Данные типа ARRAY ... OF ...

Рассмотрим вектор точек: (x1,y1), (x2,y2), ... , (xn,yn) , когда точка - представлена двумя действительными координатами . Поэтому мы можем рассмотреть два вектора: <x1 ,x2 , ... , xn> и <y1,y2, ... , yn> для которых установлено неявное соответствие по номеру элемента вектора. В каждом векторе его компонента - скалярная величина. Каждая компонента принадлежит одному и тому же типу. В языках программирования такие структуры данных называются *массивами* - это совокупность однородных (однотипных) элементов. Массив не скалярное данное, это составное данное имеющее определенную организацию.

Над массивами определена операция:

- выборка компонента из множества - нужно указать имя массива и порядковый номер компоненты, значением операции является выбранная компонента массива.

Синтакис_описания_типа_для_массивов.

TYPE

<имя> = ARRAY [<тип_индекса>] OF <тип_компоненты>

Семантика_описания_массива.

Имя - это имя типа для массива, который вводится в разделе type. Тип_индекса - это множество значений, которые может принимать индекс (порядковый номер) компоненты массива. Тип_компоненты - это тип данных, которому принадлежат значения компонент массива . Синтаксически тип индекса может быть задан по имени , а может быть анонимным.

Ограничения:

Тип_индекса - это только любой ординальный тип. Тип_компоненты - любой предопределенный или ранее введенный программистом.

Примеры:

1.

```
TYPE
RAS = 1 ..100;
TARR = array [RAS] of real;
VAR
    x: TARR;  y: TARR;
```

Оператором TYPE введены имена для типа индекса и имена для типа массивов. Оператором VAR определены массивы X и Y. Каждый из которых состоит из 100 элементов типа real. Можно и так

2.

```
TYPE
    TARR = array [1 .. 100] of real;
VAR
    x, y: TARR;
```

или так

3.

```
VAR
x, y: array [1 .. 100] of real;
```

Лучше определить тип для массивов, а потом использовать его для определения самих массивов.

Массив, тип компоненты, которых является скалярным, называется одномерным или вектором.

Примеры:

```
TYPE
    prim1 = array [-35 .. -10] of char;
    prim2 = array [‘a’ .. ‘i’] of char;
```

Синтаксис_операции_над_массивами:

Операция селекция -

<имя_массива> [< индексное_выражение>] , где индексное_выражение - любое выражение совместимое по присваиванию с переменной типа индекса для этого массива. После селекции компонента может использоваться во всех операциях как скалярная величина соответствующего типа.

Пример:

```
write(x[1]);
```

Существуют массивы с более сложной организацией: двухмерные -вектор векторов, трехмерные - вектор двухмерных массивов и т. д. Количество измерений ограничено реализацией.

*Синтаксис_описания_многомерных_массивов_*:

TYPE

<имя_типа> = array [<тип_индекса1>] of array [< тип_индекса2>] of <тип>;

Семантика_:

Аналогично одномерному массиву, но описывается структура двухмерная, моделью ее является матрица.

Известно также, что

array [<тип_индекса1>] of array [<тип_индекса2>] of <тип> ≡

array [<тип_индекса1>,<тип_индекса2>] of <тип> .

Описание трехмерных массивов и массивов с большим числом измерений идентично двухмерному с учетом количества индексов. В оперативной памяти все компоненты массива размещены последовательно, без пропуска участков памяти между компонентами.

Пример:

Вычислить произведение квадратных матриц размером n x n.

```
        <шапка>
    CONS
        N = 10;
    TYPE
        ar = array [1 .. N, 1 .. N] of integer;
    VAR
        a, b, c : ar;
        i,j, k : integer;
BEGIN
    for i:= 1 to N do  begin
        for j := 1 to N do
            read(a[i,j]);
            readln();
                End;
```

```
for i:= 1 to N do  begin
                              for j := 1 to N do
                                          read(b[i,j]);
                                          readln();
                                                End;
                             for i:= 1 to N do
                              for j := 1 to N do
                                    begin
                                       c[i, j] := 0;
                                        for k := 1 to N do
                                           c[i, j] := c[i, j] + a[i, k] * b[k, j];
                                   end;
for i:= 1 to N do  begin
                              for j := 1 to N do
                                          write(c[i,j]);
                                          writeln();

END.
```

Строковый тип данных

В диалекте введен тип string вместо описанного выше . Тип данных string иногда называют стринговым.

Пример:

```
VAR
    Name : string [20];
```

Память , отведенная для хранения значения переменной Name, составляет 21 байт, 1 байт содержит текущую длину строки. В каждом байте хранится одна литера . Каждая литера представляет собой значение типа char.

В стринговых выражениях используется операция конкатенации (слияния), которая обозначается знаком “+” .

Пример:

```
Name := Name + ‘пять’;
Name:= ‘строка’;
```

Любая строка символов - это массив символов и только.

К сожалению, в стандарте Паскаля нет специально определенных функций для работы над строками:

- определить с какой позиции строки y строка x входит в y;
- определить входит ли строка x в строку y;
- определить содержит ли строка x символы некоторой строки y и т. д.

Для этих целей используется библиотека.

К строкам применимы все шесть операций отношений, но строки при этом должны иметь одинаковую длину.

Пример:

Написать программу, которая читает строку символов не более 80 и определяет частоту вхождения в строку каждой буквы латинского алфавита от “a” до “i”.

Схема:

```
i:= 0; <установить первую позицию>
    repeat i:= i+1;
          <читать>(ai);
           < увеличить счетчик для   ai  на 1 >
    until ( ] (i< 80 и ord(ai) ≠ 255));
           <печать результатов>
```

Программа:

```
program num;
TYPE
      arcout = array [char] of integer;
 VAR
   symb: char;{читаемый и анализируемый элемент  }
   count: arcount;
   str : string [80]  ; { строка}
   i, j : integer;
 BEGIN
                                {обнулить массив счетчиков }
      for symb := chr(0) to chr(255) do
      count [symb] := 0;
  i:= 0;
```

```
repeat
        i:= i + 1; read( str[i]);
        count[str[i]] := count[str[i]] + 1;
   until ( not( (i<80) and ( ord(str[i]) <>255 );
   writeln(  ‘в этой строке’);
        for j := 1 to i do
             write( str[j]);
             writeln(‘символ встречается’ );
             for symb := ‘a’ to ‘i’ do
               write(‘’, symb, ‘’, count[symb]:10);
END.
```

Записные типы

В языке можем выделить виды данных , используя критерий сложности строения данных: скалярные и составные.

К скалярным можем отнести : целочисленные, реальные, логические, литерные, предопределенные данные, перечислимые, диапазонные.

К составным данным пока - массивы, строки, файлы.

Простые состоят из одного значения базового типа . Составные из одного или нескольких значений не обязательно скалярного типа. Для массива характерно , что все его компоненты однотипны.

Во многих задачах однотипность элементов естественна.

