**Подпрограммы**

Подпрограмма – поименованная логически законченная группа операторов языка, которую можно вызвать для выполнения по имени любое количество раз и различных мест программы.

Подпрограмма:

* Группа операторов
* Есть имя
* Имеет логический конец
* Несколько раз вызвать можно

Виды подпрограмм:

* Процедуры
* Функции

Виды подпрограмм:

* Встроенные
* Определенные пользователем

Крч подпрограмма как программа, такая же, но без uses части вроде

Блок: Тело модуля + раздел операторов

Блоки:

* Глобальные
* Локальные

Элементы программы (типы, переменные, константы, …)

* Глобальные
* Локальные

*Принцип локализации имён*

Имена, вводимые в употребление в подпрограмме, имеют силу только в данной подпрограмме (и вложенных в нее)

Если имя, вводимое подпрограммой, совпадает с именем из внешнего блока, оно перекрывает это имя

В каждой подпрограмме можно вводить свои идентификаторы

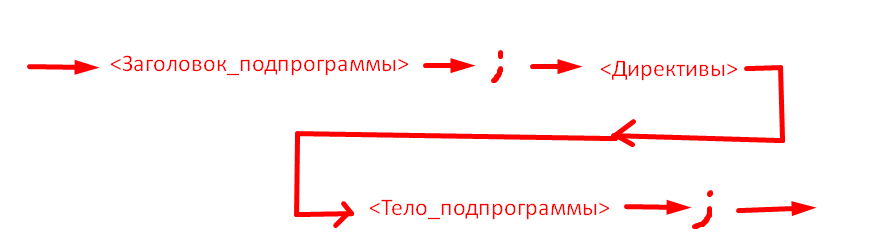
Область видимости

Область видимости (действия) имени:

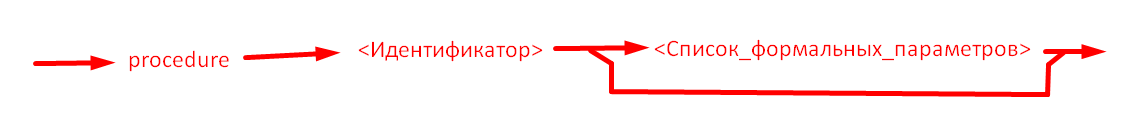
* От объявления до конца текущего уровня вложенности:
  + Включая вложенные блоки
  + Кроме вложенных блоков, где объявлено такое же имя

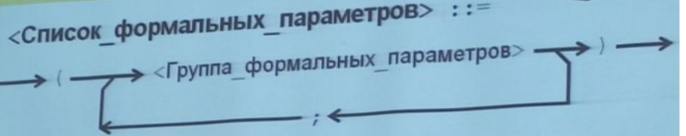
Повторные объявления имён в пределах блока не допускаются

<Объявление\_подпрограммы> ::=

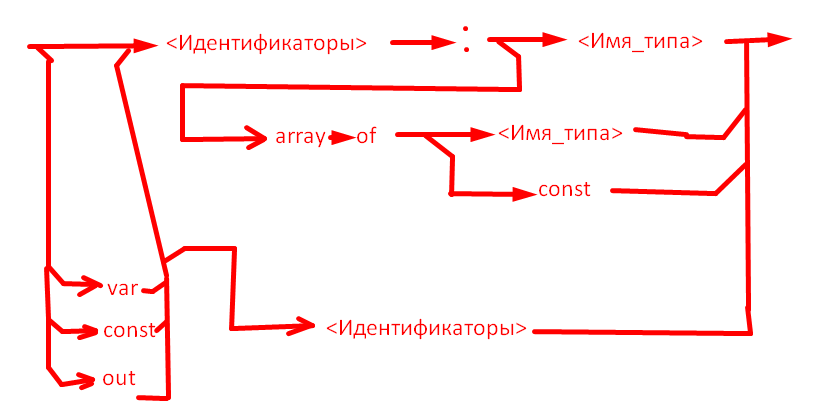


<Заголовок\_процедур> ::=





<Группа\_формальных\_параметров> ::=



Процедуры без параметров – редко юзают

* В большинстве случаев нужны исходные данные
* Передача исходных данных через глобальные переменные – дурной тон

Недостатки передачи параметров через глобальные переменные:

* Процедуру сложно повторно использовать в другой программе
* Если несколько процедур использует одну и ту же глобальную переменную, логика программы усложняется
* Меньшая гибкость по сравнению с другими способами передачи параметров

*Виды параметров*

В языке Pascal:

* Параметры-значения
* Параметры-переменные
* Параметры-константы
* Параметры без типа

В языке Delphi

* Есть и другие способы передачи параметров

Параметры (формальные/фактические)

Формальные параметр – переменная, которая будет доступна внутри процедуры/функции

Фактический параметр – то, что передано при вызове

*Передача параметров*

Для передачи параметров используется программный стек



Программный стек располагается в оперативной памяти

Отличия от остальных участков памяти:

* В процессоре предусмотрены регистры, задающие расположение стека
* В процессоре предусмотрены специальные инструкции для работы со стеком

*Параметры-значения*

Для параметров-значений при вызове создаётся локальная переменная:

* Существует только во время выполнения процедуры/функции
* При каждом вызове это новая переменная

В заголовке процедуры/функции записывается без модификаторов, указываются только имя и тип данных

Порядок передачи:

1. Вычислить значение фактического параметра
2. Создать локальную переменную с именем формального параметра и присвоить ей значение фактического параметра

В стек помещается копия значения фактического параметра

Фактическим параметром может быть любое выражение того же типа данных, что и параметр

*Параметры-переменные*

В заголовке процедуры записывается с зарезервированным словом var

Фактическим параметром может быть только переменная того же типа данных, что и параметр

В стек помещается адрес переменной-параметра

Изменение формального параметра внутри подпрограммы приводит к изменению фактического параметра

*Параметры-константы*

В заголовке процедуры записывается с зарезервированным словом const

Фактическим параметром может быть выражение того же типа данных, что и параметр

Компилятор запрещает изменение значения внутри подпрограммы

Способ передачи **выбирается компилятором**

Рекомендуется использовать при передаче параметров строкового типа и записей

* Создание копии значения для этих типов более сложно, чем для остальных
* Использование const-параметра позволяет компилятору передать адрес значения, не создавая его копию

*Параметры без типа*

Для всех параметров-значений тип должен быть указан.

