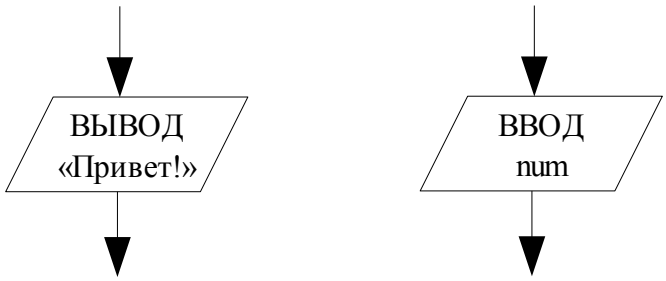
Определение алгоритмов.

Прежде чем приступать к изучению программирования необходимо четко осознать что такое алгоритм. Алгоритм – это какая-то строго заданная последовательность действий. С алгоритмами человек, сам того не осознавая сталкивается ежедневно. Возьмите, к примеру, поваренную книгу, откройте на любой странице и вы увидите текст вроде *«ингридиенты измельчить, перемешать, подогреть 10 минут на медленном огне; полить сверху кремом и выпекать 30 минут в духовке при температуре 200 градусов»*. Или другой пример – описание как добраться до какого-то места *«необходимо доехать до станции метро Юго-Западная, выход в строну первого вагона из центра, далее автобусом 707 до остановки Родниковая улица дом 14»*. Эти две ситуации являются примерами так называемых линейных алгоритмов – последовательность действий в них четко определена от начала до конца и не изменяется. Но в реальной жизни чаще всего мы имеем дело с ситуациями, когда в определенный момент приходится делать выбор или принимать решение, от которого будут зависеть наши дальнейшие действия. Например, мама отправляет дочку в магазин: *«сходи в магазин, если там будет молоко – купи два пакета, если молока не будет – возьми пакет кефира»*. Или определение из учебника математики для младших классов: *«если число делится без остатка на два, то оно называется четным, в противном случае оно называется нечетным»*. Это так называемые разветвляющиеся алгоритмы. То есть в каждом случае мы имеем некоторое условие (*молоко есть в магазине* или *число делится без остатка на два*) и дальнейшие действия, которые мы делаем или не делаем в зависимости от выполнения условия (*покупаем молоко – покупаем кефир* или *число четное – число нечетное*). Любая программа для компьютера – это алгоритм, записанный на некотором языке программирования и, чаще всего, скомпилированный в исполняемый код.

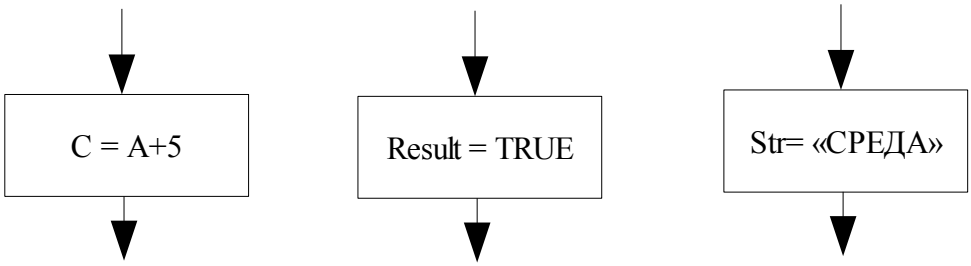
Способы описания алгоритмов.

Итак, как мы уже выяснили, алгоритм – это некоторая последовательность действий. Но как-то эти указания нам необходимо донести до непосредственного исполнителя. В повседневном общении люди использую словесную форму описания действий, как в примерах выше. Однако эта форма не всегда удобна, так как некоторые фразы могут быть двусмысленны, непонятны и неточны. В программировании используют более формализованные способы описания алгоритмов. В частности, мы будем изображать алгоритмы в виде блок-схем. Преимущества этого способа очевидны: не надо вчитываться в описание, основная идея алгоритма и его структура видны с первого взгляда. К тому же, этот способ описания широко используется во всём мире как в работе, так и в специальной литературе. Поэтому, если вам попадет в руки книга по программированию, в которой алгоритмы описываются блок-схемами, вы сможете понять смысл, даже если сама книга написана на иностранном языке.
Существует около двух десятков различных элементов блок-схемы, но основными, необходимыми для понимания, считать следующие.
Начало/окончание алгоритма. Его назначение понятно из названия – этот блок обозначает начало или конец нашей программы. Изображается в виде овала.