Но в природе есть объекты, которые имеют неоднородные свойства, и описать их однотипной матрицей невозможно.

Пример:

Анкета сотрудника:

1) порядковый номер N - число;
2) ФИО - фамилия - строка;
   имя - строка;
   отчество - строка;
3) дата рождения - день - число от 1 до 31,
   месяц - число от 1 до 12,
   год - число от 1920 до 2000;
4) пол - 0 или 1.

В теории данных для моделирования объектов с различными по виду свойствами используют понятие структуры данных (или агрегат данных).

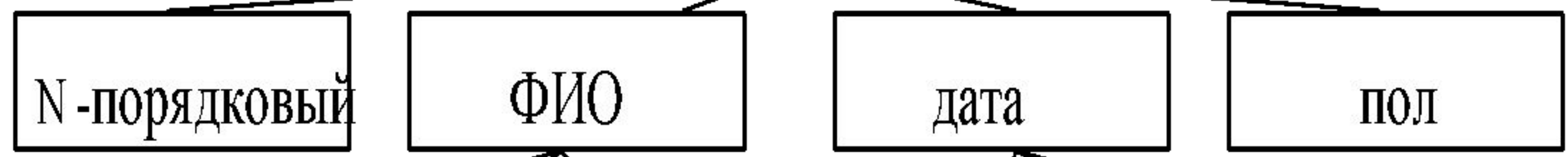
К сожалению, термин структура данных имеет два смысла: широкий - он означает строение данного, отношение между компонентами данного и узкий -данное обладает структурой.

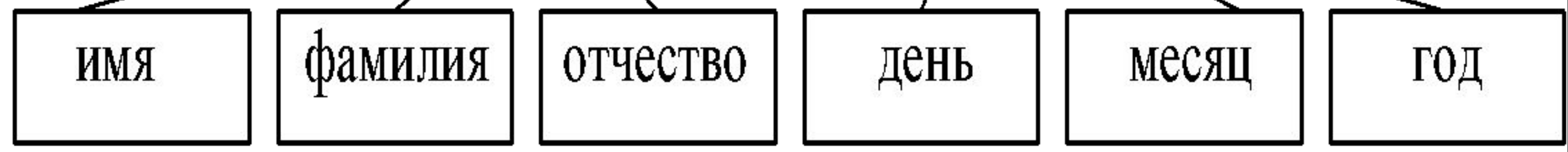
Так, например, в широком смысле массив имеет двухмерную структуру, его данные расположены и по вертикале и по горизонтали.

Анкета - сотрудник имеет иерархическую структуру:

АНКЕТА - 0 уровень

1 уровень





2 уровень

Каждая компонента анкеты называется полем. Некоторые поля могут быть составными, тогда возникает следующий уровень.

Очевидно, что если поле i - уровня составное то его компоненты расположены на i + 1 уровне, а скалярное поле -i -го уровня не имеет компонентов и, следовательно, для него отсутствует i + 1 уровень. Важно понять, что данные последних скалярных уровней образуют совокупность данных, которое рассматривается как единое целое.

{5, Кураедов , Илья, Александрович, 01,01, 1967, 1} - анкета.

Некоторая группа данных, например {Кураедов , Илья, Александрович}- новое данное, т. е. в структуре существует подструктура, но каждая структура - это совокупность скалярных данных входящих в структуру не только как непосредственные компоненты, но как компоненты подструктур.

Данные вида record подразделяются на типы record , каждый тип - бесконечное множество (в практическом смысле). Данные типа record относятся к записным типам. Строение компонент множества задается описанием:

TYPE

имя_типа_вида_RECORD =

RECORD

список_имен_полей $_1$: тип $_1$;

список_имен_полей $_2$: тип $_2$; } секция

...

список_имен_полей $_n$: тип $_n$

END;

В типе RECORD теоретически может и не быть секций, но на практике таких записей не применяют, поэтому будем считать $n \geq 1$. В секции записи может быть одно или несколько полей. Каждое поле задано именем - идентификатором (в синтаксическом плане), т. к. тип может быть любым (кроме определяемого) и обязательно ранее определенный, то возможно объявление одним типом многоуровневых записей с помощью одного оператора.

На практике желательно описать $тип_i$ - явно, а затем использовать в RECORD ... END по имени, т. е. использовать идеологию типизации языка Паскаль.

Пример:

```
{тип 3-го уровня}

NameType = packed array [ 1 .. 30];
DayType = 1 .. 31;
MonthType = 1 .. 12;
YearType = 1920 .. 2000;

{тип 2-го уровня }

FioType = record
              fam: NameType;
              nam: NameType;
              otch: NameType
          end;
DataType = record
              day: DayType;
              month: MonthType;
              year: YearType
           end;
```

{тип 1-го уровня }

```
AncetType  =  record
                 number: integer;
                  fio: FioType;
                  data: DataType;
                  pol: boolean
               end;
```

Теперь объявим запись - массив записей:

```
VAR
     AncetOne : AncetType;
     AncetSot: array [ 1 .. quty] of AncetType;.
```

Над данными класса *record* определены операции:

1) := - присваивания :

 данные данные AncetSot[k]:= AncetOn

одного типа record

AncetOne:= AncetSot[k];

2) селекция - выделение поля записи “.”. Имеет два операнда : левый - имя записи, правый - имя поля в этой записи, при этом если имя записи - на i уровне, то имя поля- на i + 1 уровне:

AncetOne.day - скаляр.