Для var- и const- параметров указание типа необязательно

* Параметры, у которых тип не указан, называются параметрами без типа (untyped)s

TBytes(Dest)[I] – явное приведение типа

Приведение переменной Dest к типу TBytes

*Директива absolte*

Позволяет создать переменную, которая будет располагаться в той же области памяти, что и какая-либо другая переменная

S: String[10];

X: Integer absolute S;

Переменная – это не сама область памяти это:

* Идентификатор (имя)
* Его связь с областью памяти

С именем связаны:

* Адрес этой области
* Ее размер в байтах

Обычное объявление переменной заставляет компилятор:

* Выделить область памяти
* «создать» идентификатор и связать его с этой областью памяти

Директива absolute позволяет создавать дополнительные имена для имеющихся областей памяти

S: String[10];

X: Integer absolute S;

RAM

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 8 | Р | Е | К | У | Р | С | И | Я | ? | ? |   S: Sting[10]  X: Intger absolute S (8рек) |

*Использование производных типов в качестве параметров подпрограмм*

Можно.

procedure SomeProc (X: array [1..10] of Integer; N: 1..100);

begin

// asdasd

end; X

type

TArrayParam = array[1..10] of Integer;

TNumber = 1..100;

Procedure SomeProc(X: TArrayParam; N: TNumber);

…

Могут быть:

* Параметрами-значениями
* Параметрами-константами
* Параметрами-переменными

Фактические параметры – только соответствующих типов

* Исключение – строковые константы

Параметры типов, занимающих много места в памяти, нежелательно по значению

**Open array parameters**

В классическом Pascal в заголовке разрешено указывать только идентификаторы типов (по синтаксису)

В Delphi введён специальный синтаксис для передачи массивов произвольного размера

Вместо имени типа записывается конструкция array of <Тип>.

Это НЕ параметр типа «динамический массив»

При вызове подпрограммы:

* Для передачи в стек помещаются две величины:
  + Адрес массива
  + Количество элементов в нем

Если open array constructor:

* Массив формируется в стеке «на лету»

Фактическим параметром может быть:

* Статический массив с тем же базовым типом
* Динамический массив с тем же базовым типом
* Переменная базового типа

Формальный параметр ведет себя как обычный статический массив, но с некоторыми ограничениями:

* Нумерация всегда начинается с 0
* Работать можно только с отдельными элементами, но не с полной переменной
* В другие подпрограммы могут быть переданы только как:
  + open array parameter;
  + параметр без типа.

Procedure Sum(const A: array of Integer; var S: Integer);

var  
 I: Integer;

begin

S := 0;

for I := Low(A) to High(A) do

S := S + A[I];

end;

Sum([1, 10, X + Y, 42], Res);

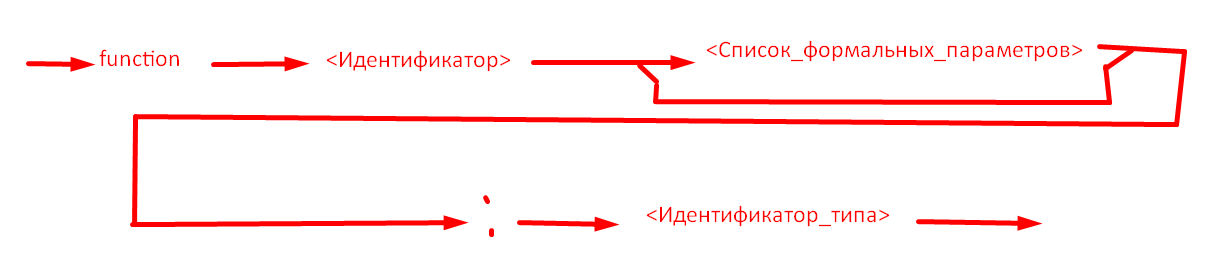
**Функции**

Функция – процедура, возвращающая значение

* Или процедура – функция, не возвращающая значения

Функции аналогичны процедурам, кроме нескольких отличий

<Заголовок\_функции> ::=



Функции могут использовать те же механизмы передачи параметров и результатов, что и процедуры.

Кроме того, каждая функция вычисляет так называемое **возвращаемое значение функции**

Тип возвращаемого значения:

* Может быть:
  + Стандартным типом
  + Предварительно описанным типом
* Не может быть:
  + Заданием типа

Классический способ:

Function GetMax(x1, x2: Real):Real

Begin

If x1 > x2 then

GetMax := x1

Else

GetMax := x2;

End;

В делфи – GetMax заменяется на Result (анти-рекурсия крч)

*Возвращаемое значение*

Оба способа присваивания возвращаемого значения могут использоваться в одной и той же функции

Если возвращаемое значение функцией не было присвоено, оно является неопределенным

Функции (в отличие от процедур) могут использоваться в составе выражений

* Сам вызов функции является выражением, его тип – тип возвращаемого значения функции

Val := GetMax(…) + Z +…

*Процедурный тип*

У процедур и функции есть адреса.

* Адрес процедуры/функции – адрес самого первого байта ее кода

Процедурные типы – типы, значениями которых являются адреса процедур/функций

if F = G then // Переменная процедурного типа в выражении – вызов соответствующей функции

if @F =@G then // Сравнение адресов функций

*Ограничения процедурных переменных*

Нельзя присваивать адреса стандартных процедур/функций

Нельзя присваивать адреса процедур/функций, объявленных с директивой inline

Процедура/функция должна соответствовать процедурному типу

Зачем:

TCompareFunc = function(X1, X2: Real):Boolean

TArray = array[1..100] of Real

Procedure Sort(A:TArray; CompareFunc: TCompareFunc)

…

If CompareFunc(A[I], A[J]) then…

// Смотря как отсортировывать и с чем сравнивать

IoC: Inversion of Control, DI

*Директива inline*

В некоторых реализациях Pascal:

* Позволяла вручную задать машинный код для программы
* Синтаксис был аналогичен вызову процедуры

В ранних версиях Delphi:

* Поддержка удалена

Начиная с Delphi 2007:

* Введена с синтаксисом директивы для подпрограмм

Обычная подпрограмма:

