Алгоритмы и исполнители

1. Алгоритмы и исполнители

Что такое алгоритм?

"Прежде, чем что-нибудь сделать, надо составить план", — говорила Алиса из сказки Льюиса Кэрролла. И в жизни мы все время составляем планы наших действий, например, утром большинство из нас действует по такому плану:

```
встать
одеться
умыться
позавтракать
выйти из дома в школу или на работу
```

В таком же виде можно записать план для того, чтобы заварить чай, сделать бутерброд с колбасой, купить себе мороженое, вымыть грязные руки, …

В информатике план действий называют **алгоритмом**. Алгоритм состоит из отдельных **шагов** – команд. Ни одну из них нельзя пропустить, чаще всего никакие команды нельзя поменять местами (что при этом произойдет?).

Для каждого шага этого алгоритма можно предложить более подробный план. Например, для действия "**позавтракать**":

```
вскипятить чайник
сделать бутерброд
съесть бутерброд с чаем
вымыть посуду
```

И тут каждый шаг, в свою очередь, тоже можно расшифровать – составить более подробный план. Где же остановиться? Ответ прост – это зависит от **исполнителя** — того, **кто будет выполнять** этот алгоритм. Надо остановиться на таком плане, в котором исполнителю будет понятно, как выполнить каждый шаг.

Исполнители

Что такое исполнитель?

Исполнители часто встречаются в сказках. В одной из них Иван-Царевич говорит Избушке-На-Курьих-Ножках: "Избушка, избушка! Встань к лесу задом, ко мне передом!". При этом команда должна быть задана **очень точно,** чтобы исполнитель ее понял. В сказке "Али-Баба и сорок разбойников" волшебная дверь открывалась по команде "Сезам, откройся!". Жадный Касым, тайно проникший в пещеру, забыл эту фразу и не смог выйти из пещеры.

И Избушка-На-Курьих-Ножках, и волшебная дверь имеют много общего: они умеют понимать и выполнять некоторые точно заданные команды, то есть являются **исполнителями**.

- **Исполнитель** – это тот, кто умеет понимать и выполнять некоторые команды.
- **Среда исполнителя** – это предметы, которые окружают исполнителя и с которыми он работает.

♦ **Список (или *система*) Команд Исполнителя (СКИ) –** набор команд, понятных исполнителю. Исполнитель может выполнить только те команды, которые входят в его СКИ.

Исполнителями могут быть

- **люди:** ученик, рабочий, учитель, бригада;
- **животные:** дрессированная собака (санитар, розыскная, охотничья), кошка;
- **машины:** станки, роботы, компьютеры;

Вообще говоря, исполнителями могут быть даже **растения:** подсолнечник (разворачивается на солнце), кувшинки (закрываются на ночь).

Человек как исполнитель отличается от всех остальных исполнителей несколькими признаками, например:

1. Понимает команды в различных вариантах (например “Сядь!”, “Садись!”, “Присядь!”).
2. Выполняя команды, «додумывает» их с учетом своего опыта.
3. Может отказаться исполнять команду, если она ему не нравится (“Ешь манную кашу!”, “Выстрели в окно из рогатки!”). То есть человек обладает волей и отвечает за свои действия.

Для решения большинства задач недостаточно отдать одну команду исполнителю, надо составить для него алгоритм — план действий, состоящий из команд, которые ему понятны (входят в его СКИ). Таким образом, можно дать определение алгоритма.

♦ **Алгоритм –** это точно определенный план действий исполнителя, направленный на решение какой-то задачи. В алгоритм можно включать только те команды, которые есть в СКИ исполнителя.

🗐 *Ошибки при работе исполнителей*

Работа исполнителя не всегда проходит гладко – иногда встречаются ошибки. Существует три вида ошибок исполнителей.

“НЕ ПОНИМАЮ”	Заданной команды нет в списке команд исполнителя, и он ее не понял. Вероятно, мы ошиблись в записи текста команды.
“НЕ МОГУ”	Исполнитель понял команду, но не может ее выполнить. Например, роботу дана команда “вперед”, а впереди стоит стенка, и он не может идти. Или собаке скомандовали “Сидеть!”, а она уже сидит.
ЛОГИЧЕСКИЕ ОШИБКИ	Исполнитель понял команду и выполнил ее, но сделал не то, что мы от него хотели. Причина этого – наша ошибка в составлении задания (алгоритма).

🗐 *Как ввести нового исполнителя?*

Введем теперь нового исполнителя, которого назовем **дядя Федор** (как у Э. Успенского). Чтобы ввести нового исполнителя надо:

- задать **среду** исполнителя – класс, столы, стулья;
- составить **СКИ**:
  - ◊ ВСТАНЬ
  - ◊ СЯДЬ
  - ◊ ПОДНИМИ РУКУ
  - ◊ ОПУСТИ РУКУ

- ◊ ПРЫГНИ
- ◊ МЯУКНИ

- определить, как передаются команды исполнителю (голосом, жестом, письменно, по рации или как-то иначе);
- определить, как исполнитель выполняет команды;
- определить, в каких случаях возникает ошибка “НЕ МОГУ”.

Старинные задачи

Переправа. Крестьянину надо переправить через реку волка, козу и капусту. Но кроме человека лодка вмещает только или волка, или козу, или капусту. Оставить на берегу без присмотра волка с козой или козу с капустой нельзя (съедят!). Как крестьянину переправить свой груз?

Переправа семьи. Отец, мать и двое детей хотят переправиться через реку. Все умеют грести, но лодка выдерживает либо одного взрослого, либо двоих детей. Как им всем переправиться на другой берег?

Фальшивые монеты. Из 9 монет одинакового достоинства одна фальшивая (более легкая). Как ее найти за два взвешивания с помощью чашечных весов без гирь?

Какие бывают алгоритмы?

Линейный алгоритм

В линейном алгоритме команды выполняются последовательно, одна за другой. Примером линейного алгоритма может служить алгоритм заварки чая:

```
вскипятить воду
сполоснуть заварочный чайник горячей водой
насыпать заварку
залить заварку кипятком
закрыть чайник чем-нибудь теплым
подождать 5 минут
... теперь можно пить чай
```

Разветвляющийся алгоритм

В разветвляющемся алгоритме порядок следования команд может быть разный в зависимости от того, какова окружающая обстановка. Примером разветвляющегося алгоритма может служить алгоритм перехода улицы:

```
подойти к пешеходному переходу
если есть светофор, то
  ждать зеленого света
  перейти улицу
иначе
  ждать, пока слева не будет машин
  перейти улицу до середины
  ждать,  пока справа не будет машин
  перейти вторую половину улицы
```

Циклический алгоритм

В циклическом алгоритме некоторые действия повторяются несколько раз (в информатике говорят, что выполняется **цикл**). Существуют два вида циклических алгоритмов.

В одном из них мы знаем заранее, сколько раз надо сделать эти действия, в другом мы должны остановиться лишь тогда, когда выполнится некоторое условие.

Примером **цикла первого типа** является наша жизнь в рабочие дни (от понедельника до субботы) – мы выполняем 6 раз почти одни и те же действия.

Пример **цикла второго типа** – алгоритм распилки бревна: мы не можем заранее сказать, сколько раз нам надо провести пилой от себя и на себя – это зависит от плотности дерева, качества пилы и наших усилий. Однако мы точно знаем, что надо закончить работу, когда очередное отпиленное полено упадет на землю.

```
/* Число шагов известно */
повторить 6 раз
    проснуться
    встать
    позавтракать
    пойти в школу
    вернуться домой
    пообедать
    сделать уроки
    поиграть в футбол
    лечь спать
/* программа на воскресенье */
спать
```

```
/* Число шагов неизвестно,
   но   ограничено   условием
*/
положить бревно на козлы
наметить место распила
пока полено не отвалится
    пилить от себя
    пилить на себя
положить полено в поленницу
```

Программы

Человек способен понимать смысл команды и часто может «додумать», что от него хотели даже тогда, когда команда задана неточно. Для того, чтобы алгоритм был понятен роботу, компьютеру или другой машине, недостаточно только написать команды, надо еще и оформить алгоритм в таком виде, в котором его понимает машина, то есть записать в **формальном виде.**

В формальной записи алгоритма можно использовать только те команды, которые входят в СКИ исполнителя. Кроме того, надо соблюдать специальные правила оформления, которые позволят исполнителю распознать команды и определить последовательность их выполнения.

```
Репка  это название алгоритма */
{    /* эта скобка обозначает начало алгоритма */
посадить репку; /*команда заканчивается знаком ;*/
вырастить репку;
пытаться вытащить репку;
позвать Бабку;  пытаться вытащить репку;
позвать Внучку; пытаться вытащить репку;
позвать Жучку;  пытаться вытащить репку;
позвать Кошку;  пытаться вытащить репку;
позвать Мышку;  вытащить репку;
} здесь алгоритм заканчивается */
```

Исполнителем для этого алгоритма является дед — именно он должен выполнять эти команды.

Правила записи алгоритмов для компьютеров

Алгоритм можно записать разными способами и даже на разных языках. Хотя при этом исполнитель может, конечно, их не понять. Вы знаете, что есть специальные виды исполнителей алгоритмов — компьютеры. Они выполняют **программы**.

♦ **Программа** – это алгоритм, записанный в форме, понятной компьютеру.

Существуют специальные правила записи программ для компьютеров. На рисунке вверху страницы их характерные элементы выделены в рамках:

1. любой алгоритм имеет **название**;
2. алгоритм начинается с открывающей фигурной скобки "**{**" и заканчивается закрывающей фигурной скобкой "**}**"; команды, расположенные между этими скобками, называются **телом алгоритма**;
3. в алгоритм могут входить только те команды, которые есть в СКИ исполнителя;
4. каждая команда заканчивается знаком "**;**", который обозначает конец команды;
5. для того, чтобы нам было легче разбираться в программах, используют комментарии - текстовые пояснения, которые начинаются знаками **/*** и заканчиваются знаками ***/**; исполнитель не обращает внимания на комментарии в алгоритме.

Задача о перевозчике

Давно известна старинная задача о крестьянине, которому надо перевезти на другой берег реки **волка**, **козу** и **капусту** на лодке, в которую помещается сам крестьянин и на **одно свободное место** он может взять или волка, или козу, или капусту. Сложность заключается в том, что коза и волк ведут себя прилично только в присутствии крестьянина, в его отсутствие коза съест капусту, а волк съест козу.

Когда крестьянин едет на другой берег в первый раз, он может взять козу, так как только волк и капуста могут остаться наедине.

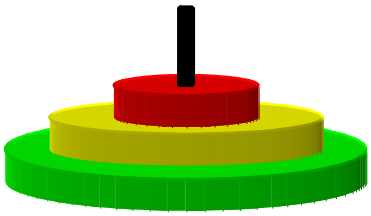
Затем он возвращается и берет с собой волка (или капусту - второй вариант решения). Но он не может оставить волка (или капусту) с козой на другом берегу и поэтому вынужден взять с собой козу обратно.

Вернувшись назад и высадив козу, он забирает волка (или капусту) и перевозит его. Теперь на другом берегу снова останутся волк с капустой и крестьянину останется только забрать козу.

```
Перевоз-1
{
перевезти козу;
вернуться;
перевезти волка;
вернуться с козой;
перевезти капусту;
вернуться;
перевезти козу;
}
```

```
Перевоз-2
{
перевезти козу;
вернуться;
перевезти капусту;
вернуться с козой;
перевезти волка;
вернуться;
перевезти козу;
}
```

Ханойские башни (рекурсивные алгоритмы)



Одна из любимых детских игрушек – пирамидка с цветными кольцами разного диаметра, насаженными на стержень.

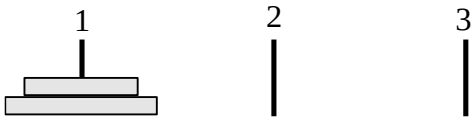
Однако есть страны, где в эту игру играют уважаемые и почтенные старцы. Придумали ее монахи древнего Ханоя (теперь это территория Вьетнама). У них была одна полная пирамидка с 64 кольцами и два пустых стержня. Считалось, что когда все кольца удастся перенести на другой стержень, соблюдая все правила (см. ниже), наступит конец света.

Правила игры

Требуется перенести пирамидку с одного стержня на другой, используя третий стержень в качестве промежуточного и соблюдая следующие правила:

- за одно действие можно переносить только одно кольцо;
- кольцо можно укладывать либо на свободный стержень, либо на большее кольцо.

Решим сначала самую простую задачу - для пирамидки из двух колец.

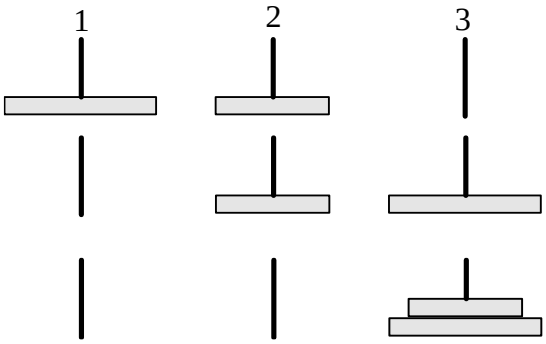


Обозначим стержни номерами:

1 левый стержень, на котором находится пирамидка в начале игры
2 средний стержень, вспомогательный
3 правый стержень, на него надо перенести пирамидку

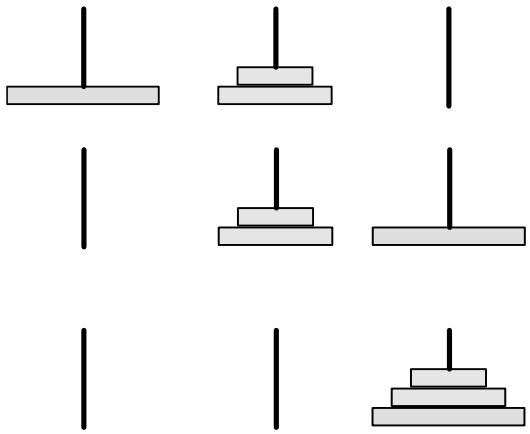
Будем обозначать ходы стрелками. У основания стрелки будем писать номер исходного стержня, с которого берем кольцо, а у острия - номер стержня, на который его переносим.

```
Ханой-2
{

1 → 2;

1 → 3;

2 → 3;
}
```



Немного сложнее решить задачу для пирамидки из трех колец. Заметьте, что нижнее кольцо можно класть только на пустой стержень. А для этого нам надо верхние два кольца переложить на средний стержень, воспользовавшись алгоритмом ***Ханой-2***. Затем переносим большое кольцо на третий стержень и, снова используя алгоритм ***Ханой-2***, переносим два меньших кольца на третий стержень.

```
Ханой-3
{
/* переносим два кольца */
/* на второй стержень */
1 → 3; 1 → 2; 3 → 2;
/* переносим большое кольцо */
/* на третий стержень */
1 → 3;
/* переносим два маленьких
кольца */
/* со второго на третий */
2 → 1; 2 → 3; 1 → 3;
}
```



В этом алгоритме мы два раза использовали алгоритм ***Ханой-2***, но при этом разные стержни выступали в качестве конечного и вспомогательного.

Решение для пирамиды из ***n*** колец можно записать в таком виде:

```
Ханой ( n, начальный, вспомогательный, конечный )
{
если n > 1, то
     Ханой ( n-1, начальный, вспомогательный, конечный );
начальный → конечный;
если n > 1, то
     Ханой ( n-1, вспомогательный, конечный, начальный );
}
```

Здесь в качестве начального, конечного и вспомогательного можно использовать любые стержни. Алгоритм ***Ханой*** фактически предлагает решать задачу для ***n*** колец через две задачи для меньшего числа колец (***n-1***). Такой прием в программировании называется **рекурсия.**

Что такое рекурсия?

♦ **Рекурсия** – специальный прием в программировании, когда алгоритм решения задачи содержит алгоритм решения подобной задачи, но с другими исходными данными.

Теперь мы познакомились с четвертым видом алгоритмов – **рекурсивным алгоритмом.** Заметим, что для переноса пирамидки из двух колец требуется всего ***3*** хода, для трех колец – уже ***3+1+3=7*** ходов, для четырех – ***15*** и т.д. Можно показать, что для переноса пирамидки из ***n*** колец нам потребуется $2^n - 1$ ходов. У монахов древнего Ханоя была пирамидка из ***64*** колец и они верили, что когда удастся перенести всю пирамидку на третий стержень, наступит конец света. Конечно это легенда, но число $2^{64} - 1$ в самом деле очень велико и для того, чтобы сделать столько ходов, не хватит нескольких человеческих жизней.

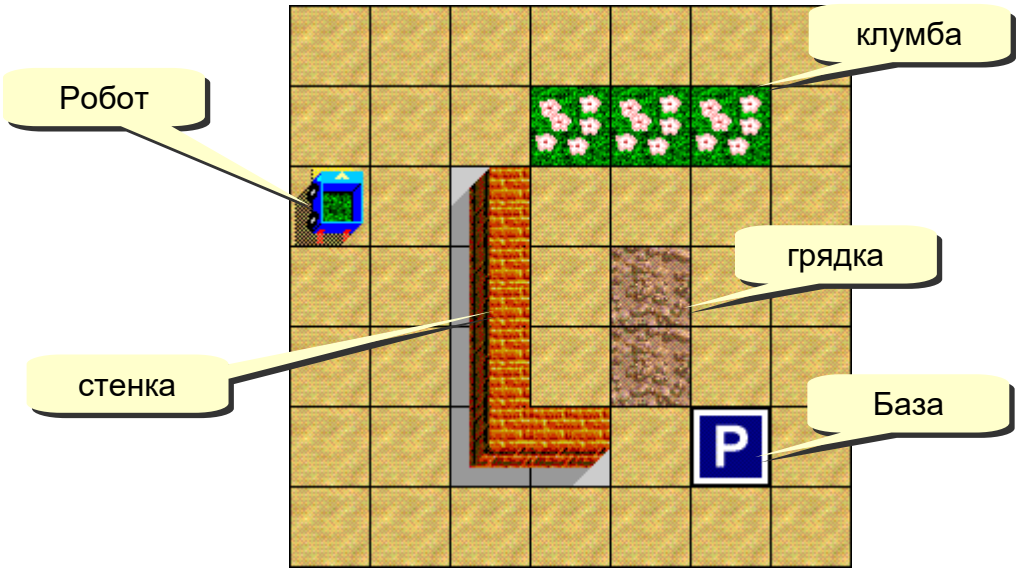
Рекурсию имеет смысл использовать тогда, когда в результате исходная задача сводится к более простой. Доказано, что любой рекурсивный алгоритм можно заменить алгоритмом без рекурсии (который иногда может быть очень громоздким). Так как использование рекурсии в реальных программах связано с некоторыми техническими проблемами, лучше ее не применять, если есть простой нерекурсивный алгоритм.

2. Исполнитель Робот

Среда Робота

Учебный исполнитель Робот предназначен для того, чтобы без участия человека сажать цветы в подготовленные для них грядки. В программе, с которой вы будете работать, Робот изображен в виде машинки, которая ездит по полю. Поле размечено на квадраты, каждый из которых может быть: 1) свободным местом  ; 2) грядкой  или 3) стенкой  . Робот может переходить из клетки в клетку по грядкам или по свободным клеткам, **ходить по клумбам с цветами запрещается**. Он должен посадить цветы на всех грядках и вернуться на Базу, обозначенную значком  , для пополнения запасов.

Робот может двигаться вперед и назад, а также разворачиваться на 90 и 180 градусов влево или вправо. Конечно, в реальной обстановке на Робота влияет ветер, дождь, неровность земли и т.п., но мы их не будем учитывать. Такое упрощенное представление называется **моделью** Робота.



Основные команды Робота

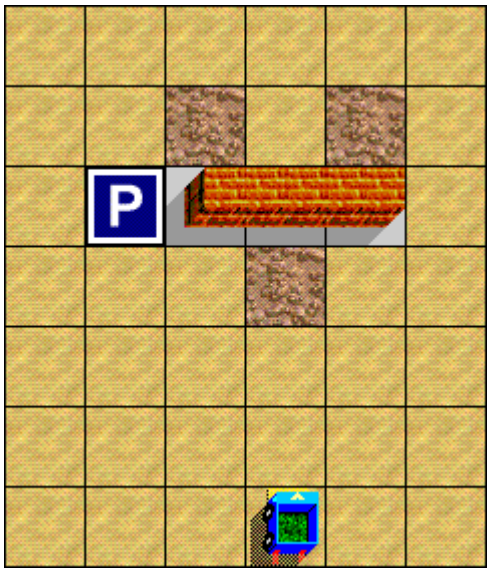
Как и любой исполнитель, Робот понимает только ограниченный набор команд, которые входят в его СКИ (список команд исполнителя). Пока нам хватит нескольких команд, перечисленных ниже:

♦ **СКИ Робота:**

`направо;`	повернуться на 90 градусов вправо
`налево;`	повернуться на 90 градусов влево
`кругом;`	развернуться кругом (на 180 градусов)
`вперед ( n );`	перейти на *n* клеток вперед
`назад ( n );`	перейти на *n* клеток назад
`посади;`	посадить цветы на грядке в том месте, где стоит Робот

Позже мы немного расширим СКИ и добавим в него новые команды. Робот не может ходить по диагонали, проходить сквозь стенки и топтать цветы на клумбах.

