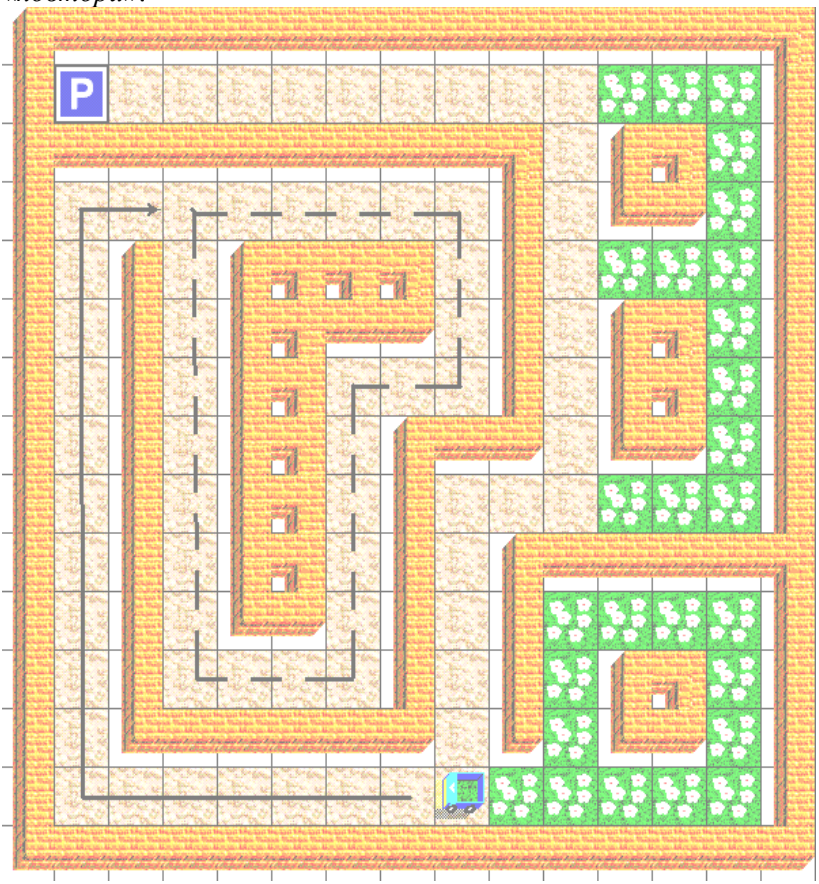


Рис.13. Путь к последнему циклу

```
ГлавнаяПодпрограмма
  {
   П;вперед(1);О;направо;О;
   направо;посади;вперед(1);О;
   повтори (3)
```

```
 {налево;вперед(1);П;}
кругом;
О;направо;О;вперед(1);
повтори (3)
 {П;налево;вперед(1);}
повтори (3)
 {О;направо;}
О;вперед(1);
повтори (3)
 {П;направо;вперед(1);}
}
```

Результат работы этой программы изображен на рис. 14. Если робот посадит сейчас на текущей грядке цветы, то цикл разомкнется и на пути к базе не окажется ни одной развилки. Можно будет написать так же, как мы делали раньше, последовательность действий, каждое из которых содержит прохождение прямого участка и поворот в нужную сторону. Но ведь теперь нет развилок и циклов, а значит, нужно лишь вовремя поворачивать в нужную сторону. Действия однотипные. Поэтому запишем алгоритм возвращения в таком виде : иди, не забывая правильно поворачивать, пока не окажешься на базе.

```
ГлавнаяПодпрограмма
 {
П;вперед(1);О;направо;О;
направо;посади;вперед(1);О;
повтори (3)
 {налево;вперед(1);П;}
кругом;
О;направо;О;вперед(1);
повтори (3)
 {П;налево;вперед(1);}
повтори (3)
 {О;направо;}
```

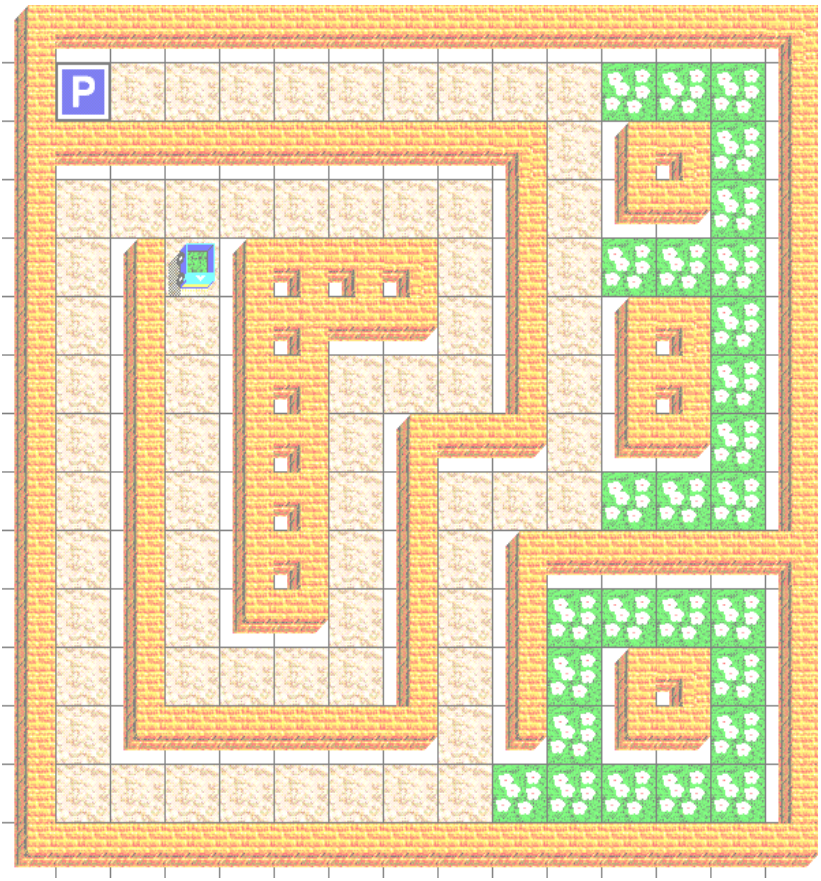


Рис. 14. В начале пути для возврата на базу

```
О;вперед(1);
повтори (3)
  {П;направо;вперед(1);}
пока (не база)
  {
   если (справа_свободно)
```

```
      {направо;посади;вперед(1);}
    иначе
      если (слева_свободно)
        {налево;посади;вперед(1);}
    О;
   }
  }

П /* Прогулка до поворота */
{
  пока(впереди_свободно и
    не (справа_свободно или
      слева_свободно))
    вперед(1);
}

О /* Озеленение */
{
  пока(впереди_свободно и
    не (справа_свободно или
      слева_свободно))
  {
    посади;
    вперед(1);
  }
}
```

**Упражнение А.** Проверьте, что эта программа работает правильно. Если под рукой не оказалось компьютера с программой «Исполнители», то нарисуйте на бумаге этот лабиринт, вырежьте из бумаги фишку размером с клетку лабиринта (назовем ее моделью робота) и передвигайте модель робота по клеткам, выполняя программу шаг за шагом и ставя пометки там, где робот садит цветы.

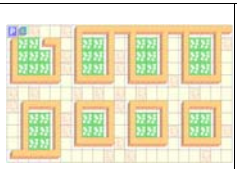
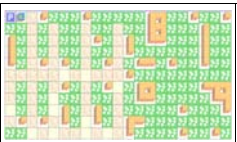
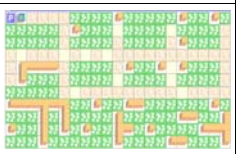
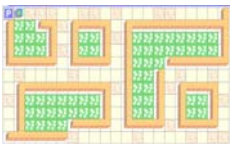
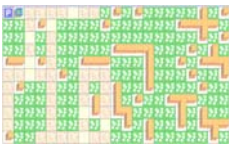
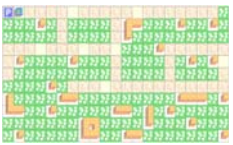
**Задача А.** В любом из указанных ниже вариантов есть три лабиринта, в каждом из которых одинаковые количество и взаимное расположение поворотов в проходах между стенами и посадками. При этом длина прямого участка прохода между двумя поворотами (или между поворотом и тупиком) в любых двух из этих лабиринтов может оказаться различной.