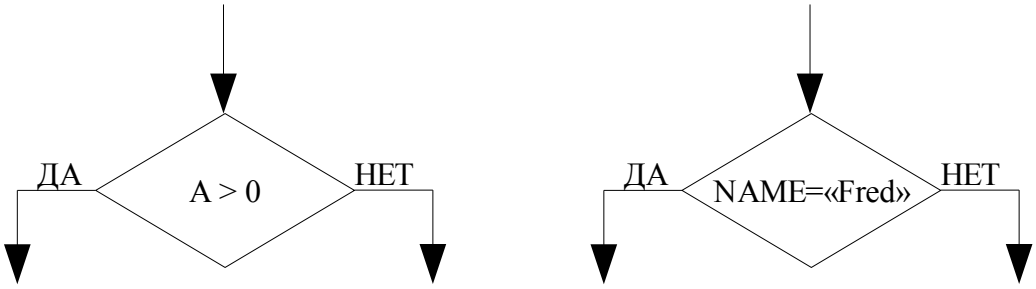
Блок ввода-вывода. Показывает место в программе, где на экран выводится какое-то сообщение, либо наоборот пользователь должен ввести с клавиатуры какие-то данные. Изображается в виде параллелограмма, внутри которого указана операция (ввод или вывод) и константа для вывода на экран, либо переменная, для ввода/вывода.



Блок действия. Изображается в виде прямоугольника. Наиболее часто встречающийся элемент алгоритма. Опять же, из названия очевидно, что любые действия, выполняемые алгоритмом, описываются именно в этом блоке.



Блок принятия решения. Именно благодаря этому блоку мы и можем описывать разветвляющиеся алгоритмы. Внутри блока имеется некоторое условие, которое может быть истинно или ложно. От этого блока всегда отходят две стрелки. Одна из них обозначается «ДА» и показывает ход алгоритма, если условие в блоке истинно, вторая обозначается «НЕТ» и показывает ход алгоритма при ложном условии. Что такое истинное и ложное условие? Всё просто. Любой утверждение либо верно, либо неверно, то есть истинно либо ложно. Например, утверждение *«Земля держится на трех черепахах»* ложно, а утверждение *«2+2=4»* - истинно. Если, например, мы имеем A=4 и B=5, то утверждения *«A+B=9»*, *«A<B»*, *«B-A>0»* истинны, а утверждения *«A*B=22»*, *«B=10»*, *«A>B+55»* ложны. Так же, вполне корректными считаются и подобные бессмысленные на первый взгляд



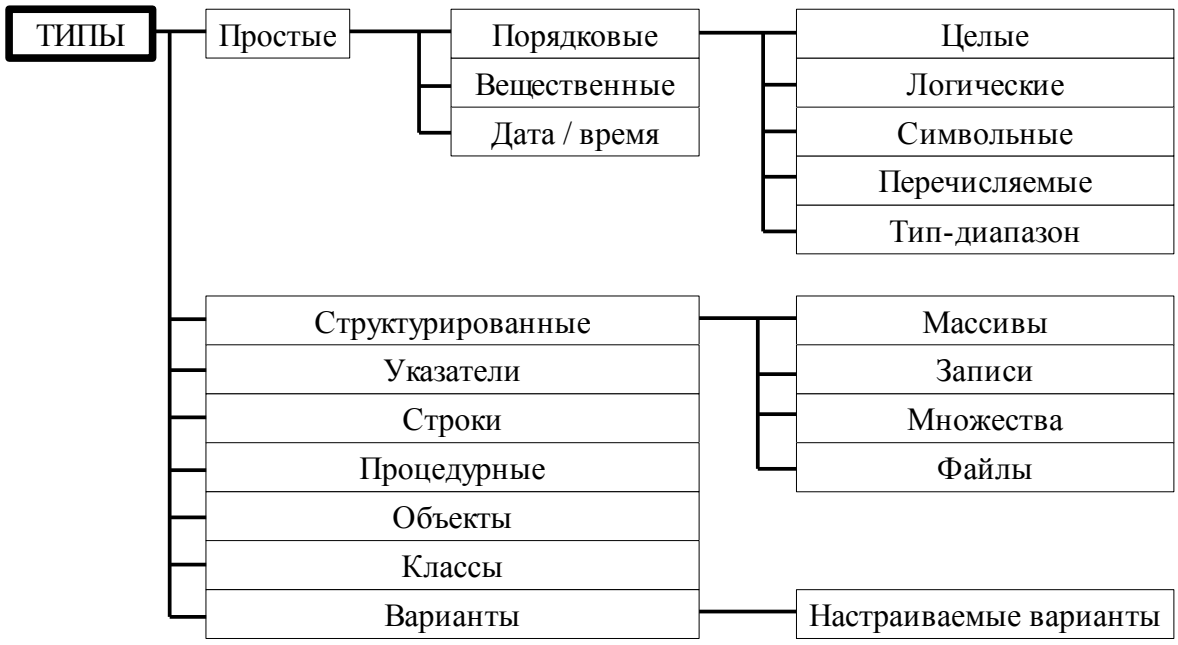
Введение в язык Pascal.