AncetOne. Data - структура,

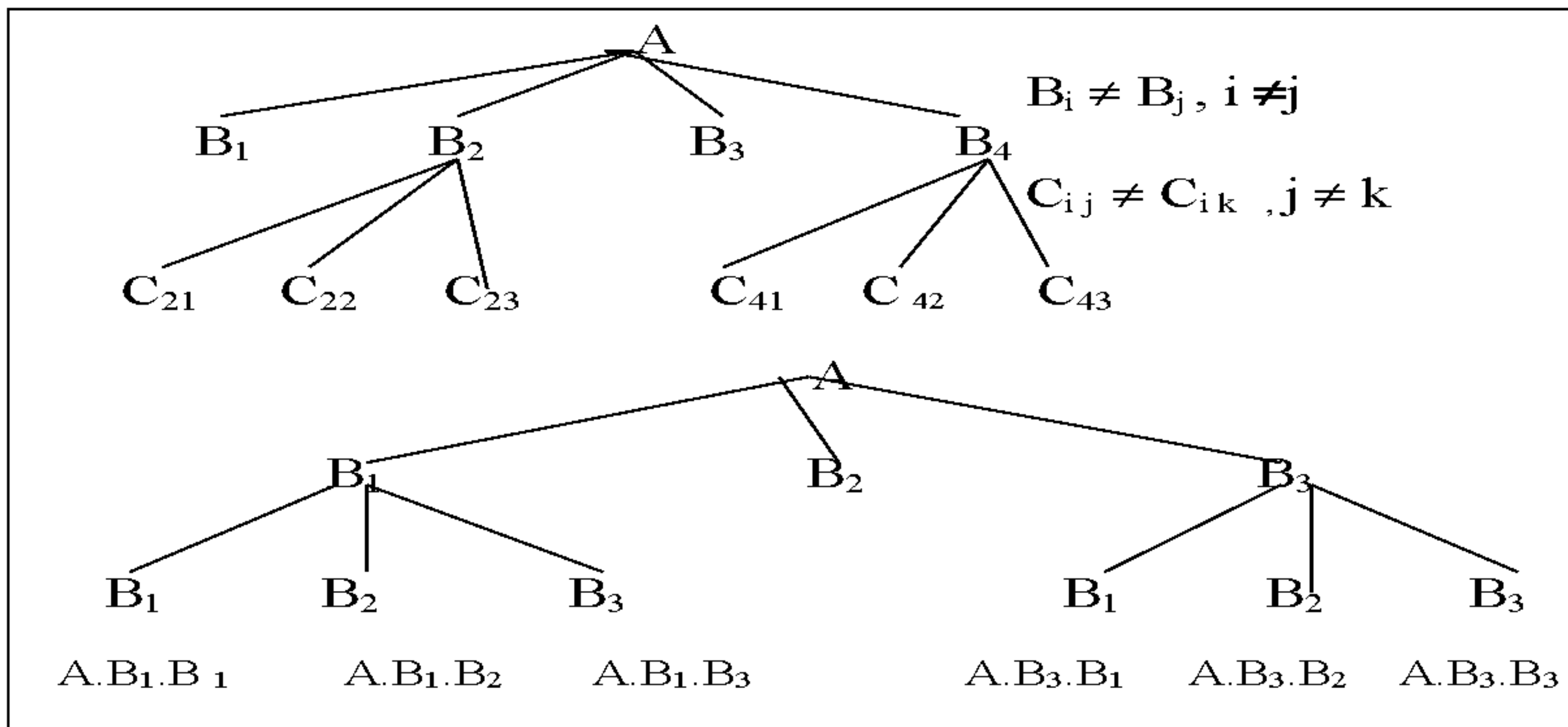
AncetOne.fio - структура.

Инициализировать (присвоить первоначальное значение) всех полей структуры можно только по полям скалярного вида:

read(AncetOne.day, AncetOne.month);

и т. д. , если нужно массив - то используется цикл.

Обратите внимание, что имена полей одного уровня в записи, если они принадлежат всей записи , должны быть уникальны по любой ветви иерархии.



Любая скалярная компонента записи может быть использована там, где допускается применение однотипной скалярной переменной.

Строение одного объекта можно моделировать разными способами, но все они дают разные типы для одного объекта, какой способ использовать зависит от опыта.

Подпрограммы

Решение каждой задачи (подзадачи) можно оформить в виде подпрограмм. *Подпрограмма* - часть программы, один раз написанная, но выполняемая в программе один и более раз. Естественно, в программе подпрограмма должна выполняться хотя бы один раз. Если она не выполняется ни разу при запуске программы, то она и не нужна.

Причины, обуславливающие применение подпрограмм в программе:

1). Размер программы. Если в программе достаточно много мест (≥ 2), где нужно выполнять одни и те же действия с одними и теме же данными, или с другими данными, но с одной и той же структурой, то эти действия желательно оформить в виде подпрограммы. Описать один раз в тексте программы, а там где в тексте программы идет ее вызов (обращение к ней):

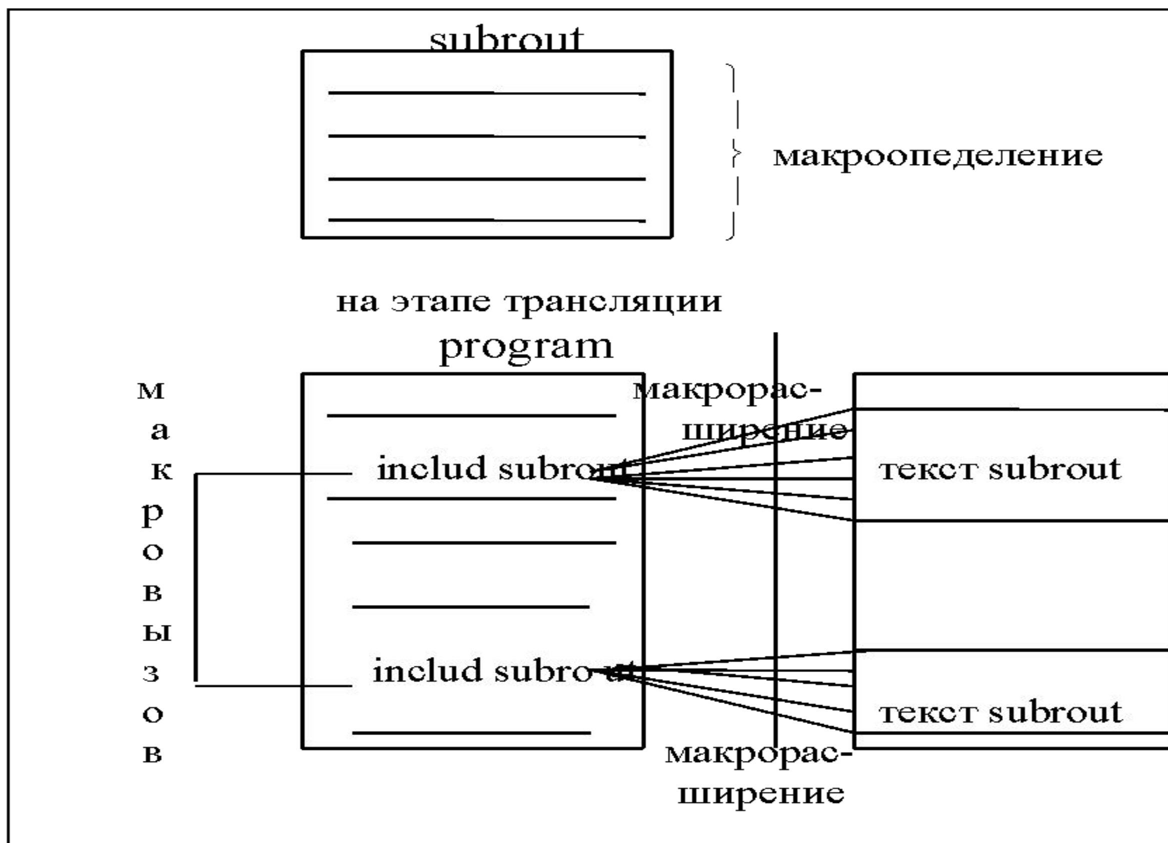
а) либо на этапе трансляции подставить текст подпрограммы;

б) либо передать управления модулю -подпрограммы.

Если применяется подстановка текста подпрограммы, то такую подпрограмму называют *макроопределением* , а сам процесс подстановки - *макроподстановка* , результат подстановки - *макрорасширение* .