Для вашего варианта, считая заранее неизвестными

- длины прямых участков проходов,
- расположение грядок и свободных клеток в проходах,
- расположение стен и клумб (вокруг проходов),

написать (на бумаге) и проверить в системе «Исполнители» одну программу для робота, исполняя которую, независимо от того, в каком из трех лабиринтов он находится, робот посадит цветы во всех проходах на все грядки и вернется на базу без отказов в работе.

В программе обязательно использовать циклы «пока», операторы «если» и составные условия, но при этом нельзя использовать ни переменные, ни рекурсивные функции.

| № варианта | Лабиринты | | |
|---|---|---|---|
| 1 |  |  |  |
| 2 |  |  |  |

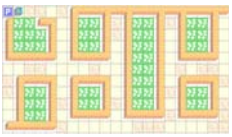
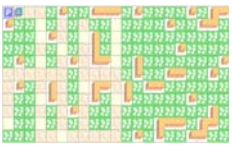
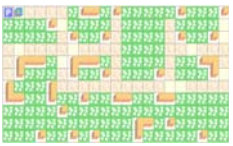
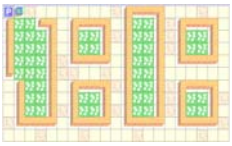
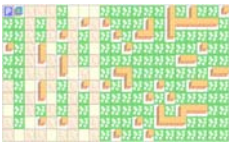
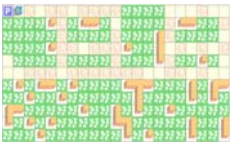
| № варианта | Лабиринты | | |
|---|---|---|---|
| 3 |  |  |  |
| 4 |  |  |  |

Таблица 1. Тестовые примеры для вариантов задачи А

## 3. Рекурсия и итерация

Словом «итерация» обозначается шаг алгоритма. Обычно под итерацией понимают не любой шаг алгоритма, а подобный каким-то другим шагам. Например, если производится подряд несколько отдельных действий, каждое из которых является сложением чисел, то каждое из этих сложений называют итерацией. Первое сложение называют первой итерацией, второе сложение — второй итерацией, третье — третьей. Описание алгоритма посредством описания итераций — не единственный способ описания, кроме итерационного описания используют еще и рекурсивное описание.

Словом «рекурсия» обозначается ситуация, когда подпрограмма вызывает сама себя, хотя и, возможно, с другими исходными данными.

Рекурсия используется для упрощения записи алгоритма, однако в самых простых случаях эффект упрощения отсутствует. Не смотря на то что сначала польза от рекурсии не будет заметна, начнем именно с очень простого примера. Этот

пример поможет понять, как именно выполняются рекурсивные вызовы подпрограммы.

**Задача Б**. Робот находится в начале коридора, ему необходимо посадить цветы в конце коридора и вернуться на базу. Длина коридора заранее неизвестна.

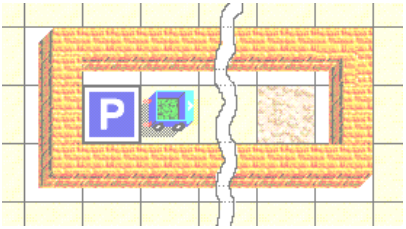


Рис.15. Коридор неизвестной длины

Итерационный способ решения

```
До_стены_и_назад
{
 пока (впереди_свободно)
    вперед(1);
 посади;
 пока (сзади_свободно)
    назад(1);
}
```

Итерация «вперед(1)» выполняется до тех пор, пока робот не дойдет до конца коридора. Затем робот посадит цветы и выполнит нужное число итераций «назад(1)», чтобы вернуться на базу.

Рекурсивный способ решения

```
До_стены_и_назад                    //Главная подпрограмма
{
 осторожный_шаг_вперед;             /*Вызов вспомогательной
подпрограммы*/
 посади;
 кругом;
 осторожный_шаг_вперед;             /*Вызов вспомогательной
```

```
подпрограммы*/
    }

    осторожный_шаг_вперед          /*Описание не главной, а
вспомогательной подпрограммы*/
    {
      если (впереди_свободно)
      {
         вперед(1);
         осторожный_шаг_вперед;
      }
    }
```

Эта программа отличается от предыдущей тем, что появилась новая подпрограмма «осторожный_шаг_вперед» и вместо циклов «пока» появились вызовы этой новой подпрограммы.

Разберем, как работает главная подпрограмма «До_стены_и_назад» в случае, когда для того чтобы попасть на грядку, роботу нужно сделать два шага вперед. В других случаях действия будут аналогичными, но количество действий будет другим. Объяснение работы программы будет дополняться снимками окна среды «Исполнители», в котором будут показаны кадры выполнения программы в режиме трассировки. Режим трассировки — это способ пошагового выполнения программы, когда отображается выполнение каждого действия, включая команды подпрограмм. Для выполнения очередной команды удобно нажимать «горячую» клавишу F7. При выполнении программы по шагам (т. е. по строкам) строка, которая должна быть выполнена при очередном шаге, отображается на специальном фоне. В таком случае говорят, что на строке установлен курсор.

Нажмем один раз (и сразу отпустим, не удерживая) клавишу F7. При этом курсор установится на первую строку «осторожный_шаг_вперед;» главной подпрограммы «До_стены_и_назад» (рис. 16).

```
До_стены_и_назад
{
  осторожный_шаг_вперед;
  посади;
  кругом;
  осторожный_шаг_вперед;
}

осторожный_шаг_вперед
{
   если (впереди_свободно)
   {
       вперед(1);
       осторожный_шаг_вперед;
   }
}
```

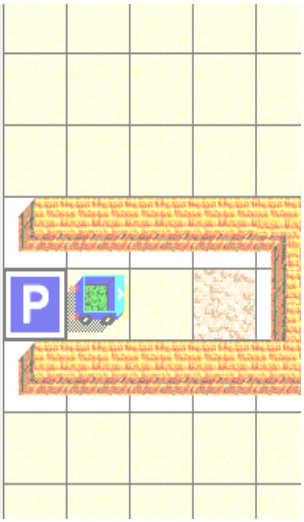


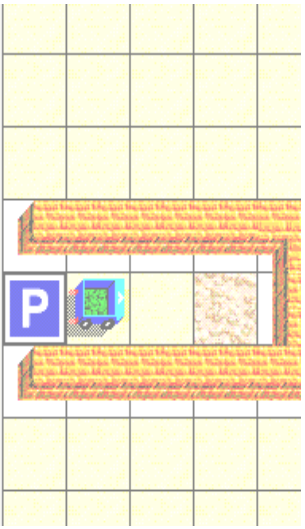

Рис.16. Первый шаг трассировки

Как только команда «осторожный_шаг_вперед;» выполнится, будет выполняться следующая команда «посади;». Однако команда «осторожный_шаг_вперед;» означает, что нужно выполнить все действия подпрограммы «осторожный_шаг_вперед», а это мы хотим разобрать подробно. Команда «осторожный_шаг_вперед;» в подпрограмме «До_стены_и_назад» называется «вызовом подпрограммы» «осторожный_шаг_вперед».