Простейшая программа (задача z1-3.maz)



```
ТриКлумбы
{
вперед(3);
посади;
направо;
вперед(2);
налево;
вперед(2);
налево;
вперед(1); посади;
вперед(2); посади;
вперед(1);
налево;
вперед(1);
}
```

Имя программы должно состоять из одного «слова», обратите внимание, что внутри нет пробелов. Каждая команда заканчивается точкой с запятой. Можно записывать несколько команд в одну строчку.

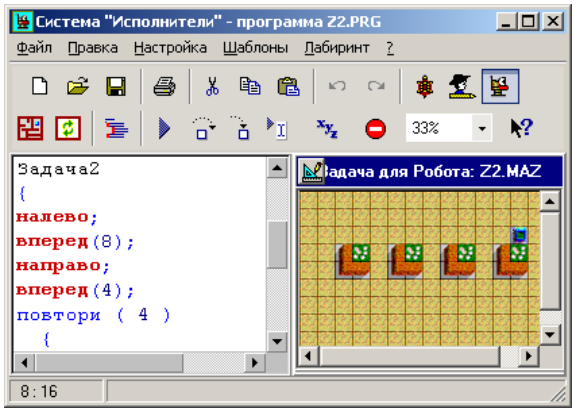
Какие ошибки могут быть у Робота?

1. **Синтаксические (“НЕ ПОНИМАЮ”)** – появляются при ошибках в написании команд, например
   ```
   влево;
   вперет ( 3 );
   направо ( 2 );
   ```
2. **Отказы (“НЕ МОГУ”) –** появляются, например, если Роботу приказывают идти прямо на стенку или сажать цветы там, где нет грядки.
3. **Логические** – возникают тогда, когда Робот понимает команды и делает все, что ему сказали, но результат совсем не тот, какой мы ожидали.

Синтаксические ошибки и отказы обнаруживает сам исполнитель. Когда вы будете работать с компьютером, вы увидите сообщения об таких ошибках. Самые сложные ошибки – логические – придется искать самим.

Работа в системе Исполнители

Чтобы проверить программу и посмотреть исполнителя **Робот** в работе, мы будем использовать систему **Исполнители**, установленную на компьютерах. Найдите на **Рабочем столе** ярлык программы и дважды щелкните по нему. Когда программа запустится, вы увидите окно, показанное на рисунке справа.



Окно состоит из трех частей: верху расположены меню и кнопки для управления исполнителем, слева – редактор программы, а

справа – поле исполнителя.

Сначала загрузите задачу для Робота, щелкнув по кнопке и выбрав заданный файл.

Затем надо набрать программу в поле редактора. Для того, чтобы ускорить ввод команд, удобно использовать меню **Шаблоны.** Там есть все команды языка программирования и команды исполнителя **Робот**.

Для того, чтобы компьютер выровнял все строки программы (привел программу в «приличный» вид), можно нажать клавишу **F6** или щелкнуть по кнопке на панели инструментов.

Когда программа готова, запишите ее на диск, нажав клавишу **F2** или кнопку на панели инструментов. В ответ на это при первой записи файла на диск появляется окно для ввода имени файла, где вам надо ввести любое имя и затем щелкнуть на кнопку **ОК**. При записи файла в следующий раз имя уже известно, поэтому система переименует старую версию, сделав у нее расширение ***.bak**, а новую запишет с тем же именем.

Для выполнения программы надо нажать клавишу **F9** или кнопку на панели инструментов. Если в программе нет синтаксических ошибок, которые машина обнаруживает, вы увидите, как Робот (в виде машинки) выполняет программу.

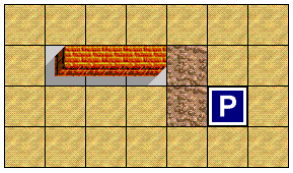
Если ошибки есть, красным цветом будет выделена строка, в которой обнаружена ошибка, и выведено сообщение на экран. Посмотрите внимательно на эту строку и на предыдущую, нажмите на клавишу **Enter** и исправьте ошибку.

Если ошибок нет, но Робот не выполнил задание, в программе есть **логическая ошибка**. Для ее обнаружения воспользуйтесь **режимом отладки**: при нажатии на клавишу **F8** исполнитель выполняет одну строку программы и останавливается. Такой режим называется **пошаговым**. Таким образом, можно определить, в какой строчке программа начинает выполняться не так, как вам хочется. Обнаружив ошибку, нажмите клавишу **Esc** для выхода из режима отладки. Когда все получилось, запишите новый вариант на диск и закончите работу, щелкнув по кнопке в правом верхнем углу окна или нажав клавиши **Alt+F4**.

Если вы забыли какую-то команду или хотите узнать то, что вам еще не рассказывали, щелкните на кнопке или нажмите клавишу **F1**, чтобы войти в справочную систему.

Если программа не доделана и записана на диск, вы сможете в следующий раз загрузить старую программу, нажав на клавишу **F3** или щелкнув по кнопке . Чтобы начать новую программу и очистить поле редактора, щелкните по кнопке .

Задачи



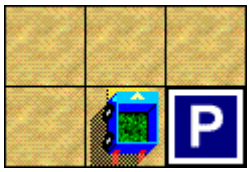
1. Определите, чего не хватает в условии этой задачи. Дополните условие и решите задачу.

2. Известно, что Робот вышел из некоторой точки *A* и туда же пришел после выполнения задания. Восстановить недостающую строчку (или строчки) в программе.

```
Вокруг
{
вперед ( 2 );
...
вперед ( 2 );
налево;
}
```

```
Вокруг2
{
вперед ( 2 );
...
налево;
вперед ( 3 );
}
```

3. Известна программа перехода Робота из одной клетки в другую. Составить программу обратного хода Робота.
4. Перевести Робота на Базу всеми возможными способами из трех команд. Можно ли сделать это в 2 шага (в 5 шагов? 10 шагов? 15 шагов? 1991 шаг?).

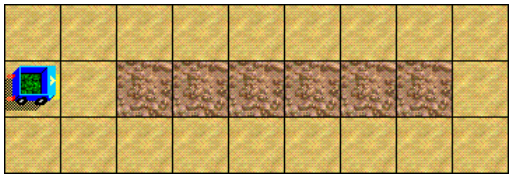


5. Составить и решить свою задачу для Робота (придумать интересное название).

3. Циклы

Что такое цикл (задача z2-3.maz)?

Часто исполнителю надо выполнить какую-то последовательность команд несколько раз. Например, в задаче на рисунке Робот должен подойти к ряду клеток, которые надо закрасить, и затем выполнить 6 раз команды ***вперед(1)*** и ***посади***.



В данном случае эти команды надо повторить только 6 раза и можно легко 6 раз написать одинаковые команды. Но представьте, что надо сделать одинаковые операции 100 или 200 раз! В программировании в таких случаях используется специальная команда (*оператор цикла*), которая говорит исполнителю, что какую-то часть программы надо сделать несколько раз.

- **Цикл** — это многократное повторение одинаковых действий
- **Тело цикла** – это команды, которые выполняются несколько раз.
- **Шаг цикла** – это однократное выполнение тела цикла.

Для нашей задачи подходит цикл ***повтори*** (или ***repeat***), в котором с известным числом шагов. Программа с использованием оператора цикла выглядит так:

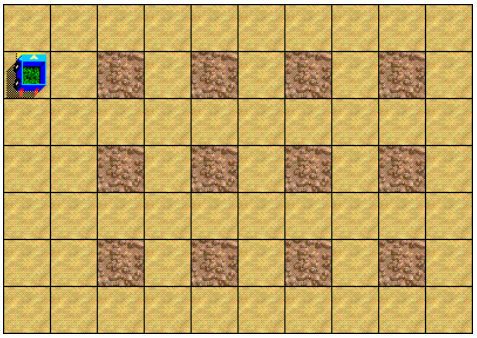
```
Ряд
{
вперед ( 1 ); /* подойти к месту работы */
повтори ( 6 )
  {
  вперед ( 1 );
  посади;
  }
}
```

Правила использования оператора цикла

1. Цикл ***повтори*** (или ***repeat***) используется тогда, когда число шагов заранее известно или может быть вычислено.
2. Оператор цикла начинается заголовком цикла – ключевым словом ***повтори***, за которым в скобках указывается нужное количество шагов.
3. Тело цикла начинается открывающей фигурной скобкой **{** и заканчивается закрывающей **}.**
4. Если тело цикла включает всего один оператор, скобки можно не ставить.
5. Для того, чтобы легче разбираться в программе, применяют специальную **систему записи с отступами**: тело цикла смещают вправо на 2-3 символа — это позволяет сразу видеть, где начинается и где заканчивается цикл. Для того, чтобы компьютер автоматически сделал отступы в программе, можно нажать клавишу **F6**.

Вложенные циклы (задача z3-3.maz)

Рассмотрим задачу для Робота, в которой требуется сажать цветы во многих местах площадки (на рисунке справа).



Как бы такая задача решалась в реальных условиях? Можно предложить такой вариант: Робот сначала сажает цветы в первом (верхнем) ряду, затем во втором и т.д.

Для обработки одного ряда можно использовать цикл ***повтори(4)***. В программе надо обработать 3 ряда, то есть написать три одинаковых цикла. Тогда получается, что можно снова использовать цикл ***повтори(3)*** для трех рядов, но внутри него также будет находиться цикл.

♦ **Вложенный цикл** – это такой цикл, который находится внутри другого цикла.

Ниже даны два возможных решения этой задачи. Они показывают, что внутренний и внешний циклы можно переставлять, если порядок обработки грядок безразличен. Главное – перевести Робота в нужную клетку и в нужное положение перед тем, как начнется следующий цикл. Кроме того, нельзя забывать, что Робот не может ходить по клумбам.

```
Способ1
{
направо;
повтори ( 3 )
  {
  повтори ( 4 )
    {
    вперед ( 2 );
    посади;
    }
  направо;
  вперед ( 2 );
  налево;
  назад ( 8 );
  }
```

```
Способ2
{
направо; вперед(2);
направо;
повтори ( 4 )
  {
  повтори ( 3 )
    {
    посади;
    вперед ( 2 );
    }
  налево; вперед(2);
  направо;
  назад ( 6 );
  }
```

4. Алгоритмы с обратной связью

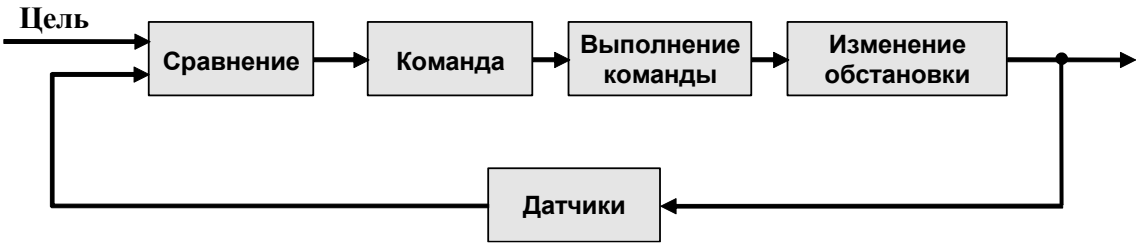
Что такое обратная связь и зачем она нужна?

До сих пор мы приказывали Роботу выполнить какую-то задачу, предполагая, что обстановка полностью известна: мы точно знаем сколько шагов до стенок, какую они имеют форму и где расположены. Мы не анализировали результаты действий Робота и обстановку на поле. Такой подход напоминает действия начальника, который отдает приказания, но не проверяет их выполнение, или шофера, который ведет машину с закрытыми глазами, полагаясь на свое знание дороги.

При решении сложных задач ситуация часто известна не полностью и надо анализировать обстановку, которая изменяется во время работы исполнителя. Человек очень часто не знает, сколько шагов ему надо пройти и не задумывается об этом, потому что он знает, куда он идет, то есть знает, где (**при каком условии**) остановиться.

♦ **Обратная связь** – это информация об окружающей обстановке, которую исполнитель использует для выбора нужного варианта выполнения алгоритма..

Действие обратной связи можно описать такой схемой:



Обратная связь дает нам возможность контролировать результаты действий исполнителя во время его работы и следить за внезапными изменениями обстановки. Если обстановка и цель не совпадают, то блок *Сравнение* вырабатывает сигнал ошибки, на основе которого исполнитель получает команду на дальнейшие действия. После выполнения очередной команды обстановка меняется и снова сравнивается с желаемым результатом.

Как Робот использует обратную связь?

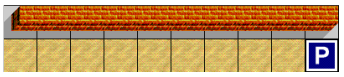
Робот имеет *датчики*, которые позволяют ему получать информацию об обстановке. Датчики определяют, например, есть ли стена в каком-то направлении. Чтобы использовать эту информацию в программе, в СКИ Робота есть специальные **логические команды**.

♦ **Логическая команда** – это условие, которое может быть верным (истинным) или неверным (ложным).

У Робота есть датчики, которые позволяют определять, что находится в той клетке, где он сейчас находится, и в соседних клетках. Вот все логические команды Робота:

```
справа_стена   справа_клумба      справа_свободно
слева_стена    слева_клумба       слева_свободно
впереди_стена  впереди_клумба     впереди_свободно
сзади_стена    сзади_клумба       сзади_свободно
грядка         база
```

Команды ***грядка*** и ***база*** определяют, есть ли грядка (или база) в клетке, где сейчас находится Робот.



Пример 1 **(задача z4-3.maz).** Роботу надо придти на Базу, которая расположена на краю стенки. Расстояние от Робота до стенки и длина стенки неизвестны.

Сначала Роботу надо подойти к стенке. Если бы мы управляли Роботом вручную, то надо было бы поступать так:

1) выдать запрос ***впереди_свободно;***
2) если Робот получил от датчиков ответ ***"нет"***, то он выполнил задание и находится у стены;
3) если получен ответ ***"да"***, то сделать шаг вперед и повторить весь процесс.

На втором этапе Роботу повернуться направо и идти вперед, пока он не придет на Базу. Заметим, что расстояние до Базы также неизвестно, но Робот с помощью логической команды ***база*** может обнаружить, что он уже пришел на место. Решение задачи в виде программы дано ниже в рамке.

Цикл с условием

Мы знаем, что многократное выполнение группы команд называется *циклом*. Однако здесь мы не можем применить цикл ***повтори***, так как число шагов заранее неизвестно – оно определяется во время работы программы.

```
Подход
{
  пока ( впереди_свободно )
    {
    вперед ( 1 );
    }
  направо;
  пока ( не база )
    вперед ( 1 );
}
```

Тем не менее, есть четкое условие, по которому Робот должен закончить работу: если перед ним оказывается стена. Таким образом, Робот должен выполнять цикл ***пока впереди свободно***. Для этой цели служит специальный вид цикла – цикл ***пока*** (или ***while***, от английского *while* – пока). Такой вид цикла называется *циклом с условием*, поскольку он заканчивается, когда нарушается условие в заголовке цикла.

Для того, чтобы придти на Базу, в программе используется цикл ***пока не база***. Это условие истинно (верно), если Робот еще на пришел на Базу и надо двигаться дальше. Если Робот вступил в клетку, где находится База, условие ***база*** стало истинным, а условие ***не база*** – ложным, поэтому цикл закончится.

Правила использования цикла *пока*

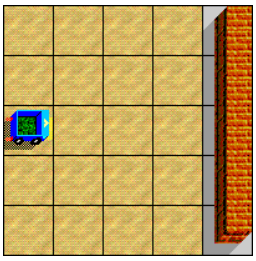
1. Цикл ***пока*** используется тогда, когда число повторений цикла заранее неизвестно, но ограничено каким-то условием.
2. Оператор цикла начинается заголовком цикла – ключевым словом ***пока***, за которым в скобках указывается **логическая команда** – условие, при котором **выполняется** цикл.
3. Если условие перестает быть верным (**истинным**), выполнение цикла заканчивается и исполнитель переходит к следующей команде.
4. Условие проверяется **в начале цикла**, то есть если перед выполнением цикла условие **ложно**, то цикл не выполнится **ни разу**.
5. В цикле выполняются все операторы, заключенные в фигурные скобки;
   Если тело цикла включает всего один оператор, скобки можно не ставить.
5. Для того, чтобы легче разбираться в программе, все команды, входящие в цикл, смещают вправо на 2-3 символа – это позволяет сразу видеть, где начинается и где заканчивается цикл.

Пример 2. При такой программе в той же задаче, что и в примере 1, Робот не будет ничего делать, так как сейчас справа от него нет стенки, и условие ***справа_стена*** не выполняется.

```
Ничего
{
пока ( справа_стена )
   вперед ( 1 );
}
```

Важно помнить, что условие **не** проверяется внутри цикла, то есть датчик срабатывает только тогда, когда выполняется команда в заголовке цикла.

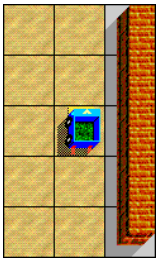
Пример 3. В этом примере программа для Робота составлена так, что он врежется в стенку и сообщит об ошибке "**НЕ МОГУ**".



```
Диверсия
{
пока ( впереди_свободно )
    {
    вперед ( 2  );
    }
}
```

С циклом ***пока*** связано одна из самых неприятных ошибок программистов – **зацикливание**. Оно происходит в тех случаях, когда условие в заголовке цикла ***пока*** никогда не становится ложным.

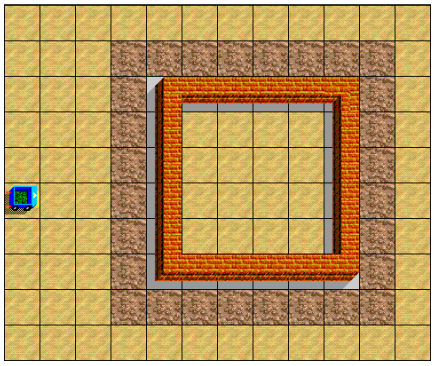
Пример 4. Эта программа приводит к зацикливанию, так как условие ***справа_стена*** выполняется всегда и Робот не меняет своего места.



```
Зацикливание
{
пока ( справа_стена )
    {
    кругом; кругом;
    }
}
```

Использование цикла ***пока*** позволяет нам решать задачи, в которых некоторые данные (например, длина стенок) заранее неизвестны.

Пример 5. Посадить цветы во всех клетках по периметру прямоугольной стены, считая, что расстояние до нее и ее размеры неизвестны.



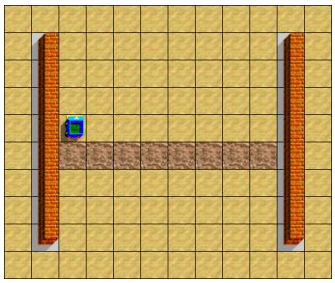
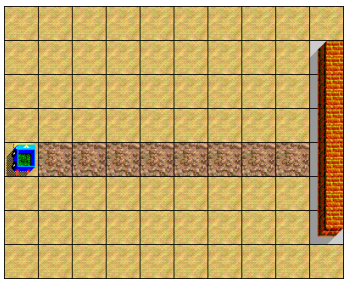
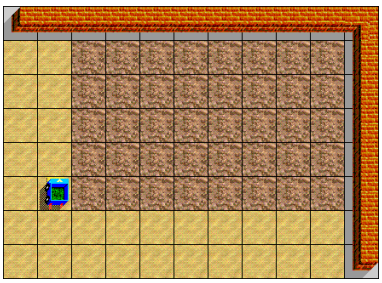
Для решения этой задачи надо использовать несколько циклов с условием. Сначала Роботу надо дойти до стенки, затем перейти к углу. Дальше он пойдет «держась за стенку», обходя таким образом прямоугольник и сажая цветы во всех нужных клетках.

Поскольку при обработке каждой из 4-х стенок Роботу надо выполнять одинаковые команды, здесь можно использовать цикл ***повтори (4)***. Тогда цикл ***пока*** становится вложенным циклом.