Многие языки программирования используют аппарат макроподстановки, особенно языки ассемблера. В стандарте Паскаля макроподстановки не используются.

Пример

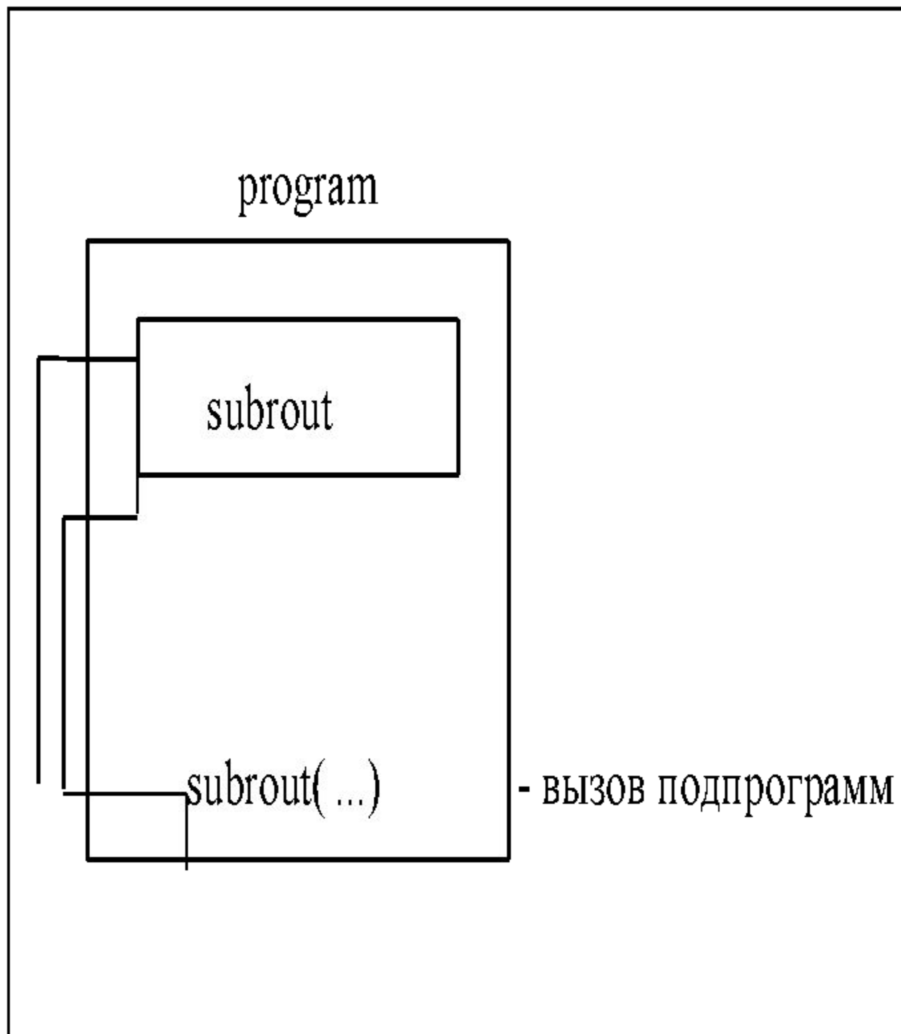


Аппарат макроподстановок хорош тем, что программист имеет возможность создать большие по размеру программы, используя небольшие тексты определений макро и исходной программы.

Непосредственная подстановка в текст программы макрорасширения, увеличивает текст и возможно в 2 - 4 раза, но скорость выполнения программы может быть наивысшей.

Если использовать обращение к подпрограмме то:

а) где-то вне программы или внутри нее имеется текст подпрограммы;



program
subrout
subrout(...)
- вызов подпрограмм

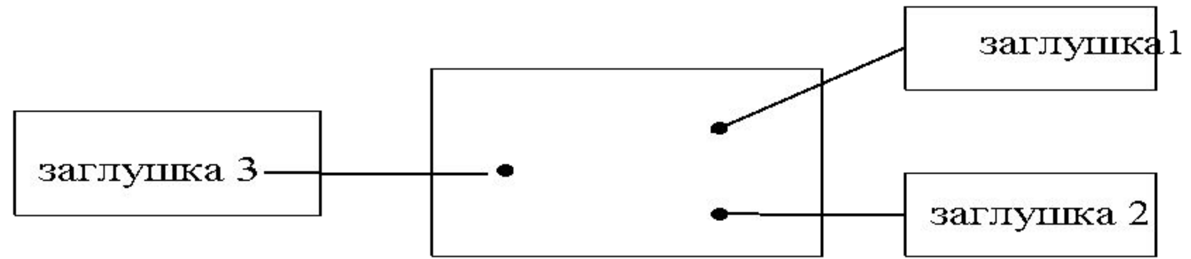
б) где-то вызов подпрограммы или таких вызовов может быть несколько.

Выполняя, операторы программы в заданной последовательности, доходим до обращения к подпрограмме. В этом месте осуществляется “переход” на начало подпрограммы subrout, и она выполняется в последовательности, определяемой логикой программы и исходными данными. Как только будет достигнут оператор return или конец подпрограммы, то осуществляется “переход” на оператор , следующий за оператором вызова подпрограммы, и программа продолжает свое выполнение .

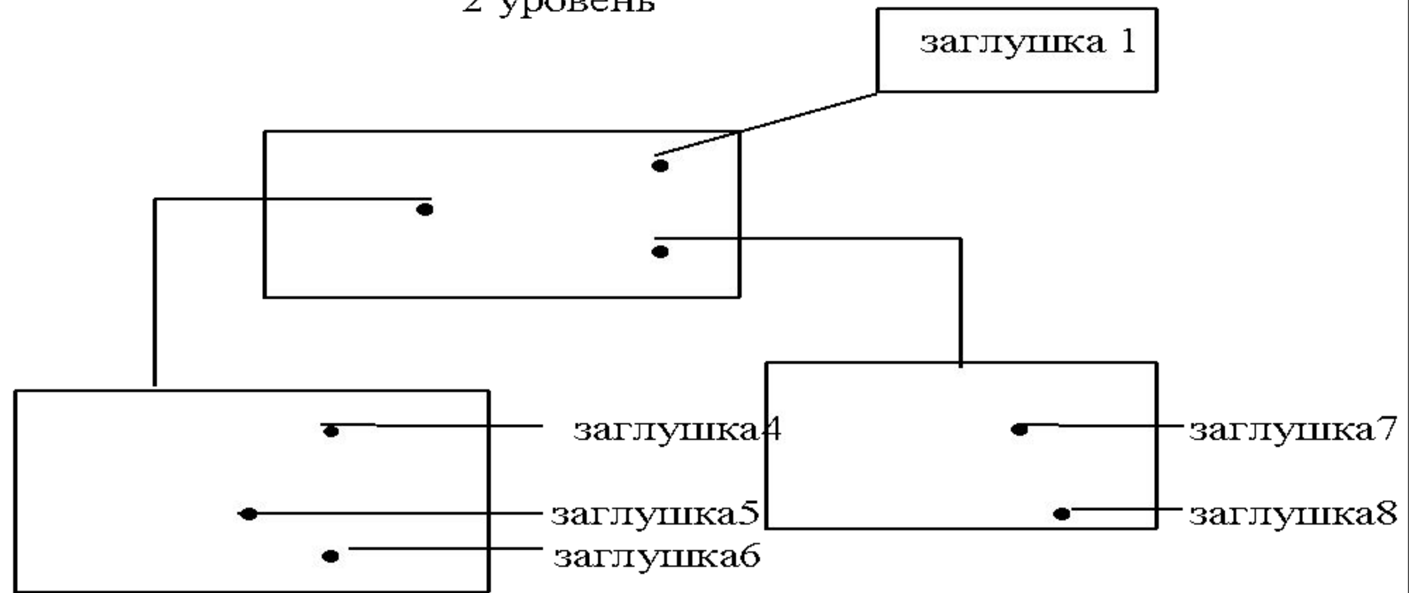
Здесь, в отличие от макроподстановки, текст не вставляется в тело программы в точке вызова подпрограммы. Вместо этого в программе на этапе трансляции генерируются коды перехода на подпрограмму, и сама подпрограмма рассматривается как некоторая отдельная часть программы, в которой наряду с обычными операторами имеются коды операторов возврата в программу. Выигрыш по длине текста очевиден, проигрыш по времени - лишние операторы перехода к подпрограмме и обратно, лишние операторы для организации интерфейса между программой и подпрограммой.