* Формируется машинный код для тела подпрограммы
* В местах вызова вставляются команды вызова этого кода

Inline-подпрограмма:

* Тело подпрограммы вставляется прямо в месте вызова

**Рекурсивные подпрограммы**

Рекурсия – определение какого-либо понятия через само это понятие

Классический пример:

n! = 1, если n = 0

n \* (n-1)!, если n > 0

При каждом рекурсивном вызове создается новый набор:

* Фактических параметров
* Локальных переменных

На время выполнения текущего рекурсивного вызова данные всех предыдущих вызовов становятся недоступными

Рекурсивность – это не свойство решаемой задачи, а особенность ее решения

Ту же задачу можно решить без рекурсии

n! = 1\*2\*3\*…\*(n-1)\*n

Преимущества

* Как правило, запись такой программы короче и нагляднее

Недостатки

* Как правило, такая программа выполняется медленнее, чем эквивалентная ей не рекурсивная

Рекурсия:

* Явная – обращение к подпрограмме содержится в теле самой подпрограммы
* Неявная (взаимная) – обращение к подпрограмме содержится в теле другой подпрограммы, к которой происходит обращение из данной подпрограммы

При организации взаимной рекурсии возникает проблема:

* Вызов подпрограммы возможен только после ее описания

Проблема решается с помощью опережающих описаний (директива forward)

Описания подпрограммы:

* Опережающее
* Определяющее

Опережающее описание:

* Заголовок подпрограммы
* Директива forward

Определяющее описание:

* Заголовок подпрограммы
* Блок (тело подпрограммы)

**Модули**

Файлы в Delphi:

* Файл проекта (.dpr) – программный модуль;
* Исполняемый файл (.exe) – результат компиляции проекта;
* Вспомогательные модули (.pas);
* Скомпилированные модули (.dcu).

Было: (еще поддерживается)

<Модуль> ::= 🡪 <Заголовок\_модуля> 🡪 <Интерфейсный\_раздел> 🡪 <Раздел\_реализации> 🡪 <Раздел\_инициализации> 🡪 . 🡪

Стало:

<Модуль> ::= 🡪 <Заголовок\_модуля> 🡪 <Интерфейсный\_раздел> 🡪 <Раздел\_реализации> 🡪 <Раздел\_инициализации> 🡪 <Раздел\_финализации> 🡪 end. 🡪

Unit MyUnitName;

Interface

Uses …

Type …

Const …

Var …

Procedure …

Function …

Implementation

Uses …

Type …

Const …

Var …

Procedure …

Function …

Initialization

…

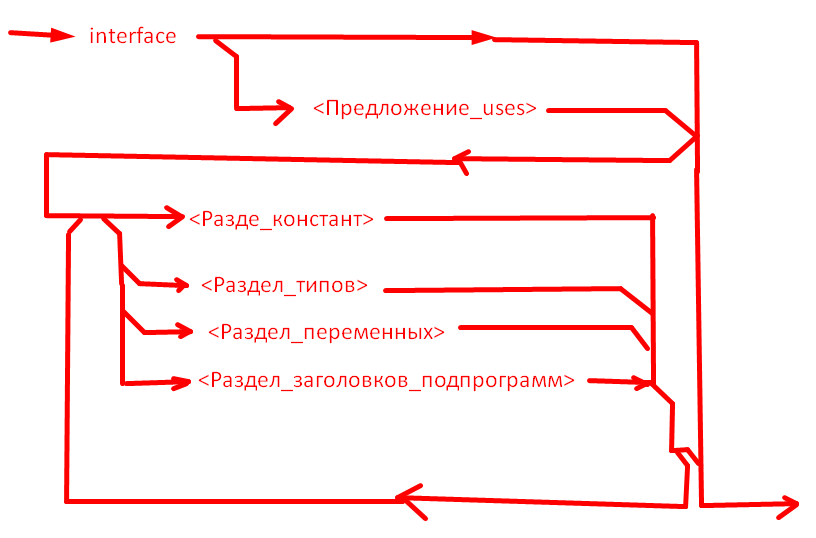
Finalization

…

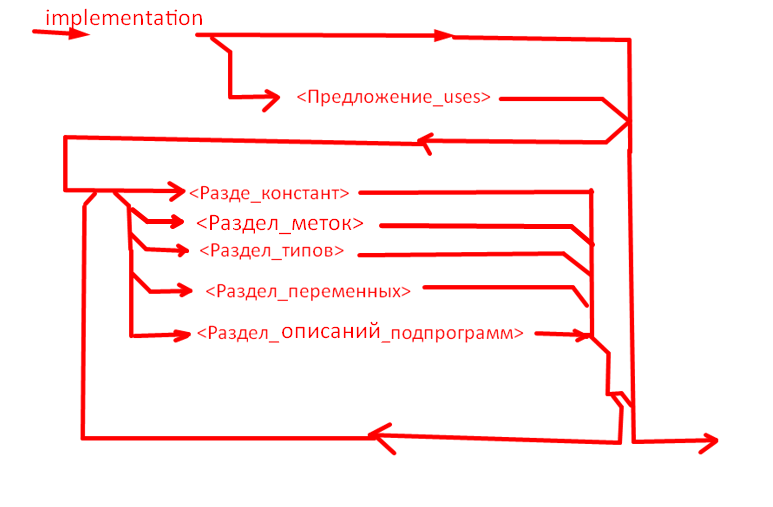
End.

<Заголовок\_модуля> ::= 🡪 unit 🡪 <Идентификатор\_модуля> 🡪 ; 🡪

<Интерфейсный\_раздел> ::=



<Реализационный\_раздел> ::=



*Директива компилятора [$I]*

Позволяет подставить в текст модуля содержимое внешнего текстового файла.

…

{$I SomeFile.inc}

…

**Записи**

Запись – структура данных, состоящая из упорядоченных разнородных компонентов.

