

1. Алгоритм и его свойства. Способы описания алгоритмов. Пример.

Алгоритм – система правил, чётко описывающая последовательность действий, которые необходимо выполнить для решения задачи.

Алгоритмизация – сведение задачи к последовательности этапов, выполняемых друг за другом так, что результаты предыдущих этапов используются при выполнении следующих.

Алгоритмизация:

1. Чёткая формулировка задачи: исходные данные, результаты;
2. Математическая постановка задачи
3. Выбор метода решения
4. Разработка алгоритма решения задачи
5. Выбор структур данных
6. Программирование
7. Тестирование и отладка
8. Выполнение программы (решение задачи)

Свойства правильного алгоритма:

1. **Дискретность** – означает, что в алгоритме могут быть выделены отдельные шаги, причём выполняться они должны **последовательно**, один за другим, и каждый **следующий** шаг должен использовать результаты, полученные на **предыдущих** шагах.
2. **Определенность** – каждое правило алгоритма должно быть **однозначным**. Значения величин, получаемых в какой-то момент времени, однозначно связаны со значениями величин, вычисленных ранее.
3. **Результативность** – алгоритм должен приводить к решению задачи за **конечное число шагов**.
4. **Массовость** – алгоритм должен разрабатываться в общем виде так, чтобы его можно было **применить для класса задач**, различающихся лишь исходными данными.

Способы описания алгоритмов: (В программировании метаязыком называется язык, предназначенный для описания языка программирования.)

1. **Запись на естественном языке (словесное)**

Этап обработки(вычисления) V:=выражение

Проверка условия: Если условие, то идти к шагу N

Переход к этапу с номером N: Идти к шагу N

Конец вычисления: Останов.

Недостаток – малая наглядность

2. **Изображение в виде схемы (графическое):** Схемой алгоритма называется графическое представление алгоритма, в котором этапы процесса обработки информации и носители информации представлены в виде **геометрических символов**, а последовательность процесса отражена **направлением линий**.

Для стандартизации и унификации языка схем алгоритмов в 1985г. был принят международный стандарт ISO 5807-85. В 1992 г. после переработки он был принят как стандарт СССР под обозначением ГОСТ 19.701-90 – Единая система программной документации – Схемы алгоритмов, программ, данных и систем – Условные обозначения и правила выполнения.

Виды схем:

- Данных
- Программы
- Работы системы
- Взаимодействия программ
- Ресурсов системы

3. **Запись на языке алгоритма (составление программы)**

Обычно программа, записанная на алгоритмическом языке – это окончательный вариант алгоритма решения задачи, ориентированный на конкретного исполнителя (компьютер или язык программирования).

2. **Графическое представление алгоритмов по ГОСТ 19.701-90. Пример.**

Схема алгоритмов - графическое представление алгоритма, в котором этапы процесса обработки информации и носители информации представлены в виде геометрических символов, а последовательность процесса отображена направлением линий.

Виды схем

1. Схема данных
2. Схема программы
3. Схема работы систем
4. Схема взаимодействия программ
5. Схема ресурсов системы

Символы в ГОСТ 19.701-90

- **Символы данных**
- **Процесса**
- **Линий**
- **Специальные символы**

1-3 группы делятся на основные и специфические символы

1. **Символы Данных:**

- Данные
 - Запоминаемые данные
 - ОЗУ
 - Ручной ввод
 - Магнитная лента(последовательная выборка)
 - Документ
 - Прямой доступ
 - Карта
 - Дисплей
2. Символы Процесса:
- Процесс
 - Предопределенный процесс
 - Ручная операция
 - Подготовка
 - Решение
 - Параллельные действия
 - Граница цикла
3. Символы Линии:
- Линия
 - Передача управления
 - Канал связи
 - Пунктирная линия
4. Специальные символы:
- Соединитель
 - Терминатор
 - Комментарий
 - Пропуск

Правила применения символов:

- Символы должны быть расположены равномерно
- Следует придерживаться минимального числа линий
- Стандарт регламентирует форму символов
- Размеры символов должны позволять включать текст внутрь символа
- Не должны изменяться углы и другие параметры формы
- По возможности символы должны быть одного размера
- Символы могут быть вычерчены в вертикальной ориентации или зеркальном отображении (нежелательно)
- Внутри символа следует помещать минимальное количество текста, необходимое для понимания его функции
- Текст записывается слева направо, сверху вниз

- Если текст не помещается внутрь символа, следует использовать комментарий
- Линии показывают направление потока управления
- Стандартное направление: сверху вниз, слева направо
- Если управление потока отличается от стандартного, оно должно указываться стрелками

Главная идея: Стремиться к наиболее понятному представлению алгоритма.

3. Разновидности структур алгоритмов. Виды циклов. Пример.

Структуры алгоритмов:

- **Линейные** – процесс, в котором направление вычислений является единственным
- **Разветвляющиеся** – процесс, в котором направление вычислений определяется некоторыми условиями
- **Циклические** – процесс, в котором отдельные участки вычислений выполняются многократно.

Цикл – участок схемы, многократно повторяющийся в ходе вычислений.

Классификация циклов

По взаимному расположению:

- **Простые** – не имеет в себе других циклов
- **Сложные** – имеет в себе другие циклы
- **Вложенные** – входит в состав других циклов
- **Внешние** – содержит внутри себя другие циклы, но сам в другие циклы не входящий

Тело цикла — часть цикла, многократно выполняющаяся последовательность действий.

Итерация — одно из повторений цикла.

От местоположения условия:

- **С предусловием**
- **С постусловием**

По виду условия выполнения:

- **С параметром:** счётчик, известное количество повторений
- **Итерационные** – циклический процесс, в котором количество повторений заранее не известно и зависит от получающихся в ходе вычислений результатов

Цикл с параметром: инициализация, вычисления, изменение параметра цикла, условие цикла

4. Основные положения теории структурного программирования.

Пример.

К концепциям структурного программирования относятся:

- **Отказ от использования оператора безусловного перехода (GoTo), а также от Break and Continue, так как по сути это GoTo с ограничениями. (иногда допускается)**
- **Применение фиксированного набора управляющих конструкций (3 базовых конструкций)**
- **Использование метода нисходящего проектирования – разбиение большой задачи на меньшие подзадачи так, чтобы каждую подзадачу можно было рассматривать независимо.**

В основу структурного программирования положено требование:

- **каждый модуль алгоритма (программы) должен проектироваться с единственным входом и единственным выходом.**
- **Программа представляется в виде множества вложенных модулей, каждый из которых имеет один вход и один выход.**

Базой для реализации структурированных программ является принцип Боме-Джакопини, в соответствии с которым **любая программа может быть разработана с использованием лишь трех базовых структур:**

- **Функциональный блок**
- **Конструкция принятия двоичного (дихотомического) решения**
- **Конструкции обобщенного цикла**

Функциональный блок – это отдельный вычислительный оператор или любая другая реальная последовательность вычислений с единственным входом и единственным выходом.

Конструкция принятия двоичного (дихотомического) решения - обычно называется конструкцией If-Then-Else (если-то-иначе), разветвлением или ветвлением. Это структура, обеспечивающая выбор между двумя альтернативными путями вычислительного процесса в зависимости от выполнения некоторого условия.

Конструкция обобщенного цикла – в качестве базовой конструкции структурного программирования используется цикл с предусловием, называемый циклом «Пока» (Do-While, пока условие истинно, тело цикла выполняется).

Достоинства:

- Упрощение тестирования программы
- Повышение производительности программистов
- Повышение читаемости программ – упрощается их сопровождение
- Повышение эффективности объектного кода программ

5. Реализация теоретических основ структурного программирования в современных языках программирования. Достоинства структурного программирования. Пример.

Реализация теоретических основ структурного программирования базируется на следующих правилах.

Все операции в программе должны представлять собой:

- 1) Исполняемые в линейном порядке выражения
- 2) Вызовы подпрограмм (обращение к замкнутому участку кода с 1 входом и 1 выходом)
- 3) Вложенные на произвольную глубину операторы If-Then-Else;
- 4) Циклические операторы (While)

Этих средств **достаточно** для составления структурированных программ. Однако иногда допускаются их некоторые расширения:

- дополнительные конструкции организации цикла: цикл с параметром, как вариант цикла с предусловием; цикл с постусловием
- подпрограммы с несколькими входами и выходами
- Оператор GoTo с жесткими ограничениями(выход из нескольких вложенных циклов, сложная обработка ошибок, автоматический сгенерированный код)

Достоинства:

- Упрощение тестирования программы
- Повышение производительности программистов
- Повышение читаемости программ – упрощается их сопровождение
- Повышение эффективности объектного кода программ

6. Преобразование неструктурированных программ в структурированные. Метод дублирования кодов. Достоинства и недостатки метода. Пример.

В общем случае произвольная программа не может быть преобразована в структурированную программу, которая реализует тот же алгоритм, построена с

применением тех же конструкций и не использует дополнительных переменных. Такое преобразование возможно при использовании трех известных методов:

- Дублирование кодов
- Введение переменной состояния
- Метод булевого признака

Сущность метода дублирования кодов: дублируются те модули исходного алгоритма или программы, в которые можно войти из нескольких мест (кроме последнего блока).

Достоинства:

- удобно использовать при нисходящем проектировании программ.
- Можно использовать как промежуточный шаг при разборе запутанного кода

Недостатки:

- Неприменимость к программам с циклами
- Увеличение объёма кода (громоздкость схемы алгоритма)
- Дополнительные затраты памяти для хранения дублируемых модулей

Поэтому метод используется, если дублируемые модули содержат незначительное число операторов. Если модули велики, то вместо дублирования кодов необходимо использовать вызываемые подпрограммы с формальными параметрами.

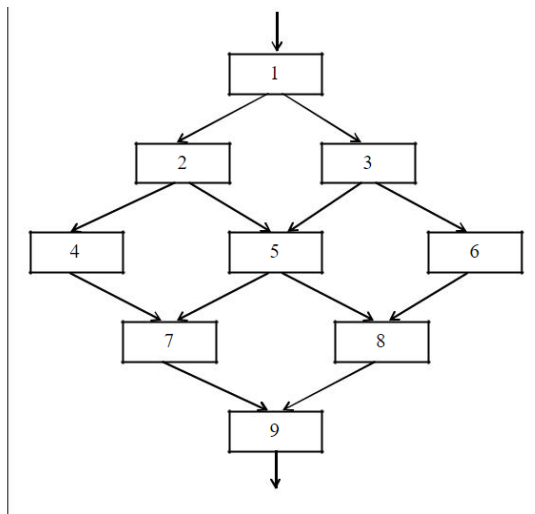
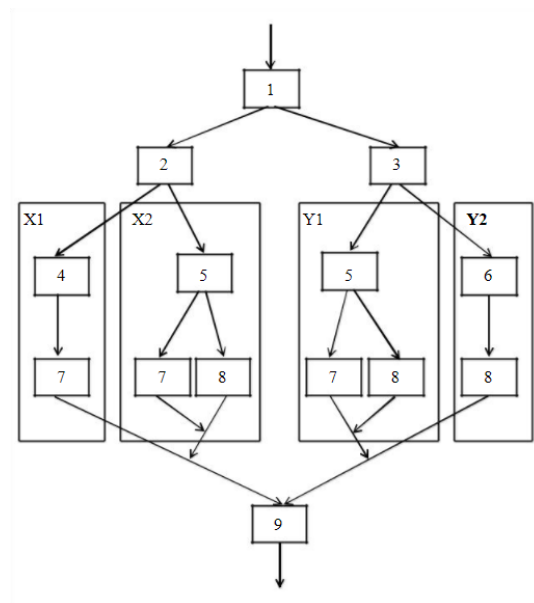


Рисунок 3.7 – Алгоритм неструктурированной программы типа «решетка»



7. Преобразование неструктурированных программ в структурированные. Метод введения переменной состояния. Достоинства и недостатки метода. Пример.

Процесс преобразования программы в структурированную состоит из следующей последовательности шагов:

- Каждому блоку неструктурированной схемы приписывается номер. Обычно первому – 1, последнему – 0
- В программу вводится дополнительная переменная целого типа, называемая переменной состояния
- Функциональные блоки исходной схемы заменяются блоками, выполняющими помимо основных функций преобразование переменной состояния: переменной присваивается значение, равное номеру блока-приемника в исходной схеме
- Аналогично преобразуются логические блоки. При этом, если в логическом блоке условие истинно, то это соответствует одному значению, если ложно – другому.
- Исходная схема перестраивается к виду, предложенному Ашкрофтом и Манной

Достоинства:

- Процесс преобразования программы отличается наглядностью и чёткостью
- Применим к алгоритмам любой структуры, в т.ч. и циклические
- Возможно автоматическое применение данного метода

Недостатки:

- Топология схемы сильно изменяется
- Затраты времени на проверку и изменение значения переменной состояния
- Громоздкость результирующей схемы алгоритма

Можно рассматривать как разновидность автоматного программирования

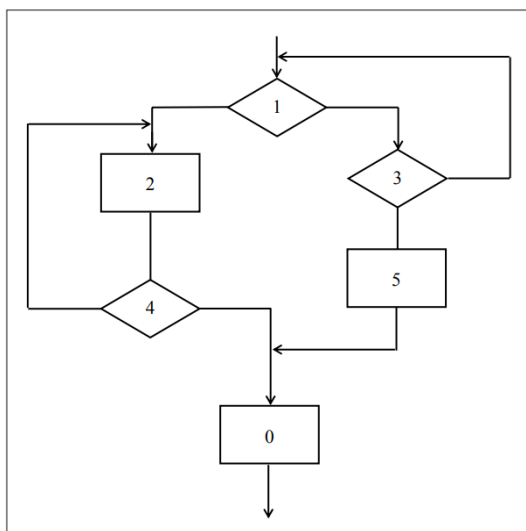


Рисунок 3.12 – Исходная схема неструктурированного алгоритма

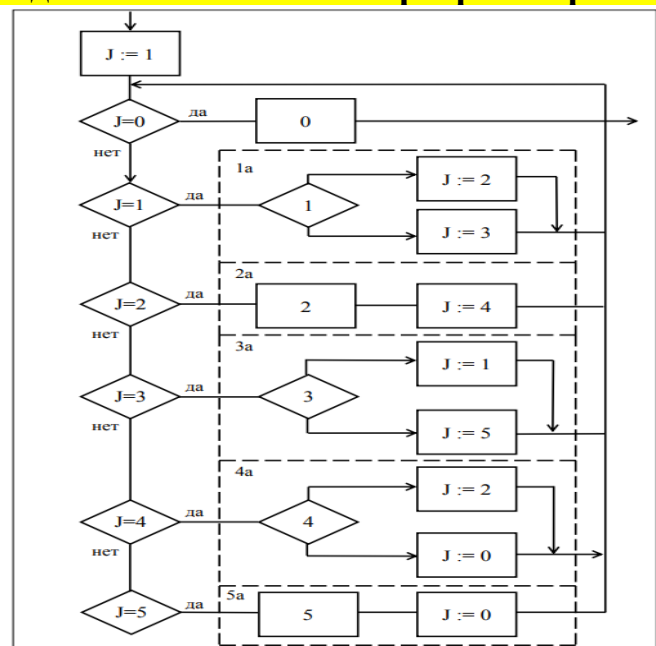


Рисунок 3.14 – Структурированная форма исходной схемы

8. Преобразование неструктурированных программ в структурированные. Метод булевого признака. Достоинства и недостатки метода. Пример.

Сущность метода булевого признака заключается в следующем.

- В программу, содержащие циклы, вводится некоторый признак
- Начальное значение признака задаётся до цикла
- Цикл выполняется, пока признак сохраняет своё исходное значение
- Значение признака изменяется при наличии некоторых условий внутри цикла

Достоинства:

- Компактность, экономичность
- Топология схемы изменяется незначительно

Недостатки:

- Применим только для алгоритмов с циклами

Иногда можно обойтись без специального признака, используя те условия, которые уже есть в исходной схеме.