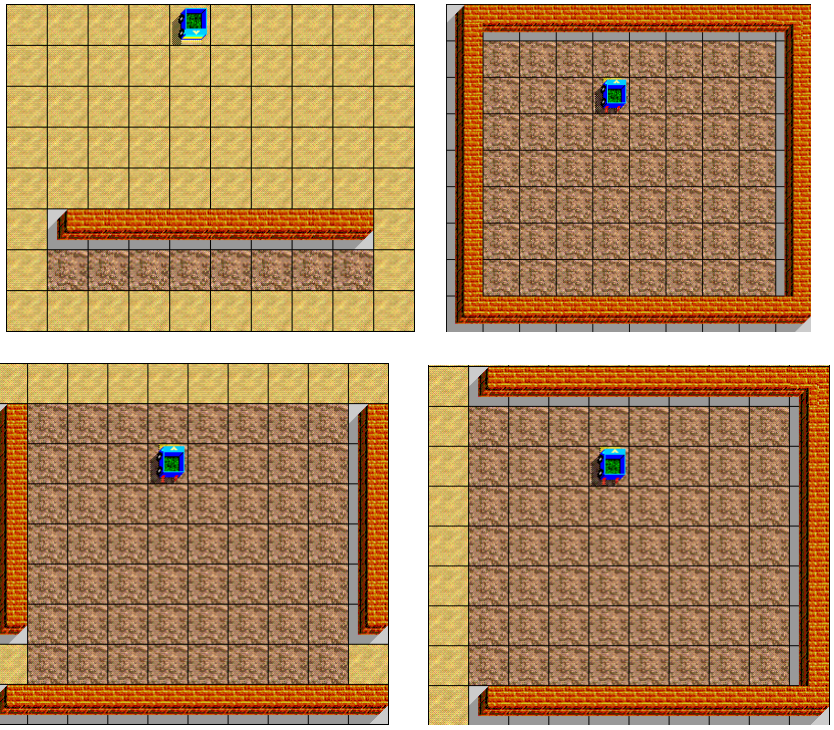
```
Контур
{
  пока ( впереди_свободно )
       вперед(1);      /* подойти к стене */
  налево;
  пока ( справа_стена )
       назад(1);       /* в левый нижний угол */
  повтори ( 4 )
    {
    вперед (1); /* теперь справа стена */
    пока ( справа_стена )
      {
      посади;
      вперед(1);
      }
    посади; /* угловая клетка */
    направо;
    }
}
```

Задачи

1. Посадить цветы во всех грядках клетках между стенками и вернуться обратно. Толщина стены – 1 клетка, остальные размеры считать неизвестными.

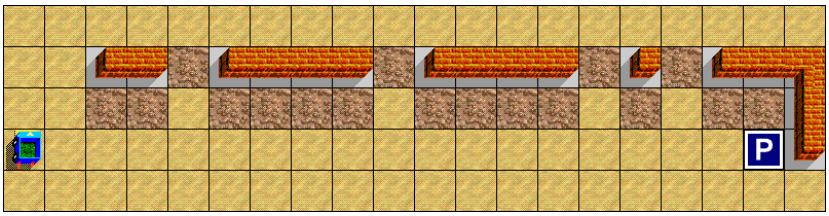
2. Посадить цветы во всех грядках. Толщина стены – 1 клетка, остальные размеры считать неизвестными. Все размеры считать неизвестными.



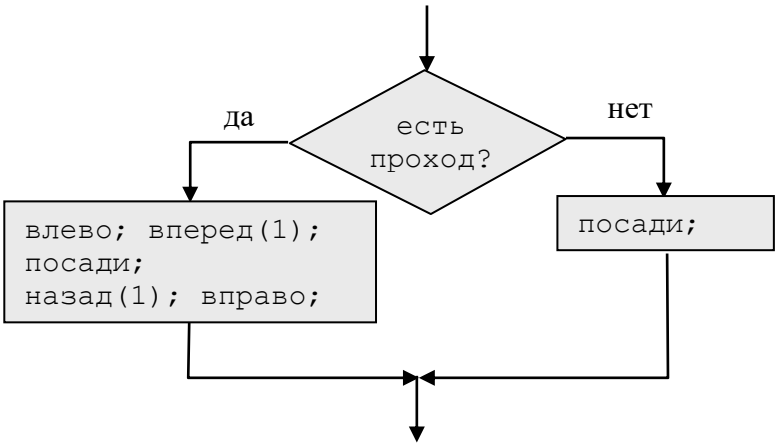
5. Условный оператор

Что такое условный оператор (задача z5-3.maz)?

Рассмотрим новую задачу для Робота. Надо посадить цветы во всех клетках вдоль стены, где нет прохода, а если в этом месте есть проход, войти в него и обработать грядку между стенок. Предполагаем, что длина стены и число проходов неизвестны.



Для решения этой задачи нам надо научить Робота выполнять разные действия в зависимости от окружающей обстановки. Это можно изобразить на схеме



Словами это можно сформулировать так: если есть проход (условие ***есть проход*** выполняется), то выполни одну группу команд, если нет – выполни другие команды. В программе для этой цели используется специальный *условный оператор* ***если***

```
Выбор
{
вперед(1); направо; вперед(1); /* подойти к началу стены */
пока ( впереди_свободно )
  {
  вперед(1);
  если ( слева_свободно )
    { /* войти в проход */
    налево; вперед(1);
    посади;
    назад(1); направо;
    }
  иначе
    { посади; }
  направо;
  вперед(1);
  }
}
```

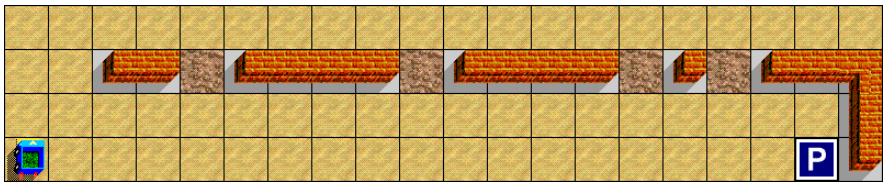
Таким образом, мы определили два варианта действий Робота - первый работает тогда, когда обнаружен проход, а второй – когда справа стена.

Правила использования условного оператора

1. Условный оператор состоит из двух частей; первая часть начинается ключевым словом ***если*** или ***if*** (от английского "*если*"), после которого в скобках записывается условие.
2. Если это условие верно (или *истинно*), то выполняется группа команд, стоящая ниже в фигурных скобках (*блок-если*).
3. Вторая часть (*блок-иначе*) начинается со слова ***иначе*** или ***else*** (от английского "*иначе*") и выполняется в том случае, когда условие в скобках *ложно.*
4. Нельзя отделять *блок-если* и *блок-иначе*, поскольку они составляют единый оператор.
5. Условие ставится только в заголовке *блока-если*.
6. *Блок-иначе* может отсутствовать, если он не нужен; в этом случае мы говорим, что условный оператор записан в сокращенной форме.
7. Чтобы было удобнее разбираться в программе, используют отступы так же, как и в циклах: тело *блока-если* и *блока-иначе* сдвигается вправо на 2-3 символа.

Сокращенная форма

Немного изменим задачу – пусть теперь Роботу надо обрабатывать только по 1 клетке в начале каждого прохода.

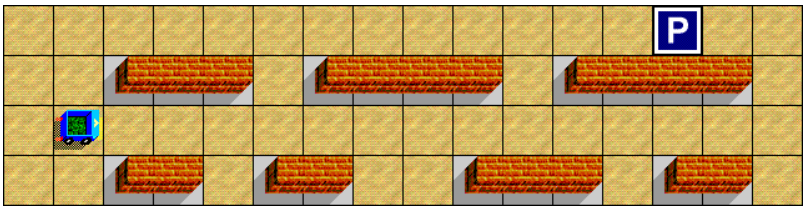


Таким образом, в *блоке-иначе* не осталось ни одной команды – если прохода нет, ничего делать не надо. Поэтому можно использовать сокращенную форму условного оператора – без второй части:

```
Сад2
{
вперед ( 1 ); направо; вперед ( 1 );
пока ( впереди_свободно )
  {
  вперед(1);
  если ( слева_свободно )
    {
    налево; вперед(1);
    посади;
    назад(1); направо;
    }
  }
}
```

Что такое сложные условия (задача z6-3.maz)?

Рассмотрим еще одну задачу для Робота. Ему нужно пройти через коридор с проходами и придти на Базу. Сложность состоит в том, что в обеих стенках есть проходы, сколько их – неизвестно.



Мы замечаем, что внутри коридора нет такой клетки, у которой слева и справа – свободные клетки. Значит, Роботу надо остановиться, когда слева и справа – свободно, это означает конец коридора. Теперь можно сформулировать алгоритм прохода через весь коридор на русском языке – иди вперед, пока слева стена **ИЛИ** справа стена.

В этом словесном алгоритмах мы объединяли логические команды Робота с помощью операции **ИЛИ,** получив из двух простых условий одно *сложное условие*. То же самое можно делать и в программе:

```
Посадка
{
вперед(1);
пока ( слева_стена или справа_стена )
  вперед (1);
налево;
вперед(2);
налево;
пока ( не база ) вперед ( 1 );
}
```

♦ **Сложное условие –** это условие, состоящее из простых условий и **логических операций**:

НЕ	отрицание
И	логическое умножение
ИЛИ	логическое сложение

Правила использования сложных условий

1. Простейшими условиями являются логические команды исполнителей (например, ***слева_стена***) и **логические отношения** между значениями[1].

> <	больше, меньше	***т > 5, 2+n < x***
>=	больше или равно	***a >= 2* x+5***
<=	меньше или равно	***c+2*d <= 5*v***
==	равно	***d == 2+c***
<>	не равно	***a != b***

[1] Разговор о числовых выражениях целесообразен только после изучения переменных.

2. В условии “равно” ставится два знака равенства; чтобы не запутаться, надо запомнить, что если переменная изменяется (*оператор присваивания*), то надо ставить один знак “=“, а если не меняется (*логическое отношение*), то два.
3. Сложные условия составляются из нескольких простых; простые условия объединяются с помощью **логических операций.**
4. Операция **"И"** требует *одновременного* выполнения двух условий, например:

   ```
   сверху_стена  И  снизу_стена
   ```

5. Операция **"ИЛИ"** обозначается требует выполнения *хотя бы одного* из двух условий (или обоих вместе), например:

   ```
   сверху_стена  ИЛИ  снизу_стена
   ```

6. Иногда удобно использовать логическую операцию **“НЕ”**, которая отрицает значение логического выражения, например условия

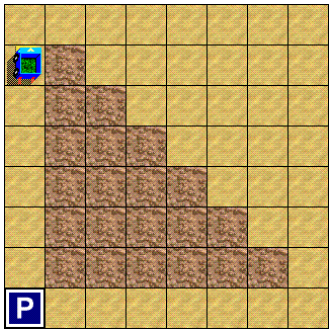
   ```
   a < b        и        НЕ (b >= a)
   ```

   означают одно и то же.
7. Устанавливается такой **приоритет** (старшинство) логических отношений и операций:
   - сначала выполняются операции в скобках, затем ...
   - операции **“НЕ”**, затем ...
   - логические отношения (>, <, >=, <=, ==, !=), затем ...
   - операции **“И”** и в последнюю очередь
   - операции **“ИЛИ”**.

   Для изменения порядка выполнения операций используются скобки.

6. Переменные и арифметические выражения

Зачем нужны переменные (задача z7-3.maz)?



Пример 1. Пусть Роботу надо посадить цветы на треугольной площадке. Для каждой «строки» можно использовать цикл ***повтори***. Если бы длины всех «строк» были равны, можно было бы использовать вложенный цикл.

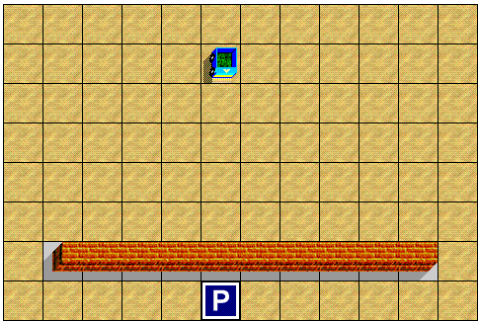
Конечно, можно пять раз написать цикл ***повтори***. Но чаще всего используют следующий подход.

1. Длина «строки» хранится в ячейке памяти.
2. В самом начале в эту ячейку записывается длина первой «строки».
3. В заголовке цикла ***повтори*** вместо числа подставляется значение, взятое из этой ячейки.
4. При переходе к следующей строке значение ячейки увеличивается на 1.

Посмотрим, как решается приведенная выше задача.

```
Ряд
{
 int n;       /* выделить место в памяти */
направо;
 n = 1;       /* присвоить значение переменной */
повтори ( 6 )   /* всего 6 строк */
  {
  повтори ( n ) /* длина строк меняется! */
    {
    вперед ( 1 );
    посади;
    }
  направо;
  вперед ( 1 );
  налево;
  назад ( n );
  n = n + 1;  /* увеличить переменную n на 1*/
  }
}
```

Подумайте, как можно решить эту задачу, используя цикл ***пока*** вместо ***повтори***?



Пример 2. Рассмотрим еще одну задачу: Роботу требуется обойти стенку и придти на Базу, которая находится точно под ним, но ... с другой стороны стенки, длина которой неизвестна. Логическую команду ***база*** использовать нельзя (отказал датчик).

С помощью цикла ***пока*** мы можем приказать ему дойти до стенки и обогнуть ее, но как остановить его именно в том месте, где нужно?

Чтобы решить эту задачу, надо в тот момент, когда Робот спустится вниз до стенки, каким-то образом запомнить, где он стоит, то есть запомнить расстояние от него до края стенки. Единственный способ сделать это – считать, *сколько шагов*

сделает Робот вдоль стенки до края, чтобы затем сделать столько же шагов обратно, уже под стенкой.

Запомнить эту информацию можно только *в ячейке памяти*. Чтобы не потерять ее, эту ячейку надо пометить, то есть дать ей *имя*. Содержимое этой ячейки нужно можно изменять во время выполнения программы, поэтому такая величина называется *переменная***.** Решение нашей задачи выглядит так:

```
ВнизСквозьСтену
{
 int n = 0;     /* объявление переменной   */
                /* выделение ячейки памяти */
                /* присвоение начального значения */
 пока ( впереди_свободно ) /* вниз до стенки */
      вперед (1);
 направо;
 пока ( слева_стена ) /* выйти влево за край стены */
      {
      вперед (1);
      n = n + 1;      /* увеличиваем счетчик шагов на 1 */
      }
 налево; вперед ( 2 ); налево;
 вперед ( n );       /* идем на нужное число шагов */
 }
```

Что такое переменная?

Запомнить информацию можно только в ячейке памяти компьютера. Чтобы не потерять ее, эту ячейку надо пометить, то есть дать ей *имя*. Содержимое этой ячейки можно изменять во время выполнения программы, поэтому такая величина называется *переменная***.**

♦ **Переменная** - это величина, которая имеет имя, тип и значение. Значение переменной может меняться во время выполнения программы. В компьютерах каждая переменная записана в свою ячейку памяти.

Объявление переменных

1. В начале процедуры все используемые переменные необходимо объявлять, при этом компьютер выделяет под них место в памяти и запоминает имена переменных. Если переменная не объявлена, то возникает ошибка "**НЕ ПОНИМАЮ**";
2. При объявлении переменных сначала указывается их тип, от этого зависит объем памяти, который выделяет компьютер. Пока мы будем рассматривать данные двух типов:

   int целые числа (сокращение от англ. *integer* - целый)

   float вещественные числа, которые могут иметь дробную часть
3. Справа от типа указывают имена переменных этого типа, списком через запятую, например:

   ```
   int       a, b, n1, mmm;
   float     c2d, fg, qwerty;
   ```
4. Имена переменных могут состоять из нескольких символов (букв или цифр), но начинаться они должны обязательно с буквы. Объявление переменных, так же как и любая другая команда, завершается точкой с запятой.

5. При объявлении мы может присвоить *начальные значения* некоторым переменным – после выделения памяти компьютер поместит эти числа в соответствующие ячейки, например:
```
int       d, b = 4, cbn, a = 6;
float     c, gh = 4.5, mmm = 7.89;
```
6. В информатике при записи вещественных чисел целая и дробная часть числа разделяется не запятой, а точкой, так, как это принято за рубежом.

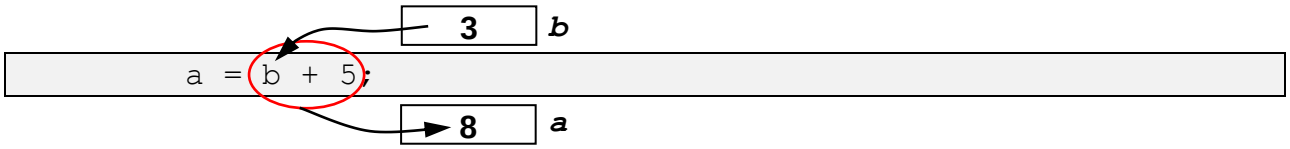
Правила работы с переменными

Для того, чтобы использовать переменные, надо уметь выполнять две основные операции
1. Считывать из памяти и использовать значение переменной.
2. Изменять значение переменной.

Как мы видели, для использования значения переменной достаточно указать ее имя, вместо которого будет автоматически подставлено значение этой переменной. Значение переменной изменяется с помощью специального *оператора присваивания*.

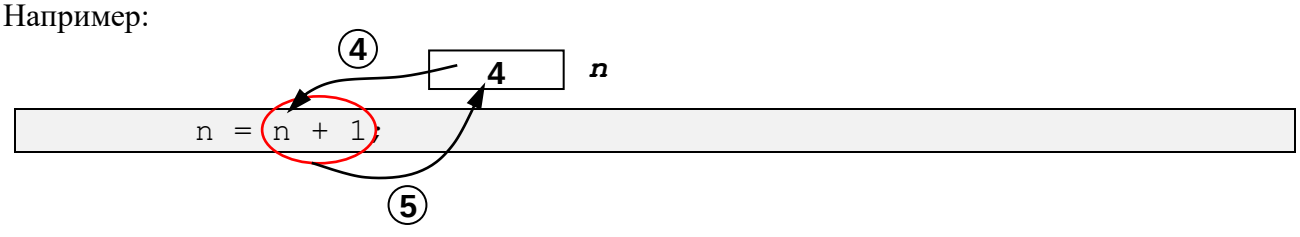
♦ Чтобы изменить значение переменной, надо использовать **оператор присваивания:** знак = показывает, что мы хотим изменить значение переменной, слева стоит имя переменной, которая изменяется, а справа - то, что мы хотим записать в эту ячейку, ее новое значение (при этом старое значение стирается!!!).

Например:
```
n = 5;
```
При этом в переменную **n** будет записано значение **5.** Справа от знака **=** в операторе присваивания может стоять какое-то арифметическое выражение, в котором участвуют другие переменные и числа, например:



Этот оператор присваивания приказывает компьютеру выполнить такие действия:
1. прочитать значение переменной **b** из памяти;
2. вычислить значение выражения **b+5**;
3. результат записать в ячейку **a**; при этом содержимое ячейки **b** не меняется, а старое содержимое ячейки **a** теряется безвозвратно.

В выражении справа можно использовать и имя той переменной, которой присваивается новое значение, в этом случае для вычислений используется **старое** значение этой переменной. Например:



Такой оператор присваивания приказывает компьютеру выполнить такие действия:
1. прочитать *старое* значение переменной **n** из памяти;
2. вычислить значение выражения **n+1**;
3. результат записать в ту же ячейку **n**; при этом фактически содержимое ячейки **n** увеличивается на единицу.

Понятно, что такой оператор нельзя рассматривать с точки зрения математики как уравнение относительно **n**, в информатике он имеет совсем другой смысл.

Арифметические выражения

Арифметическим выражением называют запись, которая содержит элементы четырех типов

- числа
- имена переменных
- знаки арифметических действий
- вызовы функций
- скобки для изменения порядка выполнения действий

Правила записи арифметических выражений[2]

1. В арифметическое выражение могут входить числа, знаки арифметических действий, имена переменных и вызовы функций.
2. Выражения должны быть записаны в виде линейной цепочки символов, индексы и степени не допускаются.
3. Для обозначения умножения используется знак **`*`**, деления **`/`**, возведения в степень **`^`.**
4. Знак операции умножения обязателен, например **`4*a`.**
5. Дробная и целая части числа отделяются **точкой**.
6. Устанавливается **приоритет** (старшинство) операций:
   - сначала выполняются операции **в скобках**, затем ...
   - вызовы **функций**
   - возведение в **степень**, затем ...
   - **умножение и деление** слева направо, затем ...
   - **сложение и вычитание** слева направо;
7. чтобы изменить порядок выполнения операций, используют скобки.