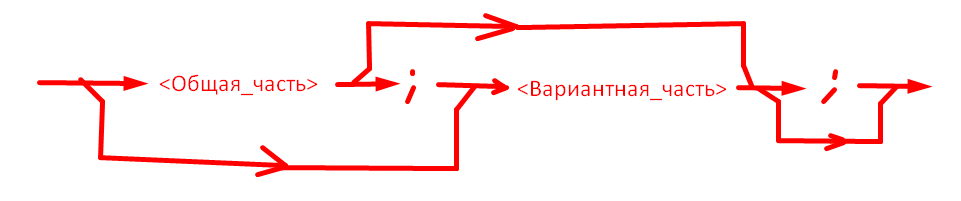
Компоненты записи – поля

Другие названия:

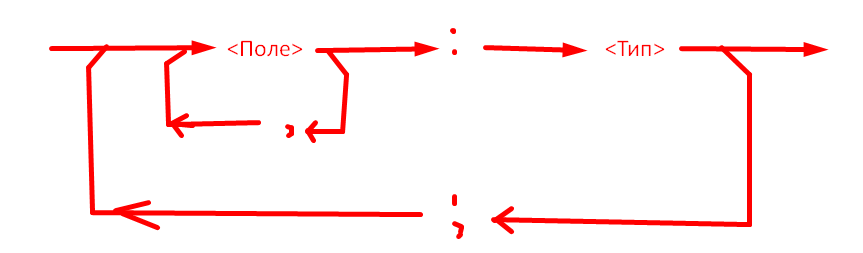
* Комбинированный тип
* Тип record
* Структура (в C-подобных языках)

<Тип\_Record> ::= 🡪 Record 🡪 <Список\_полей> 🡪 End 🡪

<Список\_полей> ::=



<Общая\_часть> ::=



Пример комплексных чисел:  
Type

TComplex = record

Re: Real;

Im: Real;

End;

Var

x, y: TComplex

В памяти:

|  |  |
| --- | --- |
| Re: Real; | Im: Real; |
| 6 байт | 6 байт |

Type

TUserInfo = record

Name: String[20];

Age: Integer;

Average: Real;

End;

|  |  |  |
| --- | --- | --- |
| Name: String[20]; | Age: Integer; | Average: Real; |
| 21 байт | 4 байт | 6 байт |

Объем памяти, занимаемой переменной-записью, складывается из размеров полей нижнего уровня вложенности.

Полная переменная – переменная, имеющая тип записи верхнего уровня вложенности.

Type

TComplex = record

Re: Real;

Im: Real;

End;

Var

X: TComplex

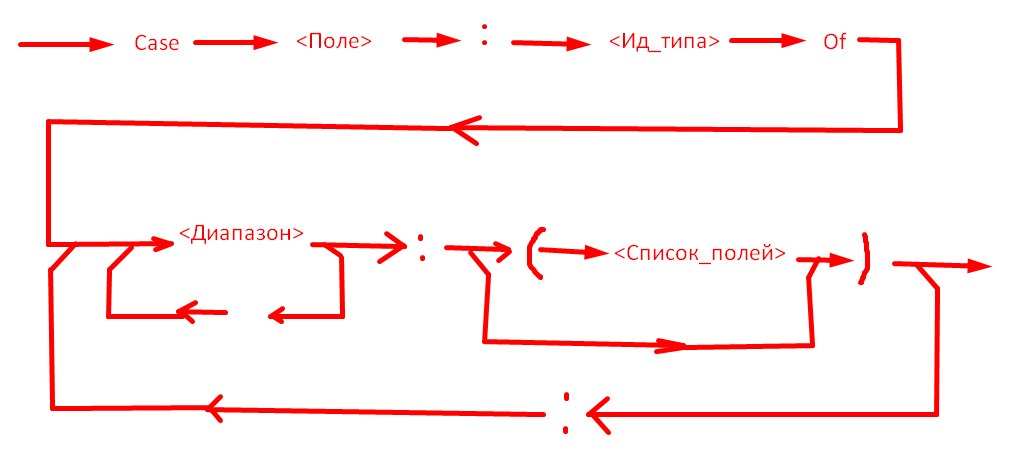
X.Re := 5;

ИмяПолнойПеременной.ИмяПоля := Значение;

Для полных переменных существует только один оператор:

* Оператор присваивания

Для определения значения полной переменной необходимо присвоить значения всем ее полям.

<Вариантная часть> ::= 

*Записи с вариантами*

Type

TEmployee = record

FirstName, LastName: String[40];

BirthDate: TDate;

Case Salaried: Boolean of

True: (AnnualSalary: Currency);

False: (HourlyWage: Currency);

End;

Поля каждого варианта занимают одну и ту же область памяти.

Размер этой области памяти определяется по наибольшему варианту.