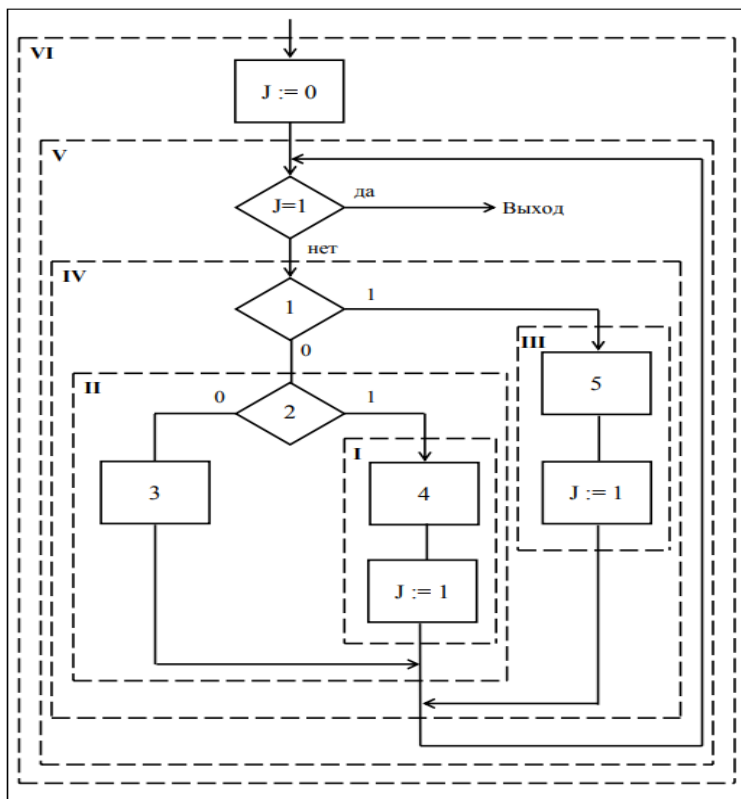


Рисунок 3.17 – Структурированная форма исходной схемы, преобразованная по методу булевого признака

9. Метод Дамке представления схем алгоритмов. Пример.

3 основные конструкции:

- Функциональный блок
- IF-Then-Else
- Конструкция цикла с предусловием (While)

Дополнительные конструкции:

- Конструкция цикла с постусловием
- Конструкция цикла с параметром
- Конструкция Case

Основным принципом при разработке структурированных схем алгоритмов по методу Дамке является **принцип декомпозиции**. Он означает, что любой элемент алгоритма, реализующий некоторую функцию (задачу), можно разделить на несколько элементов, реализующих необходимые подфункции (подзадачи).

Элементы в самой **левой** части схемы представляют **укрупнённую** структуру алгоритма. Затем элементы расширяются **вправо** по мере деления каждого элемента на **подзадачи**.

Чтобы исследовать любую подзадачу, достаточно анализировать только те элементы и управляющие структуры, которые находятся справа от нее.

Преимущества:

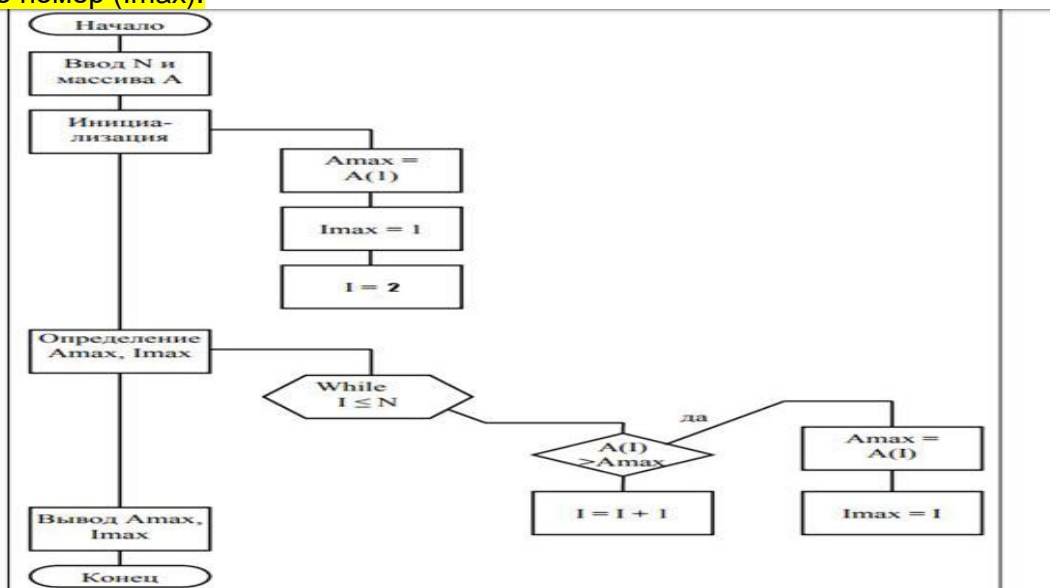
- **Нельзя** построить **неструктурированный** алгоритм
- **Удобно** использовать для **нисходящего** проектирования
- **Наглядность** для больших программ
- Удобство **коллективной** разработки

Недостатки:

- Схемы могут быть более **громоздкими**

Пример

Дан массив A, состоящий из N элементов. Найти наибольший из элементов массива (Amax) и его номер (Imax).



10. Схемы Насси-Шнейдермана. Пример.

Схемы Насси-Шнейдермана – это схемы, иллюстрирующие структуру передач управления внутри модуля с помощью **вложенных друг в друга блоков**.

Схемы используются для изображения **структурированных схем** и позволяют **уменьшить громоздкость схем** за счёт **отсутствия явного указания линий перехода по управлению**.

Схемы Насси-Шнейдермана называют ещё **структурограммами**.

Изображение основных элементов структурного программирования в схемах Насси-Шнейдермана организовано следующим образом. Каждый **блок** имеет форму **прямоугольника** и может быть вписан в любой внутренний прямоугольник любого другого блока. Информация в блоках записывается по тем же правилам, что и в структурированных схемах алгоритмов (на естественном языке или языке математических формул).

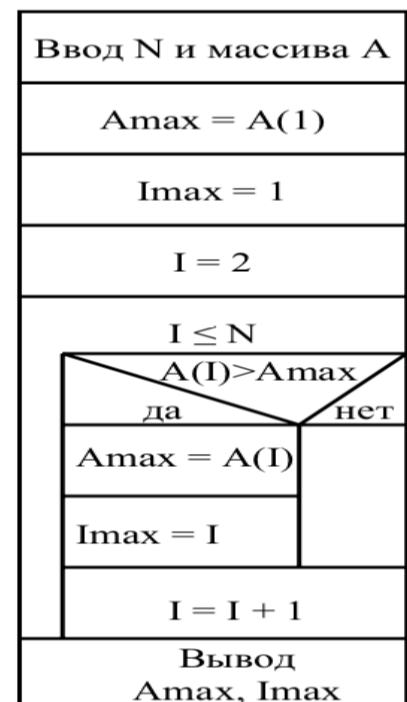
Конструкции структурированных алгоритмов в Насси-Шнейдермана:

- Функциональный блок
- Блок следования
- Блок решения
- Блок Case
- Цикл While
- Цикл Until

Укрупненная схема
Насси-Шнейдермана



Подробная схема
Насси-Шнейдермана



11. Сложность алгоритма. Оценка сложности алгоритмов. Пример.

Одна и та же задача может быть решена разными способами.

Т.е. может быть несколько алгоритмов решения одной и той же задачи.

Алгоритмы могут быть оценены:

- По количеству выполняемых операций
- По объёму потребляемой памяти
- По использованию других ресурсов

Оценка может быть:

- Минимальной(лучший случай)
- Средней
- Максимальной(худший случай)

Вычисление суммы элементов массива (N элементов)

```
Sum:=0;  
For I:=1 to N do  
    Sum:=Sum+A[I];
```

N операций изменения Sum, N операций чтения элементов массива => **O(N)** – **линейная сложность**

```
Sum:=0;  
For I:=1 to N do  
    Begin  
        Max:=A[I,1];  
        For J:=1 to M do  
            If Max < A[I,J] then  
                Max:=A[I,J];  
        Sum:=Sum+Max;  
    End;
```

N*M операций чтения элементов из матрицы

N*(M-1) операций сравнения с Max

N операций инициализации Max

N сложений Sum and Max

N инкремент I

N*M операций INC(J)

1 операция записи переменной Sum

N*M+N*(M-1)+3N+1 => O(N*M)

Линейный поиск

```
Index:=-1;  
For I:=1 to N do  
  If A[I]=Value then  
    Begin  
      Index:=I;  
      Break;  
    End;
```

Сложность $O(0.5N)$ или $O(N)$

Бинарный поиск:

- Выбрать средний
- Если равен исходному – завершить
- Если $<$, повторить для левой части
- Если $>$, повторить для правой части

Каждая итерация сокращает массив в 2 раза $\Rightarrow O(\log_2 N)$

Зная оценку сложности алгоритма, можно определить сколько времени ему потребуется для решения задачи заданного объема

Пример:

2 алгоритма:

- $O(N \log_{10} N)$
- $O(N^2)$

Предполагаемый объём данных – 1 млрд. элементов

Ограничение по времени 1 секунда

Для 1000 элементов:

- 0.003 с.
- 0.001 с.

Сложность алгоритма по объёму определяемой памяти оценивается **аналогично**

Как правило алгоритм можно **ускорить, увеличив расход памяти и наоборот**

На практике **не все операции** обходятся **одинаково дорого**

- **Выделение** динамической памяти может само иметь сложность $O(N)$
- **Присваивание** может занимать много времени для больших типов данных (сложность $O(S)$)
- Характер **обращений** к памяти может влиять на **использование кэша**
- **Ветвления** могут замедлять работу программы

12. Этапы постановки и решения задачи на компьютере. Методы автоматизации программирования. Назначение и классификация языков программирования.

Этапы постановки и решения задачи на компьютере:

- Чёткая формулировка задачи (исходные данные и их формат, результат)
- Математическая формулировка задачи
- Выбор метода решения
- Разработка алгоритма
- Выбор структуры данных
- Программирование
- Тестирование и отладка
- Выполнение программы

Чем раньше тот этап, на котором допущена ошибка, тем сложнее ее исправить.

К основным методам автоматизации программирования можно отнести следующие.

1) **Использование языков высокого уровня**, близких к естественному человеческому языку, позволяющих автоматически однозначно преобразовывать написанную на них программу в программу на языке машины, т.е. в машинные коды.

2) **Создание и использование библиотек стандартных программ и подпрограмм**, предназначенных для реализации **часто используемых задач**.

3) **Использование современных технологий** программирования.

4) **Использование Case-средств**, предназначенных для **автоматизации** процесса разработки программ.

Поколения ЯП:

- Машинные коды
- Языки ассемблера
- **Языки третьего поколения** (возможность задавать алгоритм работы программы без привязки к особенностям конкретного устройства или класса устройств)
- **Языки четвертого поколения** (К началу XXI века акценты слегка изменяются и под 4GL начинают понимать **среды программирования**, которые позволяют программисту писать минимум кода (в идеальном случае — вообще не написать ни строчки кода) для достижения требуемых результатов. Такая среда программирования значительную часть кода генерирует самостоятельно, давая возможность разработчику оперировать более абстрактными понятиями, чем в языках третьего поколения)

Уровень языка — среднее количество **машинных** команд, приходящихся на одну команду языка программирования. Чем ближе это число к единице – тем ниже язык программирования.

Основная идея автоматизации программирования заключается в **отказе от написания программ** непосредственно в **машинных кодах**. Программа пишется на некотором входном языке. **Входные языки делятся на следующие группы:**

1)машинно-ориентированные языки - Программы, написанные на языках данной группы, могут выполняться **только** на тех компьютерах, для которых **разработаны** соответствующие языки.

2)процедурно-ориентированные языки(C, Pascal, Basic) - Языки данной группы первоначально предназначались для решения **конкретного** класса задач – **инженерных, научно-технических, обработки** экономической информации, обработки списков, моделирования и т.д.

3)объектно-ориентированные языки(C#, C++) - В объектно-ориентированных языках, при сохранении свойств процедурно-ориентированных языков, дополнительно введено понятие **объекта**.

4)проблемно-ориентированные языки - Проблемно-ориентированные языки (domain-specific languages или DSLs) ориентированы на **узкий класс** однотипных задач. **Исключают** работу программиста по **разработке алгоритмов** решаемых задач.

5)языки четвертого поколения (4GL) реализуют современные технологии **визуального программирования**. Автоматически генерируют исходный текст программ целиком или в виде **отдельных фрагментов**.

13. Структура программного обеспечения. Системы программирования. Назначение, состав.

Под системой программного обеспечения (СПО) подразумевают **совокупность специальных программ, облегчающих** процесс **разработки** программ и обеспечивающих процесс их **выполнения** на компьютере, и связанную с ними **документацию**.

Программное обеспечение (ПО) по функциям и задачам, выполняемым его программами, можно разделить на две группы:

- 1) специализированное ПО;**
- 2) стандартное ПО.**

Специализированное ПО состоит из **прикладных** программ, предназначенных для решения некоторых **самостоятельных** задач, достаточно **часто** встречающихся.

К **стандартному ПО** относятся **системы программирования** и **операционные системы**.

Система программирования – совокупность программ, описаний и инструкций, предназначенных для **автоматизации** процесса **разработки** программ.

Операционная система – совокупность программ, описаний и инструкций, предназначенных для **организации** и **контроля выполнения** программ на компьютере.

Основное назначение систем программирования – максимально **облегчить** процесс общения **программиста** с **компьютером**, **освободить** его от необходимости **описания алгоритма** на машинном языке, **предоставить** возможность использования **входного языка программирования**.

Состав системы программирования:

- 1) **входной язык системы**;
- 2) **транслятор** с входного языка на машинный язык;
- 3) **редактор связей**;
- 4) **библиотеки программ**;
- 5) **средства отладки**;
- 6) **обслуживающие (сервисные) программы**;
- 7) **документация**.

- 1) **Входной язык системы**

Входной язык – это язык, на котором пишется исходный текст программы.

- 2) **Транслятор**

Транслятор – это программа, предназначенная для **преобразования** исходной программы, написанной на **входном** языке программирования, в программу на **машинном** языке. Существует два вида трансляторов:

а) **Компиляторы** – трансляторы, в которых **трансляция отделена** от **выполнения** программы (транслятор компилирует рабочую программу, которая в дальнейшем может быть выполнена). К компиляторам относятся трансляторы, например, со входных языков Паскаль, Си.

б) **Интерпретаторы** – трансляторы, в которых **трансляция совмещена** с **выполнением** программы. Каждый оператор языка читается, расшифровывается и выполняется. Интерпретаторы отличаются **малой скоростью** работы, но являются более **простыми** по сравнению с компиляторами. К интерпретаторам относится, например, транслятор со входного языка Бейсик.

- 3) **Редактор связей**

Современные системы программирования основаны на **модульном** принципе: программы оформляются в виде совокупности взаимосвязанных программ. Каждая такая программа называется модулем.

Исходный модуль – это модуль, записанный на **входном** языке программирования. Каждый исходный модуль транслируется независимо от других модулей.

Объектный модуль – это модуль, полученный в результате **трансляции**. Он содержит **текст** программы на **машинном языке** и дополнительную информацию, обеспечивающую **объединение** этого модуля с другими независимо транслированными модулями и настройку модуля по месту его загрузки в память.

Редактор связей – это программа, предназначенная для **сборки и установления связей** между **модулями**. **Результатом** работы редактора связей может являться, в зависимости от операционной системы, **загрузочный** или **абсолютный** модуль.

Загрузочный модуль – это модуль, готовый к вводу в память для настройки по месту в памяти и выполнения (модуль, готовый к загрузке).

Абсолютный модуль – это модуль, полученный в результате загрузки.

4) Библиотека стандартных программ

Библиотека стандартных программ (БСП) представляет собой **готовые программы**, предназначенные для решения **распространенных** задач. Программы, включенные в библиотеку, оформляются специальным образом, **облегчающим** их **вызов**, использование, передачу входных данных и результатов. Программы, включенные в БСП, автоматически вызываются для выполнения специальными командами вызова.

5) Средства отладки

Основная цель этапа отладки – выявление и исправление ошибок в программе. Процесс отладки состоит из многократных попыток выполнения программы на компьютере и анализа получившихся результатов.

Большинство синтаксических ошибок обнаруживается автоматически на этапе трансляции. Современные трансляторы с языков программирования выдают информацию о синтаксических ошибках (ошибках, допущенных при записи текста программы на алгоритмическом языке), указывают места ошибок и их характер.

После того, как программа становится работоспособной, проводится ее тестирование – проверка правильности ее функционирования на различных наборах исходных данных из диапазона их допустимых значений.

Под операционной системой (ОС) понимают набор программ, которые организуют и контролируют выполнение программы на компьютере без вмешательства оператора.

В настоящее время на персональных компьютерах наиболее распространены операционные системы типа MS-DOS (PC-DOS), UNIX, Windows.