Запишем в машинном виде выражение

$$x = \frac{2a + 4d}{(c - 2d)(5 - 7c)^2} + \frac{5a}{4dc}$$

С учетом правил записи выражений результат будет такой:

```
x=(2*a+4*d)/((c-2*d)*(5-7*c)^2)+5*a/(4*d*c);
```

Некоторые стандартные функции уже заложены в память компьютера и для их использования надо только вызвать их по имени. Мы рассмотрим только две функции:

`abs ( x )`	вычисление модуля (абсолютного значения) числа **`x`**
`sqrt ( x )`	вычисление квадратного корня от **`x`**

Запишем в машинном виде формулу

$$x = \sqrt{\frac{a + 2b + 1}{(c - 3d)(2a - d)} + \left|\frac{15a^2 + 3b}{5c(b - a)}\right|}$$

С использованием стандартных функций это выражение запишется так

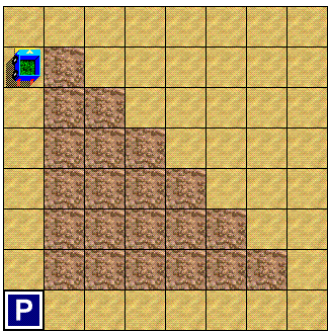
```
x = sqrt( (a+2*b+1)/((c-3*d)*(2*a-d)) +
          abs ((15*a^2+3*b)/(5*c*(b-a))) );
```

[2] Объем материала этого раздела варьируется сообразно возрасту. В 5-6 классах достаточно рассказать только 4 арифметических операции и скобки.

Цикл с параметром[3]

Зачем нам еще один вид цикла?

Рассмотрим разные способы решения следующей задачи. Роботу надо посадить цветы в области, имеющей форму прямоугольного треугольника заданного размера. Длина его катета заранее известна (на рисунке катет равен 6 клеткам). Сейчас мы знаем, как решать эту задачу с помощью цикла ***повтори***. Рассмотрим другой способ, в котором используется цикл ***пока***.



```
Ряд
{
int n;        /* выделить место в памяти */
направо;
n = 1;              /* присвоить значение переменной */
пока ( n <= 6 )     /* всего 6 строк */
  {
  повтори (  n  ) /* длина строк меняется! */
    {
    вперед ( 1 );
    посади;
    }
  направо;
  вперед ( 1 );
  налево;
  назад (  n  );
  n = n + 1;  /* увеличить переменную n на 1*/
  }
}
направо; вперед ( 1 ); налево;
назад ( n );
}
```

Как и ранее, мы должны использовать вспомогательную переменную ***n***, которая равна длине очередного ряда клеток. При этом велика вероятность того, что мы забудем присвоить ей начальное значение ***(n = 1;)*** или увеличивать ее в теле цикла ***(n = n+1;)***. Чтобы сосредоточить все операции с переменной ***n*** (она называется *параметром цикла*) в одном месте, используют третий вид цикла, который так и называется – ***цикл*** или ***for*** (от английского “*для*”), который позволяет заменить как цикл ***повтори***, так и цикл ***пока***.

```
Ряд
{
int n;
направо;
цикл ( n=1; n<=6; n=n+1 )
  {
  повтори (  n  )
    {
    вперед ( 1 );
    посади;
```

[3] Тема для дополнительного изучения, например, на кружке или факультативе.

```
        }
      направо;
      вперед ( 1 );
      налево;
      назад (  n  );
      }
    }
    направо; вперед ( 1 ); налево;
    назад ( n );
    }
```

Как видно из этой программы, все операции с переменной ***n*** теперь сгруппированы в заголовке цикла ***for*** между круглыми скобками. Три части отделяются знаком «***;***», так же как и конец команды:

1. **Начальное условие** ***n=1*** выполняется один раз перед началом цикла;
2. **Условие продолжения** ***n<=6*** говорит о том, при каком условии цикл будет выполняться (если в самом начале это условие неверно, то цикл не выполнится ни одного раза);
3. **Изменение переменной цикла** ***n=n+1*** - этот оператор выполняется каждый раз в конце очередного прохода тела цикла.

Задачи

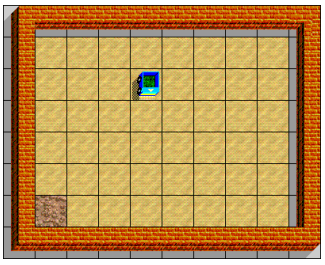
1. В начале значение переменной ***x*** равно ***3***. Как изменится ***x*** после выполнения программы:

```
x = 5; x = - x; x = x + 5;
```

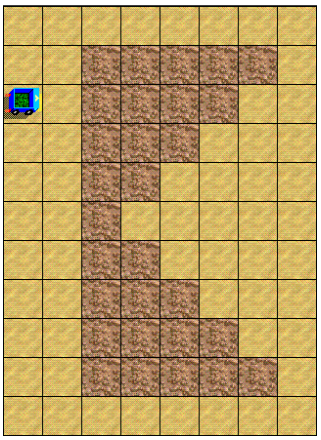
2. В начале значение переменной ***x*** равно ***3***, а значение ***y*** равно ***5***. Как изменятся ***x*** и ***y*** после выполнения программы:

```
y = 1; x = x + y; y = x; x = y;
```

3. Посадить цветы в левом нижнем углу и **вернуться обратно**. Размеры прямоугольника и начальное положение Робота считать неизвестными.



4. Решите задачу для Робота, используя цикл с параметром.



7. Диалоговые программы

Что такое диалоговая программа?

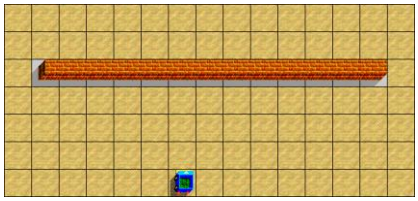
До сих пор мы составляли программу для исполнителя и затем смотрели, как он ее выполняет, не имея возможности вмешаться в его работу. Чтобы скорректировать его действия, надо было дождаться, пока он закончит работу по программе (или прервать ее выполнение), исправить программу и выполнить ее снова с самого начала.

В сложных программах требуется, чтобы человек, работающий с программой (он называется *пользователь*) смог во время работы программы ввести в нее необходимую информацию и получить ответ на экране монитора, то есть программа должна работать в диалоговом режиме.

- **Пользователь –** человек, использующий программу в своей работе.
- **Диалоговая программа** – программа, во время выполнения которой происходит диалог пользователя и программы с использованием операций ввода и вывода информации. Решения принимает человек, а программа выполняет всю рутинную работу.

Для организации диалога используются специальные операторы ввода и вывода. Вводить информацию можно с клавиатуры, мыши или джойстика, выводится информация на экран монитора или на принтер.

Вывод на экран (задача z8-3.maz)



Рассмотрим задачу, в которой Роботу надо определить и вывести на экран длину стенки (будем считать, что она заранее неизвестна).

Алгоритм решения очень прост:

1) дойти до стенки (цикл ***пока***);
2) дойти до края стенки (цикл ***пока***);
3) идти вдоль стенки, пока она не кончится, и на каждом шаге увеличивать переменную-счетчик (назовем ее ***n***).

Когда длина найдена, надо вывести ее на специальный экран:

```
вывод n;
```

После команды Робота ***вывод*** пишут имя переменной. Но при этом на экран будет выведено только значение переменной (число). Гораздо удобнее получить на экране сообщение вроде

Длина стенки 13 клеток.

Для этого надо дополнить команду вывода так:

```
вывод "Длина стенки ", n, " метров.";
```

То есть в команде ***вывод*** можно указать несколько элементов через запятую. Это могут быть строчки в кавычках (они выводятся на экран без изменения) и имена переменных (на экран выводится значения этих переменных). Вот полная программа:

```
НайтиДлину
{
int n = 0;
пока ( впереди_свободно ) вперед ( 1 );
налево;
пока ( справа_стена ) вперед ( 1 );
```

```
  назад ( 1 );
  пока ( справа_стена )
    {
    назад ( 1 );
    n = n + 1;
    }
  вывод "Длина стенки ", длина, " клеток.";
  }
```

Правила использования оператора вывода

1. Для вывода информации на экран монитора используется оператор ***вывод*** или ***print*** (от английского *print* – печать), после которого следует список элементов, разделенных запятыми.
2. В списке вывода можно использовать элементы трех видов:
   - **текст, заключенный в кавычки –** он выводится на экран без изменений;
   - **имя переменной**, значение которой надо вывести на экран;
   - **арифметическое выражение –** компьютер сначала вычислит его значение, а потом выведет результат на экран.

При использовании простейшей формы оператора вывода

```
  вывод n;
```

не совсем ясно, что же выводит на экран программа. Это считается плохим стилем и поэтому наша программе должна ясно написать, что же она подсчитала.

Ввод данных

Во многих программах надо задавать исходные данные для расчета, выбирать нужный режим работы, в общем, **вводить данные**. Для этого применяют специальную команду, которая называется **оператор ввода**. Возникает вопрос – куда Роботу (то есть его компьютеру) записать эти данные? Для этого надо объявить переменную в памяти и указать ее имя в команде ввода:

```
  ввод n;
```

Но при этом в момент ввода мы не будем знать, что же хочет компьютер и что он сделает с этим числом. Такой ввод считается признаком плохого стиля программирования. Поэтому перед именем переменной можно вставить подсказку – текстовое сообщение, которое будет выведено на экран перед тем, как компьютер будет ждать ввода данных.

Пример 1. Ввести с клавиатуры целое число и вывести на экран его квадрат.

```
 КвадратЧисла
 {
 int n, x;                      /* объявление переменных */
 вывод "Введите целое число";   /* ввод данных */
 ввод n;
 x = n*n;                       /* обработка */
 вывод "Квадрат числа ", n, " равен ", x; /* вывод */
 }
```

Обратите внимание, что в этой простейшей диалоговой программе мы выделили четыре части

1. объявление переменных;

2. ввод исходных данных;
3. обработка данных (вычисления);
4. вывод результатов.

Правила использования оператора ввода

1. Для ввода переменных с клавиатуры используется оператор **`ввод`** или **`input`** (от английского *input* – ввод), после которого следует список элементов, разделенных запятыми.
2. В списке ввода можно указать одно или несколько (через запятую) имен переменных, значение которых надо ввести с клавиатуры.
3. Переменные вводятся последовательно в порядке их перечисления в списке ввода. При вводе каждой переменной компьютер будет ждать, пока мы наберем нужное число и нажмем на клавишу ***Enter***, после этого введенное число будет записано в ячейку с указанным именем.

Задачи

1. Автомобиль движется без остановок с постоянной скоростью из Петербурга в Москву (расстояние 650 км). Составить программу, которая позволяет ввести скорость автомобиля и находит время в пути в часах (в минутах, в часах и минутах*).
2. У бабушки есть куры и утки, их всего 20 штук. Одна курица вести 3 кг, а одна утка – 10 кг. Составить программу, которая позволяет ввести количество кур и находит общий вес всех птиц.
3. Автомобиль сначала ехал по шоссе 2 часа на максимальной скорости. Затем он 3 часа ехал по лесной дороге, при этом его скорость уменьшилась в 4 раза. Составить программу, которая позволяет ввести максимальную скорость автомобиля и находит расстояние, которое проехал автомобиль, и его среднюю скорость.
4. Василий Пупкин выехал на машине в г. Мухинск. Через 3 часа у него кончился бензин и попутная машина за 2 часа притащила его в Мухинск со скоростью 20 км/ч. Составить программу, которая позволяет вести скорость автомобиля Василия в начале пути, и находит среднюю скорость, с которой Василий преодолел расстояние до Мухинска.
5. Пограничники обнаружили на расстоянии 5 км от берега судно-нарушитель морской границы, которое уходило в сторону моря со скоростью 20 км/ч. На перехват был выслан быстроходный катер. Через 4 минуты погони у катера сломался один из двигателей, и скорость упала до 30 км/ч. Составить программу, которая позволяет вести начальную скорость быстроходного катера и находит время, которое потребовалось ему для того, чтобы догнать нарушителя.
6. Крестьянин Василий Пупкин жил на рубеже XIX и XX веков. Известно, что он жил в XX веке на 53 года больше, чем в XIX. Составить программу, которая спрашивает, сколько всего лет прожил Василий Пупкин, и после этого находит год, в котором он родился?
7. Сторож обходит прямоугольный участок за 12 минут, его скорость – 5 км/ч. Длина одной из сторон участка известна. Составить программу, которая позволяет ввести длину этой стороны в метрах и вычисляет площадь участка.

Вычисления с циклами

Пример 1 **(задача Гаусса).** Найти сумму всех натуральных чисел от 1 до 100:

$$S = 1+2+3+\ldots+99+100.$$

По легенде эту задачу учитель дал классу, в котором учился великий немецкий математик К. Гаусс, рассчитывая, что ее решение займет достаточно много времени. Он очень удивился, когда Гаусс сразу же сказал ответ, подсчитав его устно (подумайте, как он это сделал).

Мы будем решать эту задачу «в лоб», выполнив суммирование с помощью компьютера без применения хитростей. Выделим в памяти ячейку-переменную *S* и в начале запишем в нее нуль.

```
int S;
S = 0;
```

S | **0**

Затем добавим к этой ячейке последовательно 1, 2, 3 и т.д.

```
S = S + 1;
S = S + 2;
S = S + 3;
...
S = S + 100;
```

S | **1**

S | **3**

S | **6**

S | **5050**

Когда мы выполним эту операцию 100 раз, в переменной *S* окажется сумма всех натуральных чисел от 1 до 100, что и требовалось. Теперь остается эти 100 сложений заменить циклом. Заметим, что в этих строчках отличается только последнее число, т.е. каждая строка имеет вид

```
S = S + i;
```

где величина *i* принимает значения 1, 2, 3 и т.д. до 100. Поэтому решение выглядит так

```
Гаусс
{
int i, S;
i = 1;                /* начальное число в сумме */
S = 0;                /* пока в сумме ничего нет */
повтори ( 100 )
  {
  S = S + i;          /* добавляем очередное число к сумме */
  i = i + 1;          /* переходим к следующему числу */
  }
вывод "S = ", S;
}
```

Пример 2. Рассчитать и вывести на экран таблицу значений функции

$$y = x^2 + 2x + 3$$

на интервале от **a** до **b** с шагом **h** (эти значения вводятся с клавиатуры). Для решения такой задачи удобно использовать цикл с параметром**.**

```
Таблица
{
float a, b, h, x, y;       /* объявление переменных */
вывод "Введите a, b, h:"; /* подсказка для ввода */
ввод  a, b, h;             /* ввод исходных данных */
for (x = a; x <= b; x = x + h ) /* вычисления и вывод */
```

```
        {
        y = x*x + 2*x + 3;
        вывод "x = ", x, " y = ", y;
        }
    }
```

Задачи

1. В земле выкопано N лунок, в каждую из которых кладут некоторое количество зерен:
   - на 1-ую клетку 1 зерно
   - на 2-ую клетку 2 зерна
   - на 3-ю клетку 4 зерна
   - на 4-ую клетку 8 зерен

   и т.д.,

   в каждую следующую – в 2 раза больше, чем в предыдущую. Программа должна подсчитать, сколько всего зерен на N клетках. Число N<30 нужно ввести с клавиатуры.
   Ответ: при N = 25 должно получиться 33554431 зёрен.
2. Найти сумму N чисел Фибоначчи (каждое следующее число равно сумме двух предыдущих)

   1, 2, 3, 5, 8, 13,…

   Число N < 40 нужно ввести с клавиатуры.
   Ответ: при N=30 сумма равна 3524576.
3. Вывести в три ровных столбика квадраты и кубы целых чисел от 5 до 25.
4. Вычислить сумму N чисел следующей последовательности

   1, 2, 4, 7, 11, 16,…

   Подсказка: разница между двумя соседними числами каждый раз увеличивается на 1.

   Ответ: при N = 10 сумма равна 175.
5. Клиент положил некоторую сумму на хранение в банк при ставке 5%. Составить программу, которая позволяет ввести начальную сумму и срок хранения, и находит сумму, которая накопится на счету к концу этого срока.

8. Процедуры

Зачем нужны процедуры?

Пока наши программы содержат не более 20-30 команд (операторов) и разобраться в них не составляет труда. В то же время профессиональные программы содержат тысячи и миллионы строк, и если бы они были написаны в виде одной длинной программы, их было бы практически невозможно понимать и редактировать. Даже книги всегда разбивают на главы и разделы.

Кроме того, в больших программах часто одни и те же действия (последовательности команд) должны выполняться несколько раз в разных местах программы. При этом приходится несколько раз переписывать (или копировать) их в текст. Если этот блок надо изменить, нужно вносить изменения в нескольких местах.

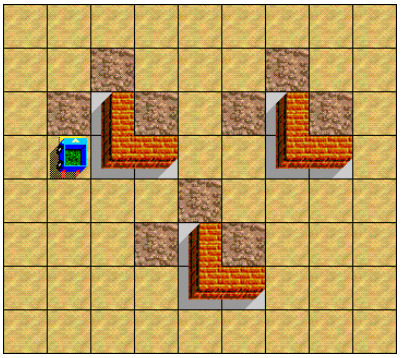
Таким образом, при работе с большими программами мы встречаемся с двумя проблемами:

1) программы получаются длинные и непонятные;
2) существуют целые группы команд, которые встречаются несколько раз в разных местах программы.

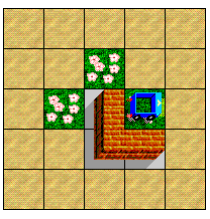
Выход из этой ситуации достаточно прост – надо разбить программу на несколько более мелких законченных задач (*подзадач*). Каждую из этих подзадач разбивают еще на более мелкие задачи так, чтобы каждая мелкая подзадача была записана в 20-50 строк программы. Для решения подзадач составляются вспомогательные алгоритмы, которые называются *процедурами*. Каждая процедура имеет свое имя, если написать в программе имя процедуры, то выполнятся все команды, входящие в процедуру.

Фактически при создании процедуры в список команд исполнителя *добавляется новая команда*. Чтобы исполнитель понял ее, надо эту команду расшифровать, то есть, объяснить, что делать при получении этой команды. Важно, чтобы в конечном счете все новые команды были расшифрованы через основные команды, входящие в СКИ исполнителя.

Как ввести новую команду (задача z10-3.maz)?



Пример 1. Рассмотрим задачу для Робота, показанную на рисунке. Попробуем решить ее, добавляя в СКИ исполнителя нужные нам команды. Сначала надо выделить те части задачи, которые одинаковы (или, по крайней мере, похожи). Легко заметить, что в нашей задаче есть три одинаковых площадки, каждая из которых состоит из угловой стенки и трех грядок.



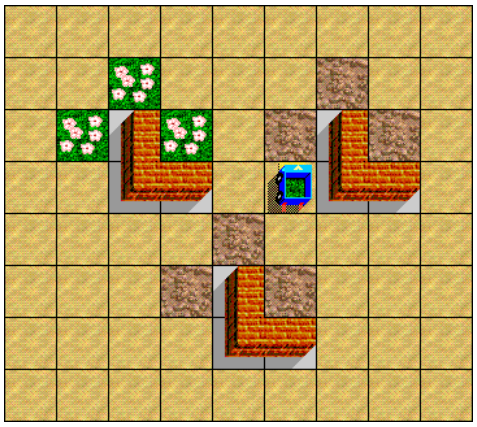
Введем новую команду ***Угол***. Будем считать, что Робот по этой команде обрабатывает одну такую площадку (см. рисунок слева). В конце работы машина Робота разворачивается на восток. Важно, что площадки совершенно одинаковые, поэтому каждую из них можно обрабатывать командой ***Угол***.

Основная программа выглядит так:

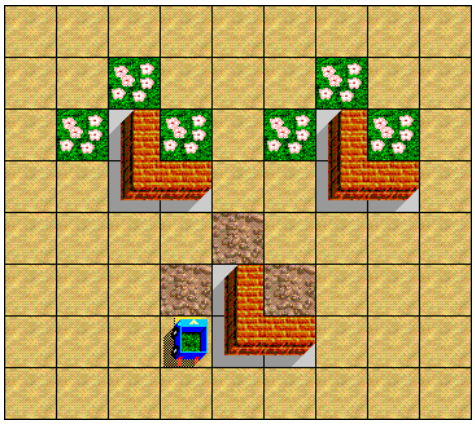
```
Программа
  {
 Угол;
  вперед(2);
  налево; назад(1);
 Угол;
  вперед(1);
  направо; вперед(2);
  направо; вперед(5);
  направо; назад(2);
 Угол;
  вперед(2);
  }
```

Обратите внимание на команды между вызовами процедуры. Они служат для того, чтобы перевести Робота в исходное положение для следующего вызова процедуры. Подумайте, что случится, если их убрать?