Язык Pascal – прародитель Delphi, в ней самой используется так называемый Object Pascal, который от обычного паскаля отличается достаточно существенно, но на уровне структуры программы, основных конструкций, базовых типов эти отличия не играют роли.
Начиная изучать язык Pascal, необходимо в первую очередь четко осознать следующие его основные отличия от языка Basic:

- Паскаль является типизированным языком. Прежде чем переменная будет использоваться в программе, необходимо явным образом указать какого типа данные в этой переменной хранятся, то есть объявить переменную.
- Программа на паскале имеет четкую структуру – имеется специальный раздел для описания переменный, раздел для задания констант, раздел для непосредственного описания алгоритма.

Типы данных.

Как вы уже знаете из предыдущих курсов, переменная – это некоторая область памяти, где хранятся данные программы, котрые в процессе работы могут модифицироваться. Но так же мы знаем, что память компьютера – это всего лишь набор битов. В виде единичек и ноликов в памяти хранится музыка, видео, тексты, сами программы – абсолютно всё. Чтобы не получалось путаницы, мы должны четко указать компилятору что именно у нас хранится в этом участке памяти: целое число, в котором, например, записан номер дома, дробное число, в котором хранятся результаты измерений в физическом эксперименте, или же это строка символов, в



Вообще говоря, существует иерархия типов в паскале, но на данном этапе вам необходимо знать как минимум эти:
Целый тип. В этом типе хранится то, что в математике называется целым числом – 1, 3 ,10, 0, 15, -44 и т.п. Целые типы отличаются друг от друга размером в памяти (от одного до четырёх байт) и тем являются ли они знаковыми или беззнаковыми. Основным типом здесь является INTEGER. Переменная этого типа может принимать любое значение в диапазоне -2147483648..2147483647
Вещественный тип. Вещественные типы применяются для хранения вещественных чисел, то есть чисел имеющих некоторую дробную часть. Вещественные типы отличаются друг от друга, во-первых, количеством занимаемой памяти (от этого зависит насколько большие числа мы можем хранить в переменной этого типа), а во-вторых, количеством бит, отводимых под хранение дробной части (от этого зависит точность вычислений). Наиболее часто используемый вещественный тип – REAL. Он занимает 8 байт и принимает значения от $5.0*10^{-324}$ до $1.7*10^{308}$

Логический тип. Тип называется BOOLEAN. Переменная данного типа может принимать два значения: TRUE (истина) или FALSE (ложь). Применяется для хранения результатов логических операций.
Символьный тип. Как следует из названия, применяется для хранения отдельных символов, например, когда мы хотим считать с клавиатуры ответ пользователя Y/N или нам нужно хранить ориентирование по сторонам света (N, E, S, W). Название типа – CHAR.
Строковый тип. Тип STRING - предназначен для хранения строки символов. То есть это любой короткий текст – имя человека, его адрес, название чего-либо. Константы строкового типа в паскале записываются в одинарных кавычках – *'это строка'* . К отдельным символам строки можно обратиться, если после имени переменной строкового типа поставить в квадратных скобках номер символа, то есть если строку предыдущего примера записать в переменную *S*, то запись *S[2]* будет указывать на второй символ этой строки, то есть на *'m'*, а *S[4]* будет указывать на пробел.

Структура программы.