Операционная система обеспечивает выполнение двух главных задач:

- 1) **поддержку работы всех программ** и обеспечение их **взаимодействия с аппаратурой**;
- 2) **предоставление пользователям возможностей общего управления компьютером.**

14. Общая характеристика языка Delphi. Достоинства языка Delphi.

Достоинства языка:

- 1) **относительная простота** (т.к. разрабатывался с целью **обучения программированию**);
- 2) **идеология языка Паскаль близка к широко используемым базовым методологиям и технологиям программирования, в частности, к структурному программированию и нисходящему проектированию** (методу пошаговой детализации) программ. Паскаль может использоваться для записи программы на различных уровнях ее детализации, не прибегая к помощи схем алгоритмов;
- 3) **гибкие возможности в отношении используемых структур данных;**
- 4) **высокая эффективность программ;**
- 5) **наличие средств повышения надежности программ,** включающих **контроль правильности использования данных различных типов и программных элементов** на этапах **трансляции, редактирования и выполнения.**

Язык Паскаль **относится к процедурно-ориентированным языкам высокого уровня.** Паскаль разработан американским ученым **Никласом Виртом** в **1971 г.** в качестве языка для обучения программированию. **Базой** при разработке явился язык **Алгол.** Начиная с версии Турбо Паскаль 5.5 в язык введено понятие **объекта** и Паскаль стал обладать свойствами **процедурно-ориентированных и объектно-ориентированных языков программирования.**

15. Алфавит языка Delphi. Классификация символов. Пример.

Грамматическое описание любого языка программирования включает:

- **Алфавит языка**
- **Синтаксис – правила построения фраз**
- **Семантика – смысловое значение фраз**

Алфавит:

- **Буквы 'A'..'Z', 'a'..'z', '_'** В программах строчные латинские буквы эквивалентны прописным везде, за исключением литералов(строковых констант)
- **Цифры 0..9**
- **Специальные символы**

Специальные символы (1 группа) :

Простые

Символы ограничителя:

- Знаки **арифметических** операций: +-*/*
- Знаки **сравнения** <>=
- Знаки **разделители** ., : ; , ' ,
- Знаки **скобок** () { } []

Другие символы @ # \$ ^ 'пробел'

Составные:

- Присваивание :=
- Не равно <>
- Диапазон значений ..
- Меньше или равно <=
- Больше или равно >=

Специальные символы (2 группа):

- Служебные зарезервированные слова: and, array, begin, end, repeat, until, eps, else, if, then

16. Основные понятия языка Delphi. Лексемы и их типы. Идентификаторы. Комментарии. Понятие оператора. Типы операторов. Пример.

Текст программы состоит из:

- **Лексем**
- **Комментариев**
- **Пробелов**

Лексема – неделимая последовательность знаков алфавита, имеющая в программе определенный смысл (Слова ЯП)

Лексемы нельзя разрывать

Лексемы:

- **Специальные символы**
- **Идентификаторы** – имя (любая последовательность букв, цифр и знаков подчеркивания, начинающаяся буквой или знаком подчеркивания. Никакие другие символы в идентификаторах использовать нельзя), которое позволяет однозначно выбрать один объект из множества объектов.

- **Литералы** (строковые и символьные константы)
- **Числовые константы**
- **Метки**

Идентификаторы:

- **Предопределенные** (идентификатор, имеющий **стандартный** смысл и **входящий в описание языка**) Предопределенные идентификаторы **не являются служебными словами**. Поэтому программист при желании может изменить их смысл, используя соответствующее описание. Однако это нежелательно, чтобы избежать лишних ошибок.
- **Определенные программистом** (идентификатор, **смысл которого определен непосредственно в программе**. Такие идентификаторы обычно задают имена некоторых элементов программы. Например, X – имя переменной)

Комментарии:

Комментарий служит для внесения **пояснений** в программу и может размещаться **везде, где допускается пробел**. В качестве комментария может использоваться любая последовательность символов, включая русские буквы, за **исключением символов-ограничителей комментариев**.

- {Комментарий} или (*Комментарий*)

Вложенность комментариев не допускается. Но комментарии разного вида могут быть вложены друг в друга:

- (*...{...}...*) или {...(*...*)...}

Комментарии игнорируются транслятором и не оказывают влияния на решение задачи. При выводе текста программы на печать комментарии выводятся вместе с текстом.

К основным видам комментариев относятся:

- **вводный комментарий** – это комментарий, записываемый перед текстом программы, в котором даются общие сведения о программе (например, назначение программы, используемые методы вычислений, длительность работы программы, необходимые ресурсы памяти, даты разработки и последнего обновления, авторы и т.д.);
- **комментарии-заголовки** – это комментарии, записываемые перед подпрограммами (или большими блоками программы), в которых описывается назначение подпрограмм (блоков программы), выполняемые в них действия, методы, положенные в основу их работы и т.д.; нормы комментариев-заголовков: 4 – 5 строк на подпрограмму;
- **строчные комментарии** – это комментарии, описывающие мелкие фрагменты программ; нормы строчных комментариев: один комментарий на каждые две-три строки исходного текста программы.

Оператор – законченное предложение записанное на ЯП.

- **Состоит из лексем** (слов ЯП)

- Представляет собой законченное описание какого-либо действия

Оператор - команда, указывающая, какие действия на данном этапе должны быть выполнены компьютером.

- **Основные** (не содержащие в своем составе других операторов)
- **Производные** (в состав входят другие операторы)

17. Способы описания синтаксиса языков программирования. РБНФ.

Синтаксис – набор правил описывающих правильные предложения языка (Правила)

Метаязык – формализованная система обозначений, применяемая для записи правил синтаксиса (Способ записи правил) (язык для описания других языков)

Метаязыки:

- Расширенная форма Бэкуса-Наура (РБНФ)
- Синтаксическая диаграмма

РБНФ основные понятия:

- **Метаконстанта**
- **Метапеременная**
- **Метасимвол**
- **Синтаксическая единица**

Метапеременная – это определенная синтаксисом конструкция языка (исключая основные символы). Для записи метапеременных используются последовательности слов русского языка и служебных слов, между которыми находится символ подчеркивания. Заключаются в угловые скобки.

Примеры записи метапеременных:

<Шестн._цифра>
<Оператор_While>
<Объявление_переменных>

Метаконстанта – это лексема языка программирования. В программе метаконстанте соответствует она сама.

В РБНФ метаконстанты заключаются в кавычки.

Примеры метаконстант:

“End”
“+”
“For”

Синтаксическая единица – это строка, описывающая состав и порядок следования элементов конструкций языка программирования. Синтаксическая единица состоит из **метапеременных, метаконстант и метасимволов**.

Метапеременная в синтаксической диаграмме означает, что соответствующий фрагмент диаграммы должен быть детализирован подстановкой синтаксической диаграммы с именем, соответствующим данной метапеременной.

Метасимволы – специальные символы, используемые в метаязыках для описания синтаксиса языков программирования.

В РБНФ используется следующий набор метасимволов;

а) = (или ::=) имеет смысл «определяется как», «по определению есть»; справа от знака ::= записывается синтаксическая единица, слева – метапеременная;

б) . точка; обозначает конец определения;

в) | вертикальная черта; обозначает выбор, альтернативу (смысл эквивалентен словам «либо», «или»);

г) { } фигурные скобки; означают возможность повторения заключенной в них конструкции ноль, один или более раз; например, {<Тело цикла>};

д) [] квадратные скобки; обозначают необязательную часть конструкции, т.е. возможность повторения заключенной в них конструкции ноль или ровно один раз; например, запись [“+”] означает, что знак + перед числом может писаться или нет;

е) (|) круглые скобки вместе с используемой внутри них вертикальной чертой; означают альтернативы внутри определения, заключенного в скобки; например, запись (“X”|“Y”|“Z”) означает вхождение в конструкцию элемента “X” или “Y” или “Z”.

Идентификатор – по определению – последовательность латинских букв, цифр и знаков подчеркивания, начинающаяся с буквы или знака подчеркивания.

В РБНФ это определение может быть представлено так:

<Идентификатор> ::=

(<Буква> | <Знак_подчеркивания>){<Буква> | <Цифра> | <Знак_подчеркивания>}.

<Буква> ::=

“a” | “b” | “c” | “d” | “e” | “f” | “g” | “h” | “i” | “j” | “k” | “l” | “m” | “n” | “o” | “p” | “q” | “r” | “s” | “t” | “u” | “v” | “w” | “x” | “y” | “z” | “A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” | “I” | “J” | “K” | “L” | “M” | “N” | “O” | “P” | “Q” | “R” | “S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z”.

<Цифра> ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”.

<Знак_подчеркивания> ::= “_”.

18. Способы описания синтаксиса языков программирования. Синтаксические диаграммы. Пример.

Синтаксическая диаграмма – ориентированный граф с размеченными ребрами, используемый для описания синтаксической конструкции языка.

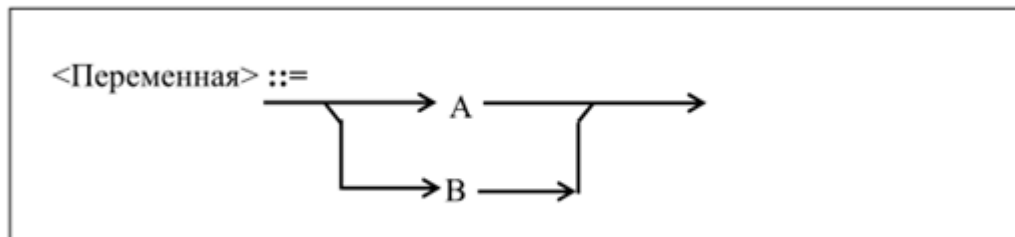
- Ребра помечены метaperеменными и метаконстантами
- Метасимволы не используются
- Метаконстанты записываются без кавычек

Язык РБНФ более строг и точен, более удобен для представления синтаксиса в памяти машины, более компактен.

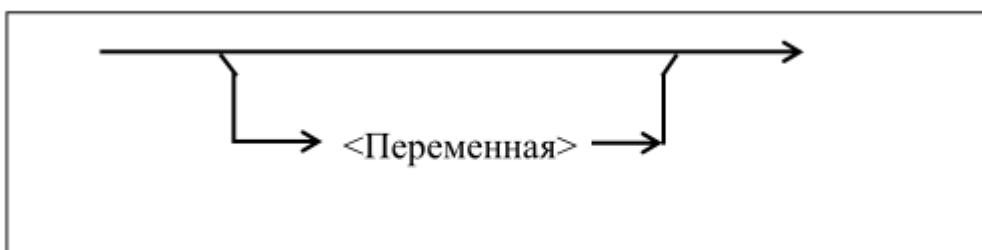
Синтаксические диаграммы более наглядны и просты для понимания, но более громоздки.

Ниже рассмотрено представление в виде ориентированных графов некоторых из метасимволов языка РБНФ.

a) **Выбору, альтернативе** (метасимволу | (Или)) соответствует разветвление в синтаксической диаграмме с последующим объединением. Например, переменная может принимать значение **A** или **B** (**A**, **B** – это лексемы языка Паскаль, т.е. метаконстанты).



b) **Необязательная часть** ([])



c) **Повторение** ({ })



19. Классификация данных в языке Delphi. Пример.

В программировании **данными** называется любой **набор знаков**, над которыми можно **выполнять** определенные **действия**. **Данные в программе** представляются в виде **констант и переменных**.

Константа – это элемент данных, имеющий **фиксированное значение**.

Переменная – элемент данных, который обозначается в программе **идентификатором** и может в процессе выполнения программы **изменять свое значение**.

Константа обозначается:

- Своим значением (42, 33);
- Именем (Pr, maxsize).

Переменная обозначается:

- Именем (x, BufSize).

Пример: X := X + 5;

X – переменная, 5 – константа

Обрабатываемые **данные** могут быть **сгруппированы** в **структуры данных**, объединенных **общим именем**. Такое объединение позволяет обрабатывать как **отдельные** элементы структур, так и **всю** структуру как **единую** переменную. Примерами структур данных являются **массивы, множества, записи**.

Тип значений определяет:

- 1) **множество** различных **значений**, которые могут принимать программные элементы данного типа;
- 2) **свойства** данных значений;
- 3) **операции**, которые могут выполняться над этими значениями.



Все типы значений в Паскале делятся на две группы:

- 1) **основные** (простые);
- 2) **производные** (структурные).

Основные типы – это типы, состоящие из **единственного** элемента данных (так называемые **тривиальные** структуры данных).

Основные типы делятся на:

- **ссылочный** тип;
- **скалярные** типы.

Скалярные типы в свою очередь делятся на:

- **стандартные** типы – это типы, зафиксированные в языке (предопределенные типы); например, типы **Integer, Real, Char**;
- **описанные** типы – это типы, определенные программистом; например, **перечислимый тип**.

В языке Паскаль большинство скалярных типов обладает свойством **перенумерованности** (упорядоченности). Оно обозначает, что среди элементов соответствующего типа данных установлен линейный порядок. Таким образом, относительно любых двух элементов данного типа определено, какой из них предшествует другому.

Производные (структурные) типы – это типы, которые образуются на основании **других типов** (как основных, так и производных). В общем случае производный тип состоит из **нескольких элементов данных**.

В Паскале производные типы данных делятся на:

- стандартные (предопределенные) – например, типы **String, Text** и др.
- описанные – например, **массивы**, записи и др.

В языках программирования наподобие Delphi, ориентированных на то, чтобы разработанная программа работала максимально правильно и надёжно, компилятор **использует информацию** о типах данных, **чтобы проверить** правильность действий над этими данными, и если в программе имеются какие-то несоответствия, компилятор укажет на это, а программа не будет скомпилирована. В результате многие логические ошибки в программе удаётся обнаружить ещё до запуска программы, причём автоматически.

20. Целочисленные типы данных. Форматы. Диапазоны представления данных. Представление в памяти. Пример.

- **Знаковые**
- **Беззнаковые**

ShortInt $-2^7/2^7-1$

SmallInt $-2^{15}/2^{15}-1$

LongInt $-2^{31}/2^{31}-1$ (2147483647)

Byte $0/2^8-1$

Word $0/2^{16}-1$

LongWord $0/2^{32}-1$

Integer $-2^{31}/2^{31}-1$ размер зависит от целевой платформы

- Множество целочисленных значений является **перенумерованным**
- **Целочисленные литералы: десятичные** (42, -542), **16-ичные** (\$42, \$COO1CODE)
- **Тип целочисленного литерала** – тип с **наименьшим диапазоном, включающим** значение литерала

В памяти машины целые числа хранятся в виде двоичного числа с фиксированной точкой (ФТ) и занимают (как видно из таблицы) 1 байт, 2 байта или 4 байта.

Внутреннее представление целых чисел

Для чисел со знаком **старший бит** соответствующего поля памяти, отведенного для хранения числа, считается **знаковым**. Если в нем **0** – число **положительное** (знак +), **1** – **отрицательное** (знак –). **Отрицательные** числа в памяти машины хранятся в **дополнительном** коде.

Дополнительный код (ДК) числа i образуется путем инвертирования (замены значения на противоположное) всех значащих разрядов прямого кода числа и прибавления 1 к самому младшему разряду. Аналогично производится преобразование из дополнительного кода в прямой код.

21. Целочисленные константы. Операции над целочисленными данными.

Пример.

Множество целочисленных значений является **перенумерованным**, порядковым номером каждого значения целочисленного типа является само это значение.

В программах целые числа обычно используются в качестве счетчиков (для управления количеством повторений цикла) и в качестве индексов. При обработке целых чисел используется арифметика с ФТ, округления не производятся, вычисления выполняются точно. Множество целочисленных значений является перенумерованным, порядковым номером каждого значения целочисленного типа является само это значение.

Целые константы делятся на два типа – **десятичные** и **шестнадцатеричные**:

Признак шестнадцатеричной константы – знак \$ перед числом.

Примеры записи десятичных констант: +16, 25, -48

Примеры записи шестнадцатеричных констант: \$F, \$9A0

Целой константе присваивается целочисленный тип с наименьшим диапазоном, включающим значение этой константы.

В двухместных арифметических и логических операциях (кроме операции деления /) тип результата будет таким же, как **общий** тип **обоих** операндов. Перед выполнением операции **оба** операнда **преобразуются** к их **общему** типу. **Общим** типом является целочисленный тип с наименьшим диапазоном, включающим все возможные значения **обоих** типов.