перед вторым вызовом процедуры:



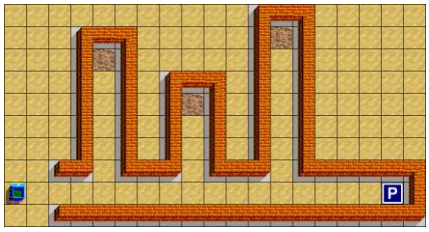
перед третьим вызовом процедуры:



Теперь остается только объяснить Роботу, что делать, когда он встретит в программе новую команду. После основной программы мы запишем расшифровку процедуры в виде отдельного алгоритма.

```
  Угол
  {
  вперед(1);посади;
  повтори(2)
     {
     вперед(1); направо;
     вперед(1);посади;
     }
  налево;
  }
```

Если бы мы не знали, что это процедура, невозможно было бы отличить ее от основной программы, поскольку обе оформляются одинаково. Первую процедуру исполнитель считает основной программой.



Пример 2. Рассмотрим еще одну задачу. Робот должен посадить цветы в конце каждого тупика (см. рисунок справа). Известно, что тупики есть только слева от Робота, справа – сплошная стена. Количество тупиков и их длины заранее неизвестны.

Идея алгоритма очень проста: Робот двигается вперед до стены, на каждом шаге проверяя, есть ли стена слева. Если ее нет, он обрабатывает тупик.

```
Поход
{
пока(впереди_свободно)
  {
  вперед(1);
  если (слева_свободно)
    Тупик;
  }
}
```

Теперь остается только записать расшифровку процедуры ***Тупик***:

```
Тупик
{
налево;
пока(впереди_свободно)
  вперед(1);
посади;
пока(сзади_свободно)
  назад(1);
направо;
}
```

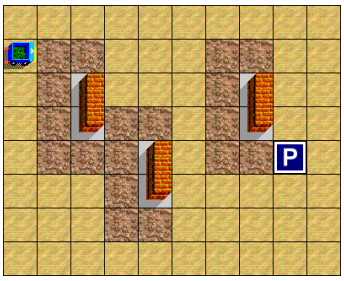
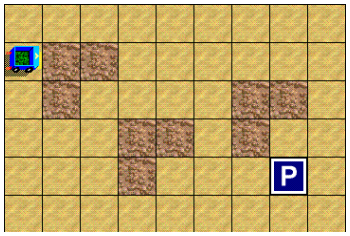
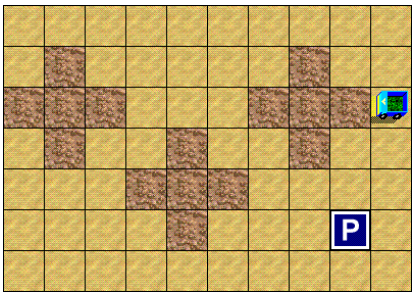
Важно, что после выполнения процедуры Робот стоит в том же положении, что и до ее вызова. Иначе основная программа была бы неправильной.

Правила использования процедур

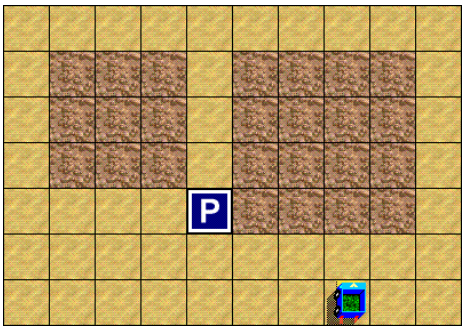
1. Процедуры – это вспомогательные алгоритмы.
2. Процедуры служат для разбиения программы на более мелкие блоки и для того, чтобы выделить подзадачи, встречающиеся в программе несколько раз.
3. Каждая процедура имеет имя и оформляется по тем же правилам, что и основная программа.
4. Каждая новая команда (процедура) должна быть расшифрована. Процедуры записываются одна за другой после основной программы.
5. Выполняется только основная программа. Процедура выполняется лишь тогда, когда она *вызывается*, то есть, ее имя встречается в основной программе или в другой процедуре и исполнитель выполняет эту строчку.
6. Когда процедура вызывается, выполняются все команды, входящие в процедуру и затем исполнитель переходит к следующей команде в вызывающей программе.
7. Одна процедура может вызывать другую. Количество процедур неограниченно.

Задачи

Составить программу для исполнителя Робот с использованием процедур:

Процедуры с параметрами (задача z11-3.maz)



Рассмотрим задачу для Робота, показанную на рисунке справа. Мы видим два похожих квадратных участка, которые надо обработать Роботу. Однако, размеры этих участков разные, поэтому невозможно два раза использовать одну процедуру.

Тем не менее, видно, что по структуре эти участки одинаковы. Было бы хорошо, если бы мы могли сказать процедуре, квадрат какого размера следует обработать Роботу, так же, как мы указываем ему, на сколько клеток двигаться вперед или назад (например, ***вперед(3)***).

Будем предполагать, что у Робота есть команда ***Квадрат***, по которой он обрабатывает квадратный участок заданного размера и возвращается в исходное положение. Тогда основная программа могла бы выглядеть так:

```
ДваКвадрата
{
направо; вперед(1); налево;
вперед(1);/* к правому нижнему углу большого квадрата */
Квадрат(4);      /* большой квадрат */
вперед(4);
налево;          /* к правому нижнему углу малого квадрата */
Квадрат(3);      /* малый квадрат */
}
```

Первые 4 строчки переводят Робота к правому нижнему углу большого квадрата – в исходное положение перед началом выполнений команды ***Квадрат*** (рисунок 1). Предположим, что по команде ***Квадрат(4)*** Робот обрабатывает большой квадрат и останавливается в таком положении, которое показано на рисунке 2.

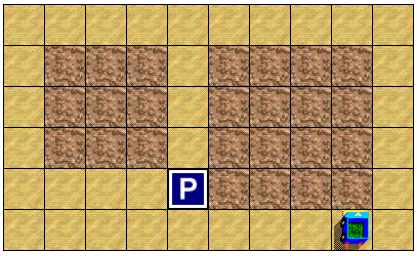


Рисунок 1.

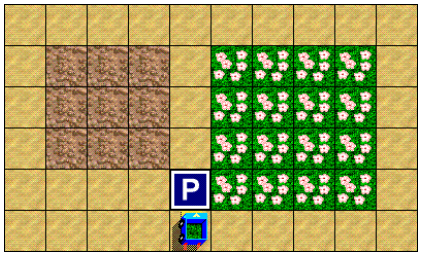


Рисунок 2.

Для обработки второго квадрата нужно вывести Робота в исходное положение (рисунок 3) и применить команду ***Квадрат(3)***, после чего Робот сразу оказывается на Базе (рисунок 4).

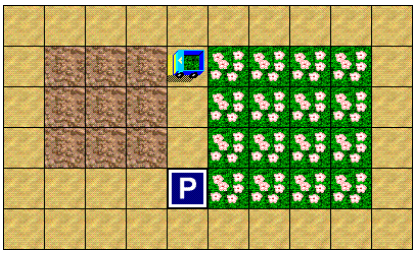


Рисунок 3.

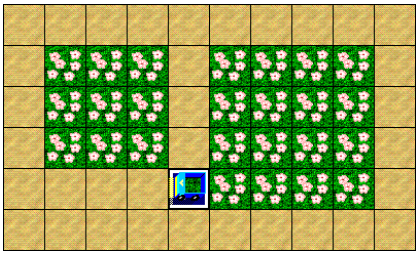


Рисунок 4.

Обработку малого квадрата можно было начинать и с любого другого угла, но тогда бы пришлось вести Робота на Базу дополнительными командами.

Теперь остается написать расшифровку процедуры ***Квадрат***, так чтобы она могла воспринимать и использовать целое число, заданное в скобках.

```
Квадрат ( int n )
 {
 повтори ( n )
   {
   повтори ( n )
     {
     вперед(1);
     посади;
     }
   налево;
   вперед(1);
   направо;
   назад ( n );
   }
 }
```

Как видим, в заголовке после имени процедуры в скобках указывается тип переменной величины (параметра) – в данном случае ***int*** (целое число) – и ее имя (здесь ***n***). При вызове процедуры число, указанное в скобках, записывается в переменную ***n*** и используется в трех выделенных строчках внутри процедуры. При первом вызове ***n*** принимает значение 4, при втором – 3.

Переменная ***n*** известна лишь процедуре ***Квадрат*** (ее называют *локальной*). Другие процедуры и основная программа могут иметь свои переменные с таким же именем, но они будут размещаться в других ячейках памяти.

В отличие от простых процедур, процедура с параметрами уже не является независимой, потому что она должна получить значения всех неизвестных параметров при вызове.

Правила использования процедур с параметрами

1. В заголовке процедуры после имени в скобках перечисляются все ее параметры (величины, которые изменяются).
2. Эти параметры имеют имена и называются *формальными* (поскольку неизвестны заранее и поэтому обозначены именами). Для каждого параметра указывается его тип (***int***, ***float*** и т.п.).

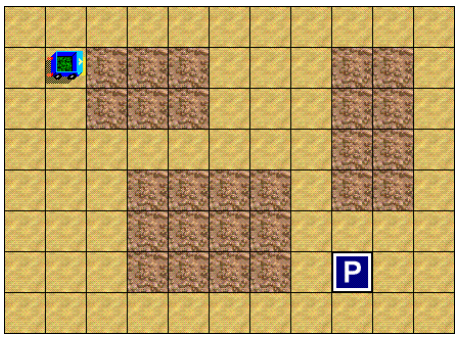
3. Если параметров несколько, они перечисляются через запятую. Например:

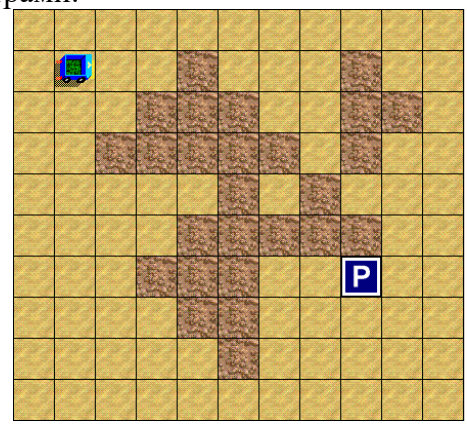
```
Группа ( int m, int n ) { ... }
```

4. При вызове процедуры после имени в скобках через запятую перечисляются значения ее параметров (числа или арифметические выражения, которые могут быть вычислены в момент вызова). Они называются *фактическими* параметрами и в процедуре подставляются вместо имен параметров.

Задачи

Решите задачи, используя процедуры с параметрами:





9. Методы составления программ

Метод “сверху вниз”

При использовании этого стиля можно выделить несколько этапов:

1. Сначала проектируется основная программа, состоящая из 20-30 команд. Часто на первом этапе составляется алгоритм на естественном (русском) языке. При этом можно вводить несуществующие команды – процедуры.
2. Затем для каждой новой процедуры пишется расшифровка так, чтобы она также состояла из 20-30 строк, при этом снова можно вводить несуществующие команды.
3. Эта процедура повторяется до тех пор, пока все процедуры не окажутся расшифрованными, то есть пока не останется ни одной неизвестной команды.

Такой способ часто называют **методом последовательного уточнения**.

Преимущества такого подхода:
- позволяет сначала рассмотреть задачу в целом, не обращая внимание на детали
- позволяет ограничить размер процедур так, чтобы их было легко понимать и отлаживать (находить и исправлять ошибки)
- позволяет легко разбить программу на части, которые выполняются разными разработчиками

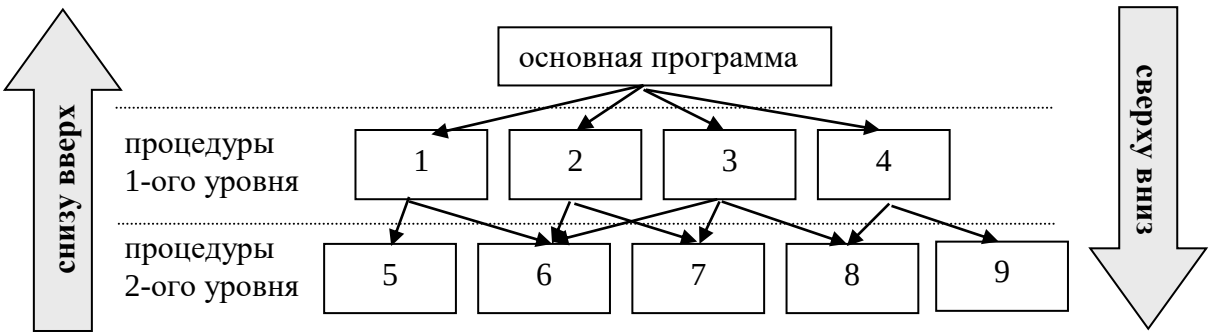
Недостатки:
- можно запутаться в большом количестве процедур, некоторые из которых могут выполнять похожие действия
- одинаковые или похожие операции могут быть по-разному реализованы в разных частях программы

Метод “снизу вверх”

При использовании этого метода сначала проектируются процедуры самого нижнего уровня, которые могут быть расшифрованы только через команды, входящие в СКИ исполнителя. После этого составляются процедуры следующего уровня, которые выполняют более сложные действия. В конце концов, образуется библиотека процедур, из которых программа собирается как из кубиков.

Преимущества такого подхода:
- программа составляется на основе единой библиотеки вспомогательных процедур

Недостатки:
- необходимо заранее продумывать набор необходимых процедур, в ходе работы он может меняться
- сложно разбить работу на части, выполняемые одновременно разными программистами
- сложно разрабатывать общую структуру программы, “стыковать” отдельные части
- процедуры могут получаться слишком длинными, это затрудняет отладку



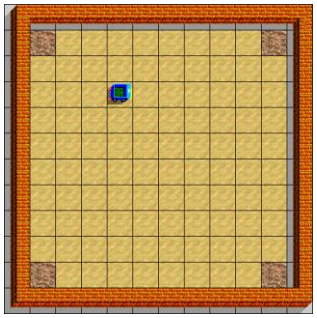
Комбинированный способ

В большинстве случаев используется комбинированный способ. Он заключается в следующем:

1. На первом этапе разрабатывается библиотека процедур самого нижнего уровня.
2. Дальнейшее проектирование идет методом “сверху вниз”.

Таким образом, удается совместить преимущества обоих способов.

Пример составления программы



Рассмотрим задачу для Робота, показанную на рисунке. Робот находится в замкнутом прямоугольнике, размеры его неизвестны. Надо посадить цветы во всех его углах. Очевидно, что надо начать с какого-то угла, например, с левого нижнего (на рисунке). Тогда основная программа может быть записана так:

```
Программа
{
ВУгол;
повтори(4)
  {
  Сторона;
  направо;
  }
}
```

Здесь мы предполагаем написать две процедуры – ***ВУгол*** и ***Сторона***. Первая должна привести Робота в левый нижний угол:

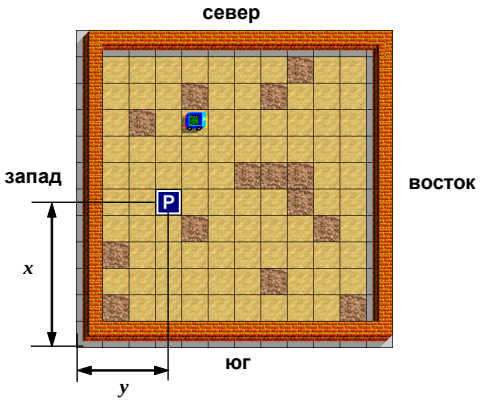
```
ВУгол
{
пока(сзади_свободно)
  назад(1);
налево;
пока(впереди_свободно)
  вперед(1);
направо;
}
```

Выполняя вторую, Робот обрабатывает одну сторону прямоугольника:

```
Сторона
{
посади;
пока(впереди_свободно)
  вперед(1);
}
```

Заметьте, что Робот остановится перед клумбой в левом нижнем углу, поскольку логическая команда ***впереди_свободно*** вернет ложное значение перед клумбой.

Сложная задача



Рассмотрим следующую довольно сложную задачу для Робота. Исполнитель находится где-то внутри замкнутого прямоугольника из стенок. Известно, что внутри нет стенок, где-то есть База, и есть несколько клеток, в которых надо посадить цветы. Составить программу для Робота так, чтобы он посадил цветы во всех клетках, где надо, и вернулся на Базу.

На примере этой задачи мы покажем, как правильно составить сложную программу. Сначала программа может состоять всего из одной строчки

```
Разведка
  {
  посадить цветы и прийти на Базу;
  }
```

Теперь надо расшифровать эту строчку через команды Робота. Удобнее всего сначала перевести Робота в один из углов прямоугольника и затем проверить каждый ряд, посадив цветы во всех клетках, где надо. При проверке надо также запомнить, где находится База, чтобы вернуться туда после прохода всех клеток.

```
Разведка
 {
 прийти в угол;
 проверить каждый ряд;
 прийти на Базу;
 }
```

Будем считать, что мы начинаем проверку из юго-западного (на рисунке – левого нижнего) угла. Затем расшифруем проверку и запишем ее с помощью цикла. Тогда получаем основную программу

```
Разведка
{
ВУгол;
пока ( справа_свободно )
   {
   ПроверитьРяд;
   направо; вперед(1); налево; /* к следующему ряду */
   }
ПроверитьРяд;
НаБазу;
}
```

Мы ввели три новых команды – ***ВУгол***, ***ПроверитьРяд*** и ***НаБазу***. Теперь остается их расшифровать. Заметьте, что строчка ***ПроверитьРяд*** после цикла нужна для того, чтобы проверить самый правый ряд клеток.

Проще всего написать процедуру ***ВУгол***:

```
ВУгол
{
пока ( сзади_свободно )
  назад(1);
налево;
пока ( впереди_свободно )
  вперед(1);
направо;
}
```

Как нам запомнить, где находится База? Простейший способ – записать ее координаты относительно какого-то из углов. Примем юго-западный угол за начало отсчета и будем запоминать координаты ***x*** и ***y*** (на рисунке). Они показывают, сколько шагов надо сделать из юго-западного угла на восток, а потом – на север, чтобы прийти на базу. Для этого введем специальные целые переменные ***x*** и ***y***. Будем считать, что если идти из юго-западного угла Тогда процедура ***НаБазу*** может быть записана так (предполагается, что Робот смотрит на север):

```
НаБазу
{
ВУгол;
направо; вперед(x);
налево;  вперед(y);
}
```

Процедуру проверки ряда ***ПроверитьРяд*** можно записать так

```
ПроверитьРяд
{
пока ( впереди_свободно )
  вперед ( 1 );
ПроверитьКлетку;
пока ( сзади_свободно )
  {
  назад ( 1 );
  ПроверитьКлетку;
  }
}
```

Как же найти координаты ***x*** и ***y***? Наиболее разумно делать это при проверке очередной клетки – если нашли Базу, сразу запомнили ее координаты. Таким образом, в процедуре ***ПроверитьКлетку*** надо сажать цветы, если это нужно, и запоминать положение Базы, если она найдена.

```
ПроверитьКлетку
{
если ( грядки ) посади;
если ( база )
  запомнить положение Базы;
}
```

Чтобы запомнить положение Базы, мы должны установить правильные значения ***x*** и ***y***. Таким образом, и процедура ***НаБазу***, и процедура ***ПроверитьКлетку*** должны иметь доступ к этим переменным. Однако если мы объявили их в процедуре ***НаБазу***, то только она может их использовать. Такие переменные называются *локальными* (от английского *local* – местный). Можно попробовать объявить их также и в процедуре ***ПроверитьКлетку***, но при этом в памяти будут созданы еще две *другие переменные*, к которым имеет доступ только процедура ***ПроверитьКлетку***. Для того, чтобы решить задачу, надо объявить *глобальные* переменные ***x*** и ***y*** выше основной программы.

♦ **Глобальные переменные** – это такие переменные, которые можно использовать во всех процедурах и в основной программе. Их надо объявлять **выше основной программы**.