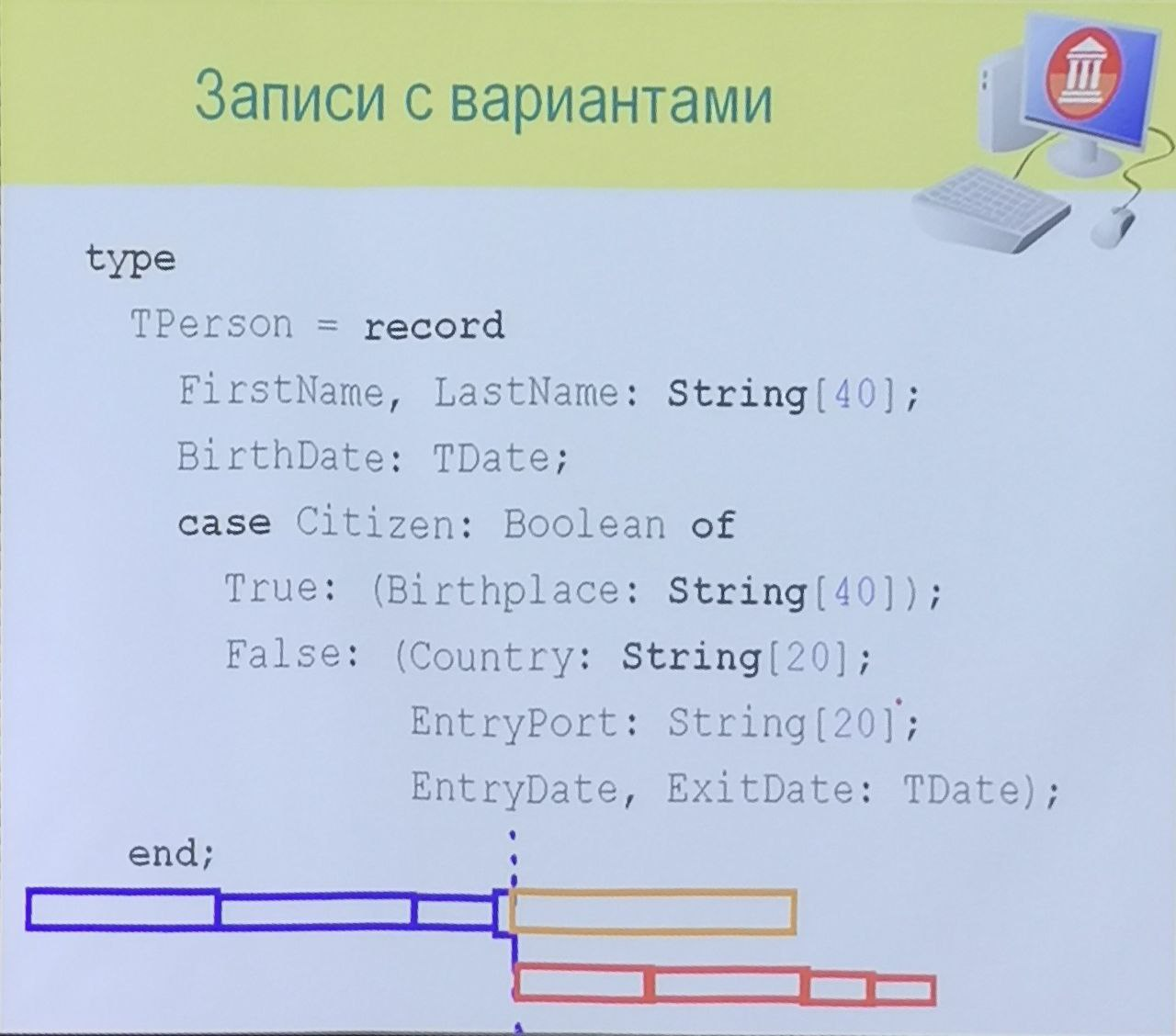
Поле, объявленное в case, - поле признака.

Поле признака относится к общей части записи.

Никаких дополнительных проверок в зависимости от значения поля признака не производится

Никаких дополнительных проверок в зависимости от значения поля признака не производится.

К любым полям вариантной части можно обращаться независимо от значения поля признака



Запись может не иметь общей части.

В этом случае вместо поля признака указывается только имя любого перенумерованного типа.

У case в записи с вариантами нет отдельного служебного слова end.

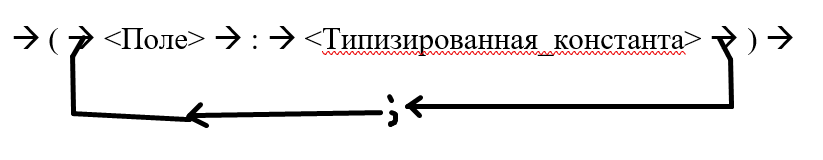
Слово end заканчивает всю конструкцию описания записи.

Вариантные поля должны находиться после полей общей части.

Имена полей должны быть уникальными:

* Даже если поля относятся к разным вариантам

<Константа\_запись> ::=



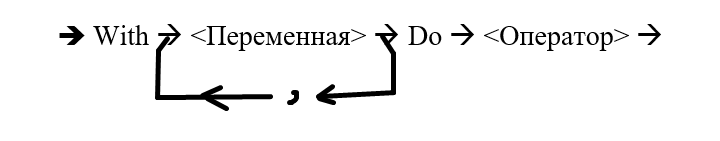
Поля должны задаваться в том же порядке, что и в объявлении типа.

Если есть вариантная часть:

* Задаваться должны поля только одного из вариантов
* Если есть поле признака, следует задавать поля выбранного варианта (в соответствии со значением поля признака).

*Оператор присоединения with*

<Оператор\_With> ::=



Var

X: TBirthDate;

…

With x do

Begin

Day := 5;  
 Month := Sep;  
…

End;

With var1, var2, var3 do =

With var1 do

With var2 do

With var3 do

При обращении к полю, которое есть в нескольких переменных, перечисленных в операторе with, предпочтение отдается переменной, указанной ближе к концу списка.

Если необходимо изменить это поведение или вообще обратиться к одноименной переменной, необходимо записать ее полное имя.

(если есть x.Year и просто Year то для обращения ко второму в юните MyUnit в with x do можно взять MyUnit.Year)

Packed record

Процессор:

* Обращается к памяти по адресам, кратным разрядности платформы
* Считывает/записывает за одно обращение количество данных, равное разрядности платформы

Чтение 1 байта по адресу 007F0001:

* Чтение 4 байт по адресу 007F0000;
* Лишние байты не используются.

Чтение 4 байта по адресу 007F0001:

* Чтение 4 байт по адресу 007F0000;
* Чтение 4 байт по адресу 007F0004;
* Объединение нужных байтов (3+1);
* Лишние байты не используются.

Выравнивание – способ размещения переменных в памяти, позволяющий минимизировать количество обращений к памяти.

В общем случае переменные следует выравнивать по адресам, кратным их размеру.

По умолчанию компилятор «выравнивает» поля всех записей.

* При этом некоторые байты могут оставаться неиспользуемыми (только для выравнивания).
* Директива {$A} позволяет управлять выравниванием

Зарезервированное слово packed заставляет компилятор располагать поля записи вплотную друг к другу.

Word + longword + byte = 8 при record

Word + longword + byte = 7 при packed record

Бывает полезно при работе с файлами сложных форматов. + Безопасность + Совместимость платформ

Зарезервированное слово **packed** также может применяться для массивов

**Указатели**

*Автоматические и динамические переменные*

Автоматическая переменная – переменная, память для которой выделяет компилятор

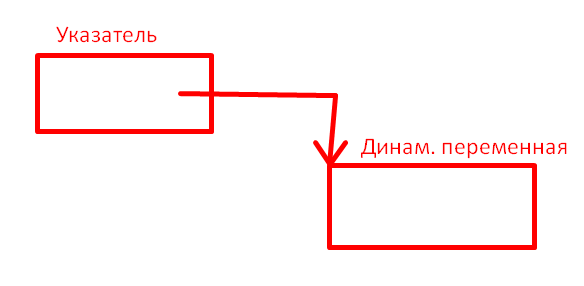
Динамическая переменная – переменная, память для которой выделяется во время выполнения программы.