Арифметические операции	+	Одноместная	Сохранение знака	Целый
	–	Одноместная	Отрицание знака	Целый
	+	Двухместная	Сложение	Целый
	–	Двухместная	Вычитание	Целый
	*	Двухместная	Умножение	Целый
	/	Двухместная	Деление	Вещественный
	<i>div</i>	Двухместная	Целочисленное деление	Целый
	<i>mod</i>	Двухместная	Остаток целочисленного деления	Целый
Логические операции	<i>not</i>	Одноместная	Поразрядное дополнение целого	Целый
	<i>and</i>	Двухместная	Поразрядное логическое умножение (И)	
	<i>or</i>	Двухместная	Поразрядное логическое сложение (ИЛИ)	
	<i>xor</i>	Двухместная	Поразрядное логическое исключающее ИЛИ	
Операции сдвига	<i>shl</i>	Двухместная	<i>i shl j</i> – сдвиг влево значения <i>i</i> на <i>j</i> битов	Тип <i>i</i>
	<i>shr</i>	Двухместная	<i>i shr j</i> – сдвиг вправо значения <i>i</i> на <i>j</i> битов	Тип <i>i</i>

Операции сравнения	=	Двухместная	Равно	Логический
	<>	Двухместная	Не равно	
	<	Двухместная	Меньше	
	>	Двухместная	Больше	
	<=	Двухместная	Меньше или равно	
	>=	Двухместная	Больше или равно	

22. Встроенные процедуры и функции, определенные над целочисленными данными. Пример.

Встроенные процедуры и функции – это процедуры и функции, которые **определены** в компиляторе языка программирования. Имена встроенных процедур и функций являются **предопределенными идентификаторами**.

Подпрограмма	Вид	Описание	Тип результата
<i>Chr(x)</i>	Функция	Возвращает символ кода ASCII с заданным номером (<i>x: byte</i>)	char
<i>Ord(x)</i>	Функция	Возвращает порядковый номер скалярного аргумента <i>x</i> (<i>Ord(5) = 5; Ord(0) = 0; Ord(-5) = -5</i>)	longint
<i>Abs(x)</i>	Функция	Абсолютное значение <i>x</i>	Тип <i>x</i>
<i>Sqr(x)</i>	Функция	Квадрат числа	Тип <i>x</i>
<i>Dec(x [, n])</i> *	Процедура	Уменьшает значение <i>x</i> на величину <i>n</i> (по умолчанию на единицу; <i>n:longint</i>)	Тип <i>x</i>
<i>Inc(x [, n])</i> *	Процедура	Увеличивает значение <i>x</i> на величину <i>n</i> (по умолчанию на единицу; <i>n:longint</i>)	Тип <i>x</i>
<i>Odd(x)</i>	Функция	Возвращает True (истину), если аргумент нечетный, False (ложь) – если четный (<i>x:longint</i>)	Boolean
<i>Pred(x)</i>	Функция	Возвращает предшествующий элемент в типе аргумента (для	Тип <i>x</i>

Подпрограмма	Вид	Описание	Тип результата
		целочисленных данных возвращает $x - 1$)	
<i>Succ(x)</i>	Функция	Возвращает следующий элемент в типе аргумента (для целочисленных данных возвращает $x + 1$)	Тип x
<i>Hi(x)</i>	Функция	Возвращает старший байт своего аргумента ($x: integer$ или $x: word$)	byte
<i>Lo(x)</i>	Функция	Возвращает младший байт своего аргумента ($x: integer$ или $x: word$)	byte
<i>Swap(x)</i>	Функция	Возвращает значение, образованное сменой младшего и старшего байта своего аргумента ($x: integer$ или $x: word$)	Тип x
<i>Random(N)</i>	Функция	Возвращает случайное число из диапазона $0..N - 1$ ($N: word$)	Word
<i>Sizeof(x)</i>	Функция	Возвращает число байт, занимаемых своим аргументом (аргумент может быть именем переменной или именем типа)	Word

23. Вещественные типы данных. Форматы. Диапазоны представления данных. Представление в памяти. Вещественные константы и способы их записи в программе. Пример.

В общем случае числа с ПТ представляются **неточно**, операции над ними выполняются по правилам действий над приближенными числами. Поэтому множества значений вещественных типов в Паскале **не обладают** свойством упорядоченности (**перенумерованности**).

Вещественные константы

Вещественные константы в программе могут быть записаны в форме числа с **ФТ** или числа с **ПТ**.

В форме с **ФТ** запись константы состоит из **целой** и **дробной** частей, разделенных **точкой**. Знак необязателен.

Наличие **точки** в вещественной константе **обязательно**.

В форме с ПТ константа задается в виде десятичной **мантиссы** m и десятичного **порядка** p с необязательными знаками, между которыми помещается символ **E** (или **e**):

Мантисса m представляется **целой** или **вещественной константой** с **ФТ**. **Порядок** – целое **десятичное число**, определяющее **истинное положение запятой** в константе.

Существует пять вещественных типов для представления переменных:

- **Real** (вещественный);
- **Single** (с одинарной точностью);
- **Double** (с двойной точностью);
- **Extended** (с повышенной (расширенной) точностью);
- **Comp** (сложный тип).

Тип **Extended** еще называется **временным**, т.к. при вычислениях в сопроцессоре математических операций все остальные типы преобразуются к нему.

Тип	Диапазон	Точность (разрядность мантиссы)		Разрядность порядка (бит)	Формат (байт)
		бит	десятичных цифр		
Real	$10^{\pm 38}$	40 *	11 ÷ 12	8	6
Single	$10^{\pm 38}$	24 *	7 ÷ 8	8	4
Double	$10^{\pm 308}$	53 *	15 ÷ 16	11	8
Extended	$10^{\pm 4932}$	64	19 ÷ 20	15	10

Примечание. *) – включая скрытый (неявный) бит F_0 (старший бит мантиссы, который при передачах чисел и хранении их в памяти отсутствует, т.к. его значение в нормализованных числах равно 1 – для всех чисел, кроме числа нуль). В значениях типа Extended F_0 хранится явно.

Нормализованным числом называется число, мантисса которого попадает в диапазон $1/q \leq m < 1$

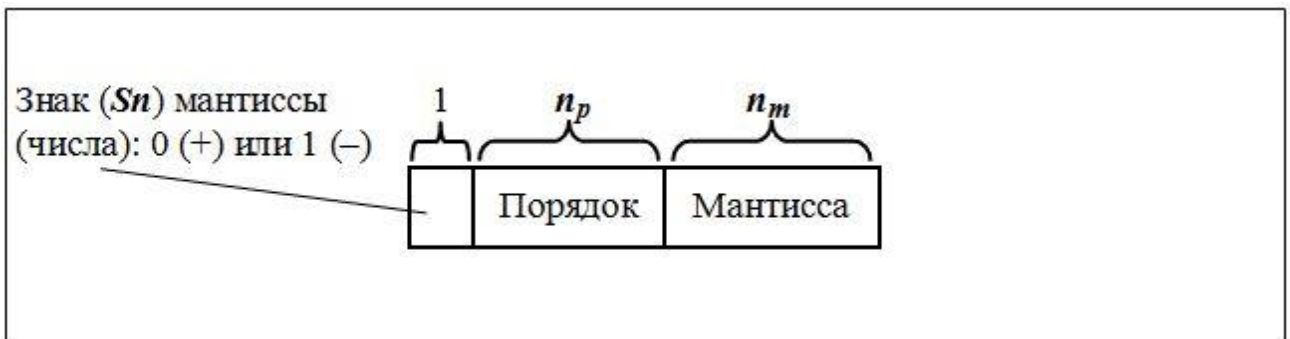
где q – основание системы счисления. Для 2СС $0.5 \leq m < 1$

то есть нормализованная мантисса представляет собой правильную дробь с единицей в старшем двоичном разряде. Например, нормализованные мантиссы могут принимать значения

0.1101

0.1000

0.1111



Значение числа равно: $m_{дв} * 2^p$

Здесь **мдв** – значение двоичного кода **мантиссы**, **р** – значение двоичного кода **порядка**.

Вещественные типы различаются разрядностью поля мантиссы (n_m) и поля порядка (n_p).

Мантисса, независимо от знака, хранится в **прямом** двоичном коде. **Порядок** хранится в виде **характеристики** – это двоичный код порядка, представленный с **избытком** (**смещением**):

$$2^{n_p-1} - 1.$$

Например, для типа Single смещение равно 127, для типа Double – 1023, для типа Extended – 16383.

Характеристика определяется выражением:

Характеристика = порядок + смещение.

Таким образом, формат вещественного числа в битах (занимаемая им область памяти) определяется выражением: $S_n + N_p + N_m$

Вещественные типы данных. Форматы. Диапазоны представления данных. Представление в памяти. Вещественные константы и способы их записи в программе. Пример.

Значениями вещественных типов являются числа с плавающей точкой (ПТ), представленные в виде мантиссы и порядка.

Например, $0,125 = 0,125 * 10^0 = 1,25 * 10^{-1} = 125 * 10^{-3} = 0,00125 * 10^2$.

В общем случае числа с ПТ представляются неточно, операции над ними выполняются по правилам действий над приближенными числами.

Вещественные константы.

Вещественные константы в программе могут быть записаны в форме числа с фиксированной точкой (ФТ) или числа с плавающей точкой (ПТ).

Примеры записи вещественных чисел в форме с ФТ: 0.25; -2.48; +31.0

Примеры записи вещественных чисел в форме с ПТ: 14.3E5; 681E-2; -5.16E-3

Вещественные типы и их представление в компьютере.

Вещественные числа в памяти компьютера описываются стандартом IEEE754.

Существует пять вещественных типов для представления переменных:

- Real (вещественный);
- Single (с одинарной точностью);
- Double (с двойной точностью);
- Extended (с повышенной (расширенной) точностью);
- Comp (сложный тип).

Диапазоны и точности представления вещественных данных:

Тип	Диапазон	Точность (разрядность мантиссы)		Разрядность порядка (бит)	Формат (байт)
		бит	десятичных цифр		
Real	$10^{\pm 38}$	40 *	11 ÷ 12	8	6
Single	$10^{\pm 38}$	24 *	7 ÷ 8	8	4
Double	$10^{\pm 308}$	53 *	15 ÷ 16	11	8
Extended	$10^{\pm 4932}$	64	19 ÷ 20	15	10

Нормализованным числом называется число, мантисса которого попадает в диапазон

$(1/q) \leq m < 1$, где q – основание системы счисления.

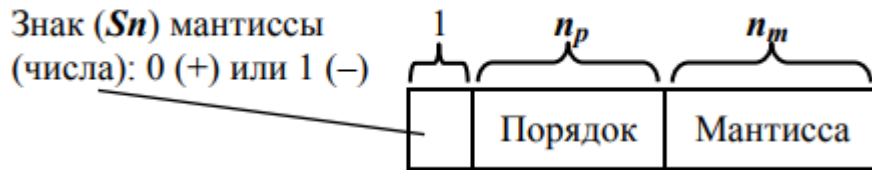
Для 2СС

$(1/2) \leq m < 1$,

то есть нормализованная мантисса представляет собой правильную дробь с единицей в старшем двоичном разряде.

Например, нормализованные мантиссы могут принимать значения 0.1101; 0.1000; 0.1111.

Представление вещественных чисел в памяти компьютера^



24. Операции над вещественными данными. Встроенные функции, определенные над вещественными данными. Пример.

Для значений вещественных типов определены следующие операции.

1) Арифметические операции:

а) одноместные

- + (сохранение знака);
- (изменение знака);

б) двухместные

- + (сложение);
- (вычитание);
- * (умножение);
- / (деление).

Арифметические операции над значениями вещественных типов дают тип результата *Real* (при отсутствии сопроцессора), *Extended* (при наличии сопроцессора).

2) Операции сравнения:

- = (равно);
- <> (не равно);
- >= (больше или равно);
- > (больше);
- <= (меньше или равно);
- < (меньше).

Операции сравнения дают тип результата *Boolean*.

Операции сравнения на точное равенство и неравенство ($=$, $<>$) над вещественными числами определены, но их лучше не использовать, т.к. для обработки вещественных чисел используется приближенная арифметика с ПТ.

Например, при выполнении цикла по условию $X <> Y$ может оказаться, что за счет погрешности вычислений X никогда не станет равным Y , хотя логика программы будет верной. Поэтому программа не осуществит выход из цикла.

Встроенные функции, определенные над вещественными данными

Над данными вещественных типов определен ряд встроенных функций

Функция	Описание	Тип результата
<i>Round(x)</i>	Округление, результат – ближайшее к <i>x</i> целое	LongInt
<i>Trunc(x)</i>	Целая часть числа <i>x</i> независимо от знака	LongInt
<i>Int(x)</i>	Возвращает целую часть <i>x</i>	Real (или Extended)
<i>Frac(x)</i>	Возвращает дробную часть <i>x</i>	Real (Extended)
<i>Abs(x)</i>	Абсолютная величина <i>x</i>	Real (Extended)
<i>Arctan(x)</i>	<i>arctg(x)</i> . Результат в радианах.	Real (Extended)
<i>Cos(x)</i>	<i>cos(x)</i> . <i>x</i> – в радианах	Real (Extended)
<i>Sin(x)</i>	<i>sin(x)</i> . <i>x</i> – в радианах	Real (Extended)
<i>Exp(x)</i>	(экспонента; <i>e</i> в степени <i>x</i>)	Real (Extended)
<i>Ln(x)</i>	<i>ln(x)</i>	Real (Extended)
<i>Sqr(x)</i>	Возведение в квадрат	Real (Extended)
<i>Sqrt(x)</i>	(квадратный корень из <i>x</i>)	Real (Extended)
<i>Sizeof(x)</i>	Количество байтов для представления вещественного значения	Word

Имеется стандартная вещественная константа

Pi = 3.1415926536E+00

25. Символьный тип данных. Способ упорядоченности. Представление в памяти. Операции и встроенные функции, определенные над символьными данными. Пример.

Переменные типа Char объявляются в разделе описания переменных.

Константой типа Char является один из допустимых символов, взятый в **апострофы**. Если значением константы является сам **апостроф**, то он записывается дважды.

Символьный (литерный) тип Char – это скалярный тип. Значениями этого типа являются элементы расширенного набора символов (литер) кода ASCII (американский стандартный код обмена информацией). Для представления значений типа Char отводится один **байт** памяти.

Элементы множества значений типа Char считаются **перенумерованными** (упорядоченными), т.е. каждому значению типа Char поставлен в соответствие свой порядковый номер. Порядковый номер символа равен его коду ASCII.

Способ упорядочения определяется в соответствии с кодом ASCII (перечисление идет по возрастанию порядковых номеров):

- 1) наименьшие порядковые номера имеют **управляющие символы и специальные символы** алфавита языка (но специальные символы не совсем упорядочены, они встречаются в **разных** местах кодовой таблицы);
- 2) **десятичные цифры** (они упорядочены по возрастанию);
- 3) **заглавные латинские буквы** (они упорядочены по алфавиту);
- 4) **маленькие латинские буквы** (по алфавиту);
- 5) **псевдографика** (в основном варианте кодовой таблицы; в альтернативном варианте псевдографика следует после русских букв);
- 6) **заглавные русские буквы** (по алфавиту);
- 7) **наибольшие порядковые номера имеют маленькие русские буквы** (по алфавиту).

Над значениями типа Char определены только **операции сравнения**:

Тип получаемого результата – Boolean. При этом сравниваются внутренние **коды** символов (в коде ASCII), т.е. фактически, **порядковые номера** символов во множестве Char (с учетом упорядоченности).

Над значениями типа Char определены **встроенные функции**.

Функция	Описание	Тип результата
<i>Ord(x)</i>	Преобразует <i>x</i> к целочисленному типу (возвращает порядковый номер символа <i>x</i> во множестве значений типа <i>Char</i> в коде ASCII)	Longint

Функция	Описание	Тип результата
<i>Pred(x)</i>	Возвращает символ, порядковый номер которого на единицу меньше порядкового номера <i>x</i> в коде ASCII	Char
<i>Succ(x)</i>	Возвращает символ, порядковый номер которого на единицу больше порядкового номера <i>x</i> в коде ASCII	Char
<i>Uppcase(x)</i>	Возвращает большую латинскую букву, если <i>x</i> – маленькая латинская буква, иначе возвращает <i>x</i>	Char
<i>Sizeof(x)</i>	Указывает количество байтов, требуемое для представления значения типа <i>Char</i> (значение функции равно 1)	Word

26. Логический тип данных. Способ упорядоченности. Представление в памяти. Операции и встроенные функции, определенные над логическими данными. Пример.