Чтобы приступить к выполнению подпрограммы «осторожный_шаг_вперед», нажмем снова клавишу F7. При этом курсор переместится на первую строку подпрограммы «осторожный_шаг_вперед» (рис. 17).

```
До_стены_и_назад
{
  осторожный_шаг_вперед;
  посади;
  кругом;
  осторожный_шаг_вперед;
}

осторожный_шаг_вперед
{
   если (впереди_свободно)
   {
       вперед(1);
       осторожный_шаг_вперед;
   }
}
```



Но кроме перемещения курсора на эту строку произошло то, что оказалось скрыто от нас, а именно: в оперативной памяти компьютера было отведено место для хранения копии подпрограммы «осторожный_шаг_вперед». Поскольку в дальнейшем подобное будет повторяться при каждом вызове этой подпрограммы, то удобно нумеровать копии подпрограммы.

Рис. 17. Курсор на первой строке подпрограммы «осторожный_шаг_вперед»

Поэтому назовем эту копию подпрограммы «осторожный_шаг_вперед» первой копией. Итак, первая копия помещена в оперативную память компьютера. Теперь мы можем приступить к выполнению первой копии. Нажмем F7, и курсор перейдет на строку с оператором «если» (рис. 18). Поскольку впереди робота свободно, то после проверки условия «впереди_свободно» курсор установится на строку «вперед(1)». Нажмем F7. После этого робот сместится вперед, а мы подберемся к особенной команде «осторожный_шаг_вперед;» (рис.19). Особенность команды, на которой находится курсор,

заключается в том, что, с одной стороны, это единственная команда, отделяющая нас от завершения выполнения первой копии подпрограммы, а с другой стороны, эта команда представляет собой вызов именно той подпрограммы, которая и содержит эту команду. Такой вызов называется рекурсивным вызовом в отличие от предыдущего вызова, когда подпрограмма «осторожный_шаг_вперед» вызывалась не из себя, а из главной

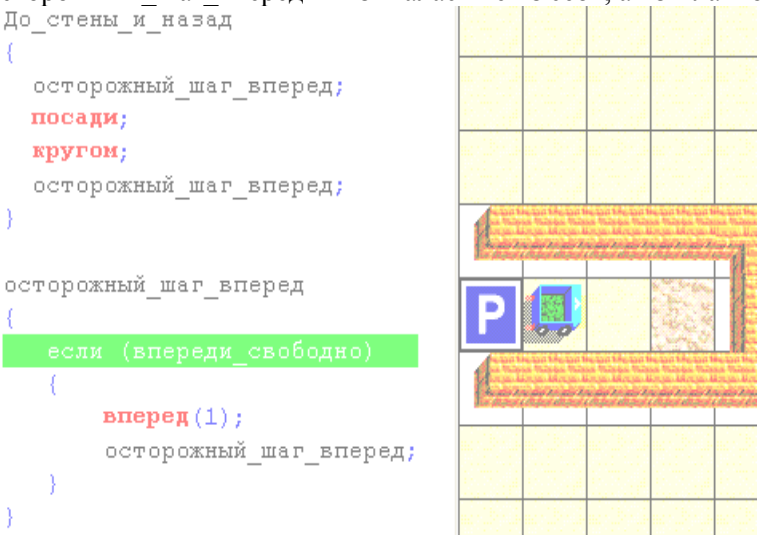



Рис. 18. Курсор на строке с оператором «если»

```
До_стены_и_назад
{
  осторожный_шаг_вперед;
  посади;
  кругом;
  осторожный_шаг_вперед;
}

осторожный_шаг_вперед
{
   если (впереди_свободно)
   {
       вперед(1);
       осторожный_шаг_вперед;
   }
}
```


Рис. 19. Курсор на строке с рекурсивным вызовом

подпрограммы «До_стены_и_назад». Если бы нам сейчас удалось сразу выполнить команду, на которой установлен курсор, то подпрограмма «осторожный_шаг_вперед» завершилась бы и ее копия (которую мы назвали первой копией) была бы автоматически удалена из оперативной памяти, а курсор установился на строку «посади;», так как предыдущая строка подпрограммы «До_стены_и_назад» оказалась бы выполненной. Однако так быстро строку, на которой пока еще установлен курсор, выполнить не удастся. Приступим к ее выполнению, нажав F7.

```
До_стены_и_назад
{
  осторожный_шаг_вперед;
  посади;
  кругом;
  осторожный_шаг_вперед;
}

осторожный_шаг_вперед
{
   если (впереди_свободно)
   {
       вперед(1);
       осторожный_шаг_вперед;
   }
}
```


Рис. 20. Начало второго прохода рекурсивной подпрограммы

Курсор снова оказался на первой строке текста подпрограммы «осторожный_шаг_вперед» (рис. 20), и снова от нас было скрыто автоматическое создание новой копии подпрограммы «осторожный_шаг_вперед». Назовем эту копию второй копией. Первая копия этой подпрограммы выполнялась при несколько иных условиях (робот находился в другой клетке), то есть исходные данные для выполнения второй копии уже другие.

Выполним и вторую копию: нажмем один раз F7.

```
До_стены_и_назад
{
  осторожный_шаг_вперед;
  посади;
  кругом;
  осторожный_шаг_вперед;
}

осторожный_шаг_вперед
{
   если (впереди_свободно)
   {
       вперед(1);
       осторожный_шаг_вперед;
   }
}
```


Рис. 21. Курсор на строке с «если»
при втором проходе

Курсор снова оказался на строке оператора «если» (рис. 21). Впереди робота находится грядка, не представляющая собой препятствие, поэтому, нажав F7, мы попадем внутрь фигурных скобок после строки условного оператора, а именно на команду «вперед(1);» (рис. 22). Выполнив эту команду при помощи F7, робот окажется на грядке перед стеной, а курсор установится на следующую команду рекурсивного вызова «осторожный_шаг_вперед». Снова повторяется ситуация, когда мы могли бы завершить выполнение очередной копии подпрограммы, но сразу сделать этого не можем, так как хотим разобрать, как же выполняется очередная копия, вызываемая этой командой. Нажатие F7 приводит к созданию теперь уже третьей копии подпрограммы «осторожный_шаг_вперед», после чего курсор снова устанавливается на первую строку текста подпрограммы.

```
До_стены_и_назад
{
  осторожный_шаг_вперед;
  посади;
  кругом;
  осторожный_шаг_вперед;
}

осторожный_шаг_вперед
{
   если (впереди_свободно)
   {
       вперед(1);
       осторожный_шаг_вперед;
   }
}
```


Рис. 22. Курсор на строке с «вперед» при втором проходе

```
До_стены_и_назад
{
  осторожный_шаг_вперед;
  посади;
  кругом;
  осторожный_шаг_вперед;
}

осторожный_шаг_вперед
{
   если (впереди_свободно)
   {
       вперед(1);
       осторожный_шаг_вперед;
   }
}
```

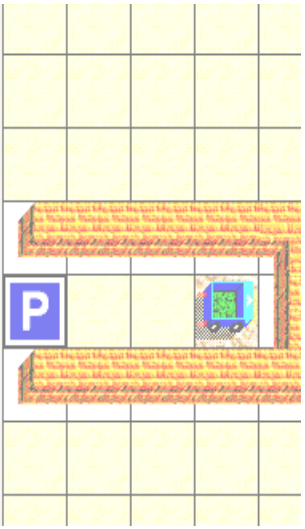



Рис. 23. Курсор на строке с «если» при третьем проходе

Приступим к выполнению третьей копии: F7. Третья копия выполняется в новых условиях : впереди робота стена (рис.23), поэтому тело оператора «если» не выполняется. Если бы после оператора «если» в подпрограмме располагались другие операторы, то они начали бы выполняться. Однако после оператора «если» нет операторов, поэтому нажатие F7 приводит к установке курсора на последнюю строку текста подпрограммы «осторожный_шаг_вперед» (рис. 24).

```
До_стены_и_назад
{
  осторожный_шаг_вперед;
  посади;
  кругом;
  осторожный_шаг_вперед;
}

осторожный_шаг_вперед
{
   если (впереди_свободно)
   {
       вперед(1);
       осторожный_шаг_вперед;
   }
}
```