2). Упрощает процесс разработки сложных или больших по объему программ. Разработка программ в этом случае соответствует некоторой технологии программирования согласно которой, достаточно объемная задача программы может быть выделена в самостоятельную программную единицу типа программа. Поэтому, если используется технология программирования “сверху вниз” (пошаговая детализация), то можно создать на первом уровне приемлемую по восприятию программу, где достаточно сложные задачи решаются подпрограммами. Причем необязательно эти подпрограммы разрабатывать немедленно. Их разработку можно отложить на более позднее время, заменив их для начало “заглушками” - подпрограммы имитирующие основные результаты соответствующих подпрограмм. Если и сама подпрограмма окажется сложной, процесс можно повторить.

1 уровень



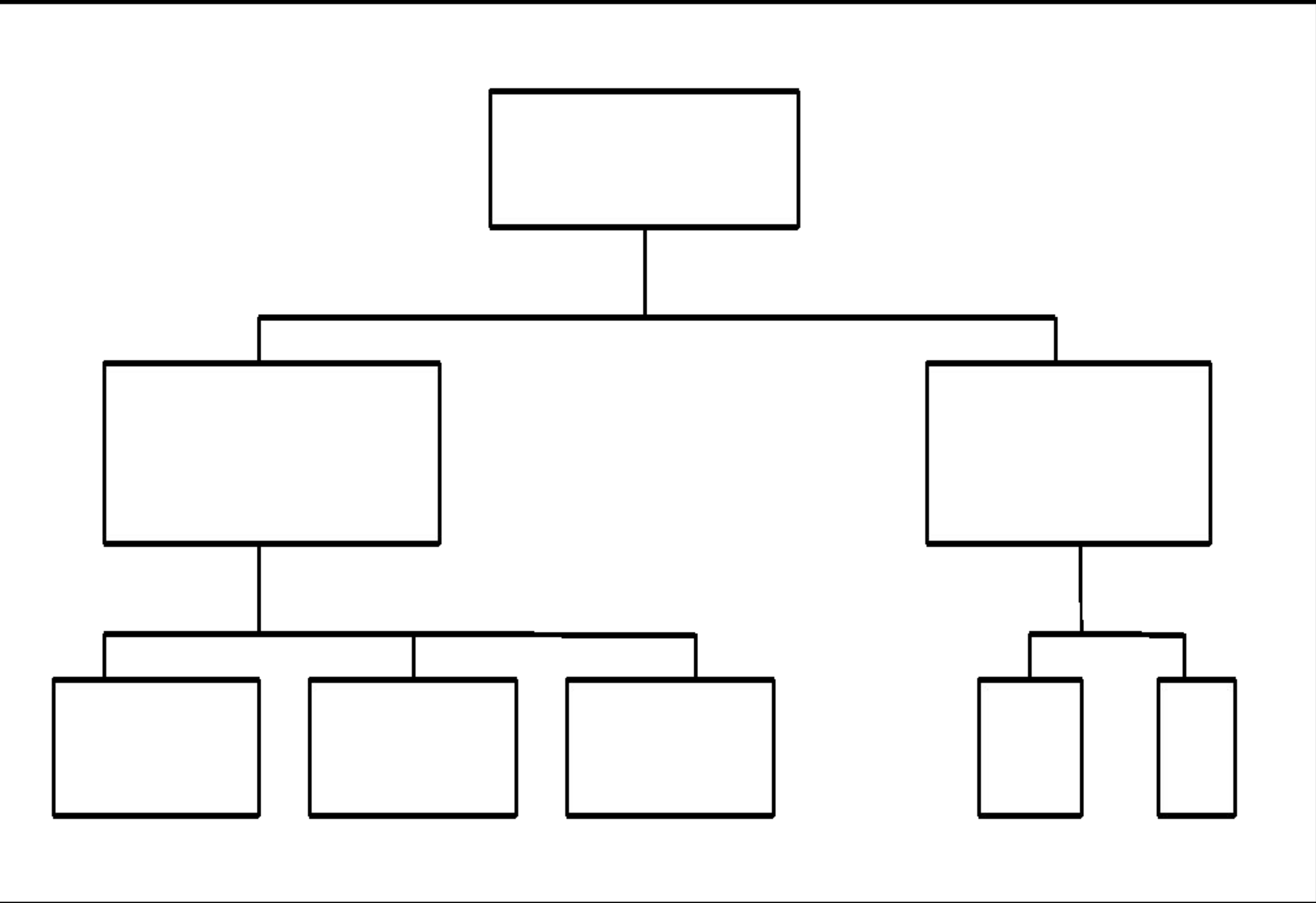
2 уровень



При технологии создания программ “снизу вверх” сначала разрабатывают подпрограммы самого низкого уровня, т. е. те, которые не обращаются к другим подпрограммам. Затем созданные подпрограммы собираются в некоторый набор других подпрограмм и т. д. пока не будет написана вся программа.

Для отладки подпрограмм низкого уровня используется *драйвер* - программа моделирующая внешнюю среду : исходные данные, общие таблицы и т. п.

Бегло рассмотренные подходы к созданию программ с применением подпрограмм можно сравнить с подходом , когда создается текст программы без подпрограмм. Начиная с некоторого момента программист теряет контроль за текстом программы т. к. психологически и физически не в состоянии осмыслить большую по объему информацию, которая заложена в программе. Другое дело, когда информация выделена в порции, она без лишнего напряжения воспринимается полностью. Здесь программист как бы мыслит в терминах подпрограмм, а не их текстов, и следовательно, легче воспринимает программу в целом, чем в мелких деталях.