Логический тип определяется как скалярный тип, множество значений которого состоит всего из двух значений:

False (ложь) и True (истина).

Значения логического типа упорядочены: значение False имеет порядковый номер 0, значение True имеет порядковый номер 1.

Значения типа Boolean занимают один байт памяти. (минимальная единица адресации)

Логические переменные объявляют в разделе переменных

Логическими константами является предопределенные в языке Паскаль идентификаторы

True и False.

Группа операций	Операция	Описание	Тип результата
Логические операции	Not	Одноместная операция (НЕ), результат равен <i>True</i> , если значение операнда <i>False</i> , в противном случае – <i>False</i> .	Boolean

Группа операций	Операция	Описание	Тип результата
	And	Двухместная операция (И), результат равен <i>True</i> , если значение обоих операндов <i>True</i> , в противном случае – <i>False</i>	Boolean
	Or	Двухместная операция (ИЛИ) результат равен <i>True</i> , если хотя бы один из операндов равен <i>True</i> , в противном случае – <i>False</i>	Boolean
	Xor	Двухместная операция (исключающее ИЛИ) результат равен <i>True</i> , если только один операнд имеет значение <i>True</i> , в противном случае – <i>False</i>	Boolean
Операции сравнения	= <> > >= < <=	Равно Не равно Больше Больше или равно Меньше Меньше или равно	Boolean

Функция	Описание	Тип результата
Ord(x)	Возвращает ноль, если $x = False$, единицу, если $x = True$	LongInt
Pred(X)	Для $x = True$ возвращает <i>False</i> , иначе не определено	Boolean
Succ(X)	Для $x = False$ возвращает <i>True</i> , иначе не определено	Boolean
Sizeof(X)	Указывает количество байтов для представления значения типа <i>Boolean</i>	Word

27. Выражения и их типы. Правила написания и вычисления выражений. Приоритет операций. Пример.

Выражение – формула для вычисления некоторого значения, состоящая из:

- **Операндов** (значения, с которыми ведется работа)
- **Знаков операций**
- **Круглых скобок**

- **Скалярные стандартные выражения** делятся на три типа:
- — **арифметические;**
- — **логические;**
- — **символьные.**

Операнды:

- **Константы**
- **Переменные**
- **Вызовы функций**
- **Выражения**

Приоритет операций:

1. Операции **Not @ (одноместные)**
2. Операции группы **умножения:**
* / div mod and shl shr
3. Операции группы **сложения:**
+ - xor or
4. Операции группы **сравнения:**
In = <> <= >= < >

Правила написания и вычисления выражений:

- **Первая группа – высший приоритет**
- В группе – **одинаковый** приоритет
- С **равным** приоритетом вычисляются **слева направо**
- Операнд, находящийся **между** операциями с **равными** приоритетами, связывается с операцией, находящейся **слева** от него
- Для уточнения последовательности действий использовать **скобки** (как в математике)
- **Последовательная запись двух знаков** операций **запрещена** (кроме not)
- **Многоэтажные формулы** в **одну строку**
- Знак * пропускать **нельзя**

Типы выражений

В Паскале каждое выражение определяет значение какого-то определенного **типа**, поэтому в нем могут фигурировать операнды тоже определенных типов. Над каждым типом значений определен свой **набор операций**.

Выражение называется **скалярным**, если его **операндами** являются **скалярные величины**. В результате вычисления скалярного выражения получается одно значение элемента данных.

Скалярные стандартные выражения делятся на три типа:

- **арифметические;**
- **логические;**
- **символьные.**

Тип выражения определяется типом результата его вычисления.

Арифметическое выражение – это выражение, результатом вычисления которого является **целое** или **вещественное** значение. **Операнды** арифметического выражения должны иметь **арифметический** (целый или вещественный) **тип**. В качестве операций могут использоваться **арифметические операции**, определенные для этих типов операндов.

Следует обратить внимание, что в Паскале не определена операция возведения числа X в степень Y . Для ее реализации можно воспользоваться следующими рассуждениями. $X^Y := \text{Exp}(Y * \text{Ln}(X))$

Однако следует помнить, что вычисление с помощью вышеприведенного выражения может приводить к существенным **погрешностям**. Поэтому, если Y имеет целочисленный тип, то лучше вычислять путем умножения X само на себя Y раз (например, в цикле).

Символьное выражение – это выражение, **результатом** вычисления которого является **символьный тип** данных Char (т.е. отдельные символы).

В Паскале **отсутствуют операции**, результатом выполнения которых являются данные **типа Char**. Поэтому символьным выражением может быть только **символьная константа**, **символьная переменная** или **функция**, дающая результат типа Char.

Примеры символьных выражений.

- 1) 'A' – символьная **константа**.
- 2) X – символьная **переменная**.
- 3) Pred (X) (X – переменная типа Char).
- 4) Chr (I) (I – переменная типа Integer).

Логическое выражение – это выражение, **результатом** которого является **логическое** значение (True или False). **Операнды** логического выражения

должны иметь тип **Boolean** (операндами могут быть переменные, константы, имена функций). Кроме того, для логических выражений существует специфический вид операнда – **отношение**.

В РБНФ отношение записывается так:

<Отношение> ::= <Выражение> <Операция_сравнения> <Выражение>.

В качестве **выражений** могут быть использованы выражения любого из упорядоченных типов и арифметические выражения. Оба выражения должны быть **совместимых типов**. **Результат** вычисления отношения имеет тип **Boolean**. Таким образом, в логическом выражении могут быть использованы все виды операций и все типы скалярных операндов.

Если отношение является не самостоятельным выражением, а одним из операндов логической операции, то оно должно быть заключено в круглые скобки (т.к. операции сравнения имеют наименьший приоритет).

28. Оператор присваивания и его типы. Составной оператор. Пустой оператор. Назначение. Формат. Пример.

Многие программы или фрагменты программ являются **линейными** – т.е. такими, в которых операторы выполняются строго **последовательно**, в порядке записи в тексте программы.

Наиболее часто используемым оператором линейной программы является оператор **присваивания**.

Оператор присваивания **предписывает** вычислить значение выражения, записанного в его **правой** части, и присвоить его **переменной**, имя которой записано в **левой** части. К моменту вычисления выражения все входящие в него **переменные уже должны быть определены** (иметь некоторые значения).

Тип переменной в **левой** части оператора присваивания и тип **выражения** должны быть **совместимыми** по присваиванию. Поэтому, с учетом классификации скалярных стандартных выражений, **существует три типа скалярных стандартных операторов присваивания:**

- арифметический,
- логический,
- символьный.

Арифметический оператор присваивания.

Служит для присваивания значения переменной **арифметического** типа (вещественного или целочисленного). В правой части оператора должно быть записано **арифметическое** выражение.

Примеры арифметических операторов присваивания:

X := 0;

Y := 2 * a / b;

Z := sin(c * 2 + a * a);

Все переменные должны иметь арифметический тип.

Логический оператор присваивания.

Это оператор присваивания, в **левой** части которого указана переменная типа **Boolean**. В **правой** части оператора должно быть **логическое выражение**.

Примеры логических операторов присваивания:

A := False;

B := G > L;

C := (E <> F) Or Odd(X);

D := Y = Z;

Здесь: A, B, C, D – переменные логического типа, X – переменная целого типа, E, F, G, L, Y, Z – переменные любых скалярных типов (совместимых между собой).

Символьный оператор присваивания.

Это оператор присваивания, в **левой** части которого указана переменная типа **Char**. В правой части оператора должно быть задано **символьное выражение**.

Примеры символьных (называемых также литерными) операторов присваивания:

A := 'A';

B := C;

D := Pred(B);

Здесь A, B, C, D – переменные типа Char.

Составной оператор:

Begin {<оператор>';'} end

Пустой оператор:

Пустой оператор не включает никаких символов, не выполняет никаких действий и используется в случае:

Для **использования символа ";"** после **последнего** оператора в блоке:

begin

 a := 1;

 b := a;

end

Поскольку в языке Паскаль символ ";" разделяет операторы, то в приведенном выше коде считается, что после последней ";" находится пустой оператор. Таким образом, ";" перед end в блоке можно либо ставить, либо нет.

29. Структура программы на языке Delphi. Синтаксис. Назначение стандартных модулей UNIT. Виды объявлений. Раздел операторов. Пример.

Паскаль-программа состоит из одного или нескольких независимо компилируемых модулей.

Различают два вида модулей:

- **программный модуль (основной)**
- **модуль Unit (вспомогательный)**

Программный модуль определяет программу, получающую управление от операционной системы.

Модуль **Unit** имеет **вспомогательный** характер, используется для создания библиотек подпрограмм и может применяться лишь вместе с **программным** модулем.

Программный модуль

<Заголовок программного модуля>----<uses часть>-----<программный блок>

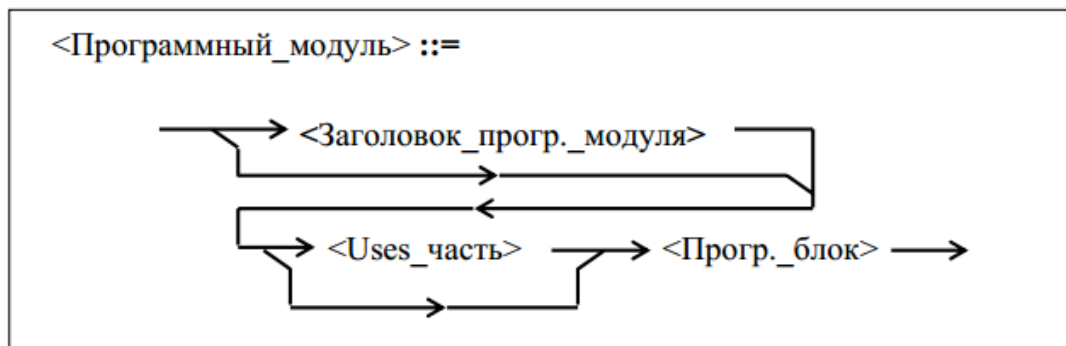


Рисунок 6.1 – Синтаксическая диаграмма структуры программного модуля

<Заголовок>::= program--- имя модуля---;

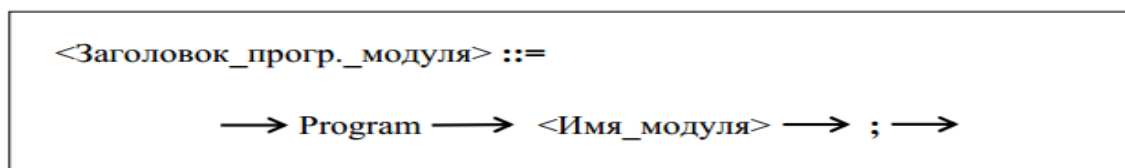


Рисунок 6.2 – Синтаксическая диаграмма заголовка программного модуля

<Uses часть>::= uses-----{‘идентификатор’;’}-----;

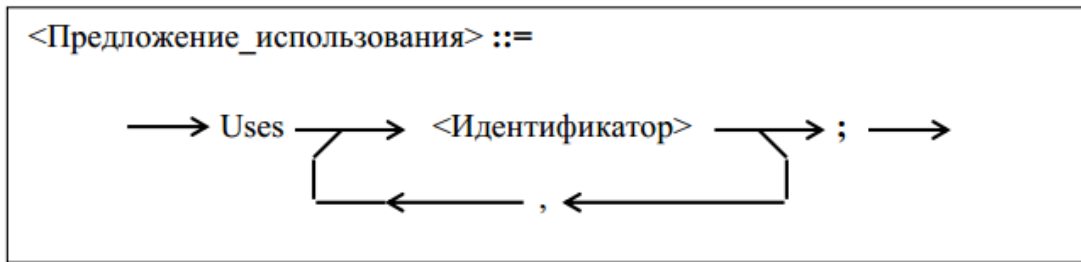


Рисунок 6.3 – Синтаксическая диаграмма предложения использования

<Программный блок> ::=
 <Раздел описаний> ----- <раздел операторов>

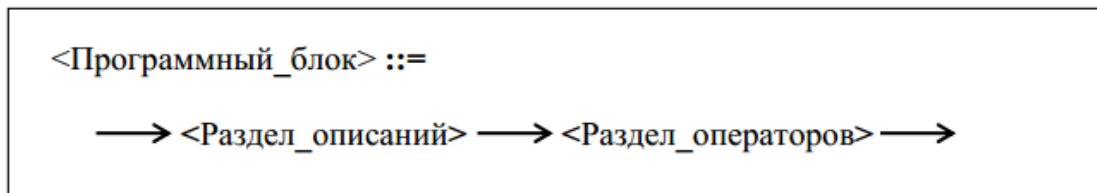


Рисунок 6.4 – Синтаксическая диаграмма тела программного модуля

Модули Unit могут быть **стандартными** и модулями, **созданными программистом**.

Существует восемь стандартных модулей Unit: System, Dos, Crt, Printer, Graph, Overlay, Turbo3, Graph3.

Unit **System** подключается автоматически, в предложении Uses он не указывается. Он содержит встроенные процедуры и функции **ввода-вывода**, обработки **строк**, **арифметические**, динамического управления памятью, поддержку арифметики с **ПТ** и т.д.

Unit **Dos** содержит процедуры обращения к **функциям MS-DOS**.

Unit **Crt** содержит переменные, процедуры и функции, используемые при управлении **выводом** информации на **экран дисплея**.

Unit **Printer** поддерживает вывод информации на **принтер** (содержит описание файла LST).

Unit **Graph** содержит процедуры для построения **графических изображений**.

Unit **Overlay** поддерживает построение программ с **оверлейной структурой (программ с перекрытиями)**.

Unit **Turbo3**, Unit **Graph3** обеспечивают **совместимость** с программами, написанными для **младших версий Паскаля**.

Виды объявлений (раздел описаний):

- объявление меток
- констант
- типов
- переменных
- процедур и функций

Раздел операторов:

begin--- {<оператор>' ;' }---end---.

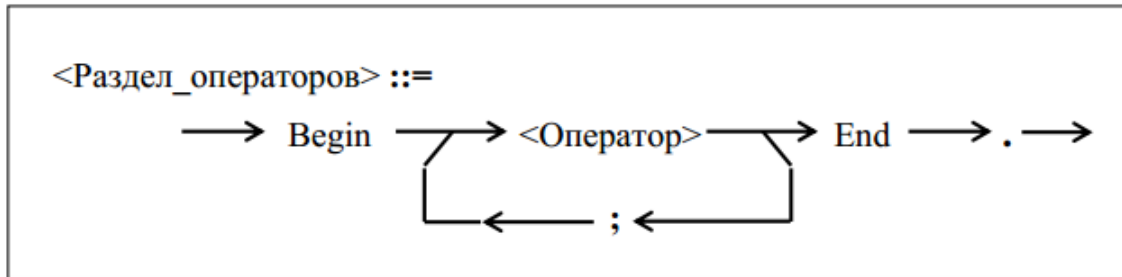


Рисунок 6.12 – Синтаксическая диаграмма раздела операторов

30. Разделы меток, типов, переменных. Назначение, синтаксис. Пример.

Метка — это идентификатор или целочисленная константа в диапазоне $0 \div 9999$, стоящая перед оператором и отделенная от него двоеточием. Метка служит для выделения оператора и играет роль имени данного оператора. Оператор с меткой называется помеченным.

Метки используются, чтобы перейти к выполнению определенного оператора, нарушая естественный последовательный порядок выполнения операторов.

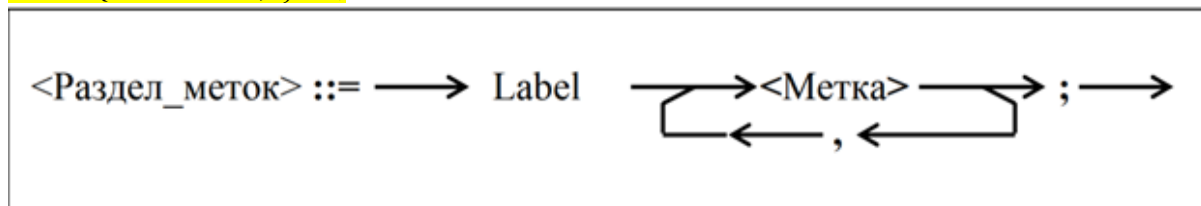
Все метки операторов должны быть различными.

Передача управления оператору, помеченному меткой, осуществляется с помощью управляющего оператора **Goto**.