Более того, нам надо в любой момент знать координаты Робота (сколько шагов он сделал от левого верхнего угла на восток и на север). Поскольку переход к следующему ряду выполняется в основной программе, надо ввести еще одну глобальную переменную ***xx***, которая показывает, сколько шагов сделал Робот на восток. Переход к следующей клетке в вертикальном ряду выполняется в процедуре ***ПроверитьРяд***, поэтому надо ввести глобальную переменную ***yy***, которая показывает, сколько шагов сделал Робот на север. Окончательный вариант основной программы дан ниже.

```
int x, y, xx, yy;
Разведка
{
ВУгол;
xx = 0;
пока ( справа_свободно )
  {
  ПроверитьРяд;
  направо; вперед(1); налево;
  xx = xx + 1;
  }
ПроверитьРяд;
НаБазу;
}
```

Теперь надо написать окончательные варианты процедур ***ПроверитьРяд*** и ***ПроверитьКлетку***. В процедуре ***ПроверитьРяд*** надо (с помощью переменной ***yy***) отслеживать, сколько шагов Робот сделал на север. Поскольку Робот проверяет клетки на обратном ходу (чтобы не ходить по цветам), мы увеличиваем счетчик ***yy***, когда Робот идет на север, а затем уменьшаем его при каждом шаге назад так, чтобы при вызове процедуры ***ПроверитьКлетку*** координаты ***xx*** и ***yy*** были правильными.

```
ПроверитьРяд
{
yy = 0;
пока ( впереди_свободно )
  {
  вперед ( 1 );
  yy = yy + 1;
  }
ПроверитьКлетку;
```

```
 пока ( сзади_свободно )
   {
   назад ( 1 );
   yy = yy - 1;
   ПроверитьКлетку;
   }
 }
```

В процедуре ***ПроверитьКлетку*** надо запоминать положение Базы, записав координаты Робота ***xx*** и ***yy*** в глобальные переменные ***x*** и ***y***, если она найдена.

```
ПроверитьКлетку
{
если ( грядка ) посади;
если ( база )
  {
  x = xx; y = yy; /* запомнить положение Базы */
  }
}
```

10. Исполнитель Черепаха

Как работает Черепаха?

Исполнитель **Черепаха** умеет делать рисунки и чертить на плоскости. Поскольку ей нужны все ее лапы, чтобы ходить, она держит перо в зубах.

Среда Черепахи – плоскость с системой координат. Система координат необходима для того, чтобы однозначно определять место Черепахи на плоскости. Черепаха редко использует прямоугольную систему координат, она поступает так же, как и человек – может развернуться в любую сторону и идти вперед или назад. Такая система координат ("вправо-влево-вперед-назад") называется *естественной системой координат*.

Какие команды понимает Черепаха?

СКИ Черепахи:

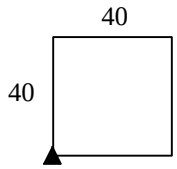
`покажись;`	Черепаха появляется на экране
`скройся;`	Черепаха исчезает
`опусти_перо;`	Черепаха оставляет за собой след
`подними_перо;`	Черепаха перемещается без следа
`в_точку ( x, y );`	переместиться в точку с координатами **(*x*,*y*)**
`вперед ( n );`	переместиться вперед на ***n*** шагов
`назад ( n );`	переместиться назад на ***n*** шагов
`влево ( a );`	развернуться влево на угол ***a*** градусов
`вправо ( a );`	развернуться вправо на угол ***a*** градусов

Как видно из этого списка команд, для Черепахи важно не только ее начальное положение на плоскости, но и ее направление. Мы будем считать, что в исходном положении Черепаха смотрит вверх ("на север").

Как управлять Черепахой?

Сначала выполним простейшую задачу для Черепахи — нарисуем квадрат со стороной 40 шагов. Черепаху будем обозначать черным треугольником. Как вы знаете, все углы квадрата равны 90 градусов, поэтому программа выглядит так:

```
Квадрат
{
покажись;
опусти_перо;
вперед ( 40 ); вправо ( 90 );
вперед ( 40 ); вправо ( 90 );
вперед ( 40 ); вправо ( 90 );
вперед ( 40 );
}
```



Как раскрасить рисунок?

Вы заметили, что Черепаха рисует все время черной линией. Используя специальные команды, рисунок можно раскрасить.

♦ Для изменения цвета линии используется команда

```
цвет ( n );          /* установить цвет линии n */
```

Цвет линии может иметь значения от 0 до 15, таким образом можно использовать всего **16 цветов**:

0	черный	**8**	темно-серый
1	синий	**9**	светло-синий
2	зеленый	**10**	светло-зеленый
3	голубой	**11**	светло-голубой
4	красный	**12**	светло-красный
5	фиолетовый	**13**	светло-фиолетовый
6	коричневый	**14**	желтый
7	серый	**15**	белый

Черепаха умеет также закрашивать замкнутую область заданным цветом.

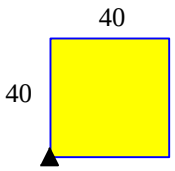
- Для закрашивания используется команда
  `закрась ( n );`
  где ***n*** — цвет краски.

При этом необходимо выполнение следующих условий:

1. Область должна быть замкнуты, то есть в границе не может быть разрывов, иначе краска "вытекает".
2. В момент закраски Черепаха должна находиться внутри этой области, перо должно быть опущено.
3. Черепаха не должна находиться в точке, которая имеет тот же цвет, что и граница.

Ниже показана программа, которая рисует желтый квадрат (номер цвета 14) с границей синего цвета (цвет 1).

```
Квадрат
{
покажись;
опусти_перо;
цвет ( 1 );
вперед ( 40 ); вправо ( 90 );
вперед ( 40 ); вправо ( 90 );
вперед ( 40 ); вправо ( 90 );
вперед ( 40 );
подними_перо;
вправо ( 135 ); вперед ( 5 );
опусти_перо;
закрась ( 14 );
}
```



Если вы не использовали команду ***`цвет`***, все линии рисуются черным цветом. Чтобы в самом начале залить экран каким-нибудь фоном, надо также использовать команду ***`закрась`***.

Окружности

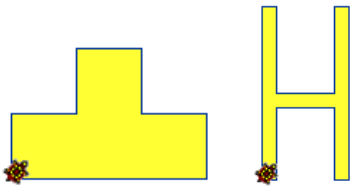
Черепаха умеет сама рисовать окружности. Для этого надо перевести ее в центр окружности и применить специальную команду.

- Для рисования окружности, центр которой находится в том месте, где стоит Черепаха, используют команду
  `окружность ( R );`

где **R** - радиус окружности

Цвет окружности определяется установленным цветом линий (то есть последней командой ***цвет***). Учтите, что Черепаха рисует окружность только тогда, когда ее перо опущено.

Задачи



Циклы

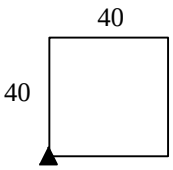
Как циклы сокращают программу

При составлении программы рисования квадрата вы заметили, что в ней несколько раз повторялась последовательность команд

```
вперед ( 40 );
вправо ( 90 );
```

Хотелось бы сказать исполнителю, чтобы он сделал эти команды ровно 4 раза. При этом будет нарисован квадрат и Черепаха вернется в исходное положение.

В данном случае эти команды надо повторить только 4 раза и можно легко 4 раза написать одинаковые команды. Но представьте, что надо сделать одинаковые операции 100 или 200 раз! В программировании в таких случаях используется специальная команда (оператор), которая говорит исполнителю, что какую-то часть программы надо сделать несколько раз.

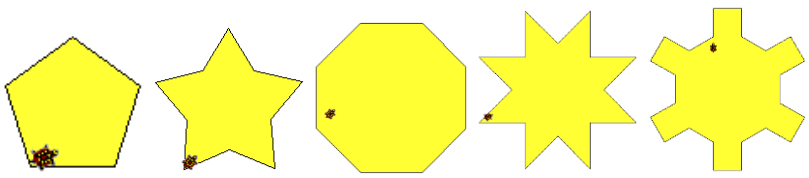


♦ **Цикл** — это многократное исполнение последовательности команд

Следующая программа рисует квадрат с использованием оператора цикла:

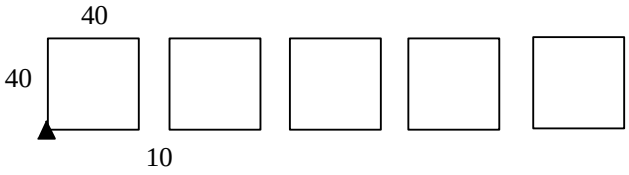
```
Квадрат
{
покажись;
опусти_перо;
повтори ( 4 )      /* заголовок цикла */
  {                /* начало цикла */
  вперед ( 40 );   /* тело цикла */
  вправо ( 90 );
  }                /* конец цикла */
}
```

Задачи



Вложенные циклы

Рассмотрим более сложную задачу, когда требуется нарисовать цепочку из пяти одинаковых квадратов, разделенных интервалом в 10 шагов:



Здесь явно напрашивается использование циклов, поскольку мы видим одинаковые элементы в рисунке, и хочется сказать исполнителю что-то вроде "Сделай 5 раз две операции:

- нарисуй квадрат и затем …
- переместись к левому нижнему углу следующего".

С другой стороны, сам квадрат рисуется с помощью цикла. Поэтому один цикл ***повтори*** будет расположен внутри другого.

♦ **Вложенный цикл** — это цикл, расположенный внутри другого цикла

Решение задачи выглядит так:

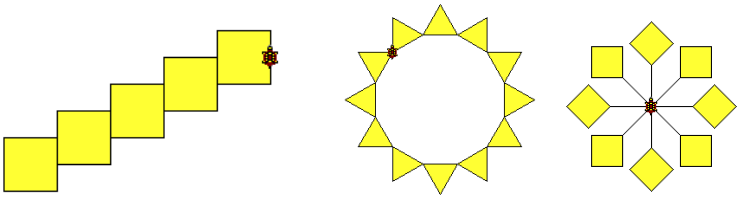
```
ПятьКвадратов
{
покажись;
повтори ( 5 )
  {
  опусти_перо;
  повтори ( 4 )
    {
    вперед ( 40 );
    вправо ( 90 );
    }
  вправо ( 90 );
  подними_перо;
  вперед ( 50 );
  влево ( 90 );
  }
}
```

внешний цикл

внутренний (вложенный) цикл

```
/* рисуем квадрат */

/* переходим к следующему */
```

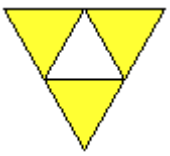
Задачи



Процедуры

Зачем нужны процедуры?

Иногда в задании явно есть одинаковые операции, но применить цикл не удается. Рассмотрим рисунок из трех равносторонних треугольников, сторона



которых равна 40 пикселей. С одной стороны, все треугольники одинаковые, с другой — они расположены так, что нельзя использовать один цикл для всех трех фигур.

В новом решении мы введем новую команду в СКИ исполнителя, назовем ее ***Треуг***. По этой команде Черепаха должна рисовать один равносторонний треугольник, расположенный так, как на рисунке слева, и закрашивать его желтым цветом. Сначала напишем программу, которая решает эту простую задачу.

```
Программа
  {
  влево ( 30 );
  опусти_перо;
  повтори ( 3 )
    {
    вперед ( 40 );
    вправо ( 120 );
    }
  вправо ( 30 );
  подними_перо;
  вперед ( 10 );
  опусти_перо;
  закрась ( 14 );
  подними_перо;
  назад ( 10 );
  }
```

Следующий важный шаг: мы вводим в СКИ новую команду ***Треуг***, которая включает все эти команды. Основная программа принимает такой вид:

```
Программа
  {
  Тр;
  }
```

основная программа

Однако если мы оставим ее в таком виде, то при запуске получим сообщение «Не понимаю, что такое 'Треуг'». И это правильно, потому что команды ***Треуг*** нет в СКИ исполнителя. Значит, надо «объяснить» Черепахе, что такое ***Треуг***. Для этого **после** основной программы включаем такой текст (расшифровку команды ***Треуг***)

```
Треуг
  {
  влево ( 30 );
  опусти_перо;
  повтори ( 3 )
    {
    вперед ( 40 );
    вправо ( 120 );
    }
  вправо ( 30 );
  подними_перо;
  вперед ( 10 );
  опусти_перо;
  закрась ( 14 );
  подними_перо;
  назад ( 10 );
```

расшифровка новой команды

```
  }
```

Запустим программу и убедимся, что она работает по-прежнему: рисует один треугольник с заливкой. Однако теперь в СКИ Черепахи добавлена новая команда, и мы можем ее использовать несколько раз. Такая команда называется **процедурой**.

Сначала рисуем первый треугольник, затем переводим исполнителя в вершину второго, и снова применяем команду ***Треуг***, и т.д. Вот что получается окончательно для фигуры из трех треугольников:

```
Программа
  {
  покажись;
  опусти_перо;
  Треуг;
  вправо ( 90 );
  вперед ( 40 );
  влево ( 90 );
  Треуг;
  влево ( 120 );
  Треуг;
  }
Треуг
  {
  влево ( 30 );
  опусти_перо;
  повтори ( 3 )
    {
    вперед ( 40 );
    вправо ( 120 );
    }
  вправо ( 30 );
  подними_перо;
  вперед ( 10 );
  опусти_перо;
  закрась ( 14 );
  подними_перо;
  назад ( 10 );
  }
```

вызов процедуры

вызов процедуры

вызов процедуры

Отдельно взятая основная программа работать не будет, однако если к ней добавить текст вспомогательного алгоритма ***Треуг***, то исполнитель сможет ее выполнить. Таким образом, программа состоит из двух частей – основной программы, которую выполняет исполнитель, и расшифровки новой команды (процедуры) ***Треуг***, которая стоит после основной программы.

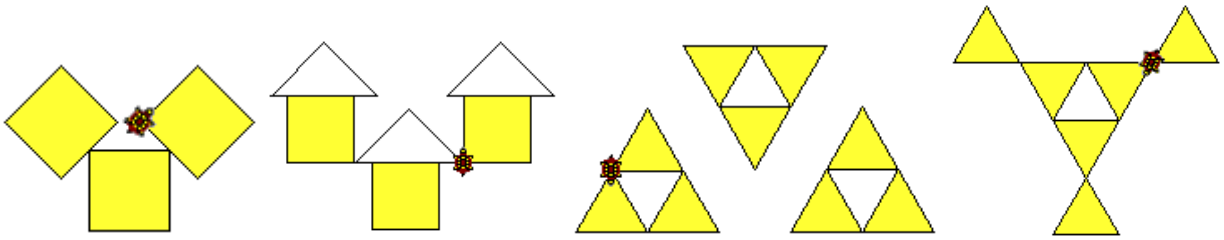
♦ **Процедуры (вспомогательные алгоритмы, подпрограммы)** – это новые команды, которые мы добавляем к СКИ исполнителя. Чтобы исполнитель знал, что делать по этой команде, после основной программы надо дать **расшифровку** процедуры через уже известные исполнителю команды.

Как правильно применять процедуры?

1. Процедуры используются в том случае, если исполнителю приходится несколько раз выполнять **одни и те же действия**. При этом удобно просто ввести новую команду и расшифровать ее один раз – это может значительно сократить текст программы.

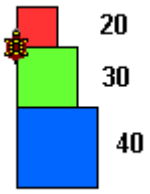
2. Основная программа всегда записывается первой. Она использует новые команды, расшифрованные после нее, при этом говорят, что **процедура вызывается.**
3. Если исполнитель встретил новую команду, которой нет в СКИ, он ищет ее расшифровку после основной программы, выполняет все действия, которые там указаны, и затем продолжает выполнять **основную программу** со следующей строчки. При этом говорят, что исполнитель **возвращается** в основную программу после выполнения процедуры.
4. В программе может быть несколько процедур. Одна процедура может вызывать другую процедуру.

Задачи



Процедуры с параметрами

Построим теперь с помощью Черепахи такую пирамидку, как на рисунке справа. Числа обозначают длины сторон соответствующих квадратов.



Сначала выделим **общее** всех трех фигур: все они — квадраты.

Затем выделим **отличия**: *длина стороны* и *цвет заливки*. Эти свойства называются **параметрами**.

Нам бы хотелось ввести новую команду ***Квадрат***, с помощью которой мы могли бы рисовать квадраты любого размера с любой заливкой. Мы уже умеем вводить новые команды-процедуры, но они всегда делают одно и то же. В СКИ Черепахи есть и другие команды, после которых в скобках указываются дополнительные данные (параметры), например:

```
влево ( 90 );
в_точку ( 30, 20 );
```

Теперь нам надо научиться вводить такие же команды – **процедуры с параметрами.**

Сначала составим программу, которая рисует *один* квадрат со стороной 40 и синей заливкой:

```
Программа
  {
  опусти_перо;
  повтори ( 4 )
    {
    вперед ( 40 );
    вправо ( 90 );
    }
  вправо ( 45 ); подними_перо;
  вперед ( 10 ); опусти_перо;
  закрась ( 1 ); подними_перо;
  назад ( 10 );
  влево ( 45 );
  }
```

Теперь, так же, как и раньше, сделаем из этой программы новую команду (процедуру) ***Квадрат***:

```
Программа
  {
  Квадрат;
  }
Квадрат
  {
  опусти_перо;
  повтори ( 4 )
    {
    вперед ( 40 );
    вправо ( 90 );
    }
  вправо ( 45 ); подними_перо;
  вперед ( 10 ); опусти_перо;
  закрась ( 1 ); подними_перо;
  назад ( 10 );
  влево ( 45 );
  }
```

Проблема состоит в том, что эта Черепаха рисует только квадраты со стороной 40 и синей заливкой. Теперь заменяем в процедуре все изменяемые величины (*параметры*) на буквенные обозначения: обозначим длину стороны через ***n*** и цвет через ***c***.

Программа не работает, поскольку неизвестно, что такое ***n*** и ***c***. Чтобы исправить ситуацию скажем, что эти величины перечисляются в скобках при вызове процедуры. Вот что должно получиться:

```
Программа
  {
  Квадрат ( 40, 1 );
  }
Квадрат ( int n, int c )
  {
  опусти_перо;
  повтори ( 4 )
    {
    вперед ( n );
    вправо ( 90 );
    }
  вправо ( 45 ); подними_перо;
  вперед ( 10 ); опусти_перо;
  закрась ( c ); подними_перо;
  назад ( 10 );
  влево ( 45 );
  }
```

вызов процедуры

заголовок процедуры

int - целое число

длина стороны

цвет заливки

В заголовке процедуры перед каждым именем параметра стоит слово ***int*** (от английского *integer*) – это означает, что данный параметр – целое число. У нас длина стороны и цвет – целые числа, поэтому перед каждым стоит слово ***int***.

Теперь, используя эту новую команду, можно дописать основную программу так, чтобы нарисовать все три квадрата:

```
Программа
  {
  покажись;
  Квадрат ( 40, 1 );
  вперед ( 40 );
  Квадрат ( 30, 10 );
  вперед ( 30 );
  Квадрат ( 20, 12 );
  }
```

сторона 40, цвет 1 (синий)

сторона 30, цвет 10 (светло-зелёный)

сторона 20, цвет 12 (светло-красный)

Не забудьте после основной программы написать расшифровку команды ***Квадрат***.

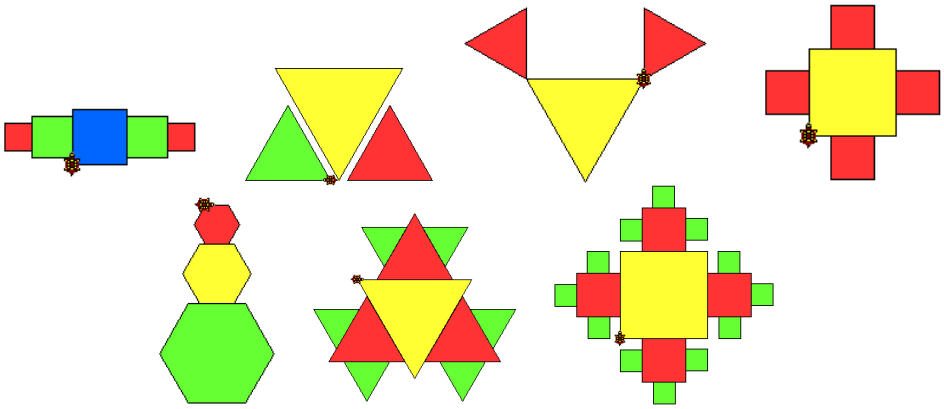
Как применять процедуры с параметрами?

1. Процедуры с параметрами используются для выполнения похожих, но не одинаковых действий.
2. Все изменяющиеся величины обозначаются символьными именами и называются **параметрами** процедуры.
3. Имена параметров могут состоять из нескольких символов (букв или цифр), но начинаются обязательно с буквы.
4. В заголовке процедуры в скобках через запятую перечисляются все параметры, для каждого из которых указывается его **имя** и **тип данных**:

   int целое число

   float число, которое может быть дробным
5. Внутри процедуры имена параметров подставляются вместо конкретных значений. Параметры в процедуре называются **формальными**, так как они обозначены именами и их значения неизвестны.
6. При вызове процедуры в скобках указываются **фактические параметры** – конкретные значения параметров, которые надо использовать в данный момент. При вызове надо указать ровно столько фактических параметров, сколько формальных параметров указано в расшифровке.
7. Во время выполнения процедуры фактические значения параметров подставляются вместо имен формальных параметров.
8. Важную роль играет порядок записи значений параметров — первым записывается значение того параметра, который указан первым в расшифровке процедуры, и т.д.