3). Удобство поиска и исправления ошибок.

4). Проще процесс тестирования программ

и т. д.

Практически любой язык программирования имеет аппарат подпрограмм. Поэтому при изучении подпрограмм следует обращать внимание на :

1) .Синтаксис описания подпрограмм;
2) .Синтаксис обращения к подпрограммам;
3) .Область *видимости* переменных, используемых в подпрограмме. Это понятие означает некоторую часть подпрограммы, где видима переменная в момент обращения к ней;
4) . Способы передачи данных от программы к подпрограмме и обратно в момент вызова подпрограмм и возвращение управления программе.

Для начала рассмотрим двухуровневую программу на языке Паскаль, т. е. программа на языке Паскаль состоит из основной программы (главной) и подпрограммы. Согласно стандарту подпрограмма является внутренней, т. е. расположенной внутри главной. По этой причине описание подпрограммы размещается после раздела описания переменных (раздел VAR).

Структура описания подпрограммы

<Заголовок_программы>;
[<раздел меток>]
[<раздел констант>]
[<раздел типов>]
[<раздел переменных>]
[<раздел подпрограмм>]
<begin- блок >

Обратите внимание, что точно такую же структуру имеет описание программы, где begin-блок - begin

<список операторов>

end.

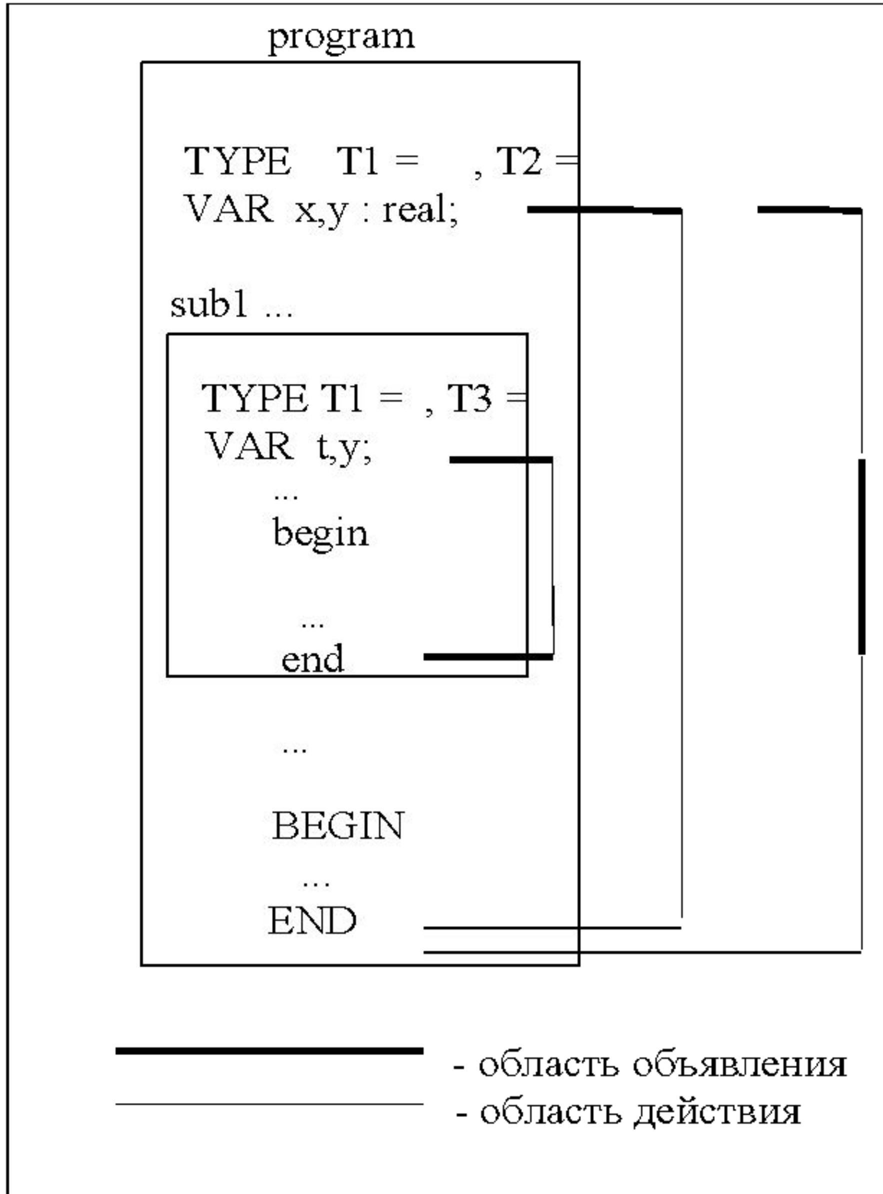
Таким образом, основную программу можно называть главной подпрограммой, она отличается от других подпрограмм на синтаксическом уровне -

program <имя_программы>[(<список_файлов>)], на семантическом уровне - к главной подпрограмме нельзя обращаться из любой другой подпрограммы, т. к. к любой другой подпрограмме можно обращаться из любой, в том числе, из той же подпрограммы и выполнение программы всегда начинается с главной.

Область действия имен

Рассматривая структуру подпрограммы, обратим внимание на begin-блок с точки зрения тех переменных, которые допускается использовать в нем. Вспомним, что в программе (в главной подпрограмме) нельзя использовать имя, если оно заранее не декларировано в одном из 5 разделов, если оно не является предопределенным. Обратим внимание, что внутри begin-блока не могут появиться разделы, следовательно, все описания имен расположены между заголовком и begin-блока процедуры в разделах.

Введем понятие блок - это часть подпрограммы, не содержащая заголовок подпрограммы, поэтому блок состоит из разделов и begin-блока одной подпрограммы. *Область объявления имени* - это часть подпрограммы между точкой объявления имени и последним end или это часть блока, где имя объявлено, начиная с точки объявления.


program
TYPE T1 = , T2 =
VAR x,y : real;
sub1 ...
TYPE T1 = , T3 =
VAR t,y;
...
begin
...
end
...
BEGIN
...
END
- область объявления
- область действия

Области объявления имен x, y - программа, t, y - подпрограмма.

Область действия имени (не только переменных, но и типов, констант, процедур и т.д.) - это часть блока, где разрешено использовать имя по первому его назначению.

Имя x в программе - переменная , ее область действия совпадает с областью объявления и ее можно использовать :

а) внутри begin-блока программы;

б) внутри begin-блока подпрограммы.

Имя y - в программе переменная, имя y - в подпрограмме переменная y' , область действия y' - только begin-блок подпрограммы, область действия y - begin-блок программы, но не begin-блок подпрограммы. Это правило касается имен констант, меток. Поскольку имена типов появляются в разделах, а не в begin-блоках, то правило определения области действия сохраняется в общем случае , но в терминах разделов.