При написании программы на паскале важно помнить две вещи.
Во-первых, после любого оператора в программе (а оператор – это любое присвоение, объявление переменной, вызов функции и т.д.) ставится точка с запятой. Исключений из этого правила очень мало и о них будет сказано отдельно.
Во-вторых, многие конструкции паскаля (о которых будет сказано ниже) подразумевают использование внутри себя одного оператора. Если же необходимо использовать несколько операторов (например, необходимо поменять значения нескольких переменных), то необходимо использовать конструкцию *begin ... end;* То есть вы пишите *begin*, пишите *end* с точкой с запятой, а между ними пишите всё, что душе угодно. Эта конструкция называется операторные скобки.
Рассмотрим основные блоки программы на примере стандартного шаблона, создаваемого для консольной (то есть работающей в текстовом режиме) программы.

```
program Project2;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

Первая строка – заголовок программы, состоит из зарезервированного слова *program* (все слова, выделенные жирным шрифтом – зарезервированные слова языка Object Pascal) и непосредственно имени программы. В качестве имени программы может выступать любой корректный идентификатор.
Идентификатор – имя программы, модуля, функции, процедуры, класса, переменной и т.д. - может состоять из букв (буквой с точки зрения паскаля считается любой символ от *a* до *z*, от *A* до *Z*, а также символ подчеркивания) и цифр, причем начинаться он должен с буквы. Название идентификатора не должно совпадать ни с одним из зарезервированных слов паскаля. Идентификаторы регистронезависимы, то есть для паскаля записи *NAME_1567_AA*, *name_1567_aa*, *NaMe_1567_aA* и *naME_1567_aa* – одно и то же.
Вторая строка программы – директива компилятора. Это своего рода инструкция для Delphi, которая указывает как надо обрабатывать последующий код. Директивы компилятору записываются в фигурных скобках и первым символом внутри скобок должен быть символ *$*. Вообще подобных директив существует много, но в данном случае вам просто надо запомнить что в данном случае должна быть именно эта, иначе ваша программа просто не будет компилироваться.
Далее следует блок *uses* – блок подключения внешних модулей. Паскаль является модульным

языком – отдельные куски программы можно выделять в процедуры и функции, а процедуры и функции выносить в другие программы. Для чего это нужно? Для облегчения работы программисту. Например, достаточно часто бывает нужна функция вычисления квадратного корня. Вычисление корня с помощью различных математических методов – задача не такая уж и простая, и её реализация на паскале займет не меньше пары десятков строк. Поэтому, можно вынести этот код в отдельную процедуру, и уже вызывать её – примерно так же, как мы в математике пишем $\sqrt{x}$, абсолютно не задумываясь как в действительности этот корень вычисляется – потом вы просто берете специальные таблицы или калькулятор и записываете полученное значение. Можно пойти дальше. Нам ведь вычисление корня нужно не в одной программе, а очень во многих вычислительных задачах, поэтому логично было бы сделать так, чтобы эту функцию описать раз и на всегда, а все последующие разы на неё ссылаться. Модульная структура паскаля позволяет это делать. Существует функция *sqrt()*, которая описана в модуле *System*. Этот модуль автоматически подключается к программе, а если бы не подключался это нужно было бы сделать самим указав в блоке *uses* имя модуля – *System*.
С процедурами вы уже сталкивались в Бейсике. Вспомните, например, процедуры рисования на экране. Чтобы нарисовать точку, линию, окружность на экране вам не нужно было задумываться о том, как всё это реально хранится в памяти видеокарты, как сигнал идет на монитор. При рисовании линии или окружности вам не нужно было просчитывать каждую их точку (а рисуются они именно по отдельным точкам)- всё это уже реализовано в соответствующих процедурах, и вам остается только указать координаты объекта, его цвет – и вот вы уже видите на экране свою линию, точку, окружность...
Вернемся к программе. Далее идет тело программы – всё то, что заключено между *begin* и *end.* Обратите внимание – в данном случае после оператора *end* стоит точка. Это оператор означает конец программы и весь текст, который идет за ним, компилятор игнорирует. А всё, что находится между этими двумя операторами – и есть программа, непосредственная реализация алгоритма.
В теле программы вы так же можете увидеть *{ TODO -oUser -cConsole Main : Insert code here }*. Это комментарий. Комментарии предназначены для того, чтобы облегчить понимание программы. Программист может пояснить назначение какое-то переменной или функции, пояснить действие алгоритма или, например, оставить комментарий в начале программы о том, что автор именно он. Так же комментарии появляются в автоматически создаваемых Delphi шаблонах. Например, этот комментарий подсказывает нам, что именно здесь, на его месте и нужно писать код программы. Комментарием считается любой текст находящийся внутри фигурных скобок *{ ... }*, внутри круглых скобок со звездочками *(* ... *)* или за двумя наклонными чертами *//* и до конца строки.
Кроме перечисленных выше элементов программы, для начала даже простейшего программирования на паскале вам обязательно потребуется еще один блок – блок объявления переменных. Как уже было сказано, прежде чем использовать какую-то переменную в паскале, нужно её объявить, то есть указать какого она типа. Происходит этот процесс как раз в блоке объявления переменных. Блок имеет очень простую структуру: начинается с зарезервированного слова *var*, и далее идут конструкции вида *список_переменных : тип_переменных;* имена отдельных переменных в списке разделяются запятыми.

```
program my_program; // до конца строки - комментарий
{$APPTYPE CONSOLE}
uses
  SysUtils; (* и это комментарий *)
var
  a,b:integer; // объявление двух переменных целого типа
  s1,s2,s3:string; // объявление трёх строковых переменных
  cn:char; // объявление символьной переменной
  x,y,z:real; // объявление трех вещественных переменных
begin
  { это тоже комментарий }
end.
```

Итак, теперь мы готовы написать простейшую программу на паскале. Пусть она спрашивает имя пользователя, и потом выводит приветствие с этим именем на экран.