Все идентификаторы имеют смысл только в исходном коде.

* В скомпилированной программе есть только обращения по адресам, но не имена переменных

Динамические переменные не объявляются в программе.

* Единственный способ обратиться к ним – по адресу

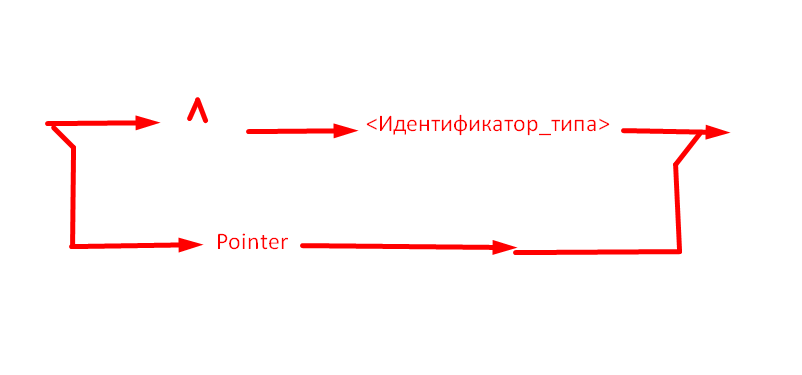


Ссылочный тип (тип указатель):

* Значения – адреса в памяти.

Внимание! В некоторых языках понятия «ссылка» и «указатель» могу не совпадать.

<Тип\_указатель> ::=



Var

P: ^TSomeType;

Указатели:

* Типизированный
* Нетипизированный

Ссылочные типы можно объявлять до объявления базовых типов.

PlistItem = ^TListItem;

TListItem = Record

Data: Integer;

Next: PListItem;

End;

*Нулевой указатель (пустая ссылка)*

Когда переменная-указатель не связана ни с какой переменной, используют специальное значение nil

Оно может быть присвоено переменной-указателю любого типа

Для получения адреса используется унарная операция @.

Вместо нее можно использовать функцию Addr().

* Самостоятельно.

Var

P: ^Integer;

X, Y: Integer;

If X > Y then

P := @X

Else

P := @Y;

*Операция разыменования указателя*

Для доступа к динамической переменной через указатель используется унарная операция разыменования указателя ^.

…

WriteLn(‘Max = ‘, P^);

…

*Понятие LValue*

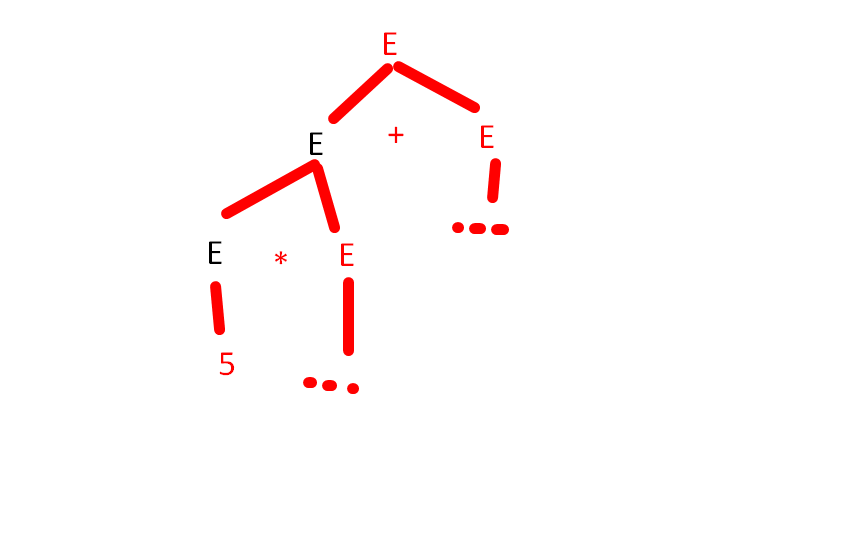
В теории компиляторов (и при описании синтаксиса ЯП) применяется понятие LValue

LValue – это выражение, которое задает определенное место в памяти.

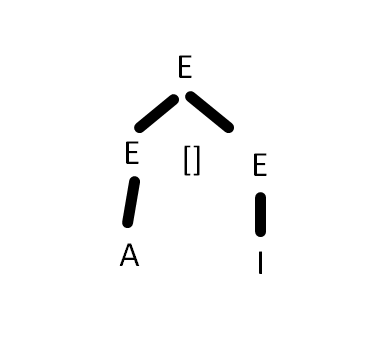
*Выражение*

Выражение может быть константой, переменной, одним или несколькими выражениями, с примененными к ним операциями

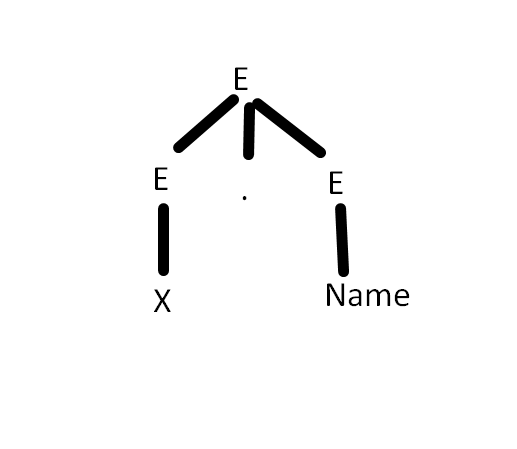
Выражение может быть представлено деревом:



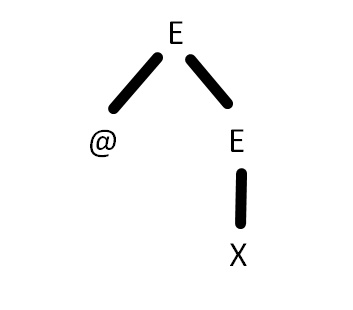
Обращение к элементу массива:

A[I] 

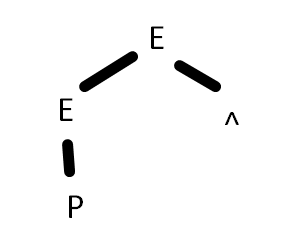
Обращение к полю записи:

X.name 

Взятие адреса:

@X 

Разыменование указателя:

P^ 

*Выражение и LValue*

Некоторые выражения могут задавать «что-то, имеющее адрес»:

* Т.е. область памяти.