Все метки, используемые в разделе операторов модуля, должны быть обязательно предварительно описаны в разделе меток.

Данный раздел начинается служебным словом Label:

Label{<metka>' ;' }---



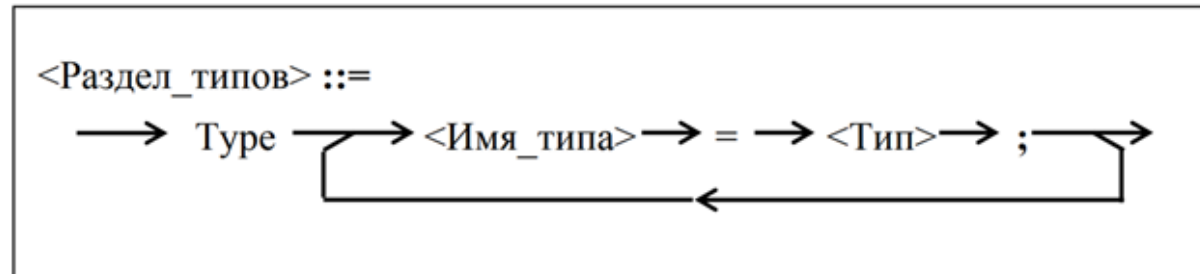
Раздел типов:

Язык Паскаль позволяет программистам **наряду со стандартными** типами значений вводить в употребление **свои типы** данных.

Каждый вводимый тип необходимо **описать** и присвоить ему **имя**. Это выполняется в разделе типов.

Type---{<имя типа>---=---<тип>---;}

<тип>::= (<имя типа>|<задание типа>) (описанный ранее или нестандартный)



Раздел переменных:

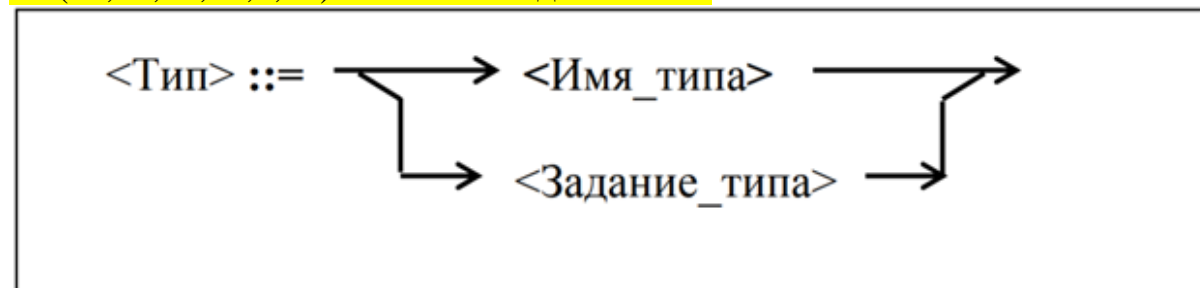
Каждая переменная, используемая в программе, обязательно должна быть объявлена (описана). Это значит, что переменной должно быть присвоено **имя** и указан **тип значений**, которые может **принимать** данная переменная.

Описание переменных содержится в разделе **переменных**. Раздел начинается служебным словом **Var**

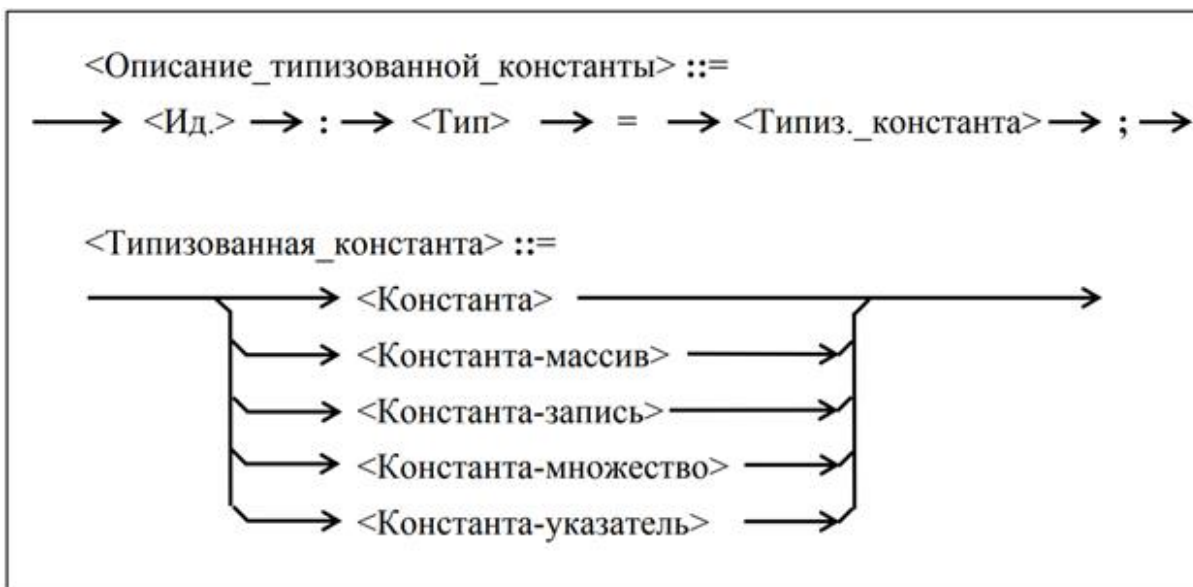
Var----{<имя переменной>-----:-----<тип>---;}

A: TDaysInMonns – явное задание типа (в приоритете)

A: (31,28,31,30,...,31) – неявное задание типа



31. Раздел констант. Типизованные и нетипизованные константы. Назначение, синтаксис. Типизованные скалярные константы. Пример.



Раздел констант позволяет назначить имена используемым в программе **постоянным** величинам.

Const----{<описание константы>|<описание типиз.константы>}---

Константы могут быть двух видов:

- ☐ **типизированные;**
- ☐ **нетипизированные.**

Описание **нетипизированной** константы имеет следующий синтаксис:

`<идентификатор>-----=-----<константное выражение>-----;` (**тип определяется типом данного выражения**)

Объявление констант может быть особенно полезным, если какая-либо величина встречается в тексте программы многократно.

Типизированная константа аналогична инициализированной переменной.

Инициализированная переменная – переменные, которым при входе в блок присваиваются начальные значения.

Описание **типизированной** константы:

`<идентификатор>-----:-----<тип>-----=-----<типизированная константа>`
`<типизированная константа>::={<константа>|<константа массив>|<константа запись>|<константа множество>|<константа указатель>}`

Константа выражение – типизированная скалярная константа.

Константное выражение – это выражение, вычисляемое на этапе компиляции.

Константное выражение **не может содержать** имен переменных и **типизированных** констант, оператора получения адреса @ или включать обращение к **определенным пользователем функциям**. В константных выражениях допустимо обращение к следующим стандартным функциям: Abs, Chr, Hi, Lo, Length, Odd, Ord, Pred, Ptr, Round, Sizeof, Succ, Swap, Trunc.

В константных выражениях могут быть использованы **только имена типизованных констант**.

32. Оператор GOTO. Формат. Назначение. Ограничения использования. Недостатки. Пример.

Его еще называют **оператором безусловного перехода**.

Данный оператор служит для **безусловной передачи управления** помеченному оператору. Оператор Goto указывает, какой оператор должен выполняться **следующим**.

<Оператор_goto> ::=
Goto → <Метка> ->

```
Label 1,2 ;  
...  
i:=1;  
1:  if i > 10 then  
    goto 2;  
    writeln ( i : 7 );  
    Inc(i) ;  
    goto 1 ;  
2:  writeln ('Mission complete') ;
```

. В языке имеется ряд **ограничений** на использование операторов Goto:

- 1) с помощью Goto нельзя **переходить внутрь производных операторов**, не содержащих данный оператор Goto (**составного оператора**, операторов **For, Repeat, While, If, Case, With**);
- 2) с помощью оператора Goto запрещен переход **из одной альтернативы в другую в выбирающих операторах** (If, Case);
- 3) с помощью оператора Goto нельзя **входить в подпрограмму или выходить из нее**.

В рамках парадигмы **структурного программирования** использование оператора goto считается **нежелательным**

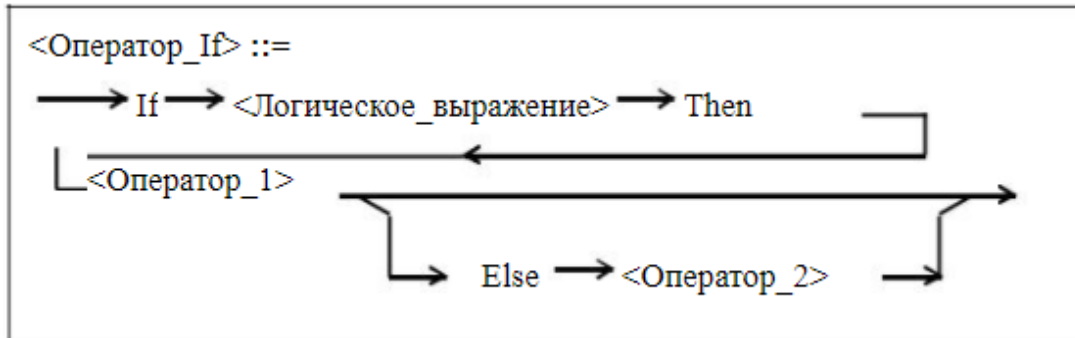
Применение оператора goto иногда может быть оправдано:

- **выход из нескольких вложенных циклов**
- **сложная обработка ошибок**
- **автоматически сгенерированный код**

Наличие оператора Goto делает программу **ненаглядной, трудночитаемой, трудноотлаживаемой**. Программа с Goto **не является структурированной**. Желательно использовать операторы Goto **минимально**, а лучше вообще не использовать.

33. Оператор IF. Формат. Назначение. Полная и сокращенная форма. Пример.

Оператор If является **производным оператором**. Относится к группе **выбирающих операторов**. Используется в **разветвляющихся** программах для **выбора** того или иного участка вычислений в зависимости от выполнения некоторого условия. Поэтому его еще называют оператором **условного перехода**.



If---<логическое выражение>----then-----<оператор 1>-----[else---<оператор 2>]

Смысл оператора: если логическое выражение истинно (принимает значение True), то выполняется <Оператор_1>. В противном случае выполняется <Оператор_2>.

После выполнения любого из <Операторов_1 или_2> следующим выполняется оператор, записанный за оператором If.

В операторе If после слов Then и Else можно записать только один оператор. Если необходимо записать группу операторов, используют операторные скобки Begin End, т.е. **составной оператор**.

Конструкция Else <Оператор_2> в операторе If необязательна. Это соответствует алгоритму: если выражение истинно, выполняется <Оператор_1>, в противном случае следующий оператор.

Конструкция оператора **If без Else** называется **сокращенной** формой оператора **If**, конструкция с Else – **полной** формой.

Возможно использование **вложенного оператора If**, т.е. оператор, записанный после Then или Else, также может быть условным.

34. Оператор CASE. Формат. Назначение. Правила выполнения. Пример.

Является **производным оператором**. Относится к группе **выбирающих операторов**.

Используется в разветвляющихся программах, если процесс нужно разветвить **более чем по двум** возможным направлениям. Является обобщением условного оператора If.

Case----<выражение>----of----{ {<диапазон>---,}-----:-----<оператор>-----;}-----[;]--
 --[else----<оператор>]---[;]----end
 <диапазон>::=<константа 1>-----[.-----<константа 2>]

Пример:

Case Symbol of

```
'A'..'Z','a'..'z':   writeln('Letter');  
'0'..'9':           writeln('Digit')  
else                 writeln('Other character')  
end
```

- выполняется только **одна** ветвь – та, которая **соответствует** значению селектора (**выражения**)
- выражение-селектор должно быть **перенумерованного** типа кроме Longint, Word, String
- если подходящей ветви **нет**, выполняется ветвь **else**
- если нет и её – не выполняется **ни одна из ветвей**

Селектор — **один или несколько диапазонов**, задающие условие выполнения этой ветви.

Значения, которыми помечаются ветви, **не могут пересекаться**. Попытки нарушения этого условия приводят к **ошибке компиляции**

Выполнение:

- 1) вычисление селектора
- 2) выбор из всех операторов того, у которого среди значений констант выбора имеется значение из пункта 1)
- 3) если 2) нет, то выполняется ветвь, помеченная Else
- 4) если и 3) нет, то выполняется оператор следующий, за Case

35. Оператор цикла с параметром. Формат. Назначение. Правила выполнения. Пример.

Цикл For используется для программирования циклов с **известным числом повторений**.

Параметром цикла называется **переменная**, служащая для **проверки условия выполнения цикла**.

For---<идентификатор>----:=-----<выражение 1> ----(to|downto)----<выражение 2>--
---do-----<оператор>-----

<Выражения 1 и 2> – это выражения того же типа, что и параметр цикла.

<Оператор>- это тело цикла. Он может быть только один (можно составной)

Выполнение

Перед каждым выполнением тела цикла происходит сравнение **текущего значения параметра** с его **предельным значением**, определяемым

<Выражением_2>. Если текущее значение меньше либо равно (в случае To) предельного значения или больше либо равно (в случае Downto) предельного значения, тело цикла выполняется

- перебирает значение переменной цикла из заданного диапазона
- Для каждого из значений выполняет тело цикла/оператора
- Следующее значение переменной цикла получает:

1)увеличение на 1, если **to**

2) уменьшение на 1, если **downto**

Ограничения:

- Внутри цикла запрещено изменять параметр цикла
- После завершения цикла параметр неопределен
- Можно использовать все перенумерованные типы

```
s := 0;  
for i := 1 to n do  
  s := s + i*i;  
Writeln(s);
```

36. Оператор цикла с постусловием. Формат. Назначение. Правила выполнения. Пример.

В общем случае используется для программирования цикла с **заранее неизвестным числом повторений**, окончание которого зависит от какого-либо условия.

Тело цикла выполняется до тех пор, пока <Логическое_выражение> принимает значение False. Как только после очередного выполнения цикла <Логическое_выражение> станет равным True, выполнение цикла прекращается.

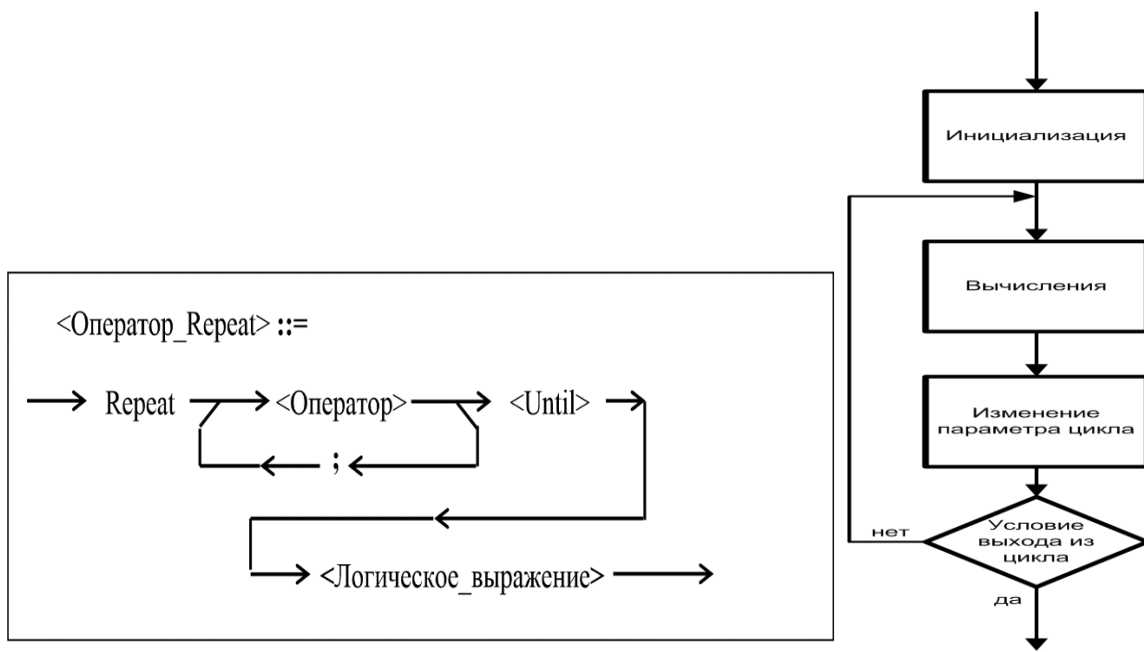
Проверка условия осуществляется каждый раз после выполнения тела цикла.