```
program ask_name;                    // 1
{$APPTYPE CONSOLE}                   // 2
uses                                 // 3
  SysUtils;                          // 4
var                                  // 5
  name:string;                       // 6
begin                                // 7
  write('Vvedite imya:');            // 8
  readln(name);                      // 9
  writeln('Hello, ',name,'!!!!');    // 10
  readln;                            // 11
end.                                 // 12
```

Итак, наша программа состоит всего из 12 строк:

1. задается имя программы
2. директива компилятора – указывает на то, что программа будет работать в консольном режиме
3. начало блока подключения внешних модулей
4. подключается внешний модуль *SysUtils*
5. начало блока объявления переменных
6. объявляется переменная *name* строкового типа
7. начало программы
8. вывод на экран подсказки для ввода имени
9. считывание имени с клавиатуры
10. вывод на экран приветствия с именем пользователя
11. ждем пока пользователь не нажмет на Enter – чтобы окно программы не закрылось сразу же после выполнения предыдущих команд
12. конец программы

Строки с 7 по 12 можно считать алгоритмом работы программы и при желании можно изобразить в виде блок-схемы.
Здесь для вас появились новые операции - ввод-вывод на экран. Эти процедуры паскаля соответствуют блоку ввода-вывода при представлении алгоритма в виде блок-схемы.
write(список вывода) – выводит на экран элементы, заданные в списке вывода
writeln(список вывода) – выводит на экран элементы, заданные в списке вывода, и переводит курсор на новую строку
read(список ввода) – заполняет с клавиатуры переменные, заданные в списке ввода
readln(список ввода) – заполняет с клавиатуры переменные, заданные в списке ввода, и при этом очищает буфер
Список ввода – список переменных, которые по очереди будут заполняться вводом с клавиатуры.
Список вывода – это разделенный запятыми список констант и переменных, содержимое которых мы хотим вывести на экран.
То есть, если пользователь в ответ на подсказку введет с клавиатуры *Vasya* и нажмет Enter, то в переменную *name* будет записана именно эта строка (9я строчка программы). Далее, согласно списку вывода в десятой строчке, на экран сначала будет выведена строковая константа *'Hello '*, далее будет выведено содержимое переменной *name* и следом за ней – строковая константа *'!!!'*. В результате на экране мы увидим *'Hello Vasya!!!'*.

Математические операции.

Прежде чем начать говорить непосредственно о математических операциях, отметим, что в паскале оператор присваивания выглядит как *:=* (двоеточие и знак равенства). Отдельный знак

равенства обозначает операцию сравнения, используемую в условных операторах, о которых речь будет идти чуть дальше. То, есть, если вам надо записать с переменную x число 2, вы должны записать *x:=2;* Если бы в предыдущем примере имя пользователя вводилось не с клавиатуры, а задавалось жестко в программе, то такое присваивание выглядело бы следующим образом:
name:='Vasya';
Теперь о математических операциях. Их шесть: сложение (+), вычитание (-), умножение (*), деление (*/*), деление нацело (*div*), остаток от деления (*mod*). Принцип действия первых четырех операций очевиден. Чтобы понять последние две, вспомните начальную школу, когда вы еще не знали дробный чисел. На уроках арифметики вы делили 5 яблок на двоих и получали по 2 яблока на каждого и 1 яблоко в остатке. Как раз такое деление и обеспечивают операции div и mod. Действие операций проиллюстрировано в таблице. Пусть предварительно у нас были выполнены команды *a:=15; b:=10;* Тогда получим

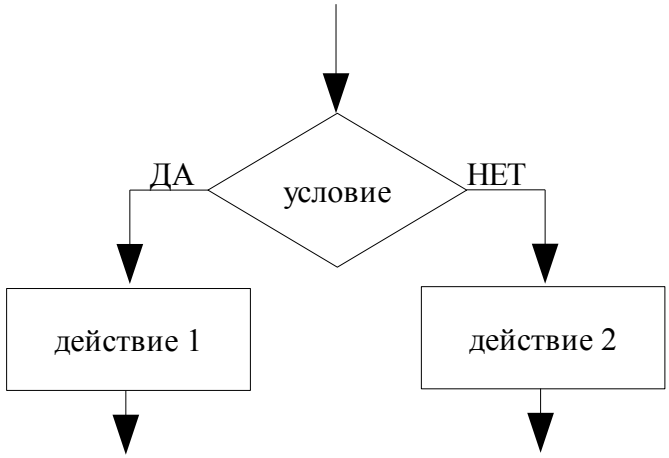
a + b	*25*
a – b	*5*
*a * b*	*150*
a / b	*1,5*
a div b	*1*
a mod b	*5*

Кроме того, существует множество математических функций, в частности:
abs(x) - модуль числа
sqr(x) - квадрат числа
srt(x) - квадратный корень
sin(x), cos(x), tan(x) – тригонометрические функции (полный список можно найти в справочной системе Delphi по ключевому слову *Trigonometry routines*)

Условный оператор IF.

Для реализации простейших нелинейных алгоритмов используется условный оператор IF. Фактически он является полный