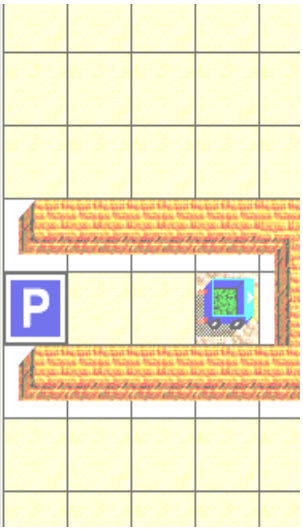



Рис. 24. Завершение рекурсивной подпрограммы

При этом произошло три события, которые опять были от нас скрыты. Сначала завершилась третья копия. Поскольку эта копия полностью отработала, то стала не нужна и потому была удалена из памяти. Затем компьютер приступил к продолжению выполнения второй копии. Обнаружилось, что последняя команда (а это был рекурсивный вызов «осторожный_шаг_вперед;») второй копии подпрограммы уже выполнена, а значит, вторая копия завершилась. Компьютер тут же автоматически удалил вторую копию из своей оперативной памяти. Тогда оказалось, что и первая копия выполнилась целиком, а значит,

после нажатия F7 компьютер продолжит выполнение главной подпрограммы, поставив курсор на строку «посади;» (рис. 25) и удалив первую копию подпрограммы.

```
До_стены_и_назад
{
  осторожный_шаг_вперед;
  посади;
  кругом;
  осторожный_шаг_вперед;
}

осторожный_шаг_вперед
{
   если (впереди_свободно)
   {
       вперед(1);
       осторожный_шаг_вперед;
   }
}
```

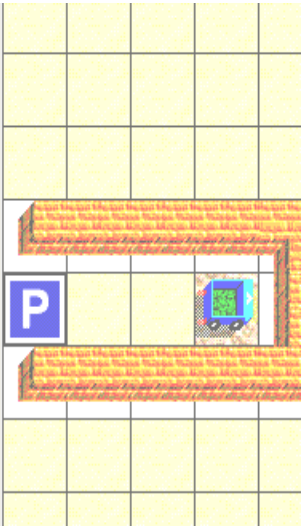



Рис. 25. Продолжение работы после выхода из рекурсивной подпрограммы

Осталось посадить цветы, развернуться и выполнить путь до конца коридора, оказавшись в результате на базе.

Итак, при вызовах рекурсивной подпрограммы ее копии располагались в оперативной памяти строго в порядке их вызова. Затем при завершении копий они удалялись из памяти строго в порядке завершения, то есть в обратном порядке: добавлялись в память копия 1, копия 2, копия 3, а удалялись сначала копия 3, потом копия 2, а в конце копия 1. Такое устройство механизма создания-удаления копий называется стеком. Принято говорить, что исполняемый код подпрограммы добавляется в программный стек (удаляется из программного стека).

**Упражнение Б**. Выполните по шагам и проанализируйте выполнение команд в таком варианте рекурсивного решения

задачи:

```
До_стены_и_назад
{
  прогулка;
}

прогулка
{
  если (впереди_свободно)
  {
     вперед(1);
     прогулка;
  }
  если (грядка)
     посади;
  назад(1);
}
```

Конец упражнения.

Рекурсивные алгоритмы принято описывать следующим образом:

*Базис рекурсии* (т. е. то условие, при котором завершаются рекурсивные вызовы).

*Шаг рекурсии* (это те действия, которые осуществляются до момента завершения рекурсии).

Например, рекурсивный алгоритм решения задачи Б, используемый в первой программе в виде подпрограммы «осторожный_шаг_вперед», можно описать так:

*Базис рекурсии*. Впереди себя роботом обнаружена стена.

*Шаг рекурсии*. Если впереди робота свободно, сделать шаг вперед.

**Задача В**. Робот находится в начале коридора. Коридор имеет следующую форму : прямой участок до стены, разворот вправо и снова прямой участок до тупика. В тупике находится база (рис. 26). Роботу необходимо прийти на базу. Длины прямых

участков коридора заранее неизвестны.

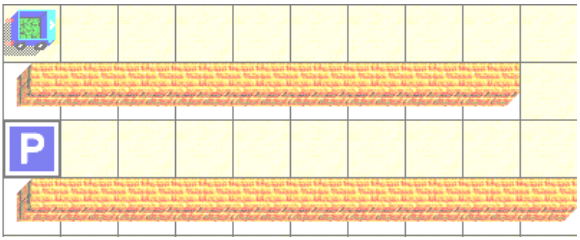


Рис. 26. Коридор с поворотом назад

Итерационный алгоритм

```
Домой
{
 пока (впереди_свободно)
    вперед(1);
 направо;
 вперед(2);
 направо;
 пока (впереди_свободно)
    вперед(1);
}
```

Рекурсивный алгоритм

Как мы делали раньше, можно для прохождения прямого участка пути использовать «осторожный_шаг_вперед» :

```
Домой
{
 осторожный_шаг_вперед;
 направо;
 вперед(2);
 направо;
 осторожный_шаг_вперед;
}
```

```
осторожный_шаг_вперед
{
  если (впереди_свободно)
  {
    вперед(1);
    осторожный_шаг_вперед;
  }
}
```

**Задача Г**. Робот находится возле базы в начале диагонального зигзагообразного коридора (рис. 27). Длина коридора заранее неизвестна. Необходимо посадить цветы на грядку в конце коридора и вернуться на базу. Написать две программы для робота : одну итерационную, другую рекурсивную.

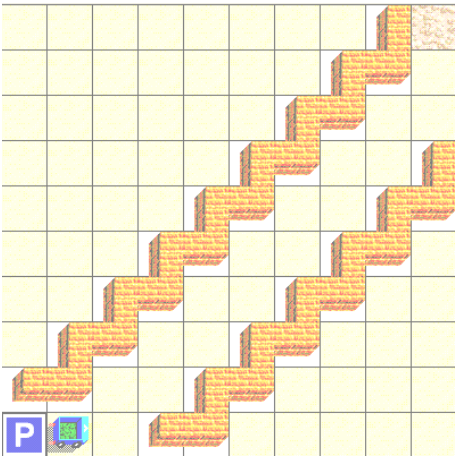


Рис. 27. Зигзагообразный коридор

**Задача Д**. Робот должен обойти по периметру сад и в конце прямых отрезков пути посадить цветы, а затем вернуться на базу (рис. 28). Длины прямых участков пути заранее неизвестны. Написать одну итерационную программу и одну

рекурсивную программу.

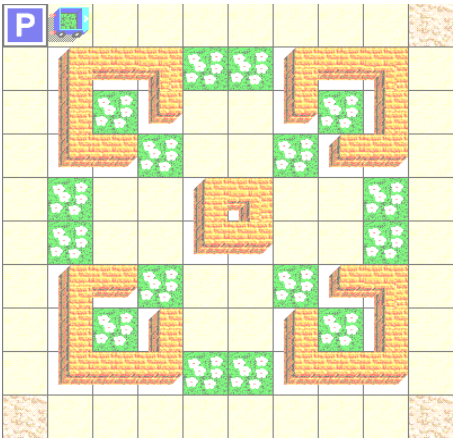


Рис. 28. Сад

**Задача Е**. Робот находится в лабиринте, имеющем форму улитки (рис. 29). Число поворотов и длины прямых участков проходов заранее неизвестны. В тупике в глубине лабиринта находится грядка. Робот должен посадить на эту грядку цветы и вернуться на базу. Написать две программы : рекурсивную и итерационную.

Для решения этой задачи можно использовать обход лабиринта, придерживаясь стены. Это означает, что робот должен стремиться к тому, чтобы с одной стороны, например с правой, всегда была стена. Например, чтобы пройти до ближайшего поворота направо и встать в направлении дальнейшего движения можно использовать такой алгоритм :

```
пока (справа_стена)
    вперед(1);
направо;
вперед(1);
```

Если принять такой алгоритм за итерацию, то условием окончания итераций, а именно достижения положения для

посадки цветов, будет попадание на грядку.