Поэтому тело цикла будет выполнено по крайней мере **один раз**.

Repeat----{<оператор>---;}----Until---<логическое выражение>---

Цикл с постусловием — цикл, в котором условие проверяется **после** выполнения тела цикла. Отсюда следует, что тело **всегда выполняется** хотя бы один раз.

Формат оператора Repeat:



Пояснение к схеме: Тело цикла выполняется до тех пор, пока <Логическое_выражение> принимает значение False. Как только после очередного выполнения цикла <Логическое_выражение> станет равным True, выполнение цикла прекращается.

37. Оператор цикла с предусловием. Формат. Назначение. Правила выполнения. Пример.

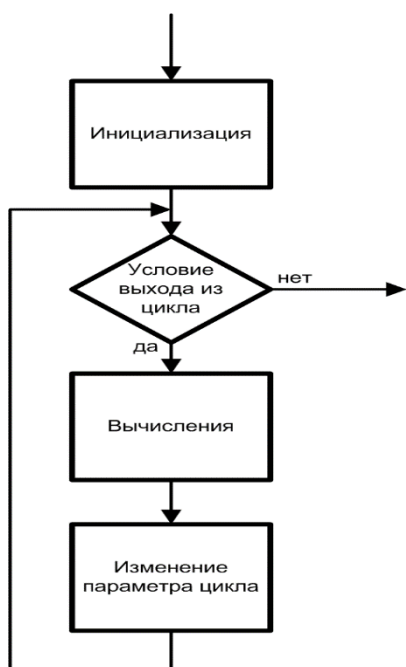
Используется для программирования цикла с заранее **неизвестным числом повторений** в тех случаях, когда тело цикла при некоторых условиях **может не выполняться ни разу**.

While---<логическое выражение>---do----<оператор>---

<Оператор> определяет тело цикла. Перед каждым его выполнением вычисляется значение <Логического_выражения>. Если оно равно True, то тело цикла выполняется. Выполнение цикла прекращается, когда <Логическое_выражение> впервые станет равным False. Если к началу выполнения цикла значение <Логического_выражения> равно False, то тело цикла не выполняется ни разу.

Если в теле цикла необходимо выполнить несколько операторов, используется составной оператор.

Оператор While является наиболее **универсальным** из операторов цикла. Его можно использовать и вместо операторов For и Repeat. Однако это не всегда удобно.



38. Сравнительная характеристика операторов цикла. Пример.

For самый быстрый, но ограничен больше других, т.к подвергается жестким оптимизациям со стороны компилятора. Нельзя изменять параметр цикла. После завершения цикла – параметр неопределен

Repeat-until по меньшей мере выполнится 1 раз, выполняется до тех пор, пока ложно условие, используется, когда заранее неизвестно число повторений

While используется, когда число повторений неизвестно, является наиболее универсальным из операторов цикла, может не выполниться ни разу.

<i>Цикл с предусловием while (пока условие истинно)</i>	<i>Цикл с постусловием repeat (до истинности условия)</i>
1. До начала цикла должны быть сделаны начальные установки переменных, управляющих условием цикла, для корректного входа в цикл.	
2. В теле цикла должны присутствовать операторы, изменяющие переменные условия так, чтобы цикл через некоторое число итераций завершился.	
3. Цикл работает пока условие истинно (пока True).	3. Цикл работает пока условие ложно (пока False).

4. Цикл завершается, когда условие становится ложным (до False).	4. Цикл завершается, когда условие становится истинным (до True).
5. Цикл может не выполниться ни разу, если исходное значение условия при входе в цикл равно False .	5. Цикл обязательно выполняется как минимум один раз.
6. Если в теле цикла требуется выполнить более одного оператора, то необходимо использовать составной оператор.	6. Независимо от количества операторов в теле цикла использование составного оператора не требуется.
<i>Цикл со счетчиком for</i>	
1. Начальная установка переменной счетчика циклов до заголовка не требуется.	
2. Изменение в теле цикла значений переменных, стоящих в заголовке цикла, не допускается.	
3. Количество итераций цикла неизменно и точно определяется значениями нижней и верхней границ и шага цикла.	
4. Нормальный ход работы цикла может быть нарушен оператором goto или процедурами Break и Continue .	
5. Цикл может не выполниться ни разу, если шаг цикла будет изменять значение счетчика от нижней границы в направлении, противоположном верхней границе.	

39. Операторы Continue и Break. Пример.

Данные операторы предназначены для **гибкого управления** операторами циклов For, While, Repeat.

Оператор **Continue** осуществляет передачу управления на **конец тела цикла**.

Данный оператор используется, если при некоторых условиях тело цикла или его часть выполнять не нужно. Осуществляет переход на конец тела цикла, после чего выполняется анализ условия дальнейшего выполнения цикла.

Является по сути оператором goto с ограничениями

Оператор **Break** служит для **безусловного выхода** из операторов For, Repeat и While. (не по условию цикла)

Является по сути оператором goto с ограничениями

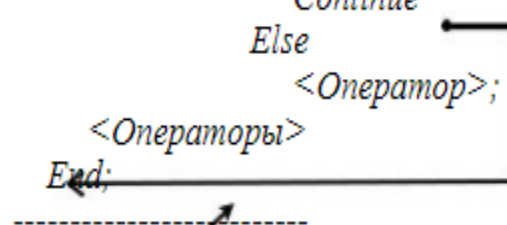
Операторы break и continue нарушают принципы структурного программирования т.к.

- Break позволяет создать цикл с несколькими выходами
- Continue позволяет создать конструкции с несколькими выходами внутри цикла
- Тем не менее, использование иногда допускается

Пример 7.18.

Фрагмент программы, использующей оператор Continue.

```
-----  
For I := 1 To N Do  
  Begin  
    <Операторы>;  
    If X > 0 Then  
      Continue  
    Else  
      <Оператор>;  
    <Операторы>  
  End;  
-----
```

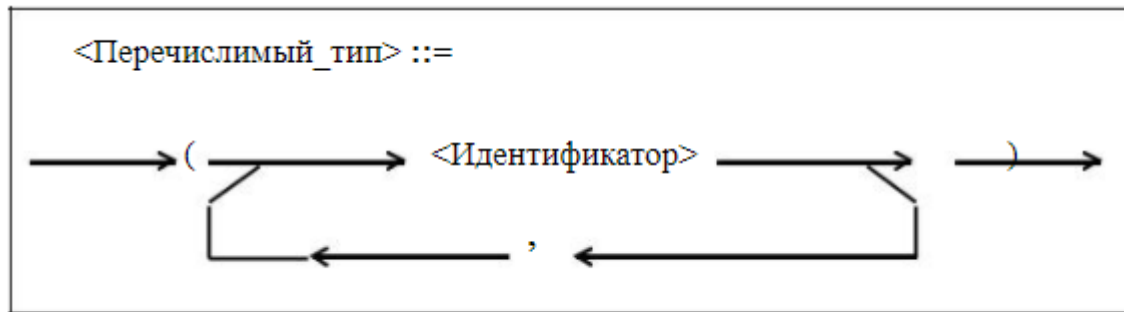


По этой ветви осуществляется выход на конец тела цикла, затем изменяется значение параметра цикла *I* и выполнение тела цикла при следующем значении данного параметра.

40. Перечислимый тип данных. Формат задания. Способ упорядоченности. Представление в памяти. Операции и встроенные функции, определенные над данными перечислимого типа. Пример.

Перечислимый тип задаётся перечислением всех своих значений. Каждый идентификатор считается константой этого типа.

<задание перечислимого типа> ::= ---- '(---- { <идентификатор> ---- ['=' ---- <константное выражение>] ---- ', } ----)'



В памяти значения перечислимых типов хранятся как целые числа, причём, если ничего не предпринимать, то литералам, использованным при объявлении, эти числа назначаются последовательно, начиная с 0.

Количество памяти, выделяемое под переменную перечислимого типа, определяется двумя факторами:

- ☐ количеством и значениями этого типа;
- ☐ настройками компилятора.

По умолчанию размер перечислимого типа совпадает с размером минимального целочисленного типа, необходимого для хранения чисел, которые поставлены в соответствие значениям этого перечислимого типа.

Тем не менее, имеется директива компилятора `{Z}`, которая позволяет задавать минимальный размер для переменных перечислимого типа. `{Z1}` устанавливает минимальный размер в 1 байт, `{Z2}` — в 2 байта, при `{Z4}` любая переменная перечислимого типа будет занимать не менее 4 байт. Для совместимости с ранними версиями Delphi допускаются также записи `{Z+}` и `{Z-}`, которые эквивалентны `{Z1}` и `{Z4}` соответственно. Эта более тонкая настройка бывает необходима при взаимодействии с библиотеками, написанными на других языках, при работе с файлами сложных форматов и т.п. Ещё одна полезная возможность — явное задание значений, соответствующих литералам перечислимого типа. Например:

```
type
  TFieldSize = (
    fsTiny = 5,      // 5
    fsSmall,        // 6
    fsMedium = 10,  // 10
    fsLarge,        // 11
    fsGiant = fsTiny + fsMedium // 15
  );
```

В основном применение этой возможности такое же, как и у директивы `{Z}`. Каждое явное задание значения становится новой точкой отсчёта, и каждый последующий литерал, не имеющий явно заданного значения, получает значение на 1 больше. В приведённом примере значения, которыми будут представлены в памяти литералы перечислимого типа, указаны в комментариях.

Идентификаторы, используемые в качестве литералов перечислимых типов, не должны совпадать с идентификаторами переменных, констант, типов и значений других перечислимых типов.

Над значениями перечислимого типа определены только операции сравнения.

Большим считается то значение, которое в памяти представлено большим числом. Кроме того, доступны следующие функции: Определённые для всех перенумерованных типов:

Ord(x) Функция Возвращает порядковый номер значения аргумента во множестве значений типа Integer

Pred(x) Функция Возвращает значение, следующее во множестве значений типа перед значением аргумента Тип x

Succ(x) Функция Возвращает значение, следующее во множестве значений типа после значения аргумента Тип x

SizeOf(x) Функция Возвращает размер типа в байтах Integer

Как нетрудно догадаться по первым трём функциям, значения перечислимых типов обладают свойством перенумерованности. Порядковый номер значения, возвращаемый функцией Ord, совпадает с числовым представлением этого значения в памяти.

41. Тип данных диапазон. Формат задания. Базовый тип. Способ упорядоченности. Операции и встроенные функции, определенные над данными типа диапазон. Пример.

Так же ограниченный тип, интервальный тип

Создается путем накладывания ограничений на уже существующий тип

```
<Тип_диапазон_скалярного_типа> ::=  
——><Константа1> —> .. —> <Константа2> ——>
```

- Тип, на который наложили ограничения, называется базовым
- Базовым может быть любой скалярный перенумерованный тип (т.е. все, кроме вещественных)
 $\langle \text{тип диапазон} \rangle ::= \text{---} \langle \text{константа 1} \rangle \text{---} .. \text{---} \langle \text{константа 2} \rangle \text{---}$
- Тип диапазон – перенумерованный
- Порядковый номер значения совпадает с порядковым номером этого значения в базовом типе
- Применимы все операции, которые применимы к базовому типу

Достоинства типа диапазон:

- 1) позволяет транслятору **экономнее использовать память** при представлении значений переменных;
- 2) **обеспечивает возможность контроля как на этапе трансляции, так и во время выполнения программы за корректностью присваиваний**, что помогает исправлять ошибки в программе;
- 3) **обеспечивает наглядную форму** представления решаемой задачи.

Тип диапазон широко применяется в комплексе с производными типами, в частности, в задачах обработки массивов.

- 1)Любой перенумерованный тип;

2) Количество элементов массива определяется количеством возможных значений этого типа;

//типа пример

type

TMyArray = array[1..10] of integer;

TSomeArray = array[Boolean] of Byte;

Tothearray = array[Char] of Word;

В памяти массив представляет собой последовательно расположенные друг за другом переменные базового типа.

Элементами массива могут быть массивы (многомерные массивы)

Две формы записи:

- Сокращенная
- Полная

A: array [1..100] of array [1..10] of integer; (полная)

B: array [1..100,1..10] of integer; (сокращенная)

Обращение к многомерному массиву:

- A[I,j,k]
- A[i][j][k]

Оба способа взаимозаменяемы

- A[i][j,k]

Выход за границы массива - грубейшая ошибка!!!

- Может приводить к повреждению данных .
- В некоторых языках (например ,C/C++) приводит к неопределенному поведению.
- Часто такая ошибка долго не проявляется и обнаруживается при внесении значительных изменений в программу.

Подмассив — массив меньшей размерности, являющийся частью многомерного массива.

43. Действия над массивами и над их элементами. Одномерная и многомерная константа-массив. Формат задания. Назначение. Пример.

Инициализация – присвоение всем элементам массива исходного значения

Над полной переменной массива не определено никаких операций, кроме присваивания, причем полной переменной может быть присвоено только значение полной переменной того же типа.

Low(x) Функция Возвращает значение индекса для первого элемента массива (минимальное значение индекса) Тип индекса

High(x) Функция Возвращает значение индекса для последнего элемента массива (максимальное значение индекса) Тип индекса

Length(x) Функция Возвращает количество элементов в массиве Integer

SizeOf(x) Функция Возвращает размер массива в байтах Integer

Над индексированными переменными (элементами массива) определены те же самые операции, процедуры и функции, что и для базового типа массива. Другими словами, элемент массива ведёт себя так же, как и обычная переменная того же самого типа.

Для инициализации массива в языке Паскаль могут быть использованы типизованные константы типа массив.

```
//
type
    TArray = array [1..5] of Byte;
const
    A:Tarray = (12,6,73,8,2);
//
//
type
    TMatrix = array [1..2,1..2] of integer;
const
    M:TMatrix = ( (11,2),(55,-9));
```

Константа массив:

<описание типиз.константы>::=---<идентификатор>---‘:’---<тип>---‘=’---<типиз.константа>---

<константа массив>::=-----{ ---‘(’---<типиз.константа>---‘,’---‘)’--- }---

<типиз.константа>::=[константное выражение|константа массив|константа указатель|константа множество|константа запись]

44. Динамические массивы. Синтаксис объявления. Представление в памяти. Основные операции над динамическими массивами. Подсчёт ссылок.

Динамический массив — это массив, количество элементов в котором может быть изменено во время работы программы.

Поскольку заранее выбранного фиксированного количества элементов у таких массивов нет, соответствующая часть объявления — перечисление типов индексов в квадратных скобках — просто исключается:

<задание типа динамический массив>::=---array---of---<тип элементов>---

Изначально такой массив содержит **0** элементов. Разумеется, пользы от такого массива немного, поэтому, когда становится понятно, сколько элементов нужно для работы, следует воспользоваться процедурой **SetLength**, например, так: **SetLength(Numbers, 10);**

Процедура **SetLength** может как **увеличивать**, так и **уменьшать** количество элементов в массиве. **Вновь добавленные** элементы считаются **неинициализированными**, т.е. их значения могут оказаться совершенно произвольными. На практике чаще всего в этих элементах оказываются нулевые значения, но это гарантируется только для некоторых типов элементов. Если же количество элементов **уменьшается**, **пропадают** (т.е. становятся **недоступными**) элементы в конце массива.

Элементы динамического массива имеют целочисленные индексы, начинающиеся с 0.

Динамические массивы **похожи** на **динамические строки**. Как и в случае с динамическими строками, **переменная типа динамический массив** содержит в себе **не сам массив**, а лишь информацию о том, где в памяти **он находится**. Это позволяет **нескольким** переменным совместно **использовать один и тот же** экземпляр массива. При этом внутри этого экземпляра отводится место для хранения **количества элементов** и **счётчика ссылок**, который в каждый момент времени содержит ответ на вопрос о том, сколько переменных на него ссылается, и позволяет организовать **автоматическое освобождение памяти** в тот момент, когда массив становится ненужным.

Тем не менее, есть и несколько отличий. Одно из них заключается в том, что в динамическом массиве **отсутствует последний элемент с нулевым значением**. У динамических массивов нет необходимости быть совместимыми с чем-либо, поэтому надобность в таком элементе отсутствует.

При **попытке изменения** строки, счётчик ссылок которой не равен 1, происходит **создание её отдельной копии** и все изменения производятся над этой **копией**

Для динамических массивов, опять же, необходимости быть совместимыми со статическими массивами не было, т.к. синтаксис их объявлений отличается. При этом создание копии массива — операция, которая в общем случае требует копирования большого объёма данных.