Задачи



Переменные

Зачем нужны переменные?

Построим на экране ряд столбиков увеличивающейся длины, как на рисунке справа. Высота первого столбика – 20 шагов Черепахи, каждый следующий – на 20 шагов выше, чем предыдущий. Такие столбики нередко используются для того, чтобы показать установленный уровень громкости (например, у телефона).

Начнем с более простой задачи, в которой все столбики имеют одинаковую высоту:

Несложно написать такую программу с использование цикла:

```
Программа
  {
  повтори ( 5 ) {
    опусти_перо;
    вперед ( 20 );
    назад ( 20 );
    подними_перо;
    вправо ( 90 );
    вперед ( 20 );
    влево ( 90 );
    }
  }
```

Числа, обведенные в рамку, обозначают длину столбика. Поэтому если мы хотим использовать цикл, это значение нужно как-то **менять**. Таким образом, эта величина становится **переменной**. Для хранения переменной в памяти выделяется отдельная ячейка, которой присваивается имя.

Выберем имя ячейки **n**. Тогда вместо

```
вперед ( 20 );
назад ( 20 );
```

мы напишем

```
вперед ( n );
назад ( n );
```

Откуда компьютер возьмет значение **n**? Во-первых, сначала оно равно 20, поэтому перед циклом нужно написать

```
n = 20;
```

Кроме того, после каждого шага цикла (когда мы переходим к следующей линии), нужно увеличивать значение **n** на 10. На языке программирования это запишется в виде

```
n = n + 10;
```

Такая запись – это не уравнение (в отличие от математики), а команда исполнителю: «возьми значение **n**, добавь к нему 10 и запиши результат в ячейку **n**».

Остается один шаг – нужно как-то сказать компьютеру, что бы будем использовать переменную с именем **n**. Такую команду называют **объявлением переменной**. В нашем случае объявление переменной выглядит так:

```
int n;
```

Слово **int** (от английского *integer* – целый) означает, что эта переменная предназначена только для хранения целых чисел. В нее можно записать числа 4 и 5, а вот 4,4 или 4,7 – нельзя. Вот полная программа:

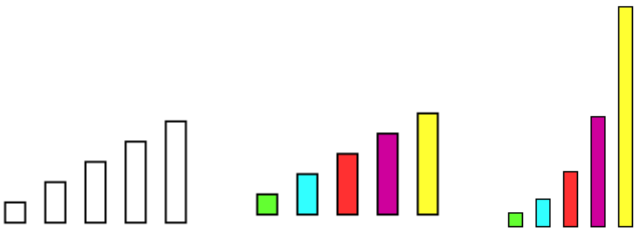
```
Программа
  {
  int n;
  n = 20;
  повтори ( 5 ) {
    опусти_перо;
    вперед ( n );
    назад ( n );
    подними_перо;
    вправо ( 90 );
    вперед ( 20 );
    влево ( 90 );
    n = n + 10;
    }
  }
```

выделить место в памяти для ячейки с именем **n**

начальное значение для переменной **n**

увеличить значение переменной **n** на 10

Задания



Что такое переменная?

Запомнить информацию можно только в ячейке памяти компьютера. Эту ячейку надо пометить, то есть дать ей *имя*. Содержимое ячейки можно изменять во время выполнения программы, поэтому такая величина называется *переменная*.

♦ **Переменная** – это величина, которая имеет имя, тип и значение. Значение переменной может меняться во время выполнения программы. В компьютерах каждая переменная записана в свою ячейку памяти.

Объявление переменных

1. В начале процедуры все используемые переменные необходимо объявлять, при этом компьютер выделяет под них место в памяти и запоминает имена переменных. Если переменная не объявлена, то возникает ошибка "**НЕ ПОНИМАЮ**";
2. При объявлении переменных сначала указывается их тип, от этого зависит объем памяти, который выделяет компьютер. Пока мы будем рассматривать данные двух типов:

int	целые числа (сокращение от англ. *integer* - целый)
float	вещественные числа, которые могут иметь дробную часть

3. Справа от типа указывают имена переменных этого типа, списком через запятую, например:

```
int     a, b, n1, mmm;
float   c2d, fg, qwerty;
```

4. Имена переменных могут состоять из нескольких символов (букв или цифр), но начинаться они должны обязательно с буквы. Объявление переменных, так же как и любая другая команда, завершается точкой с запятой.
5. При объявлении мы может присвоить *начальные значения* некоторым переменным – после выделения памяти компьютер поместит эти числа в соответствующие ячейки, например:
```
int       d, b = 4, cbn, a = 6;
float     c, gh = 4.5, mmm = 7.89;
```
6. В информатике при записи вещественных чисел целая и дробная часть числа разделяется не запятой, а точкой, так, как это принято за рубежом.

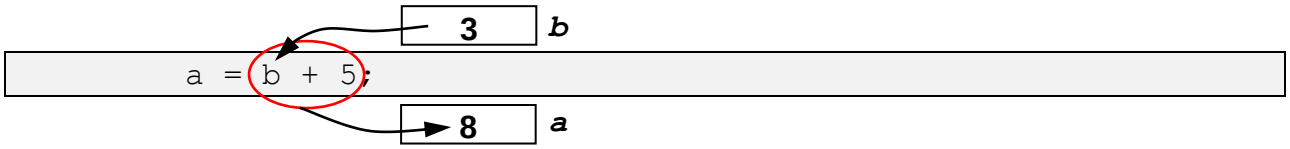
Правила работы с переменными

Для того, чтобы использовать переменные, надо уметь выполнять две основные операции
1. Считывать из памяти и использовать значение переменной.
2. Изменять значение переменной.

Как мы видели, для использования значения переменной достаточно указать ее имя, вместо которого будет автоматически подставлено значение этой переменной. Значение переменной изменяется с помощью специального *оператора присваивания*.

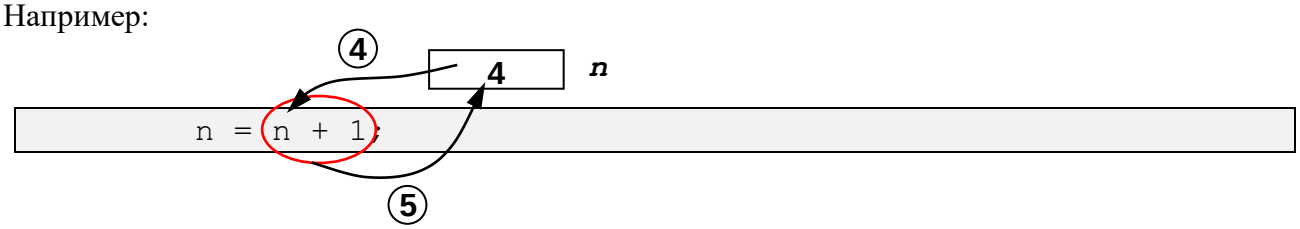
♦ Чтобы изменить значение переменной, надо использовать **оператор присваивания:** знак = показывает, что мы хотим изменить значение переменной, слева стоит имя переменной, которая изменяется, а справа - то, что мы хотим записать в эту ячейку, ее новое значение (при этом старое значение стирается!!!).

Например:
```
n = 5;
```
При этом в переменную **n** будет записано значение **5.** Справа от знака **=** в операторе присваивания может стоять какое-то арифметическое выражение, в котором участвуют другие переменные и числа, например:



Этот оператор присваивания приказывает компьютеру выполнить такие действия:
1. прочитать значение переменной **b** из памяти;
2. вычислить значение выражения **b+5**;
3. результат записать в ячейку **a**; при этом содержимое ячейки **b** не меняется, а старое содержимое ячейки **a** теряется безвозвратно.

В выражении справа можно использовать и имя той переменной, которой присваивается новое значение, в этом случае для вычислений используется **старое** значение этой переменной. Например:



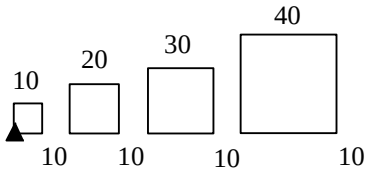
Такой оператор присваивания приказывает компьютеру выполнить такие действия:
1. прочитать *старое* значение переменной **n** из памяти;
2. вычислить значение выражения **n+1**;

3. результат записать в ту же ячейку ***n***; при этом фактически содержимое ячейки ***n*** увеличивается на единицу.

Понятно, что такой оператор нельзя рассматривать с точки зрения математики как уравнение относительно ***n***, в информатике он имеет совсем другой смысл.

Переменные и процедуры

Построим ряд из четырех квадратов со сторонами 10, 20, 20 и 40, расположенные так, как на рисунке.



Конечно, можно написать процедуру с параметром и вызвать ее 4 раза:

```
Программа
  {
  Квадрат ( 10 );
  вправо ( 90 ); вперед ( 20 );
  влево ( 90 );
  Квадрат ( 20 );
  вправо ( 90 ); вперед ( 30 );
  влево ( 90 );
  Квадрат ( 30 );
  вправо ( 90 ); вперед ( 40 );
  влево ( 90 );
  Квадрат ( 40 );
  }
Квадрат ( int a )
  {
  опусти_перо;
  повтори ( 4 )
    {
    вперед ( a );
    вправо ( 90 );
    }
  подними_перо;
  }
```

Процедура Квадрат рисует квадрат, сторона которого задана в скобках, и возвращает Черепаху в исходное положение.

Заметим, что если бы размеры квадратов были одинаковы, можно было бы использовать цикл, поскольку они расположены «в одну линию». Проблема в том, что сторона квадрата меняется для каждого шага этого цикла. Кроме того, меняется расстояние, на которое надо сдвинуться Черепахе по команде ***вперед*** для того, чтобы перейти к углу следующего квадрата.

Здесь также можно применить цикл, но с некоторыми хитростями:

1) длина стороны квадрата хранится в переменной, назовем ее ***n***;
2) в самом начале в переменную ***n*** записывается длина стороны первого квадрата (10);
3) при вызове процедуры в качестве фактического параметра подставляется значение, взятое из переменой ***n***;

4) когда нарисован очередной квадрат, для перехода к следующему нужно сдвинуть Черепаху на расстояние ***n+10***;

5) при переходе к следующему квадрату значение переменной ***n*** увеличивается на 10.

Основная программа принимает такой вид:

```
Программа
  {
  int n = 10;
  повтори ( 4 )
    {
    Квадрат ( n );
    вправо ( 90 );
    вперед ( n+10 );
    влево ( 90 );
    n = n + 10;
    }
  }
```

Не забудьте, что после основной программы надо привести также расшифровку процедуры ***Квадрат***, такую же, как и раньше. Обратите внимание, что длина стороны по разному обозначается в основной программе (переменная ***n***) и в процедуре (параметр ***a***).

Цикл с параметром

Предыдущую задачу с четырьмя квадратами можно решить и иначе, с помощью цикла ***пока***:

```
Программа
  {
  int n;
  n = 10;
  пока ( n<=40 )
    {
    Квадрат ( n );
    вправо ( 90 );
    вперед ( n+10 );
    влево ( 90 );
    n = n + 10;
    }
  }
```

Недостаток этого способов состоит в том, что действия с переменной ***n*** выполняются в трех разных строчках программы:

1) присвоение начального значения (***n=10***);
2) условие выполнения цикла (***n<=40***);
3) изменение переменной в конце каждого шага цикла (***n=n+10***).

При этом велика вероятность того, что мы забудем присвоить ей начальное значение или изменить его. Чтобы сосредоточить все операции с переменной ***n*** (она называется *параметром цикла*) в одном месте, используют третий вид цикла, который так и называется – ***цикл*** или ***for*** (от английского “*для*”), который позволяет заменить как цикл ***повтори***, так и цикл ***пока***:

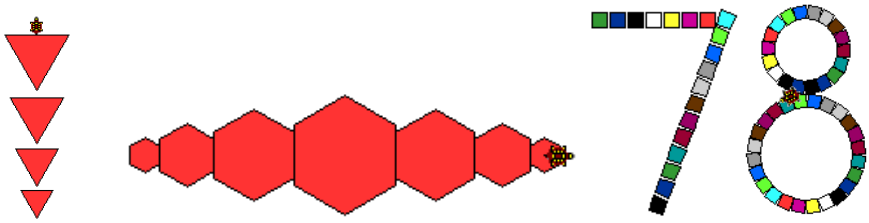
```
Программа
```

```
{
int n;
 цикл( n=10; n<=40; n=n+10 )
  {
  Квадрат ( n );
  вправо ( 90 );
  вперед ( n+10 );
  влево ( 90 );
  }
}
```

Как видно из этой программы, все операции с переменной **n** теперь сгруппированы в заголовке цикла между круглыми скобками. Три части отделяются знаком «**;**», так же как и конец команды:

1. **Начальное условие** `n=10` выполняется один раз перед началом цикла;
2. **Условие продолжения** `n<=40` говорит о том, при каком условии цикл будет выполняться (если в самом начале это условие неверно, то цикл не выполнится ни одного раза);
3. **Изменение переменной цикла** `n=n+10` - этот оператор выполняется каждый раз в конце очередного прохода тела цикла.

Задачи



Арифметические выражения

Арифметическим выражением называют запись, которая содержит элементы четырех типов

- числа
- имена переменных
- знаки арифметических действий
- вызовы функций
- скобки для изменения порядка выполнения действий

1. Выражения должны быть записаны в виде линейной цепочки символов, индексы и степени не допускаются.
2. Для обозначения умножения используется знак *****, деления **/**, возведения в степень **^.**
3. Знак операции умножения обязателен, например **4*a.**
4. Дробная и целая части числа отделяются **точкой**.
5. Устанавливается **приоритет** (старшинство) операций:
   - сначала выполняются операции **в скобках**, затем ...
   - вызовы **функций**
   - возведение в **степень**, затем ...
   - **умножение и деление** слева направо, затем ...
   - **сложение и вычитание** слева направо;
6. чтобы изменить порядок выполнения операций, используют скобки.

Запишем в машинном виде выражение

$$x = \frac{2a + 4d}{(c - 2d)(5 - 7c)^2} + \frac{5a}{4dc}$$

С учетом правил записи выражений результат будет такой:

```
x=(2*a+4*d)/((c-2*d)*(5-7*c)^2)+5*a/(4*d*c);
```

Некоторые стандартные функции уже заложены в память компьютера и для их использования надо только вызвать их по имени. Мы рассмотрим только две функции:

abs (x)	вычисление модуля (абсолютного значения) числа ***x***
sqrt (x)	вычисление квадратного корня от ***x***

Запишем в машинном виде формулу

$$x = \sqrt{\frac{a + 2b + 1}{(c - 3d)(2a - d)} + \left|\frac{15a^2 + 3b}{5c(b - a)}\right|}$$

С использованием стандартных функций это выражение запишется так

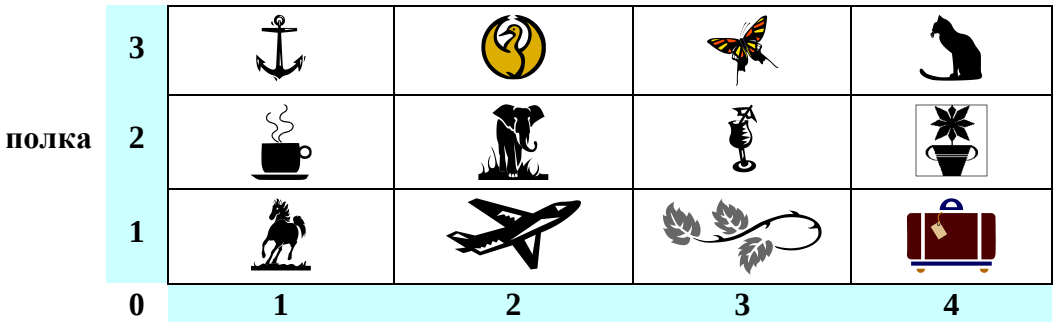
```
x = sqrt( (a+2*b+1)/((c-3*d)*(2*a-d)) +
          abs ((15*a^2+3*b)/(5*c*(b-a))) );
```

11. Исполнитель Чертежник

Еще один исполнитель, с которым мы будем работать - **Чертежник.** Он умеет рисовать на плоскости, используя прямоугольную (декартову) систему координат.

Прямоугольная система координат

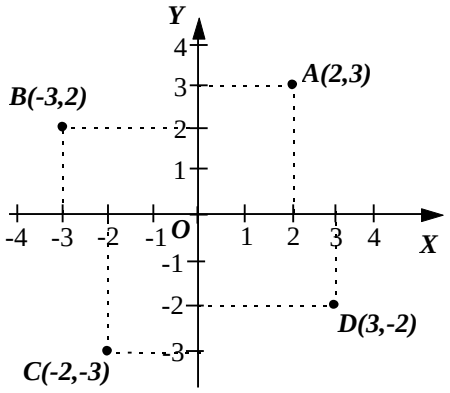
Представим себе, что в магазине на полках стоят товары, и вы хотите сказать продавщице, где находится тот товар, который вы хотите посмотреть.



Будем определять положение любого объекта двумя цифрами, первая из которых — стеллаж, а вторая — полка. Например, слон имеет координаты **(2,2)**, а кошка — **(4,3)**. Отсчет ведется от левого нижнего угла. Эта точка называется **началом координат**.

Теперь вспомним математику. На плоскости выбирается точка ***O***, которая принимается за начало координат. Через нее проводятся две оси: горизонтальная ось **OX** вправо и вертикальная ось **OY** вверх. На обеих осях наносят разметку с выбранным шагом.

Координатами любой точки считаются два числа: расстояние до оси **OY** (координата ***x***) и расстояние до оси **OX** (координата ***y***). Считается, что точка **O** имеет координаты ***(0,0)***. Точки, расположенные справа от оси **OY**, имеют положительные координаты ***x***, слева - отрицательные.



На рисунке показано, как определяются координаты точек на плоскости.

Как управлять Чертежником?

Среда Чертежника – плоскость с системой координат, которая необходима для того, чтобы однозначно определять место точки. Для задания системы координат надо определить

- направление осей координат

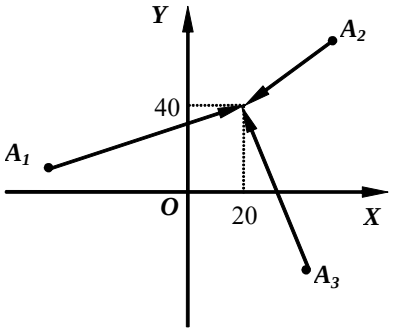
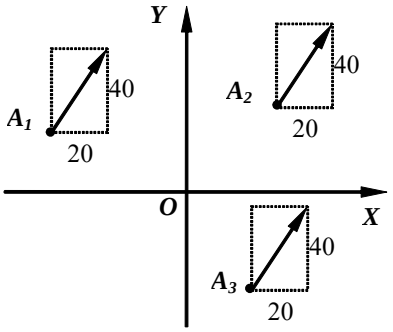
- единичные отрезки на осях
- начало отсчета – точку с координатами (0,0)

Оси координат не рисуются на экране — вы видите чистое белое поле. Началом координат считается центр поля исполнителя.

♦ **СКИ Чертежника:**

`покажись;`	Чертежник появляется на экране
`скройся;`	Чертежник исчезает
`опусти_перо;`	после этого остается след
`подними_перо;`	не оставлять след
`в_точку ( x, y );`	переместиться в точку с координатами ***(x,y)***
`вектор ( dx, dy );`	переместиться на вектор ***(dx,dy)*** относительно текущего положения

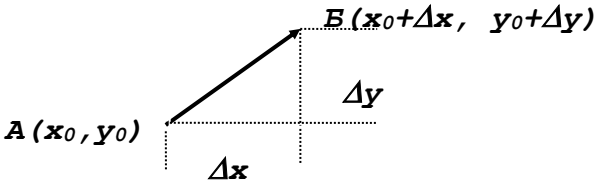
Покажем на примере разницу между командами ***в_точку*** и ***вектор***.

Команда ***в_точку(20,40)*** перемещает исполнителя в точку с **абсолютными координатами** ***(20,40)***, независимо от того, где он находился до этого.	Команда ***вектор(20,40)*** смещает исполнителя в точку, расположенную на 20 шагов правее и на 40 шагов выше его **текущего положения**.
	