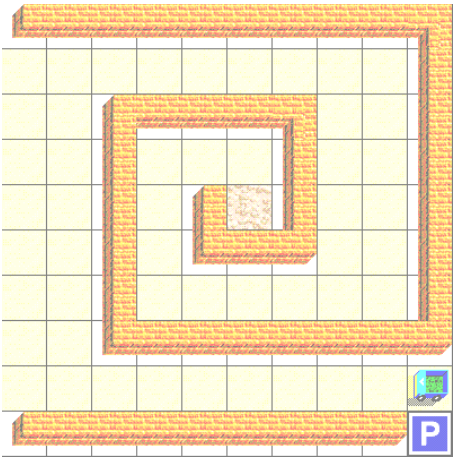


Рис. 29. Лабиринт в форме улитки

```
До_грядки
{
 пока (не грядка)
    до_поворота;
}

до_поворота
{
 пока (справа_стена)
    вперед(1);
 направо;
 вперед(1);
}
```

Теперь остается посадить цветы на грядку и вернуться, придерживаясь такого положения, чтобы стена была слева.

```
До_грядки_и_обратно
{
```

```
 пока (не грядка)
    до_правого_поворота;
 посади;
 кругом;
 пока (впереди_свободно)
    до_левого_поворота;
 назад(1);
}

до_правого_поворота
{
 пока (справа_стена)
    вперед(1);
 направо;
 вперед(1);
}

до_левого_поворота
{
 пока (слева_стена и впереди_свободно)
    вперед(1);
 налево;
 если (впереди_свободно)
    вперед(1);
}
```

Итерационная программа написана. Теперь займемся рекурсивной программой. Можно было бы как раньше заменить циклы «пока» на рекурсивные программы осторожного шага по прямой. Но мы попробуем способ, который использовался в упражнении А. Другими словами, чтобы не придерживаться сначала левой стеночки, а потом правой стеночки, мы воспользуемся тем, что информация о пройденном пути хранится в программном стеке вместе с копиями рекурсивной подпрограммы, а значит, по этой «нити Ариадны» всегда можно

автоматически вернуться назад.

```
До_грядки_и_обратно
{
 посадить_и_вернуться;
 направо;
 назад(1);
}

посадить_и_вернуться
{
 если (справа_свободно)
 {
    направо;
    вперед(1);
    посадить_и_вернуться;
    назад(1);
    налево;
 }
 иначе если (впереди_свободно)
 {
    вперед(1);
    посадить_и_вернуться;
    назад(1);
 }
 иначе если (грядка)
 {
    посади;
 }
}
```

Рекурсивная программа написана. В целом сложность программ примерно одинакова, но из рекурсивной программы ясно, как ведет себя робот в конкретной ситуации: делает шаг вперед, а

потом назад, что гарантирует его возврат в точку старта.

**Задача Ж**. Робот находится в лабиринте (рис. 30), в котором есть единственная развилка : войдя в нее, можно продолжить путь в любом из двух направлений (либо вернуться назад). То есть развилка представляет собой перекресток, из центра которого можно двигаться только в трех направлениях. Кроме того, в лабиринте нет циклов, то есть невозможно ходить кругами или, другими словами, невозможно прийти в одну и ту же клетку двумя разными путями, не побывав в ней же на пути к ней. Робот должен побывать во всех свободных клетках лабиринта и вернуться на базу. Длины прямых участков, расположение поворотов и положение развилки заранее неизвестны. Написать два варианта программы : итерационный и рекурсивный.

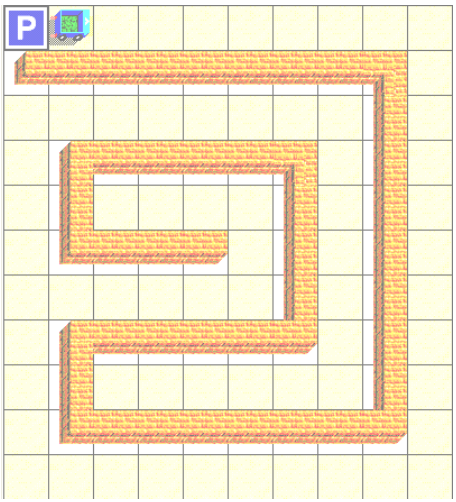


Рис. 30. Лабиринт с развилкой

Развилка мешает обходить лабиринт по единственному простому правилу: «поворачивай направо, как только представится возможность», так как на развилке нужно выбрать

оба направления поочередно.

Итерационный алгоритм

```
обойти_по_левой_стеночке
{
 пока(впереди_свободно и не база)
 {
    пока (впереди_свободно и не слева_свободно)
       вперед(1);
    если (слева_свободно)
    {
       налево;
       если (впереди_свободно)
          вперед(1);
    }
    иначе если (справа_свободно)
    {
       направо;
       если (впереди_свободно)
          вперед(1);
    }
    иначе кругом;
 } // завершение цикла «пока(впереди_свободно и не база)»
}
```

Рекурсивный алгоритм

```
обход
{
   шаг;
   назад(1); //на базу
}

шаг
{
```

```
   если (слева_свободно  или  справа_свободно
                         или  впереди_свободно)
      перебрать_направления;
}

перебрать_направления
{
   если (слева_свободно)
   {
      налево;
      шаг_с_возвратом;
      направо;
   }
   если (впереди_свободно)
      шаг_с_возвратом;
   если (справа_свободно)
   {
      направо;
      шаг_с_возвратом;
      налево;
   }
}

шаг_с_возвратом
{
   вперед(1);
   шаг;
   назад(1);
}
```

Заметьте, что в этой программе нет ни одной подпрограммы, которая в своем тексте содержала бы вызов себя. Подпрограмма «шаг» вызывает себя через другие подпрограммы: в ее тексте содержится вызов «перебрать_направления», а в тексте подпрограммы «перебрать_направления» содержится вызов

«шаг_с_возвратом», а уже из подпрограммы «шаг_с_возвратом» снова вызывается подпрограмма «шаг». Такая ситуация называется косвенной рекурсией, в отличие от ситуации прямой рекурсии, когда подпрограмма содержит свой собственный вызов в своем же тексте.

**Задача 3**. Робот находится в лабиринте, имеющем только две развилки (рис. 31), а также несколько поворотов в коридорах. В лабиринте нет замкнутых путей (т. е. «циклов»). Длины коридоров, взаимное расположение поворотов и развилок и количество поворотов заранее неизвестны. В каждом тупике располагается по одной грядке, других грядок в лабиринте нет. Количество тупиков и грядок заранее тоже неизвестно. Написать две программы посадки цветов на все грядки : одну итерационную и одну рекурсивную.

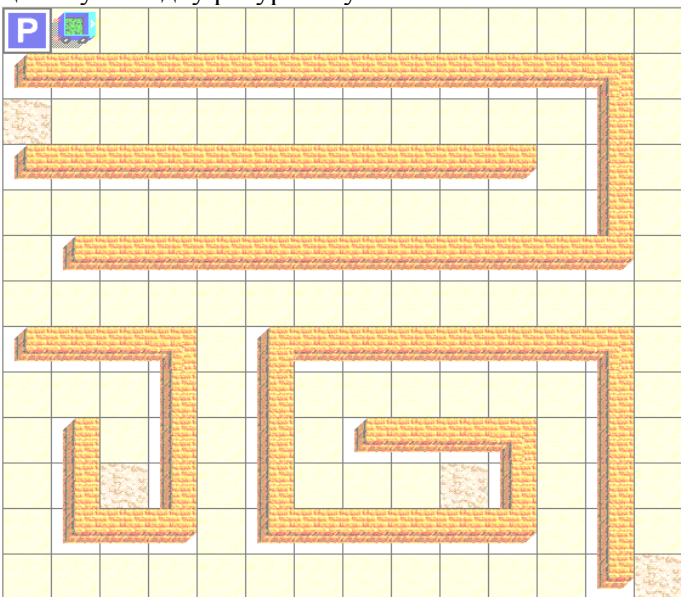


Рис. 31. Лабиринт с двумя развилками

## 3. Подпрограммы с параметрами

До сих пор мы разбирали только такие программы, которые не имели явно обозначенных данных кроме карты лабиринта. Теперь используем переменные в подпрограммах.