Применительно к динамическим массивам это означает, что сложная операция копирования должна записываться сложнее, чем просто присваивание полных переменных. На самом деле в результате выполнения **оператора присваивания** type

```
TDynArray = array of Integer;
```

```
var
```

```
  A, B: TDynArray;
```

```
...
```

```
// Заполнение массива A
```

B := A;

обе переменные, A и B, будут ссылаться на один и тот же экземпляр динамического массива. В результате попытка изменения какого-либо элемента через одну из этих переменных будет сразу же «видна» через другую переменную. Для того, чтобы создать копию динамического массива, можно использовать функцию Copy:

B := Copy(A);

В результате использования этой функции переменная **B** будет связана с отдельным экземпляром динамического массива — копией массива, на который ссылается переменная **A**.

Кроме того, функцию **Copy** можно применять к динамическим массивам не только с одним, но и с тремя параметрами, как к строкам:

B := Copy(A, 2, 3);

В результате будет создан новый динамический массив из 3 элементов — копий элементов **A[2]**, **A[3]** и **A[4]**. Все правила работы с функцией **Copy** для строк аналогичным образом применимы и к динамическим массивам.

Разумеется, многомерные динамические массивы тоже возможны. Единственное затруднение заключается в том, что запись будет чуть более многословной, чем для статических:

var

A: array of array of Real;

Приведённый пример соответствует объявлению двумерного динамического массива. Изменить его размеры можно, например, так:

SetLength(A, 5, 10);

В результате **A** станет матрицей размером **5×10**. Однако при необходимости можно построить и более сложную конструкцию:

SetLength(A, 5);

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

SetLength(A[I], I + 1); (получится пирамидка)

Чтобы понять, как это работает, следует рассматривать **A** как динамический массив, каждый элемент которого также является динамическим массивом. Результат выполнения приведённого фрагмента кода будет следующим:

Элемент массива **A[0]** содержит ссылку на экземпляр динамического массива с одним элементом, **A[1]** — с двумя и т.д.

Следует иметь в виду, что массивы (особенно многомерные) могут занимать много памяти. Использование динамических массивов позволяет ограничиться использованием только необходимого её количества, однако иногда имеет смысл **освободить занятую ненужным массивом память принудительно, не дожидаясь, пока это будет сделано автоматически.**

Для того, чтобы освободить занятую экземпляром динамического массива память, необходимо удалить все ссылки на него. Удаление ссылки можно произвести присваиванием специального значения **nil** полной переменной. В

этом случае переменная считается никуда не ссылающейся и, если она была единственной переменной, использовавшей экземпляр динамического массива, его счётчик ссылок станет равным 0 и в результате память будет освобождена. Полная переменная типа «динамический массив», которой присвоено значение `nil`, считается эквивалентной динамическому массиву нулевой длины. Именно это значение имеют все такие переменные до начала работы с ними.

44. **Динамический массив** – массив, размер которого может изменяться во время выполнения программы.

Синтаксис представления.

< Тип _динамического_ массива > ::=

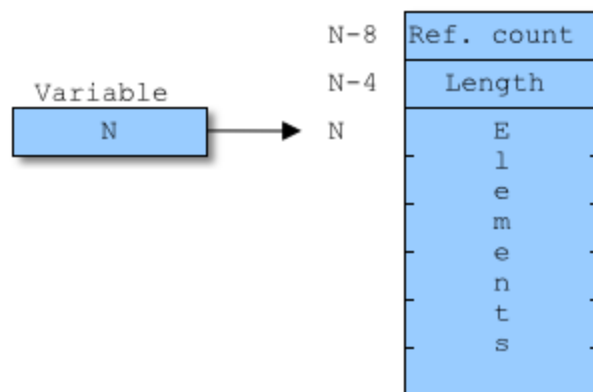
→ array → of → < Тип _элементов > →

Пример:

```
var da_MyArray : array of integer;
```

Представление в памяти.

В участке памяти ниже адреса, на который указывает ссылка динамического массива, располагаются служебные данные массива: два поля - число выделенных элементов и *счётчик ссылок*.



Если, как на диаграмме выше, `N` - это адрес в переменной динамического массива, то *счётчик ссылок* массива лежит по адресу `N - 8`, а число выделенных элементов (*указатель длины*) лежит по адресу `N - 4`. Первый элемент массива (сами данные) лежит по адресу `N`.

Для каждой добавляемой ссылки (т.е. при присваивании, передаче как параметр в подпрограмму и т.п.) увеличивается счётчик ссылок, а для каждой удаляемой ссылки (т.е. когда переменная выходит из области видимости или при переприсваивании или присваивании `nil`) счётчик уменьшается.

Для выделения памяти для динамического массива в Delphi используется процедура **SetLength**:

```
SetLength(da_MyArray,20);
```

Операции над динамическими массивами.

Как только динамический массив был распределен, вы можете передавать массив стандартным функциям **Length, High, Low** и **SizeOf**. Функция **Length** возвращает число элементов в динамическом массиве, **High** возвращает самый высокий индекс массива (то есть **Length - 1**), **Low** возвращает 0. В случае с массивом нулевой длины наблюдается интересная ситуация: **High** возвращает -1, а **Low** - 0, получается, что **High** меньше **Low**. :) Функция **SizeOf** всегда возвращает 4 - длина в байтах памяти указателя на динамический массив.

Для копирования необходимо использовать функцию **Copy**. Пример:
`da_A:=Copy (da_B);`

45. Строковые константы. Правила их записи в программе. Строковые переменные постоянной длины. Формат задания. Представление в памяти. Операции, определенные над строковыми данными постоянной длины. Пример.

Строковая константа – это последовательность любых символов, допускаемых для представления в компьютере, заключенная в апострофы

В языках Pascal и Delphi **строковые константы(литералы)** представляют собой фрагменты текста, записанные в **апострофах**

- Для отображения апострофа его повторяют дважды (экранирование)
- При подсчёте длины учитывается пробел
- Допускаются пустые константы
- Паскаль разрешает вставлять в строку символов управляющие символы. Разрешено использовать управляющие символы (начинаются со знака #), если надо записать несколько управляющих символов подряд, то между ними не должно быть никаких символов
Например: 'TURBO'#13#10'TEKST'.

Строковая переменная постоянной длины определяется как одномерный массив символов:

```
Array [1 .. N] Of Char  
    <Тип_индекса>
```


<Тип_индекса> может быть задан только с помощью типа диапазон, где N – длина строки ($N \geq 1$), определяемая как целое число без знака. При таком способе объявления строка обладает всеми свойствами массивов.

Существует особое значение строкового типа — пустая строка, т.е. строка нулевой длины. Такое значение возникает, например, когда пользователь ничего не вводит в ответ на запрос программы. Строковый литерал с таким значением записывается просто как два апострофа, один из которых «открывает» литерал, а второй тут же «закрывает»

Экранирование символа происходит самим же собой

Строки постоянной длины — это не что иное, как массивы типа Char и обладает всеми его свойствами

Особенности строковых переменных по сравнению с массивами:

- **Можно Присваивать строковые константы равной длины.** При присвоении строк разной длины лишние элементы будут отброшены
- **Сравнивать строки одинаковой длины** Над значениями строковых переменных одинаковой длины можно выполнять операции сравнения (=, <>, >, <, >=, <=)

Type

{Одномерный массив символов (строка)}

Stroka = Array [1 .. 9] Of Char;

{Двумерный массив символов (одномерный массив строк)}

Stranitsa = Array [1 .. 30] Of Stroka;

Var

Stroka1, Stroka2: Stroka;

Stranitsa1, Stranitsa2: Stranitsa;

I, K, J: 1..30;

X, Y: Boolean;

Const

Literal = 'Программа';

Begin

{К этому моменту Stroka2 должна быть определена:}

Stroka1 := Stroka2;

{Одной строке может быть присвоено значение другой строки той же длины. Здесь K-ой строке страницы присваивается значение строки:}

Stranitsa1[K] := Stroka2;

{Обращение к отдельным символам строковой переменной:}

Stroka1[I] := 'A';

Stroka1[J] := Stroka2[I];

{J-ому символу I-ой строки страницы присваивается значение K-ого символа строки:}

Stranitsa1[I, J] := Stroka1[K];

46. Строки переменной длины. Формат задания. Представление в памяти. Операции, определенные над строковыми данными переменной длины. Пример.

Тип string задает строки переменной длины. N – максимальная длина строки. N должно быть ≤ 255 . Если не указать N, то считается, что длина строки максимальная – 255. Текущая длина строки определяется длиной последнего занесенного в нее значения. Если длина присваиваемого значения больше указанной длины, лишние символы отсекаются

Переменной типа string выделяется количество байтов памяти на единицу, превышающее максимальную длину, указанную в определении типа. **В левом байте с номером 0 хранится текущая длина строки.**

Доступ к символам осуществляется по индексам. Нумерация начинается с 1 (в динамических массивах с 0)

Строковое выражение – выражение, результатом которого является значение строкового типа.

<задание типа string> ::= string-----['(N)']-----

Переменной типа String может быть присвоено значение другой строки любой длины. Если длина присваиваемой строки меньше максимальной длины данной строки, то данная переменная типа String имеет **текущую длину**. Текущая длина строки определяется длиной **последнего** занесенного в нее значения.

Если длина присваиваемого значения **превышает** указанную в объявлении максимальную длину, то **лишние** символы справа **отсекаются**.

Переменной типа String выделяется количество байтов памяти на единицу превышающее максимальную длину, указанную в определении типа. В левом байте (с номером 0) хранится текущая длина строки в двоичном коде.

Возможен доступ к отдельным символам строки типа String. При этом используются **индексные переменные**. Правила индексации аналогичны массиву символов с диапазоном индексов

Если при обращении к отдельным символам строки произойдет **выход за текущую длину строки**, то считанные из строки символы будут случайными, а присваивания элементам строки, находящимся вне текущей длины, не повлияют на значение строковой переменной.

Строковым выражением называется выражение, результатом вычисления которого является **строковое значение**. Строковые выражения состоят из строковых **констант**, строковых **переменных**, имен **функций** и **знаков операций**.

Над данными типа String определены операции **сравнения** (=, <>, >, <, >=, <=) и операция **конкатенации** (сцепления). Операция **конкатенации** имеет более **высокий** приоритет, чем операции **сравнения**.

Сравнение производится в соответствии с упорядочением символов в коде ASCII. Сравниваются символы строк последовательно слева направо до первого несовпадающего символа. **Большей** считается строка, у которой **первый несовпадающий** символ имеет **больший код** в таблице ASCII. Если строки имеют разную длину и их символы совпадают в общей части, то более **короткая** строка считается **меньшей**. Строки считаются равными, если они имеют **одинаковую текущую длину** и одни и те же **символы**.

Операция сцепления обозначается символом +. При ее выполнении две строки соединяются в одну результирующую строку. Длина результирующей строки **не должна превышать 255 символов**.

47. Встроенные процедуры и функции, определенные над строками переменной длины. Пример.

1) UpCase (Ch)

Преобразует строчную латинскую букву в прописную. В остальных случаях возвращает аргумент Ch. Параметр и результат имеют тип Char. Функция

SetLength(S, NewLength)

Процедура SetLength, применённая к короткой строке, позволяет принудительно задать ей длину. Разумеется, новая длина не должна превышать максимальной длины, заданной при объявлении переменной. Для коротких строк SetLength просто записывает новое значение в байт, хранящий текущую длину строки. Значения символов на позициях, которые ранее не входили в состав строки, не определены, т.е. могут оказаться любыми, поэтому следует не забывать их проинициализировать. Процедура

Встроенные функции:

Copy (S, from, count)

Копирует count символов из строки S, начиная с from.

Concat (s1, s2, ..., sn)

Создает строку конкатенацией строк, переданные в качестве параметров.

Length (s)

Возвращает длину строки s в символах.

Pos (St1, St2) (что ищем, где ищем)

Обнаруживает первое появление подстроки St1 в строке St2.

Встроенные Процедуры:

Delete (s, from, count)

Удаляет count символов, начиная с позиции from из строки s

Insert (substr, s, from)

Вставляет substr в строку s начиная с позиции from

Str (I, st)

Преобразует числовое значение величины I и помещает результат в строку St. Величина I должна иметь целочисленный или вещественный тип.

Val (s, value, errorcode)

Помещает в переменную value числовое значение строки s. В errorcode индекс первого недоступного символа(если таких нет, то errorcode = 0).

48. Динамические строки (Delphi-строки). Синтаксис объявления. Представление в памяти. Отличия в реализации от динамических массивов. Подсчёт ссылок.

В языке Delphi по умолчанию используется **особый** вид строк.

- **Подсчет ссылок** широко используется для решения задачи автоматического управления памятью.
- Значение счетчика ссылок – количество переменных ссылающихся на объект (строку, массив)
- Когда счетчик ссылок становится равным 0, память, занятая объектом, (строкой массивом) освобождается
- Тип string может соответствовать различным видам строк
 - 1) String [n] – всегда pascal-строки
 - 2) String – Pascal/Delphi строки в зависимости от настроек компилятора

У строковых **констант** счетчик ссылок **всегда равен -1** и **никогда не изменяется**



Особый случай Delphi-строк — строковые константы. Они также занимают место в памяти, однако память для них выделяется не динамически, а сразу при запуске программы, поэтому освобождение памяти к ним неприменимо. Чтобы отличать их от обычных Delphi-строк, значение их счётчика ссылок устанавливается в число, эквивалентное -1: все биты равны 1. Для строки с таким значением счётчика ссылок подсчёт ссылок не ведётся: счётчик ссылок всегда остаётся равным - 1

Таким образом, основной строковый тип данных — String — представляет собой универсальный тип данных, который аналогично Integer, подстраивается под актуальные возможности языка программирования и платформы. В большинстве случаев, когда конкретное представление строки в памяти (в т.ч. кодировка) не имеют значения, следует использовать именно его. В случае, если представление строки в памяти (в т.ч. кодировка) имеют значение, следует применять другие типы. При записи данных в файл следует помнить, что

переменная одного из типов, относящихся к динамическим строкам, **хранит не саму строку, а информацию о её расположении в памяти.**

Делфи-строка на самом деле является указателем

Присваивание **nil** позволяет **удалить** ссылку на строку

Значение **nil** эквивалентно **пустой строке**

Преимущества:

- **Быстрое выполнение операций** над строками (не нужно каждый раз определять длину строки)
- **Совместимость с С-строками** (позволяет передать делфи-строки операционной системе, не выполняя дополнительных преобразований)
- **Экономия памяти** (при присваивании строки не выполняется их копирование (в памяти остается 1 экземпляр))

Недостатки: **Для коротких строк**

Динамическая строка - последовательность символов неограниченной длины (теоретическое ограничение - 2Гб). В зависимости от присваиваемого значения строка увеличивается и сокращается динамически.

Delphi-строки - управляемый тип данных. Он совмещает в себе лучшие качества Pascal-(реальная длина строки перед самой строкой в памяти) и C-(наличие символа конца строки - '\0') строк.

Это дает им следующие **преимущества**:

- **быстрота** выполнения операций над ними (не нужно всякий раз высчитывать длину строки);
- **Совместимость** с С-строками (Позволяет передавать Delphi-строки операционной системе, не выполняя дополнительных преобразований);
- **Экономия** памяти (подсчёт ссылок).

Объявление динамической строки:

```
var StringName: String;
```

(в зависимости от настроек компилятора данная запись может означать и объявление Pascal-строки максимальной длины - 255)

Переменная **StringName** хранит в себе **ссылку** на первый символ строки (отмечено розовым на схеме).

При инициализации delphi-строки она равна **nil** (пустая строка, == ' ').

Индексация символов начинается с **1**.

Строки **представляются в памяти** следующим образом:

(пример строки 'Hello!')

..	0	1	2	3	4	5	6	7
----	---	---	---	---	---	---	---	---

refCnt	6	'H'	'e'	'l'	'l'	'o'	'l'	0
--------	---	-----	-----	-----	-----	-----	-----	---

Перед самой строкой располагается несколько ячеек с дополнительной информацией о данной строке (больше двух):

- **счетчик ссылок** (refCnt) - количество переменных, ссылающихся на эту строку;
- **длина строки** (элемент с нулевым индексом) - реальная длина строки.

Когда счетчик ссылок становится равным нулю - строка очищается.

У строковых констант счетчик ссылок равен -1 и никогда не изменяется.