В языках программирования для характеристики имени (переменной) по месту расположения его в программе используют термины локальное или внутреннее, глобальное или внешнее имя.

1). Если имя f объявлено в подпрограмме x, то оно локально (или внутренние) в этой подпрограмме.

2). Если имя f локально в подпрограмме x , но используется и во внутренней по отношению к x подпрограмме y, то оно внешнее по отношению к y.

Подпрограммы с параметрами

procedure<имя_процедуры>(<список_формальных_параметров>);
<блок>;

Список_формальных_параметров - это последовательность из одного или нескольких подсписков. Подсписки, если их несколько разделены “;”.

<список_формальных_параметров>⇒
<подсписок_формальных_параметров$_1$>; ... ;
<подсписок_формальных_параметров$_n$>

Каждый подсписок имеет структуру:

а) [VAR] < список_имен> : <имя_типа>,
б) <заголовок подпрограммы>.

Изучим сначала подсписок вида а). Подсписок имен содержит одно или несколько имен одного типа. Разделитель имен - запятая.

Пример:

```
x1, x2, x3 : real
Var  x4,x5 : char
Var x6:  DayType
```

! После списка имен не может быть использован анонимный тип, только имя типа. Нет ограничения на типы параметров, т. е. в качестве параметра можно использовать переменную любого типа. Поэтому в стандарте должен использоваться массивный тип, а не его задание (т. е. тип_массив должен быть определен заранее) :

```
procedure NoCorrect ( Var A : array [ 1 .. 10 ] of real);
```

Вызов процедуры:

```
<имя_подпрограммы>(<список_фактических_параметров>);
```

Если в списке фактических параметров несколько элементов, то они разделены “,”. Число элементов списка фактических параметров равно общему числу формальных параметров. При этом одному и только одному формальному параметру соответствует один и только один фактический, причем соответствие - по порядку следования элементов обоих списков. А так же должны соответствовать:

а) по типу - фактические и формальные параметры совместимы по присваиванию,

б) по виду значения.

Если перед подсписком формальных параметров стоит слово Var в заголовке процедуры, то соответствующий ему список фактических параметров должен состоять только из имен параметров-переменных.

Procedure PR3(Var f1,f2: char); - заголовок.

PR3 (v5.x1, v5.x2); - вызов.

Если же перед подсписком формальных параметров нет Var, то соответствующий список фактических параметров может содержать выражения (константы - частный случай) - параметр-значение.

Procedure PR1 (v1, v2, v3); - заголовок,

PR1 (v1,v2, v3); - вызов.

Слово Var в начале подсписка формальных параметров или его отсутствие определяет способы передачи значения между формальными и фактическими параметрами, т. е. интерфейс между подпрограммами на уровне параметров.

Если отсутствует слово Var перед подсписком, то любому имени-параметру-значению в процедуре отводится память того объема, которая требуется для значения указанного типа. Формальный параметр типа файл не может появиться в таком списке. В момент вызова подпрограммы выполняются служебные подпрограммы, которые вычисляют значения выражений фактических параметров и присваивают их формальным параметрам-значениям как начальные. По окончанию работы подпрограммы значения формальных параметров, даже если они изменялись, назад не передаются соответствующим фактическим параметрам. Такой *способ* передачи значений от фактических параметров к формальным называется передачей *по значению* .

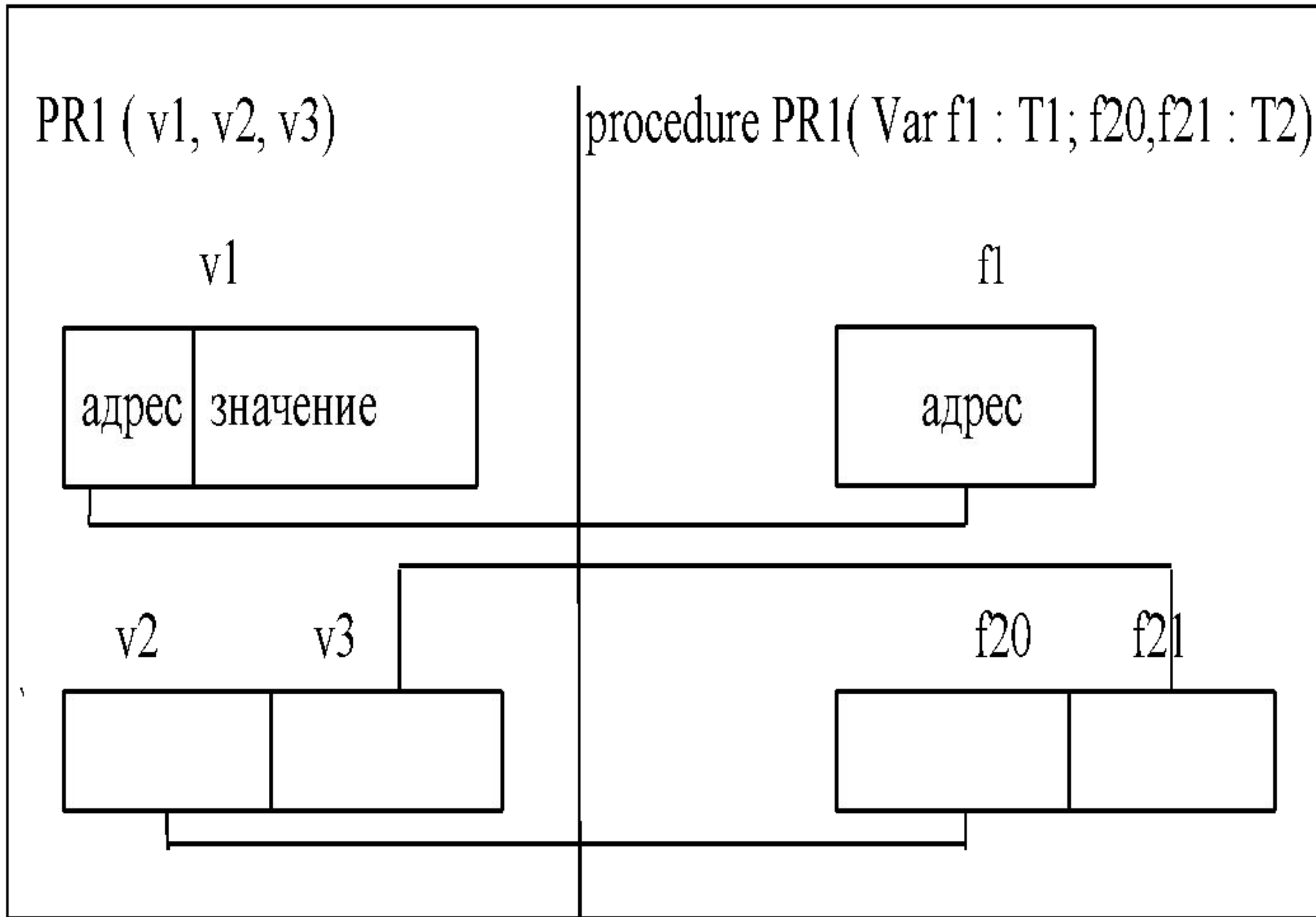
Передача по значению - это передача в одну сторону - от фактических параметров к формальным.