**Задача И**. С клавиатуры вводятся *n* чисел. Вычислить их сумму. Для вычисления суммы написать рекурсивную и итерационную программы.

Итерационная программа

```
Сумма
{
 int i,n;
 float sum, s;
 вывод «Сколько чисел суммировать ?»;
 ввод n;
 sum=0;
 for (i=1;i<=n;i=i+1)
  {
     вывод «Введите число №», i;
     ввод s;
     sum=sum+s;
  }
 вывод «Сумма этих чисел равна », sum;
}
```

Рекурсивная программа

```
/*01*/  Сумма
/*02*/  {
/*03*/   int i,n;
/*04*/   float sum, s;
/*05*/   вывод «Сколько чисел суммировать ?»;
/*06*/   ввод n;
/*07*/   sum=summa(n);
/*08*/   вывод «Сумма этих чисел равна », sum;
/*09*/  }
/*10*/
```

```
/*11*/  float summa(int n)
/*12*/  {
/*13*/     float s1,s2;
/*14*/     if (n>0)
/*15*/     {
/*16*/        s2=summa(n-1);
/*17*/        вывод «Введите число №», n;
/*18*/        ввод s1;
/*19*/        return s1+s2;
/*20*/     }
/*21*/     else
/*22*/        return 0;
/*23*/  }
```

Разберем на примере, как выполняется эта программа. Ответим на вопрос о количестве суммируемых чисел : 3. Тогда переменная n в строке 06 примет значение 3. В строке 07 будет вызвана подпрограмма summa(*n*) при значении параметра *n*=3. В оперативную память помещена первая копия подпрограммы summa. Так как в строке 14 условие 3>0 выполнено, то в строке 16 рекурсивно вызывается подпрограмма summa(*n*-1). Таким образом, в качестве параметра *n* новой (уже второй по счету) копии подпрограммы summa передано не число 3, а число 2. Теперь в строке 14 условие 2>0 оказывается выполнено, так что в строке 16 вызывается подпрограмма summa с значением параметра, равным 1. В очередной (теперь уже третьей) копии подпрограммы summa в строке 14 выражение 1>0 оказывается верным и опять в строке 16 вызывается подпрограмма summa, то есть создается четвертая копия. Наступил момент, когда параметр этой подпрограммы стал равным 0, а значит, мы оказываемся в условиях базы рекурсии: «не *n*>0». Поскольку выражение 0>0 неверно, то выполняется действие в строке 22, которое приводит к завершению выполнения четвертой копии подпрограммы summa, удалению четвертой копии из памяти и возвращению в качестве ее значения числа 0. При этом в строке

16 предыдущей (т. е. третьей) копии переменная s2 принимает это самое значение 0. Выполнение третьей копии подпрограммы продолжается: в строке 17 на экран выводится текст «Введите число №1», так как значение параметра *n* третьей копии равно 1. После ввода (в строке 18) с клавиатуры значения переменной s1 (пусть, например, вводится число 8) управление передается команде в строке 19, которая завершает работу третьей копии подпрограммы с одновременным удалением ее из памяти и возвратом в точку вызова (находящуюся в строке 16) предыдущей (т. е. второй) копии того числа, которое получено сложением значений переменных s1 и s2 (это значение есть результат операции 0+8, то есть число 8). Итак, значение переменной s1 второй копии подпрограммы вычислено в строке 16, теперь выполнение этой подпрограммы продолжается и выполняется строка 17. Выводится на экран текст «Введите число №2», в строке 18 вводится значение переменной s1 (пусть это будет число 5). Выполнение строки 19 приводит к завершению копии 2 с возвратом в точку вызова (в строке 16 первой копии) значения выражения s1+s2, т. е. 8+5=13. Аналогично, после ввода с клавиатуры третьего числа (пусть это будет 7) в точку вызова (из строки 07 главной подпрограммы Сумма) первой копии подпрограммы summa возвращается значение 13+7, то есть 20. Теперь переменная sum главной подпрограммы получает значение 20, которое в строке 08 выводится на экран.

Программа полностью выполнилась, сумма вычислена и сообщена пользователю.

Заметьте, что рекурсивные вызовы всегда приводят к созданию новых копий подпрограмм и потому никогда нет возможности попасть в одну и ту же копию подпрограммы дважды. Структура, в которой из одного элемента можно попасть в любой другой элемент единственным способом, называется деревом. По этой причине последовательность рекурсивных вызовов называют деревом рекурсии. Так, скажем,

в рассмотренном примере дерево рекурсии выглядит следующим образом :

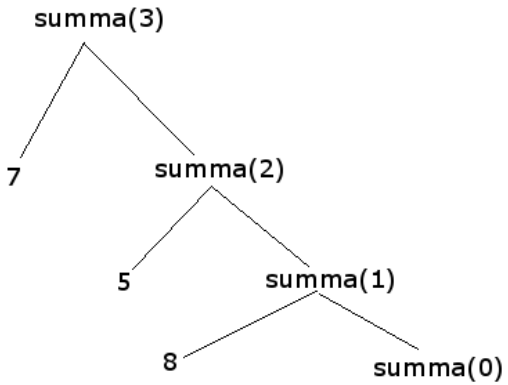



Рис. 32. Дерево рекурсии для функции summa

В этом дереве три рекурсивных вызова, в таком случае говорят, что глубина рекурсии равна трем.

**Задача К**. Числа Фибоначчи определяются рекуррентным соотношением : f(*n*)=f(*n*-1)+f(*n*-2), если *n*>1, и f(1)=1, f(0)=1. Например, второе число Фибоначчи f(2) = f(1)+f(0), то есть f(2)=1+1=2. Третье число f(3)=f(2)+f(1)=3, четвертое f(4)=5, далее идут 8, 13, 21, 34 и т. д. Написать две программы вычисления *n*-го числа Фибоначчи : одну итерационную, а другую рекурсивную. Построить дерево рекурсии для вычисления пятого числа Фибоначчи.

## 4. Ключевые примеры программ на языке Паскаль

Краткое описание программ на языке Паскаль, включающее типичный набор основных конструкций, составляет не менее 50 страниц текста. При этом чтение этого текста без проработки материала каждого раздела оказывается бесполезно, а ведь чтение с такой проработкой занимает довольно много часов. В то же время для чтения типовых конструкций фрагментов программ на языке Паскаль оказывается вполне достаточным умение понимать, как выполняются некоторые отдельные конструкции программ. Самый простой и при этом весьма эффективный способ получить навык чтения и понимания таких типовых фрагментов программ ─ это изучение их на примерах. Поэтому читаемый вами текст значительно короче обычного пособия по языку Паскаль.

**Пример 1**. Блок-схема линейного алгоритма.

В блок-схемах в овале (в эллипсе) изображаются начало и конец алгоритма, а в прямоугольнике ─ однозначно понимаемое действие. Например, пусть производятся такие действия:

1) в переменную **x** записывается число 3, это обозначается так : **x:=3**;
2) в переменную **y** записывается число 5, это обозначается так : **y:=5**;
3) в переменную **x** записывается сумма тех числовых значений, которые находились в переменной **x** и в переменной **y**, это обозначается **x:=x+y**;

Последовательность этих действий записывается в виде блок-схемы, изображенной на рис. 33.

При этом значение переменной **x** становится равным 8.