Например:

* Имя переменной;
* Обращение к элементу записи/массива;
* Разыменование указателя.

Такое выражение называется LValue.

LValue – left Value.

LValue – это выражение, которое может быть записано в левой части оператора присваивания.

* Т.е. ему может быть присвоено значение.

Типам-указателям принято давать имена, начинающиеся с буквы P:

Type

PListItem = ^TListItem;

*Динамическое выделение памяти*

Динамическое выделение памяти сводится к 2 операциям:

* Выделение блока памяти заданного размера
* Освобождение блока памяти

Дополнительно могут быть доступны другие операции:

* Изменение размера блока
* Получение размера блока
* И др.

Стандартная библиотека Pascal|Delphi предоставляет 2 способа работы с динамической памятью:

* New/Dispose
* GetMem/FreeMem

В Pascal дополнительно могли использоваться:

* Mark/Release

*GetMem/FreeMem*

Procedure GetMem(var: P: Pointer; Size: Integer);

Procedure FreeMem(var P: Pointer[; Size: Integer]);

GetMem() выделяет блок размером Size байт и помещает его адрес в P.

FreeMem() освобождает блок с адресом P/

* Необязательный параметр Size должен иметь то же значение, что и при вызове GetMem().

*New/Dispose*

Procedure New(var P: Pointer);

Procedure Dispose(var P: Pointer);

New() выделяет блок памяти и помещает его адрес в P.

* Размер блока равен размеру типа в объявлении переменной P.

Dispose() освобождает блок P.

*Mark/Release*

В Delphi эти процедуры признаны устаревшими (deprecated)

В Turbo Pascal можно было:

* С помощью Mark() сохранить состояние динамической памяти
* Выделять память с помощью New()/GetMem()
* Восстановить сохраненное состояние динамической памяти с помощью Release()

**Файлы**

**Физический файл** – поименованная область памяти на внешнем носителе, в которой хранится некоторая информация.

**Логический файл** – представление физического файла в программе

**Файловый тип** – произвольная последовательность элементов, длина которой заранее не определена, а конкретизируется в процессе выполнения программы.

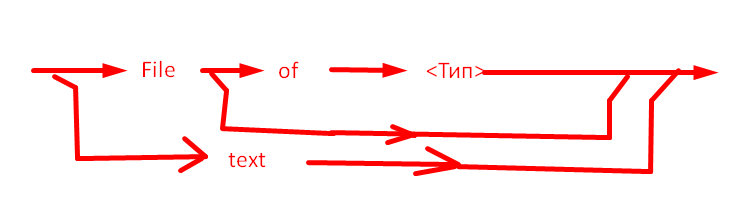
Способы доступа:

* Последовательный
* Прямой

Виды переменных файлового типа:

* «Текстовые файлы»
* «Файлы с типом»
* «Файлы без типа»

<Файловый\_тип> ::=



Над значениями файлового типа не определены никакие операции

Работа с файлами осуществляется с помощью так называемых процедур и функций ввода-вывода

*Текущая позиция в файле*

С каждым открытым файлом связан т.н. указатель файла.

* Другие названия: окно файла, текущая позиция файла.

Указатель файла определяет позицию доступа – элемент файла, с которым будет выполняться следующая операция ввода/вывода.

*Конец файла*

Позиция файла, следующая за его последним элементов, считается концом файла.

**Конец файла никак не помечается!**

*Файл с типом*

file of <Тип>

Файл считается состоящим из элементов, каждый из которых имеет тип <Тип>

*Файлы без типа*

Файл считается состоящим из элементов, размер которых определяется при открытии файла

*AssignFile*

Procedure AssignFile(var F; FileName: String)

Связывает файловую переменную F с файлом FileName

*Reset*

Procedure Reset(var F: file)

Procedure Reset(var F: file; RecSize: Word)

Открывает существующий файл.

* Если F уже открыт, закрывает и снова открывает его
* Указатель файла устанавливается на его начало
* Режим определяется переменной FileMode

*Rewrite*

Procedure Rewrite(var F: file)

Procedure Rewrite(var F: file; RecSize: Word)

Создается новый файл и открывает его

* Если F уже открыт, закрывает и снова открывает его
* Указатель файла устанавливается на его начало (файл создается пустым)

*CloseFile*

Procedure CloseFile(var F)

Закрывает файл

* F – файловая переменная, открытая с помощью Reset(), Rewrite() или Append()

*Eof*

Function Eof(var F): Boolean

Проверяет, является ли текущей позицией конец файла.

* F – файловая переменная, открытая с помощью Reset(), Rewrite() или Append()

*Read*

Procedure Read(F, V1, …, Vn);

Считывает компоненты файла в соответствующие переменные

* F – файловая переменная, открытая с помощью Reset() или Rewrite()
* Для файлов с типом

*Write*

Procedure Write (F, V1, …, Vn);

Записывает в файл компоненты из соответствующих переменных

* F – файловая переменная, открытая с помощью Reset() или Rewrite()
* Для файлов типом

*BlockRead*

Procedure BlockRead(var F: file; var Buf; Count: Integer [; var Done: Integer]);

Считывает не более Count компонентов в переменную Buf из файла F

* F – файловая переменная, открытая с помощью Reset() или Rewrite()
* Для файлов типом

*BlockWrite*

Procedure BlockWrite(var F: file; var Buf; Count: Integer [; var Done: Integer]);

Записывает не более Count компонентов из переменной Buf в файл F

* F – файловая переменная, открытая с помощью Reset() или Rewrite()
* Для файлов типом

*Seek*

Procedure Seek(var F; N: LongInt);

Перемещает указатель файла F в позицию N

*FilePos*

Procedure FilePos(var F): LongInt;

Возвращает текущее положение в файле F, в байтах

*FileSize*

Procedure FileSize(var F): Integer;

Возвращает размер файла F в компонентах

***Текстовые файлы***

*Text*

Файл считается последовательностью символов, интерпретируемой как текст

* **НЕ** эквивалентен file of Char

*AssignFile*

такой же.

*Reset*

Procedure Reset(var F: Text);

Открывает существующий файл.