```
if условие then
  действие1
else
  действие2;
```



Данный оператор работает по следующему правилу: если выражение, записанное в условии принимает истинное значение, то выполняется действие 1, в противном случае выполняется действие 2.
Следует обратить внимание на следующие моменты:

- Здесь мы имеем как раз то редкое исключение, когда после оператора не ставится точка с запятой – в операторе IF перед ELSE точка с запятой не ставится.
- Под *действие1* и *действие2* подразумевается один оператор. Если же нам необходимо использовать несколько операторов (например, нужно изменить значение какой-то переменной и при этом вывести это значение на экран), тогда нужно использовать операторные скобки (см. раздел «Структура программы»).

Для формирования условий можно использовать операторы сравнения и логические операции. Операторы сравнения: равно (=), не равно (<>), больше (>), меньше (<), больше или равно (>=), меньше либо равно (<=). То есть, например, проверка на четность будет выглядеть так:

```
if a mod 2=0 then ...
```

С помощью логических операций можно комбинировать отдельные условия таким образом, что выражение будет принимать истинное значение, если истинны оба условия (операция И – and), хотя бы одно из условий (операция ИЛИ – or), только одно из условий (операция ИСКЛЮЧАЮЩЕЕ ИЛИ – xor).

A	B	*not* A	A *and* B	A *or* B	A *xor* B
FALSE	-	TRUE	-	-	-
TRUE	-	FALSE	-	-	-
FALSE	FALSE	-	FALSE	FALSE	FALSE
FALSE	TRUE	-	FALSE	TRUE	TRUE
TRUE	FALSE	-	FALSE	TRUE	TRUE
TRUE	TRUE	-	TRUE	TRUE	FALSE

Например, если необходимо проверить принадлежит ли точка отрезку [3;5), то в программе будет использоваться следующее условие:

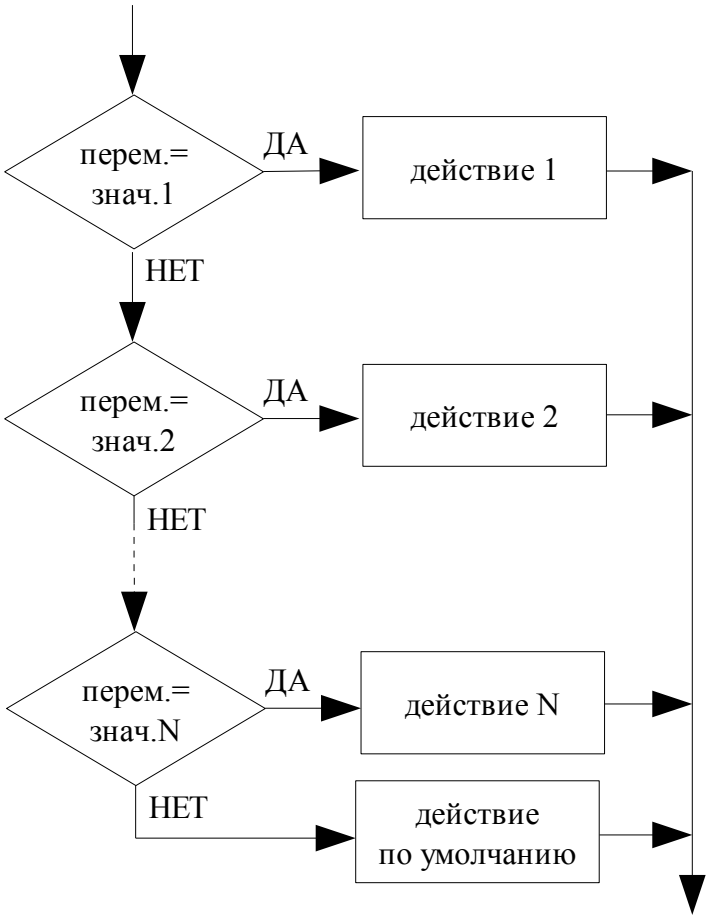
```
if (x>=3) and (x<5) then
  writeln('Prinadlegit');
```

Обратите внимание: когда мы объединяем несколько условий с помощью логических операций, каждое из этих условий должно быть помещено в круглые скобки.

Оператор выбора CASE.

Достаточно часто возникаю ситуации, когда требуется проверить не выполнение конкретного условия, а нужно узнать какое условие из нескольких условий выполнено. Как в старой сказке – стоит богатырь на распутье, а на камне надпись «налево пойдёшь...» - типичный вариант множественного выбора.