На Паскале эти действия записываются так:

```
x:=3;
y:=5;
x:=x+y;
```

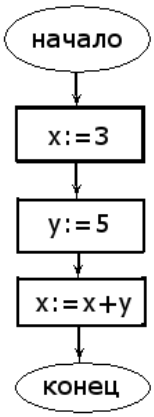
Рис. 33. Блок-схема линейного алгоритма

Символ **:=** называется оператором присваивания. Действия, выполняемые оператором присваивания, тоже называются оператором присваивания. Оператор присваивания выполняет два действия:

1) вычисляет значение выражения, стоящего справа от символа **:=** ,
   а после этого
2) записывает вычисленное значение в переменную, стоящую слева от символа **:=** .

Обычно перемена порядка действий в линейном алгоритме приводит к получению совсем другого алгоритма. Например, если переставить в рассмотренном выше алгоритме **x:=x+y** и **y:=5**, то получится алгоритм :

**x:=3;**




**x:=x+y;**

**y:=5;**

Выполняя действия этого алгоритма, получаем: **x=3**, затем **x=3+y**, а **y** к этому моменту еще не задано, то есть может иметь любое неизвестное нам значение (иначе говоря, нельзя указать, какое именно число записано в переменную **x**), а потом задается значение **y**=5. В результате в рассмотренном ранее алгоритме **x**

принимает значение 8, а в только что рассмотренном ─ какое-то неизвестное значение. Результаты вычислений разные, значит, и алгоритмы разные.

**Пример 2**. Разветвляющийся алгоритм.

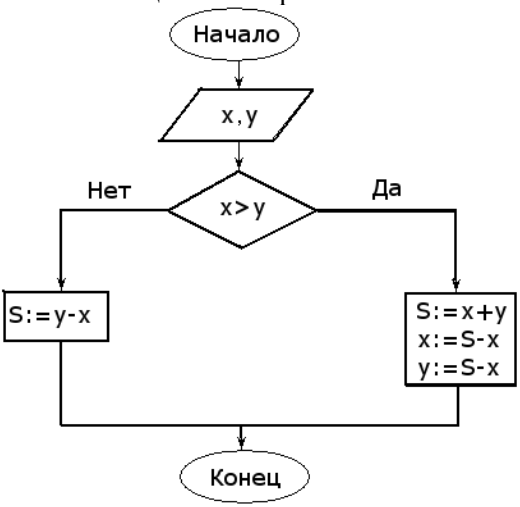



Рис. 34. Блок-схема разветвляющегося алгоритма

В случае необходимости производить ту или иную обработку данных в зависимости от выполнения или невыполнения условий применяют ветвление. Нередко в разветвляющихся алгоритмах встречается конструкция, обозначающая ввод данных (например, с клавиатуры). Эта конструкция обозначается параллелограммом с списком переменных, значения которых вводятся. Конструкция же ветвления обозначается ромбом, с записанным внутри него условием. К ромбу подходит одна стрелка от выполненного элемента блок-схемы, а отходит две стрелки. По одной из этих двух стрелок алгоритм продолжается, если условие выполнено, а по другой ─ если не выполнено. Этот же алгоритм записывается на Паскале следующим образом :

```
Read(x,y);
if  x > y  then
 begin
   S:=x+y;
   x:=S-x;
   y:=S-x;
 end
else
   S:=y-x;
```

Первое действие ─ это введение данных с клавиатуры. Пусть в переменную **x** вводится значение 9, а в переменную **y** вводится 1. Следующее действие ─ это проверка условия **x>y** и решение о том, по какой ветви выполнять алгоритм дальше. При **x=9** и **y=1** условие **9>1** оказывается выполнено, и поэтому выполнение алгоритма продолжается по стрелке «Да». При этом то, что записано по стрелке «нет», не выполняется. По стрелке «Да» записана последовательность действий :

**S:=9+1; x:=10-9; y:=10-1**.

Так что после выполнения этих действий значения переменных **x** и **y** поменялись местами, причем значение переменной **y** стало больше значения переменной **x**, а в переменной **S** оказалась сумма значений **x** и **y**. Выполнение алгоритма завершено.

Чтобы пройти по другой ветви алгоритма, выполним алгоритм с самого начала и зададим в качестве входного значения **x** число 2, а в качестве входного значения **y** число 4. Выражение **x>y** при подстановке этих значений принимает вид 2>4, что неверно. Значит, выполнение алгоритма продолжается по стрелке «Нет», а не по стрелке «Да». В переменную **S** записывается разность 4-2, т.е. число 2. Алгоритм завершается.

Итак, в результате работы алгоритма из этого примера в переменной **S** оказывается сумма значений **x** и **y**, когда **x>y**, и разность **y-x**, когда **x≤y**. При этом в **y** независимо от того, какие и в каком порядке вводятся числа, всегда оказывается наибольшее значение, а в **x** наименьшее.

**Пример 3**. Составные условия.

Составные условия возникают тогда, когда простых условных выражений (например, **x>y**) оказывается недостаточно для отражения в одном выражении всех желаемых условий.

Пусть требуется ответить на вопрос о том, находится ли число **y** между числом **x** и квадратом числа **x**, то есть верно ли, что **y** больше одного из них и меньше другого.

Алгоритм принятия решения о том, какой ответ нужно дать на этот вопрос, можно записать так :

```
otv:='No';
if  x > x*x  then
begin
   if  y > x*x then
      if  y < x then
         otv:='Yes';
end
else
   if  y < x*x then
      if  y > x then
         otv:='Yes'
```

Разберем работу этого алгоритма. Сначала в otv записывается 'No'. Если это значение не изменится в ходе дальнейшего выполнения алгоритма, то ответ станет 'No', то есть «Нет, **y** не лежит между **x** и **x·x**». Чтобы узнать, больше какого из двух значений **x** и **x·x** должно быть значение **y**, нужно выяснить, какое из этих двух значений меньше другого. Если **x > x·x**, то проверяется сначала, что **y>x·x**, если это верно, то проверяется, что **y<x**. Если и это верно, то в качестве ответа предлагается 'Yes', то есть «Да, **y** лежит между **x** и **x·x**». Если же неверно, будто **x > x·x**, то проверяется сначала, что **y<x·x**, если это верно, то проверяется, что **y>x**. Если и это верно, то в качестве ответа тоже предлагается 'Yes'. Итак, если указанное условие о расположении **y** между **x** и **x·x** не выполнено, то otv='No', а если

выполнено, то otv='Yes'.
Эту громоздкую запись алгоритма можно сделать короче за счет компактной записи условия :
« **x > x·x** и при этом как **y>x·x**, так и **y<x**,
или
**x ≤ x·x** и при этом как **y<x·x**, так и **y>x** », то есть коротко :
(**x > x·x** и **y>x·x** и **y<x**) или (**x ≤ x·x** и **y<x·x** и **y>x**).
На языке Паскаль это условие записывается так :

```
((x>x*x)and(y>x*x)and(y<x))  or
        ((x<=x*x)and(y<x*x)and(y>x))
```

поскольку слово «and» обозначает «и», «or» обозначает «или», «<=» обозначает «≤».
Итак, сокращенный вариант программы :

```
if
((x>x*x)and(y>x*x)and(y<x))  or
        ((x<=x*x)and(y<x*x)and(y>x))
then otv:='Yes'
else otv:='No'
```

**Упражнение В**. Напишите программу, в которой решается та же задача, при этом в программе есть только одно слово **if**, а ни одного из условий **x>x*x** , **x<=x*x** нет.