Определение:

Если подпрограмма при своем выполнении изменяет переменную внешнею (или глобальную) , то она обладает *побочным эффектом*.

Фактические параметры при передачи “по значению” защищены от побочного эффекта.

Передача значения между фактическим и формальным параметром-переменной двухсторонняя, т. е. фактические переменные при передаче “по ссылке” не защищены от побочного эффекта.



значение

значение

значение

значение

При передаче “по ссылке” в процедуре создается поле-указатель на структуру данных указанного типа. В момент вызова подпрограммы адрес (а не значение) фактического параметра передается в качестве значения формальному параметру. Подпрограмма, выполняя операции с формальными параметрами, на деле выполняет их с фактическими, следовательно если меняется значение формального, то меняется и значение фактического параметра.

Выводы:

1). Передача значений от фактического к формальному параметру может быть двух видов: “по значению” или “по ссылке”;

2). Если планируют передачу по значению, то в заголовке не используют Var, иначе (передача по ссылке) - Var;

3). Если формальному параметру-значению присвоить любым способом значение, то это не изменит соответствующий ему фактический параметр, поэтому фактический параметр может быть любым выражением совместимым по присваиванию с формальным параметром;

4). Если формальному параметру-переменной присвоить значение любым способом, то оно становится значением фактического параметра;

5). Формальный параметр-значение может быть любого типа, кроме файлового или составного типа, содержащего файловый тип.

Любой вызов процедуры при выполнении программы *активизирует* эту процедуру, это значит:

1). Все локальные переменные имеют память, но не имеют значения ;

2). Все глобальные переменные сохраняют память и значения перед началом выполнения первого оператора подпрограммы;

3). Устанавливается связь между фактическими и формальными параметрами по ссылке или по значению;

4). Управление передается первому оператору в begin-блоке процедуры.

Как только процедура закончила работу - дошла до конца begin-блока , то она *деактивизируется* :

1). Все локальные переменные теряют память, и они становятся недоступными;

2). Все внешние переменные сохраняют память и имеют те значения , которые могли получить при выполнении процедуры, или значения сохраняются, если процедура их не меняла;

3). Фактические параметры, переданные по ссылке, сохраняют те значения, которые они получили (или не получили) при выполнении процедуры;

4). Фактические параметры, переданные по значению , не изменяют своих значений;

5). Управление передается в вызывающую процедуру - к первому оператору, следующему после вызова процедуры.

Перейдем к рассмотрению подсписка формальных параметров процедуры вида б) (параметр-процедура).

Параметром процедуры может быть процедура. В этом случае список формальных параметров содержит подсписок в виде параметр-процедуры. В этом подсписке только один параметр - процедура:

procedure<имя_параметра_процедуры>[(список_формальных_параметров)]

Фактический параметр - имя конкретной процедуры без параметров.

Пример:

```
            ...
    procedure D ( x: real; Var y: real);
        begin
            y:= x * x;
        end;
            ...
procedure REZ ( A: mas; Var rel: real; procedure CVAR (b: real;
                                        Var c: real); n: integer);
     VAR
        i: integer;
        r: real;
     begin
        r:= 0;
        for i:= 1 to n do
           begin
              CVAR ( a[i], c);
              r:= r +c ;
           end;
              ...
        REZ( b, rel, D, 10);            {обращение}
              ...
```

Функции

Функции - это подпрограммы, которые возвращают одно скалярное значение в вызывающую программу (подпрограмму) через свое имя.

Во многом описание функции совпадает с описанием процедуры, но есть и отличия.

Заголовок функции:

function<имя_функции>[(<список_формальных_параметров>)]
:тип;

Если отсутствует список формальных параметров, то это функция без параметров, иначе - правила описания списка параметров идентичны правилам процедуры.

Если нет опережающего описания (forward), то имя функции обязательно - оно задает тип возвращаемого значения.

Пример:

```
function min (x, y : real) : real;
  begin
    if x> y then min:= x else min:= y;
  end;
```

В теле функции обязательно должен присутствовать хотя бы один оператор присваивания вида:

<имя_функции> := <выражение>;

Функция завершает свою работу по достижению конца своего блока.

Обращение к функции называют указателем функции. Указатели употребляются только в выражениях. При этом тип функции не должен противоречить той операции, в которой функция использована как операнд. Все остальное сказанное о procedure верно и для function.

Рекурсивность подпрограмм

Рекурсивная подпрограмма (процедура, функция) предназначена для решения задачи путем многократного повторения одного и того же метода. В некотором смысле рекурсивность похожа на цикл типа while<условие> do. Поэтому в рекурсии нужно соблюдать два требования :

а) необходимо условие окончания рекурсии;

б) каждый шаг рекурсии должен приближать к условию окончания рекурсии.

Пример:

1. Ввод строки с клавиатуры и ее распечатка в обратном порядке при помощи смешанной рекурсии (рекурсивный процесс длится, пока не нажата клавиша ENTER (конец вводимой строки)).

```
Program Main_program;

procedure Down_Up;

var ch: char;

begin if not Eoln then // условие продолжения рекурсивных вызовов
```

```
 begin read(ch);
                                   // BLOCK_1 (действия ДО рекурсивного вызова)
 Down_Up;                  // рекурсивный вызов
write(ch);
                                 // BLOCK_2 (действия ПОСЛЕ рекурсивного вызова)
 end;
end;
begin Down_Up; end.
```

2.«Ханойские башни». Имеются три колышка 1, 2 и 3 и n дисков разного размера, перенумерованных от 1 до n в порядке возрастания их размеров. Сначала все диски надеты на колышек А. Требуется переставить все диски с колышка А на колышек С, соблюдая при этом правила: за один раз можно перенести только один диск; больший диск нельзя ставить на меньший.

```
program Towers; // Ханойские башни
 type Position = (Left, Center, Right);
 var N : integer;
procedure MoveDisk(From, Tol : Position);
procedure WritePos(P : Position);
begin case P of
Left : write('1');
Right : write('3');
Center : write('2');
 end;
end;
```

```
begin WritePos(From);
write('-');
 WritePos(Tol);
writeln;
end;
procedure MoveTower(Hight : integer; From, Tol, Work : Position);
begin if Hight > 0 then begin MoveTower(Hight-1, From, Work, Tol);
 MoveDisk(From, Tol);
 MoveTower(Hight-1, Work, Tol, From);
end;
 end;
begin readln(N); MoveTower(N, Right, Left, Center); end.
```

Каждая активация рекурсивной подпрограммы создает в оперативной памяти новые локальные переменные с неопределенными значениями и формальные параметры согласно способу передачи значения и сохраняет старые локальные переменные и фактические параметры, таким образом при рекурсии идет дополнительный расход оперативной памяти.