```
case переменная of
  значение1: действие1;
  значение2: действие2;
  значение3: действие3;
  ...
  значениеN: действиеN;
else
  действие_по_умолчанию;
end;
```



Оператор выбора CASE работает по следующему правилу: по очереди проверяются все значения, если переменная равна одному из них, то выполняется соответствующее действие. Если ни одно из значений ни совпало со значением переменной, то выполняется действие по умолчанию. Следует отметить, что здесь действует точно такой же принцип относительно действий, как и в предыдущем случае – это либо один оператор, либо несколько, но они заключены в операторные скобки. Кроме того, как в операторе IF, так и в операторе CASE блок ELSE является необязательным. В этом случае, если не будет выполнено ни одно из условий, то не будет выполнено никаких действий.
В качестве значений могут использоваться не только отдельные константы, но и их списки, разделенные запятыми (например *1,16,44,68,79,23*) или диапазоны значений (например *1..10*). Например, мы имеем следующую задачу: Дан номер месяца — целое число в диапазоне 1–12. Определить количество дней в этом месяце для не високосного года.

```
program month_case;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  month:integer;
begin
  write('Vvedite noner mesyaca:');
  readln(month);
  case month of
    2: writeln('28 dnej');
    4,6,9,11: writeln('30 dnej');
    1,3,5,7,8,10,12: writeln('31 den');
  end;
  readln;
end.
```

Цикл FOR.

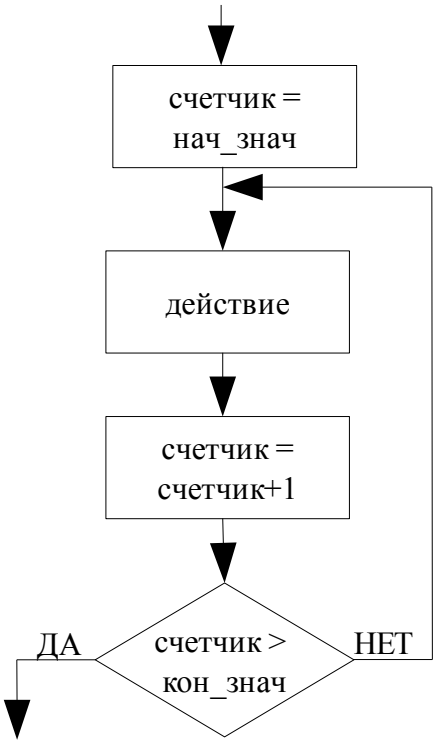
Достаточно часто нам приходится некоторые действия повторять по нескольку раз. Например, нужно получить сумму всех чисел от 1 до 10. В общем-то, не очень сложно написать *S=1+2+3+4+5+6+7+8+9+10*. А если надо будет сложить числа от 1 до 500? Ведь намного проще выполнять действие *«возьми по очереди каждое число от 1 до 500 и прибавь к общей сумме»*. Вроде то же самое? Но ведь на самом деле добавилось слово *«по очереди, каждое»*. То есть мы знаем начальное значение, конечное значение, к которому надо двигаться и знаем действие, которое надо выполнять на каждом шаге.

```
for счетчик := нач_знач to кон_знач do
  действие;
```

Здесь предполагается, что начальное значение меньше конечного. Если же ситуация обратная – вместо *to* используется *downto*:

```
for счетчик := нач_знач downto кон_знач do
  действие;
```

Соответствующим образом изменится и блоксхема для второго случая.



Данный цикл работает следующим образом (в варианте с *to*): счетчик цикла поочередно проходит все значения от начального значения (нач_знач) до конечного значения (кон_знач), увеличиваясь на каждом шаге на единицу. При этом на каждом шаге выполняется *действие*.
Отмечаем следующие моменты:

- *действие*, как и раньше – либо отдельный оператор, либо несколько, но заключенных в операторные скобки
- счетчик цикла – переменная целого типа, начальное и конечное значения – переменные или константы целого типа
- мы можем внутри цикла сами менять шаг, на который будет меняться счетчик (то есть если счетчик цикла у нас *i*, и внутри цикла мы вставим выражение *i:=i+3;* то счетчик на каждом шаге будет увеличиваться не на единицу, а на три)
- число повторений данного цикла заранее известно – мы сами его задаем за счет начального и конечного значений

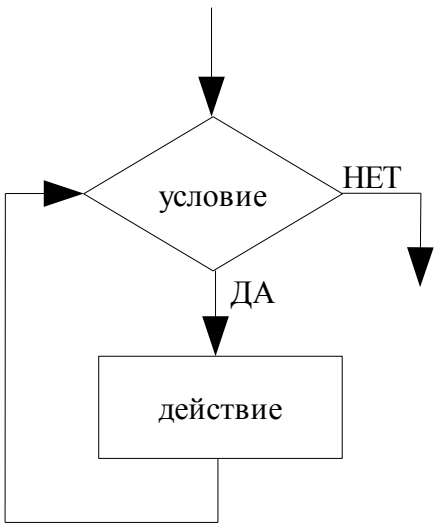
В качестве примера использования цикла FOR рассмотрим вычисление факториала (факториал числа – произведение всех чисел от единицы, до данного числа, то есть 5!=1*2*3*4*5).