**Пример 4**. Повторение действий. Цикл по условию.
Для выполнения повторяющихся действий используются конструкции, называемые циклами (рис. 35).
Пусть требуется узнать, какой по счету элемент последовательности 1, 1, 2, 3, 5 и т.д. будет не меньше числа 20. Разберем алгоритм решения этой задачи.
Сначала задаются первые два значения в последовательности: **x=1**, **y=1** и **n=2**. Затем проверяется условие **x<20**, то есть **1<20**. Так как условие выполнено, то переходим по стрелке «Да». Теперь определяется номер следующего элемента последовательности : **n:=2+1**, то есть **n** становится равным 3.
В переменной **z** сохраняется текущий элемент последова-

тельности : **z:=x** и вычисляется следующий элемент : **x:=x+y**,

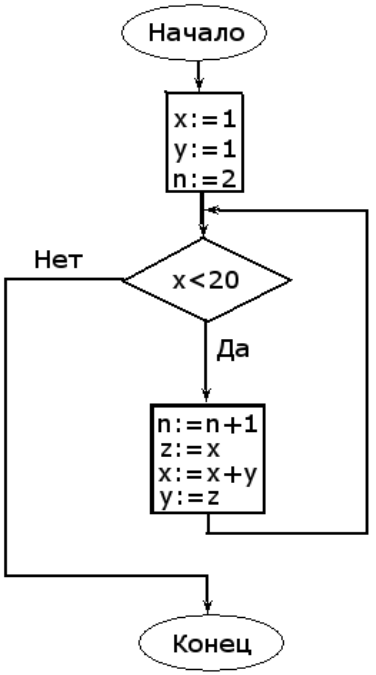



Рис. 35. Блок-схема циклического алгоритма

после чего в **y** предыдущий элемент заменяется текущим : **y:=z**. Теперь **x=2**, а **y=1**. Теперь 2**<20** и точно так же получаем **x=3**, **y=2**, **n=4**. Снова проверяется условие **x<20**, то есть 3<20. Так как условие выполнено, снова осуществляются действия **n:=n+1**, **z:=x**, **x:=x+y**, **y:=z**. Так что **n=5**, **x=5**, а **y=3**. Поскольку 5<20, то действия, которые принято называть телом цикла : **n:=n+1**, **z:=x**, **x:=x+y**, **y:=z** снова повторяются и дают результат : **n=6**, **x=8**, а **y=5**. Выражение 8<20 верно, и выполняется тело цикла. В этот раз **n=7**, **x=13**, а **y=8**. В силу истинности выражения 13<20 тело цикла выполняется еще раз, но теперь **n=8**, **x=21**, а **y=13**. Выражение **21<20** имеет ложное значение, и мы вынуждены перейти по стрелке «Нет». Значит, восьмой член указанной в

условии задачи последовательности не меньше 20.
Алгоритм, описанный при помощи блок-схемы, на языке Паскаль записывается так :

```
x:=1;y:=1;n:=2;
while x<20 do
begin
  n:=n+1; z:=x; x:=x+y; y:=z;
end;
```

Цикл **while** называется циклом с предусловием, так как его тело выполняется только после предварительной проверки условия. Если бы в самом начале оказалось, что условие не выполнено, то тело цикла ни разу не выполнилось бы.
Этот же алгоритм можно реализовать при помощи цикла с постусловием :

```
x:=1;y:=1;n:=2;
Repeat
  n:=n+1; z:=x; x:=x+y; y:=z;
until x>=20;
```

Однако в этом цикле тело выполняется до проверки условия и поэтому **n** не смогло бы оказаться меньше 3, даже если бы мы заменили число 20 в условии на 2.

**Пример 5**. Цикл с заданным числом повторений. Цикл по счетчику.
Если число повторений действий заранее известно, то удобно пользоваться циклом по счетчику.
Пусть требуется вычислить сумму кубов целых чисел, начиная с числа 2 и заканчивая числом 4.
Эта задача на языке Паскаль решается следующим образом :

```
S:=0;
for n:=2 to 4 do
  S:=S +  n*n*n;
```

Выполним прокрутку этого фрагмента программы (то есть выполним этот фрагмент шаг за шагом). Сначала в

переменную **S** записывается значение **0**. Затем начинается выполнение цикла. В переменную **n** записывается значение **2**. Значение **S** заменяется числом **8**, полученным в результате вычисления выражения **0 + 2·2·2**. Теперь переменная-счетчик **n** принимает следующее значение : **3**, а значение переменной **S** устанавливается равным **8 + 3·3·3**, то есть **35**. Наконец, счетчик принимает последнее значение **4,** и в **S** записывается число **99**, полученное сложением старого значения **35** и числа **64**. Значит, сумма кубов чисел **2**, **3** и **4** равна **99**.

Цикл **for** обладает двумя особенностями :

1) « **for n:=3 to 2 do** » не выполнится ни разу, так как 3>2, это же относится ко всем фрагментам вида :
« **for n:=K to L do** », где **K > L** ;
2) Для изменения значений счетчика в обратном порядке используется слово «**downto**» :
« **for n:=3 downto 2 do** » выполнится для значений **n** в таком порядке : сначала 3, потом 2.

**Пример 6**. Массивы.

Для хранения и обработки значительного объема данных используются массивы. Массив ─ это структура данных, в которой хранятся элементы одного и того же типа, причем к каждому элементу можно обратиться по его уникальному номеру (то есть по индексу). Массивы будем обозначать перечислением элементов, например : M=( 1, 0, 3, -2, 4, -1, 2 ) ─ массив из семи элементов.

| индекс | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| элемент | 1 | 0 | 3 | -2 | 4 | -1 | 2 |

Чтобы использовать элемент с номером **i** , нужно записать имя переменной-массива, а затем в квадратных скобках указать номер **i** : M[**i**]. Так, например, значение M[4] есть число -2, элемент M[6] хранит число -1.

Пусть надо вычислить сумму кубов тех элементов массива M, которые меньше 2 (то есть не их индексы меньше 2, а именно их значения меньше 2).
Решение:

```
S:=0;
for i:=1 to 7 do
  if  M[i] < 2  then
     S:=S +  M[i]*M[i]*M[i];
```

Выполним прокрутку. Инициализируем S значением 0 (то есть выполняем начальную установку значения переменной S, равным 0). Приступаем к выполнению цикла. При i=1 значение M[i]=1. Так как 1<2, то значение S увеличивается на 1·1·1 и становится равным 1. При i=2 выражение M[2]<2 тоже оказывается верным, так как 2 является положительным числом. Но S не изменяется по причине того, что M[2]=0. Далее результаты этапов прокрутки запишем в таблицу :

| i | M[i] | M[i]<2 | S |
|---|---|---|---|
| 3 | 3 | 3<2, нет | 1 |
| 4 | -2 | -2<2, да | -7 |
| 5 | 4 | 4<2, нет | -7 |
| 6 | -1 | -1<2, да | -8 |
| 7 | 2 | 2<2, нет | -8 |

Таким образом, сумма кубов элементов массива M, значения которых меньше 2, равна -8.

## Список литературы


1. Абрамов В.Г. и др. Введение в язык Паскаль. -М.: Наука, 1988. -320 с.
2. Абрамов С.А. и др. Задачи по программированию. -М.: Наука, 1988. -224 с.
3. Абрамов С.А., Зима Е.В. Начала программирования на языке Паскаль. -М.: Наука, 1987. -112 с.
4. Анеликова Л. Алгоритмика в теории и практике (+CD). - М.: Солон-пресс, 2007. -72 с.
5. Князева М.Д. Алгоритмика : от алгоритма к программе (+ CD). -М.: КУДИЦ-ПРЕСС, 2006. -192 с.
6. Грогоно П. Программирование на языке Паскаль. -М.:Мир, 1982. -382 с.
7. Прайс Д. Программирование на языке Паскаль. Практическое руководство. -М.:Наука, 1978. -208 с.

**Олег Всеволодович Бабич**

**Основы алгоритмики**

Методическое пособие

Подписано в печать 05. 09. 08
Формат 60×84 1/16. Печать офсетная.
Усл. печ. л. 3,49. Уч.-изд. л. 3,2.
Тираж 80 экз. Заказ №____
Редакционно-издательский отдел УдГУ
Типография ГОУВПО «Удмуртский госуниверситет»
426034, Ижевск, Университетская 1, корп. 4.