* Если F уже открыт, закрывает и снова открывает его
* Указатель файла устанавливается на его начало
* Для текстовых файлов – в режиме только для чтение (read-only)

*Rewrite*

Procedure Rewrite(var F: file)

Создается новый файл и открывает его

* Если F уже открыт, закрывает и снова открывает его
* Указатель файла устанавливается на его начало (файл создается пустым)
* Для текстовых файлов – в режиме только для записи (write-only)

*Append*

Procedure Append(var F: Text);

Открывает существующий файл для добавления.

* Если F уже открыт, закрывает и снова открывает его
* Указатель файла устанавливается на его начало
* Файл открывается в режиме только для записи (write-only)

*CloseFile*

такой же.

*Read*

Procedure Read(F, V1, …, Vn);

Считывает компоненты файла в соответствующие переменные

* F – файловая переменная, открытая с помощью Reset() или Rewrite()
* Для текстовых файлов преобразует текстовое представление значения к типу переменной

*Write*

Procedure Write(F, V1, …, Vn);

Записывает в файл значения переменных.

* F – файловая переменная, открытая с помощью Reset() или Rewrite()
* Записывает в файл текстовое представление значений переменных

*Переменные Input и Output*

В модуле System объявлены две стандартные переменные:

Var

Input: Text;

Output: Text;

Эти переменные связаны с так называемыми стандартными потоками ввода (input) и вывода (output)

По умолчанию связаны с консолью, т.е.:

* Чтение из Input дает вводимые пользователем с клавиатуры данные
* Запись в Output приводит к отображению записываемых данных в консоли (на экране)

*Read и Write для текстовых файлов*

Первый параметр F может быть опущен

* В этом случае используется стандартный ввод-вывод.

*Eof*

Function Eof([var F: text]):Boolean;

Проверяет, является ли текущей позицией конец файла

* F – файловая переменная, открытая с помощью Reset(), Rewrite() или Append()

*EoLn*

Function EoLn([var F: Text]):Boolean;

Проверяет, является ли текущей позицией конец строки

* F – файловая переменная, открытая с помощью Reset(), Rewrite() или Append()

*Буферизация ввода-вывода*

Процедура SetTextBuf() и функция Flush() – **самостоятельно**

SetTextBuf – позволяет задать пользовательский буфер для текстового файла вместо буфера по умолчанию

Procedure SetTextBuf(var F: TextFile; var Buf [; Size: Integer]);

Flush – asdas

Asdsad

Asdasd

Asd

**Множества**

Множественный тип (set) соответствует понятию множества в математике (лучше так не базарить на экзе, ага)

* Неупорядоченный набор значений

Максимальное количество элементов – 256

Все элементы должны быть одного и того же типа – базового типа множества

*Базовый тип*

Любой скалярный тип:

* Кроме вещественных
* Для остальных типов есть ограничения

Общее правило:

* Ord(x) для любого X базового типа должно попадать в диапазон от 0 до 255

Занимает от 1 до 32 байт

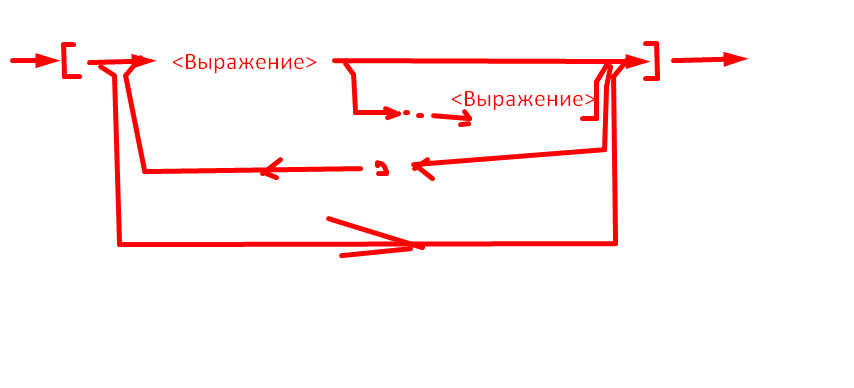
* В зависимости от количества элементов

Каждому из возможных элементов ставится в соответствие 1 бит

* Если бит содержит 1, элемент включен во множество
* Если бит содержит 0, элемент не включен во множество

Значение множественного типа – множество

<Конструктор\_множества> ::=



Конструктор множества – список разделенных запятыми выражений или диапазонов, заключенный в квадратные скобки (порядок перечисления не важен)

[2,3,5]

[‘A’..’Z’]

…

Задание множественного типа:

<Тип\_Set> ::= 🡪 Set 🡪 Of 🡪 <Базовый\_скалярный\_тип> 🡪

Set of 1..3

Set of Boolean

…

*Операции над множествами*

=, <>,

<= если множество 1 входит в 2 то 1 <= 2

>=

In – вхождение элемента во множество

+ – объединение множеств

\* – пересечение множеств

- – разность множеств

*Set of Char*

В старых версиях Delphi:

* Char = AnsiChar
* Set of Char = set of AnsiChar

В Unicode-версиях Delphi:

* Char = WideChar
* Set of Char = set of AnsiChar

**Задачка на 7 строк**

По умолчанию Delphi использует fastcall:

* Первый параметр – EAX
* Второй параметр – EDX
* Третий параметр – ECX
* Все остальные в стеке
  + Порядок помещения – слева направо

Аналогичны var-параметрам, но:

* Ключевое слово out означает, что процедура не использует значение параметра…
* … и использует его только как место для сохранения результатов работы

Введены в язык для поддержки технологии COM (не com-программы)

Технология COM предоставляет языконезависимый механизм взаимодействия программ

* Программы на разных языках могут обмениваться использовать объекты друг друга (через т.н. интерфейсы)

Managed-типы в Delphi:

* Динамические строки
* Динамические массивы
* Интерфейсы

Компилятор автоматически управляет памятью для данных этих типов

Переменные managed-типов в каждый момент времени должны содержать:

* Корректный адрес
* Или nil

Процедура не будет использовать значение out-параметра, а просто перезапишет его результатом

* Если не освободить ресурс, на который ссылается переменная, до вызова, то после вызова будет уже поздно

Переменные managed-типов, передаваемые как out-параметры, обнуляются перед вызовом

[ТУТ КОНЕЦ]