```
program factorial;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  i,n:integer;
  fact:int64;
begin
  write('Vvedite chislo:');
  readln(n);
  fact:=1;
  for i:=1 to n do
    fact:=fact*i;
  writeln('Factorial=',fact);
  readln;
end.
```

Обратите внимание, что для переменной *fact*, в которой и хранится значение факториала, был взят тип *int64* – самый емкий из целых, чтобы не случилось переполнения.

Циклы WHILE и REPEAT-UNTIL.

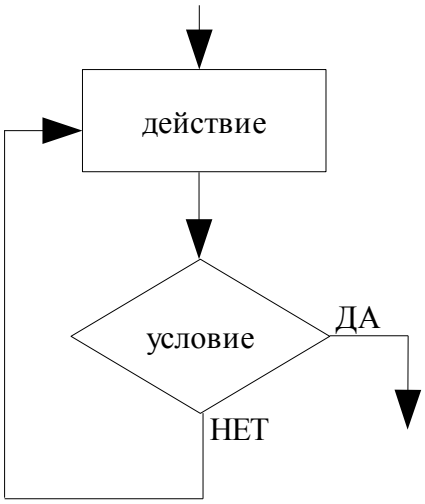
Кроме повторяющихся процессов, где число повторений нам известно, в жизни мы часто сталкиваемся с процессами, длительность которых нам неизвестна. Простой пример: мы подошли к светофору, видим – красный свет. Стоим ждем. Снова посмотрели на светофор – красный. Стоим ждем. Уже стало скучно – осмотрелись по сторонам, посмотрели на часы. Посмотрели на светофор – красный. Полезли в карман за плеером, включили другую песню, но все равно стоим ждем. И так до тех пор пока на светофоре не загорится зеленый свет. То есть здесь окончание циклического процесса (ожидания) зависит не от того, сколько раз уже этот цикл выполнен, а от какого-то условия (в данном случае – от цвета светофора).




```
while условие do
  действие;
```

Механизм действия цикла WHILE: пока выполняется условие, будет выполняться и действие. Практически также работает и цикл REPEAT-UNTIL: действие будет выполняться до тех пор, пока не будет выполнено условие.

```
repeat
  действие;
until условие;
```



Для этих двух циклов стоит отметить следующее:

- количество повторений цикла заранее неизвестно, но при этом так как в REPEAT-UNTIL условие проверяется после выполнения действия, то действие в любом случае будет выполнено хотя бы раз; из тех же соображений делаем вывод, что в цикле WHILE действие может не выполниться и ни разу
- в цикле WHILE *действие*, как обычно, либо отдельный оператор, либо нечто заключенное в операторные скобки; конструкция же REPEAT-UNTIL сама заменяет операторные скобки, поэтому на месте *действия* может стоять любое количество операторов

В качестве примера приведем программу, которая запрашивает с клавиатуры натуральное число и подсчитывает количество цифр в нём.

```
program cif;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  i,n,k:integer;
begin
  write('Vvedite chislo:');
  readln(n);
  i:=0;
  k:=1;
  repeat
    k:=k*10;
    i:=i+1;
  until n<k;
  writeln('Kol-vo cifr:',i);
  readln;
end.
```

Средства отладки Delphi.

Как только программа выходит за рамки 'HELLO, WORLD!', программист начинает сталкиваться с ситуациями, когда его программа работает не так, как должна. Это может быть и результатом ошибки программиста, и результатом неполного понимания языка – когда программист думает, что какая-то конструкция или компонент работают так, а на самом деле они работаю иначе. Такие ошибки проще всего выловить с помощью встроенных средств отладки.
Порядок действия очень простой. Сначала в исходном коде программы необходимо поставить breakpoint – точку останова. Для этого в любом месте программы (лучше перед предполагаемым местом возникновения ошибки) поставить курсор и нажать F5. Снимается breakpoint той же клавишей. Когда вы запустите свою программу и её выполнение дойдет до этого места, программа остановится и вы перейдёте к окну с исходным кодом. Дальше можно продолжить выполнение программы в пошаговом режиме, нажимая клавиши F7 или F8. Причем шаги буду соответствовать реальному ходу программы и будет видно не напутали ли вы что-то в структуре программы. Если выделить какую-то переменную и нажать CTRL+F5 – появится окошко WatchList и эта переменная уже будет добавлена в список. Редактировать список переменных можно с помощью клавиш INS и DEL. В процессе пошагового выполнения программы в этом окне рядом с именем переменной будет отображаться её значение. Это позволяет увидеть совпадают ли данные, хранящиеся в переменной, с тем, что вы ожидали там увидеть.