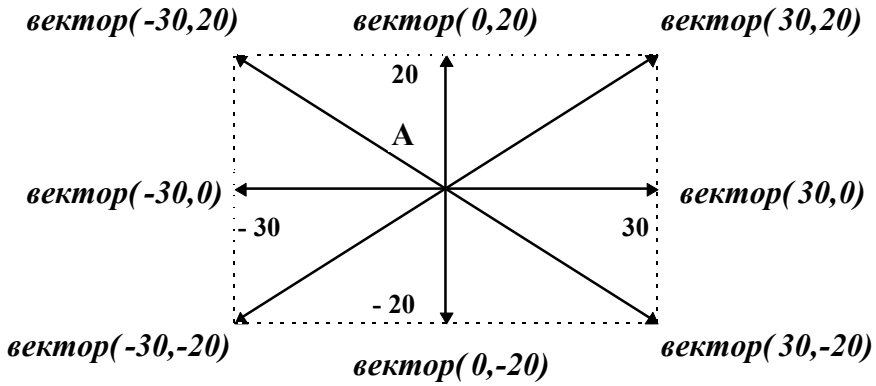
Важно, что для использования команды ***в_точку*** нам требуется знать, где находится начало отсчета, а при использовании команды ***вектор*** – нет, так как отсчет ведется от текущего положения исполнителя. Поэтому в большинстве случаев мы будем использовать только команды ***вектор***.

♦ **Вектор** - это отрезок, имеющий направление.

Начало вектора ***(Δx,Δy)*** находится там, где был исполнитель до выполнения команды, в точке ***(x_0, y_0)***, конец имеет координаты ***(x_0+Δx, y_0+Δy)***.

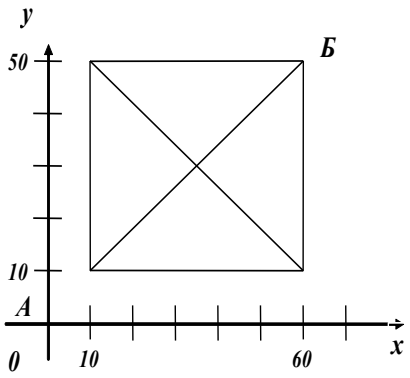


Координаты вектора могут быть как положительными, так и отрицательными числами. Если, например, координата ***x*** положительна, это значит, что исполнитель должен переместиться в ту сторону, в которую увеличивается ось ***OX***, а если координата ***y*** отрицательна, то исполнитель перемещается в направлении, противоположном оси ***OY.***



Задание для Чертежника представляет собой рисунок, состоящий из отрезков. Перед началом работы исполнитель находится в точке ***А,*** после окончания работы ему надо прийти в точку ***Б.***

Пример. Исполнитель находится в начале координат. Неизвестно, в каком положении находится его перо (опущено или поднято). Ниже даны два варианта решения задачи: с помощью команд ***в_точку*** и ***вектор***.



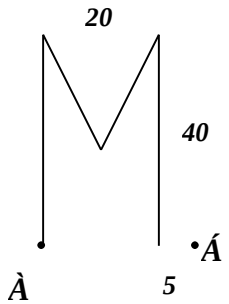

```
КонвертТоч
{
подними_перо;
в_точку (10, 50);
опусти_перо;
в_точку (60, 50);
в_точку (60, 10);
в_точку (10, 10);
в_точку (10, 50);
в_точку (60, 10);
в_точку (10, 10);
в_точку (60, 50);
}
```

```
КонвертВект
{
подними_перо;
вектор (10, 50);
опусти_перо;
вектор (50, 0);
вектор (0, -40);
вектор (-50, 0);
вектор (0, 40);
вектор (50, -40);
вектор (-50, 0);
вектор (50, 40);
}
```

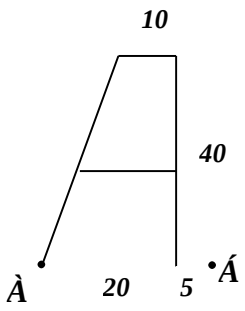
Использование процедур

Напишем с помощью Чертежника слова МАМА на экране. Заметим, что оно состоит из двух одинаковых букв А и двух одинаковых букв М. Это наводит на мысль об использовании процедур.

Научим Чертежника рисовать на экране буквы ***М*** и ***А*** одинаковой высоты. Обе буквы вписаны в прямоугольник ***20*** на ***40***, точка ***Б*** находится на расстоянии ***5*** от правой ножки. Заметим, что переход в точку ***Б*** означает, что исполнитель уже готов рисовать следующую букву.




```
БуквaМ
{
опусти_перо;
вектор (0, 40); вектор (10, -20);
вектор (10, 20); вектор (0, -40);
подними_перо; вектор (5, 0);
}
```




```
БуквaA
{
опусти_перо;
вектор (10, 40); вектор (10, 0);
вектор (0, -20); вектор (-15, 0);
вектор (15, 0);  вектор (0, -20);
подними_перо; вектор (5, 0);
}
```

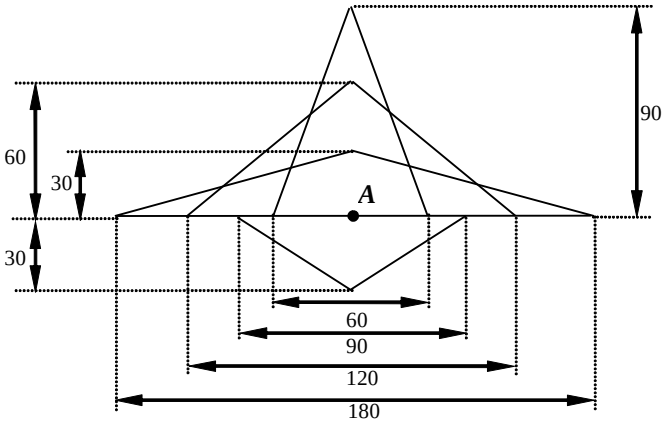
Теперь очень легко написать основную программу:

```
СловоМАМА
{
БуквaM; БуквaA;
БуквaM; БуквaA;
}
```

При проверке на компьютере после нее надо записать обе процедуры.

Процедуры с параметрами

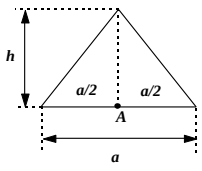
Теперь решим задачу, с которой Черепаха сможет справиться только с очень большим трудом (здесь надо использовать тригонометрические функции). Построим на экране такую фигуру:



Исполнитель находится в точке ***A***, именно отсюда удобнее рисовать все треугольники — там находятся середины их оснований. Как обычно, выделяем общее и отличие:

- **общее**: все треугольники равнобедренные, основание горизонтально
- **отличия**: разная длина основания и высота — они будут **параметрами** процедуры

Включим в параметры процедуры еще цвет линии ***c***:



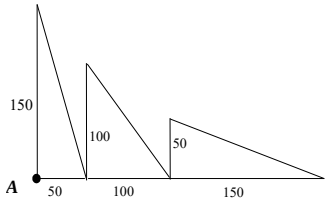

```
Треугольник(int a, int h, int c )
{
опусти_перо;
цвет ( c );
вектор ( -a/2, 0 );
вектор ( a/2, h );
вектор ( a/2, -h ); вектор ( -a/2, 0 );
}
```

С верхними тремя треугольниками все понятно. Сложность — с тем, который "опрокинут" вниз. Однако для того, чтобы нарисовать его, достаточно задать **отрицательную высоту**. Окончательно основная программа принимает вид:

```
Программа
  {
  Треугольник (60, 90, 9);
  Треугольник (120, 60, 10);
  Треугольник (180, 30, 2, 11);
  Треугольник (90, -30, 6, 12);
  }
```

Циклы и переменные

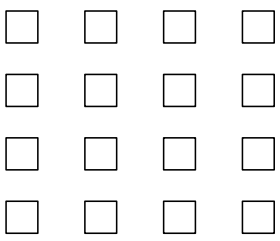
Чертежник понимает команду цикла ***повтори*** и может использовать переменные. Это демонстрирует следующий пример.




```
Зубы
  {
  int a = 10, h = 30;
  повтори ( 3 )
    {
    Треуг (a, h);
    a = a + 10;
    h = h - 10;
    }
  }
Треуг (int a, int h)
  {
  опусти_перо;
  вектор (0, h); вектор (a, -h);
  вектор (-a, 0);
  подними_перо;
  вектор (a, 0);
  }
```

Здесь одновременно меняются две переменные: основание и высота, причем основание увеличивается, а высота уменьшается.

Рассмотрим еще одну задачу: с помощью исполнителя Чертежник нарисовать узор из квадратов:



Сторона каждого из квадратов и расстояние между строками и между столбцами равны ***10***. Приведем два способа решения этой задачи:

```
Узор
  {
  повтори ( 4 )
    Ряд ( 4 );
  }
Ряд ( int n )
  {
  повтори ( n )
    {
    Квадрат ( 10 );
    вектор ( 20, 0 );
    }
  вектор ( 0, -20 );
  вектор ( -20*n, 0 );
  }
```

```
Узор
  {
  повтори ( 4 )
    {
    повтори ( 4 )
      {
      Квадрат ( 10 );
      вектор ( 20, 0 );
      }
    вектор ( 0, 20 );
    вектор ( - 80, 0 );
    }
  }
```

Первый способ использует процедуру ***Ряд***, параметром которой является количество квадратов в строке. Обратите внимание, что тело цикла не ограничено скобками, так как содержит только один оператор. При этом соблюдается запись с отступами – тело цикла сдвинуто вправо.

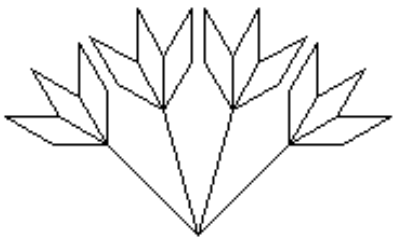
Во втором способе мы фактически сразу подставили тело процедуры в основную программу. При этом получилось, что в теле *внешнего цикла* в основной программе находится еще один цикл (*внутренний*), который раньше был в процедуре. Такая конструкция называется *вложенный цикл***.** При этом в записи тело внутреннего цикла также сдвинуто вправо относительно его заголовка.

Заметьте, что для правильной работы этих программ к ним необходимо добавить текст процедуры ***Квадрат*****.**

```
Квадрат (  int a )
{
опусти_перо;
вектор (0,  a ); вектор ( a, 0);
вектор (0, - a ); вектор (- a, 0);
подними_перо;
}
```

Сравнение Чертежника и Черепахи

Черепаха и Чертежник имеют разные команды и работают в разных системах координат: Черепаха в **естественной**, а Чертежник - в прямоугольной. Поэтому Черепаха очень легко рисует развернутые фигуры, например вот такую:



А

```
Букет
  {
  влево ( 45 );
  повтори ( 4 ) {
    Цветок;
    вправо ( 30 );
    }
  }
```

где используется процедура для построения одного цветка

```
Цветок
  {
  опусти_перо;
  вперед ( 60 ); влево ( 45 );
  повтори ( 3 ) {
    повтори ( 2 ) {
      вперед ( 25 ); вправо ( 30 );
      вперед ( 25 ); вправо ( 150 );
      }
    вправо ( 30 );
    }
  влево ( 45 ); назад ( 60 );
  подними_перо;
  }
```

Благодаря естественной системе координат Черепаха отлично справилась с работой. Теперь вообразите, сколько усилий надо затратить, чтобы научить Чертежника рисовать такую фигуру.

Однако у Черепахи тоже есть недостатки. Так ей трудно начертить прямоугольный треугольник, потому что она должна знать длины всех его сторон и все углы (Чертежник легко решает эту задачу, зная длины только двух сторон, которые пересекаются под прямым углом).

Переменные и использование памяти

Пример 1. Построить на экране с помощью исполнителя Чертежник квадрат, длину его стороны ввести с клавиатуры и рассчитать площадь этого квадрата.

```
ДиалогКвадрат
  {
  int a;
  вывод "Введите длину стороны квадрата";
  ввод a;
  Квадрат ( a );
  вывод "Площадь квадрата со стороной ", a,
      " равна ", a*a;
  }
Квадрат ( int a )
  {
  опусти_перо;
  вектор ( 0, a ); вектор ( a, 0 );
  вектор ( 0, - a ); вектор ( - a, 0 );
  подними_перо;
  }
```

Пример 2. Составить программу для решения системы двух линейных уравнений:

$$\begin{cases} a_1 x + b_1 y = c_1 \\ a_2 x + b_2 y = c_2 \end{cases}$$

Сначала мы должны решить эту систему в общем виде, то есть “в буквах”. Это можно сделать, выразив ***y*** из первого уравнения

$$y = \frac{c_1 - a_1 x}{b_1}$$

и подставив результат во второе:

$$a_2 x + \frac{b_2 c_1}{b_1} - \frac{b_2 a_1 x}{b_1} = c_2$$

Выразив отсюда ***x***, получаем

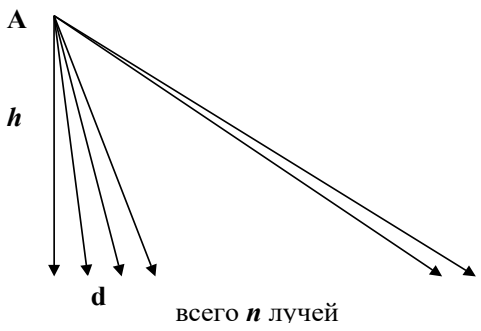
$$x = \frac{c_2 - \frac{b_2 c_1}{b_1}}{a_2 - \frac{b_2 a_1}{b_1}}$$

Таким образом, исходными данными являются значения $a_1, b_1, c_1, a_2, b_2, c_2$, по ним мы сначала вычисляем ***x***, а затем – ***y***, которые и будут результатами программы. Заметим, что систему нельзя решить этим способом, когда $b_1=0$ или знаменатель выражения для ***x*** равен нулю. Эти случаи надо обрабатывать отдельно, сообщая об ошибке.

```
СистемаУравнений
  {
  float a1, b1, c1, a2, b2, c2, x, y, d = 0;
  вывод "Введите a1, b1 и c1 ";
  ввод a1, b1, c1;
  вывод "Введите a2, b2 и c2 ";
  ввод a2, b2, c2;
  если ( b1 != 0 )
    d = a2 - b2*a1/b1;
  если ( d != 0 )
    {
    x = (c2 - b2*c1/b1) / d;
    y = (c1 - a1*x) / b1;
    вывод "Решение системы: x = ", x, ", y = ", y;
    }
  иначе
    вывод "Система вырожденна.";
  }
```

Цикл с параметром

Нарисуем на экране с помощью исполнителя Чертежник веер из ***n*** лучей, концы которых находятся на одном уровне по оси ***y***, а расстояние между концами соседних лучей равно ***d***. Исполнитель находится в точке ***A***.



Рассмотрим разные способы решения этой задачи. Первый отрезок направлен вертикально вниз, то есть ему соответствует ***вектор(0,-h)***. После этого лучше поднять перо и вернуться обратно в точку ***A***.

Следующий отрезок рисуется командой ***вектор(d,-h)***, дальше будет ***вектор(2d,-h)*** и т.д. Заметим, что в команде ***вектор*** изменяется только координата ***x***, которая принимает последовательно значения ***0,d,2d,3d,...,(n-1)d***, то есть, начальное значение ***x*** равно ***0***, цикл выполняется пока ***x<nd*** и с каждым шагом значение ***x*** увеличивается на величину ***d***.

Для решения удобно использовать *цикл с параметром*, который так и называется – ***цикл*** или ***for*** (от английского *for – "для"*).

```
Beep
{
int x, h, d, n;
вывод "Введите d, h и n ";
ввод d, h, n;
цикл (  x = 0;  x < n*d;  x = x + d )
  {
  опусти_перо;
  вектор ( x, -h );
  подними_перо;
  вектор ( -x, h );
  }
}
```

Как видно из этой программы, все операции с переменной ***d*** сгруппированы в заголовке цикла внутри круглых скобок. Три части отделяются знаком **";"**, так же как и конец команды:

1. **Начальное условие** ***x=0*** выполняется один раз перед началом цикла;
2. **Условие продолжения** ***x<n*d*** говорит о том, при каком условии цикл будет выполняться (если в самом начале это условие неверно, то цикл не выполнится ни одного раза);
3. **Изменение переменной цикла** ***x=x+d*** - этот оператор выполняется каждый раз в конце очередного прохода тела цикла.

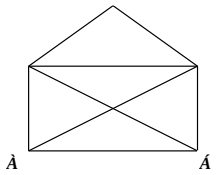
? Как можно заменить этот цикл на цикл ***повтори***? на цикл ***пока***?

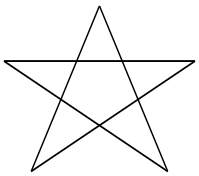
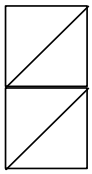
Задачи

1. Что нарисует Чертежник , выполнив программу:

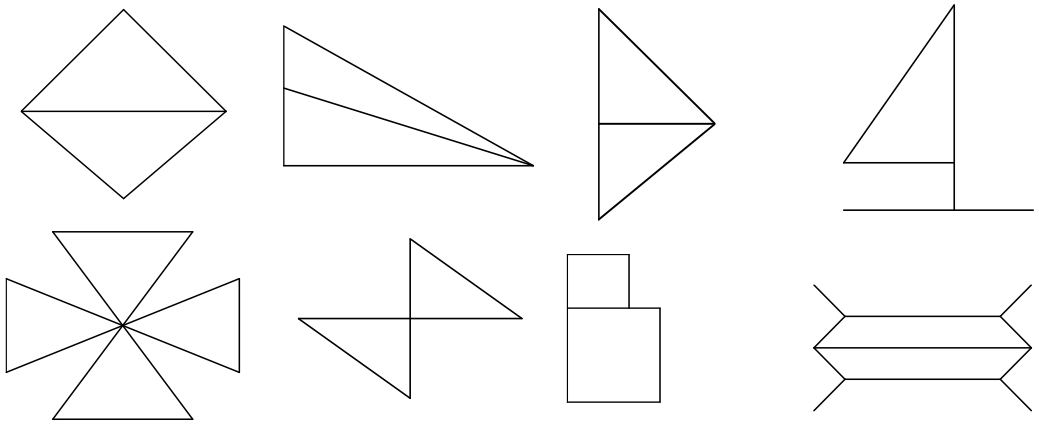
```
Домик
  {
  опусти_перо;
  вектор (4, 0); вектор (0, 4);
  вектор (-4, 0); вектор (0, -4);
  подними_перо;
  вектор (0, 4); опусти_перо;
  вектор (2, 2); вектор (2, -2);  подними_перо;
  вектор (-4, -4);
  }
```

2. Не «отрывая пера от бумаги» и не проводя одну линию два раза нарисовать

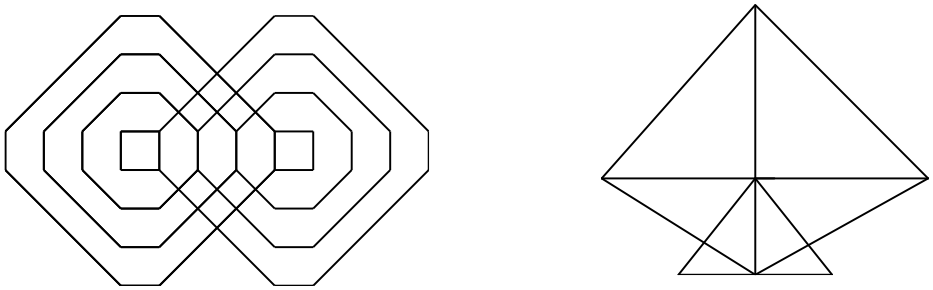


3. Составить программы для исполнителя Чертежник



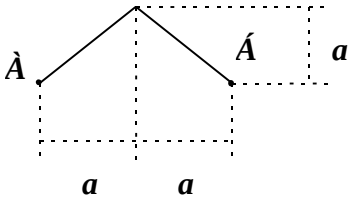
4. Составить программу для исполнителя Чертежник, используя процедуру с параметром.



5. Какую фигуру нарисует Чертежник, выполнив программу

```
Спираль
  {
  опусти_перо;
  Виток ( 10 );
  Виток ( 30 );
  Виток ( 50 );
  Виток ( 70 );
  Виток ( 90 );
  подними_перо;
  }
Виток ( int n )
  {
  вектор ( n, 0 );
  вектор ( 0, - n );
  вектор ( - n - 10, 0 );
  вектор ( 0, n + 10 );
  }
```

6. Составить процедуру для исполнителя Чертежник, по которой он рисует ***n*** звеньев ломаной размером ***a.*** На рисунке изображено одно звено.



Что произойдет, если при вызове процедуры задать отрицательное значение параметра **a?**

7. Составить программу для исполнителя Чертежник, используя цикл с параметром: