**Раздел 1. о программном обеспечении**

* 1. **Принцип программного управления**

*Суть программного управления* - **сигналы управления** работой отдельных частей компьютера **вырабатываются внутри компьютера** в процессе вычислений.Источник информации - **код команды**.

***Команда*** указывает на **операцию** и данные для выполнения.

***Программа =*** последовательность команд, реализующих заданный **алгоритм (заданную последовательность действий** решения задачи).

**принцип хранимой программы** - программа, задающая последовательность операций, записывается в **память** компьютера **до начала** вычислений.

**система команд** = ∑ть команд, которые комп м выполнить. более 200 команд. Команда изображается в виде **двоичного кода** и имеет *структуру*:

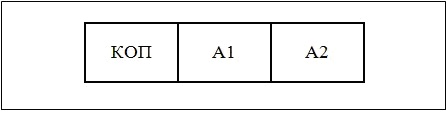


Рис 1.1 – Структура двухадресной команды (2÷6 байтов)

***КОП*** – **К**од **оп**ерации. 1 байт.

***А1, А2*** – адрес 1го и 2го **операнда** – **данные,** которые участвуют в операции.

результат записывается на **место 1го** операнда ***А1***.

***(А1 + А2)*** команды занимает 1÷ 5 байтов, а вся команда –2÷6 байтов.

**основные типы команд *(5)***:

**1) арифметические (сложение, вычитание, умножение, деление);**

**2) логические (логические сложение и умножение, отрицание, сравнение и др.);**

**3) передачи управления;**

**4) обработки адресов (специальные);**

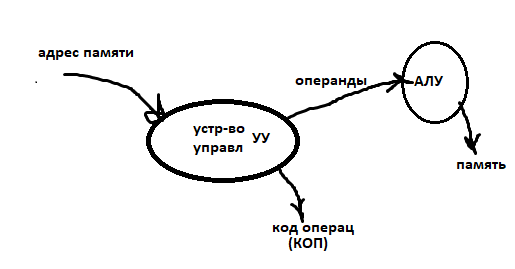
**5) ввода/вывода.**

**1.2. Автоматическое выполнение команд программы**

= **обработка входных (исходных) данных, поступивших на вход компьютера, без** участия **человека.**

**Суть:** заносятся в **память**. В **устройство управления (УУ)** передается **адрес** памяти (**АП**). УУ вырабатывает сигналы и из нее извлекается **код операции (КОП).** Если анализ результата операции требует изменения хода вычислений, то УУ изменяет адрес памяти.

Выполнив считывание **операндов** из памяти, УУ направляет их в **арифметико-логическое устройство (АЛУ** **алло**!), где производится преобразование информации. промежуточные и окончательные результаты хранятся в **памяти**. последние команды определяют операции передачи информации из памяти **в устр-тво вывода** информации (УВИдь), там она фиксируется на **носитель инфы** (**НИ**).



**1.3. Этапы постановки и решения задачи на компьютере**

1. **формулировка задачи, выделение исходных данных и формы представления результатов;**
2. **формальная (= математическая) постановка задачи – представление ее в виде уравнений и т д ;**
3. выбор **метода** решения;
4. разработк**а алгоритма;**
5. выбор **структуры данных;**
6. **программирование (на языке программирования);**
7. **тестирование и отладка;**
8. **выполнение программы.**

чтобы программа была понятна компьютеру, она должна быть составленной на ***машинном языке*** – 0 и1 (в машинных кодах). « – » = трудоемкость и стоимость программирования. => потребность в облегчении общения человека с компом. => **методы автоматизации программирования:**

1. Использование языков высокого уровня, близких к естественному человеческому, позволяют автоматически переводить прогу в машинные коды.

2. Создание библиотек стандартных программ и подпрограмм для часто используемых задач.

3. Использование современных технологий программирования.

4. Использование Case-средств разработки программ.

**1.4. Назначение и классификация языков программирования**

**Основная идея** автоматизации программирования = отказ от написания программ в машинных кодах. Программа пишется на некотором входном языке. делятся на *группы 5шт*:

1. машинно-ориентированные языки;
2. процедурно-ориентированные языки;
3. объектно-ориентированные языки (ООП);
4. проблемно-ориентированные языки;
5. языки четвертого поколения (4GL). 2-5 = машинно-независимые языки

1.4.1. Машинно-ориентированные языки

Программы могут выполняться только на тех компьютерах, для которых разработаны = *языки* ***низкого уровня*. относятся языки символического кодирования, автокоды, ассемблеры.**

***Уровень языка*** **= отношение количества машинных команд к количеству операндов языка** для написания программы**. Чем ближе это соотношение к 1, тем ниже уровень языка.**

«-» – трудоемкое программирование, машинная ориентированность, высокие требования к уровню подготовки программистов.

1.4.2. Машинно-независимые языки

2-5 группы = *машинно-независимые языки высокого уровня.* программы могут быть выполнены на любых компьютерах, имеющих транслятор с данного языка.

**2). Процедурно-ориентированные языки**

для решения **конкретного** класса задач. Со временем превратилось в универсальные. Характеристика первых версий языков:

**Фортран**. ***для решения инженерных и научно-технических задач.***

**Алгол** (1958г.). Первый язык с блочной структурой. для решения математических задач.

**Кобол** (1959г.). решение коммерческих (планово-экономических) задач.

**Бейсик** (1965г.). обучения программированию.

**Паскаль** (1971г.). Основан на Алголе. Являлся языком с блочной структурой, способствовал структурному подходу к программированию.

**Си** (1972г.). для мини-ЭВМ, использующих операционную систему UNIX. Включал средства для проектирования на уровне ассемблера, средства для эффективного использования аппаратуры. программ вычислительного характера, программ операционных систем, программ реального времени.

**Ада** (1979г.). расширение языка Паскаль для больших машин.

**3). Объектно-ориентированные языки (ООП)**

являются развитием процедурно-ориентированных языков(2). Поэтому не выделяются в отдельную группу языков.

при сохранении всех свойств процедурно-ориентированных языков **введено понятие объекта**.

версии языка Турбо Паскаль 5.5 и выше, Си++ и ряд других.

**4). Проблемно-ориентированные языки**

узкий класс однотипных задач. Исключают работу прогера по разработке алгоритмов решаемых задач. язык РПГ (Report Program Generator – **генератор отчетов**). для печати отчетов. достаточно задать информацию о форме входных и выходных документов отчета, о носителях информации.

**5). Языки четвертого поколения**

(4GL) технологии **визуального программирования**. Автоматически генерируют исходный текст программ целиком или в виде отдельных фрагментов.

визуальные среды программирования Delphi, Builder C++.

**1.5. Структура программного обеспечения СПО**

**Система программного обеспечения (СПО)** = ∑ть спец. программ д/облегчения процесса создания программ и их выполнения на компьютере + документацию.

2 группы:

1. специализированное ПО;
2. стандартное ПО.

***Специализированное ПО*** состоит из прикладных программ для решения часто встречающихся самостоятельных задач (начисления зар/платы, решения систем уравнений, 1С пр., и т.п.).

К ***стандартному ПО***относятся **системы** прогр-ния (СП) и операционные **системы (ОС), т е СП+ОС.**

**Система програм-ния** (**СП**) – ∑ть программ, описаний и инструкций **д/автоматизации** разработки программ. max облегчить общение прогера с компом

**Операционная система (ОС)** – ∑ть программ, описаний и инструкций д/**организа**ции и **контроля выполнения** программ на компьютере.

**Системные программы *-*** программы, входящие в стандартное ПО.

1.5.1.Системы программирования (**СП**) **д/автоматизации**

Основное *назначение* – max облегчить общение прогера с компом.

**Состав СП:**

1. входной язык системы – на нем пишется исходный текст проги;
2. транслятор с входного языка на машинный язык;
3. редактор связей; (***Исходный модуль***)
4. библиотеки программ; (решения типовых задач)
5. средства отладки; (ошибки)
6. обслуживающие (сервисные) программы;
7. документация.

**2. Транслятор (*Компиляторы, Интерпретаторы*)**

**Транслятор** – это прога для преобразования исходной проги на входном языке - в прогу **на машинном**. Есть 2 *вида* трансляторов:

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

б) **Интерпретаторы** – трансляторы, в которых трансляция **совмещена** с выполнением программы. Каждый оператор языка читается, расшифровывается и выполняется. Малая скорость работы, но являются более простыми по сравнению с компиляторами.

**3)Редактор связей (*Исходный модуль, Объектный модуль, Редактор связей*)**

Современные СП основаны на **модульном принципе:** программы оформляются в виде ∑ти взаимосвязанных программ. Каждая такая программа называется **модулем.**

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

**Объектный модуль** – модуль, полученный в результате **трансляции.** Он содержит текст программы **на машинном языке** и д/п инфу для объединения этого модуля с другими независимыми модулями и **настройку** модуля по месту его **загрузки** в память.

**Редактор связей** –программа **для сборки** и установления связей между модулями. Результат его работы - загрузочный или абсолютный модуль.

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

**Абсолютный модуль** – полученный в результате загрузки.

**4) Библиотека стандартных программ**

(БСП) - готовые программы для решения типовых задач. Программы в библиотеке оформляются специальным образом, облегчающим их **вызов, использование, передачу** входных данных и результатов. Они автоматически вызываются для выполнения специальными командами вызова.

**5) Средства отладки (тестирование, валидатор)**

Основная цель **этапа** отладки – выявление и исправление **ошибок** в программе. Баги.

Большинство синтаксических ошибок обнаруживается автоматически на этапе трансляции. выдают информацию о синтаксических ошибках, места ошибок и их характер, неправильное оформление подпрограмм, логические ошибки программы (÷на 0, бесконечный цикл и т.п.).

валидаторы.

После того, как прога становится работоспособной, ее **тестируют** – проверка правильности ее функционирования на различных наборах исходных данных из диапазона их допустимых значений. **многократные выполнения** программы и анализа результатов.

1. **Системные сервисные программы -** носят вспомогательный характер и упрощают работу с системой программирования.

1.5.2. Операционные системы (**ОС**)

***-*** набор программ, которые организуют и контролируют выполнение программы на компьютере без вмешательства оператора.

операционные системы типа **MS-DOS, UNIX, Windows, Linux.**

ОС обеспечивает выполнение 2 главных *задач*:

1. поддержку работы всех программ и их взаимодействия с аппаратурой;
2. предоставление пользователям общего управления компьютером.

**Раздел 2. Основы алгоритмизации**

**2.1. Алгоритм и его свойства**

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

**Алгоритм -** система правил, описывающая последовательность действий для решения задачи.

**свойства** алгоритма**:**

**1) Дискретность** – значения величин в каждый следующий момент времени должны получаться **по определенным правилам** из предшествующих значений.

2) **Определенность (детерминированность)** – каждое **правило** алгоритма должно быть **однозначным!** Значения величин, получаемых в какой-то момент времени, связаны со значениями, вычисленных ранее.

**3) Результативность (конечность)** – алгоритм должен приводить к решению задачи за **конечное число шагов**.

**4) Массовость** – его можно применить **для класса задач**, различающихся лишь исходными данными.

**2.2. Способы описания алгоритмов**

1. запись на естественном языке (**словесное**);
2. изображение в виде схемы (**графическое**);
3. запись на **алгоритм**ическом языке (**программы**).

2.2.1. Словесное описание

отличаются применяемыми метаязыками. **Метаязык** *-*  **язык** для описания **языка** программирования. Пример *типовых этапов:*

**1) Этап обработки (вычисления)** описывается в виде

V = выражение

V – переменная.

Любые вычисления можно выполнить только на этом этапе. используется *символ присваивания:* **:=**

Слева от символа (**:=**) записывается переменная, которой присваивается значение, записанное справа. Например, запись X **:=** Y означает, что переменной X присваивается значение переменной Y.

**2) Проверка условия.**

описывается в виде:

**если условие, идти к N**

Если условие выполняется, то б переход к этапу с номером (меткой) N. Иначе - переход к следующему по порядку этапу.

**3) Переход к этапу с номером N.**

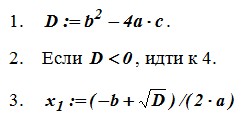
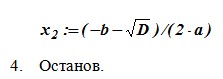
**идти к N**

**4) Конец вычислений.** Останов.

**Пример 2.1.**

Словесное описание алгоритма решения квадратного уравнения

мета1



«-» словесного описания – малая наглядность.

2.2.2. Графическое описание

- позволяет представить алгоритм в наглядной форме в виде схемы.

**Схема алгоритма** - графическое представление алгоритма, где 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 - международный стандарт **ISO 5807-85**. В 1992 г. - **ГОСТ 19.701-90:** «Единая система программной документации – Схемы алгоритмов, программ, данных и систем – Условные обозначения и правила выполнения». продолжает действовать в странах СНГ. Регламентированы условные обозначения символов, программ, данных и систем (далее – схем) и правила выполнения схем. Стандарт не ограничивает форму записей и обозначений **внутри** символов или рядом с ними для уточнения выполняемых функций.

схемы состоят из: 1. символов, 2. краткого пояснительного текста и 3. соединительных линий. Уровень детализации зависит от размеров и сложности задачи и должен быть таким, чтобы части схемы и связи между ними были понятны.

**1. Виды схем**

1. Схема **данных**

отображает **путь** данных при решении задач и определяет **этапы** обработки, а также применяемые **носители данных**.

1. Схема **программы (==схема алгоритма)**

отображают последовательность операций в программе. Понятие схемы программы аналогично понятию **схемы алгоритма.** Различие - в уровне детализации схемы.

1. Схема **работы системы**

- **отображают управление операциями** и **потоки** данных в системе: каждая программа может изображаться более чем в 1 потоке управления.

1. Схема **взаимодействия программ**

- отображают **путь активации программ** и взаимодействий с соответствующими данными. Каждая программа здесь показывается 1 раз.

1. Схема **ресурсов системы**

- отображают конфигурацию блоков данных и обрабатывающих блоков.

**2. Символы**

1. **символы данных;**
2. **символы процесса;**
3. **символы линий;**
4. **специальные символы.**

1-3 группы делится на *две* *подгруппы*:

* **Основные символы;**
* **Специфические символы.**

**2.1. Символы данных**

**-** символы, не конкретизирующие носитель данных (**носитель данных** – это источник или приемник данных: клавиатура, дискета - при вводе информации, экран дисплея, дискета - при выводе). Сюда входят символы:

**2.1.1. Данные**

Символ отображает данные, носитель которых **не определен**. определяется вводимая/выводимая инфа, если не имеет значения, откуда информация вводится или куда она выводится.

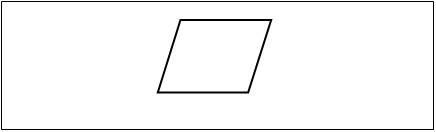


Рис 2.1 – Графическое представление символа «Данные»

Например, надо отразить ввод значений **Х**, **У** и вывод значения **Z**, причем конкретный источник и приемник информации неважны.

* + 1. **Запоминаемые данные**

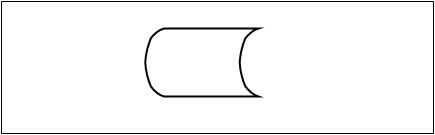


Рис 2.2 – Графическое представление символа «Запоминаемые данные».

Символ отображает **хранимые данные**. Конкретный носитель данных не определяется. Используется в схемах для изображения **результирующей информации**, которую надо запомнить, тип приемника инфы **не важен**.

С**пецифические символы данных -**  **конкретизирующие носитель** входных/выходных данных:

**2.1.3.Оперативное запоминающее устройство (ОЗУ)**

Символ отображает данные, хранящиеся в **(ОЗУ).**

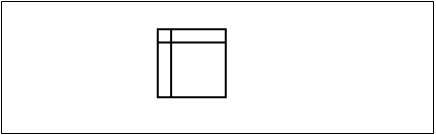


Рис 2.3 – Графическое представление символа «ОЗУ»

**2.1.4. Запоминающее устройство (ЗУ)с прямым доступом**

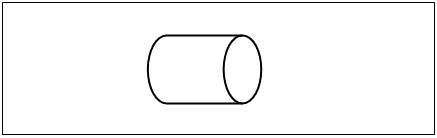


Рис 2.4 – Графическое представление символа « ЗУ с прямым доступом»

Символ отображает данные, хранящиеся в ЗУ с прямым доступом (например, **магнитный диск, флешка**).

**2.1.5. Ручной ввод**

Символ отображает данные, вводимые вручную с устройств любого типа (например, с **клавиатуры**).

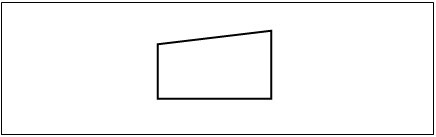


Рис 2.5 – Графическое представление символа «Ручной ввод»

**2.1.6. Дисплей**

Символ отображает данные **на отображающем** устройстве (например, на экране дисплея).

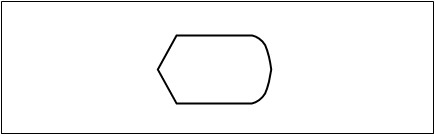


Рис 2.6 – Графическое представление символа «Дисплей»

**2.2. Символы процесса**

- отображает **функцию обработки данных** любого вида (выполнение определенной операции или группы операций).

**2.2.1. Процесс**

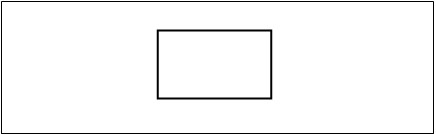


Рис 2.7 – Графическое представление символа «Процесс»

**Вычислительные операции** любого вида.

К специфическим относятся символы процесса узкого назначения:

**2.2.2. Предопределенный процесс**

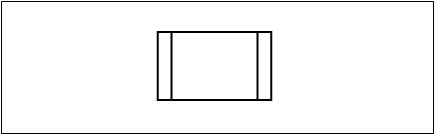


Рис 2.8 – Графическое представление символа «Предопределенный процесс».

процесс из 1 или нескольких **операций** или шагов программы, которые **определены в другом месте**. например, подпрограммы, модули программ.

**2.2.3. Подготовка**

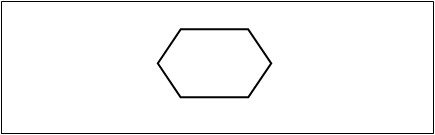


Рис 2.9 – Графическое представление символа «Подготовка»

отображает **модификацию команды** (/группы команд) д/**воздействия на следующую функцию** (установка переключателя, модификация индексного регистра или инициализация программы).

+ д/изображения работы программ **на языках низкого уровня (**Ассемблер изменяет значения индексного регистра или флагов микропроцессора). Эти действия будут изображены «Подготовкой».

**2.2.4. Решение** if … then… else

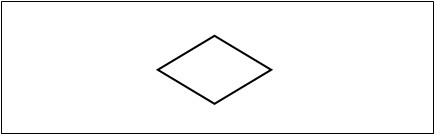


Рис 2.10 – Графическое представление символа «Решение»

отображает функцию **переключательного типа**: 1 вход и ряд альтернативных выходов, 1 из которых активизируется после вычисления условий внутри символа. Соответствующие результаты вычисления записываются рядом с линиями, отображающими эти выходы.

**2.2.5. Граница цикла**

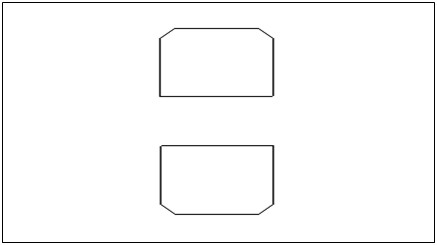


Рис 2.11 – Графическое представление символа «Граница цикла»

состоит из 2 частей, отображающих **начало и конец цикла**. Обе части символа должны иметь 1 и тот же **идентификатор:** схеме каждого цикла должно быть присвоено **имя - оно** записывается в обоих частях символа. Условия выполнения цикла (инициализация, завершение, приращение параметра и т.д.) помещаются в 1й или 2й части символа (в зависимости от расположения операции, проверяющей условие).

Например, при изображении **цикла с предусловием** (**цикл *While*)** и цикла **с параметром** (**цикл *For*)** условие выполнения цикла записывают в верхней части символа, а **цикла с постусловием** (**цикл *Repeat***) условие выхода из цикла - в нижней части.

обратить внимание: при использовании символа «Граница цикла» цикл изображается без обратных связей.

**3. Символы линий**

Подгруппа *основных символов линий* содержит (!) символ.

**Линия**

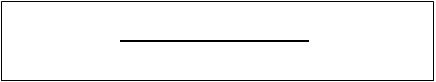


Рис 2.12 – Графическое представление символа «Линия»

для повышения удобочитаемости к линии м б добавлены стрелки:

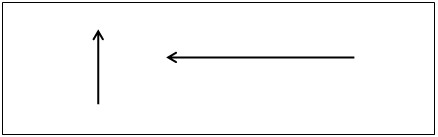


Рис 2.13 – Графическое представление символа «Линия» со стрелками

**Специфические** символы **линий - Пунктирная линия**

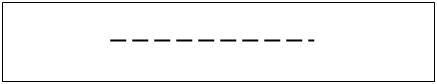


Рис 2.14 – Графическое представление символа «Пунктирная линия»

используется для обведения выделяемого участка + как часть символа комментария.

**4. Специальные символы**

**4.1. Соединитель**

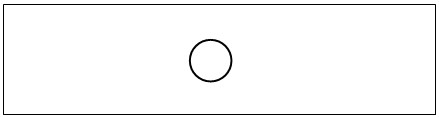


Рис 2.15 – Графическое представление символа «Соединитель»

отображает **вых**од в другую часть схемы и **вх**од из другой части схемы, для обрыва линии и продолжения её в другом месте. Соответствующие символы-соединители должны содержать 1 и то же уникальное обозначение - буквы или арабские цифры (рис 2.16).

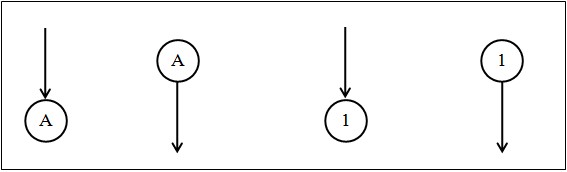


Рис 2.16 – Обозначения в символах-соединителях

**4.2. Терминатор**

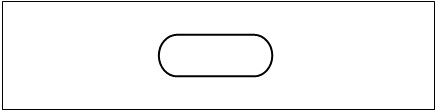


Рис 2.17 – Графическое представление символа «Терминатор»

отображает **вых**од во внешнюю среду и **вх**од из внешней среды (**начало или конец алгоритма**).

**4.2.3. Комментарий**



Рис 2.18 – Графическое представление символа «Комментарий»

- для добавления комментариев (пояснительных записей). Пунктирная линия в символе комментария связана с соответствующим символом и может обводить группу символов. Текст комментариев помещается справа от ограничивающей его сверху и снизу **квадратной скобки [ ]**.

**4.2.4. Пропуск**

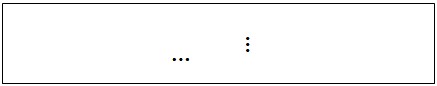


Рис 2.19 – Графическое представление символа «Пропуск»

- в виде 3х точек: пропуск символа/группы символов, если **не определены** тип или число символов. используется только в символах линии или между ними:

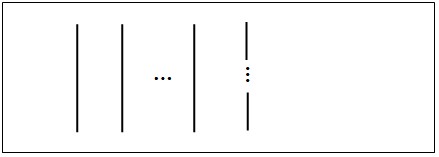
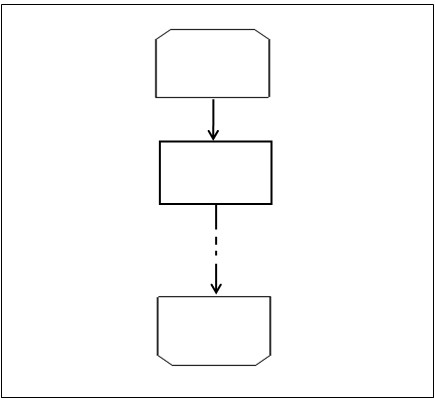


Рис 2.20 – Использование символа «Пропуск».

в схемах: **общие решения с неизвестным числом повторений:**



**А. Правила применения символов в схемах** (Стандарт)

1). должны быть расположены **равномерно**. min длинных линий. Стандарт не регламентирует конкретные **размеры:** должны позволять включать текст внутрь символа. Сохранять углы и другие параметры. **одного размера.**

2). могут быть в **вертикальной** ориентации или в зеркальном изображении. Но это не является предпочтительным.

3). Внутри символа помещать **минимальное количество текста** для понимания функции данного символа. Текст должен записываться слева направо и сверху вниз независимо от направления потока.

Например (рис 2.22), вначале вычисляется значение ***В***, а затем – значение ***С***.

4). Если объём текста не помещается внутри символа - использовать **комментарий.**

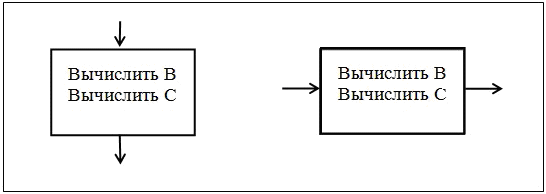


Рис 2.22 – Представление текста внутри символов

Если использование символов комментария может запутать или разрушить ход схемы, то текст помещать **на отдельном листе** и давать ссылку на символ.

5). В схемах м использовать **идентификатор символов** - определяет символ для использования в справочных целях в других элементах документации (например, в листинге программы). Идентификатор символа располагается слева над символом.

Для удобства описания схем символы нумеруют и идентификатор символа - № символа в схеме.

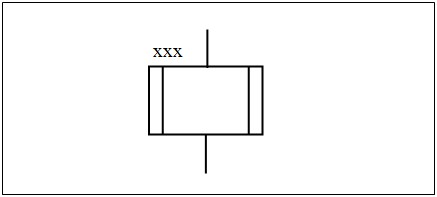


Рис 2.23 – Графическое представление идентификатора символа

6). может использоваться **описание символов** - любая другая информация (например, для улучшения понимания функции как части схемы). Описание символа располагать справа над символом:

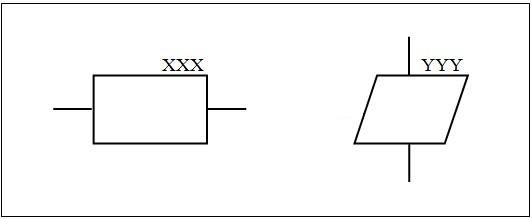


Рис 2.24 – Графическое представление описания символа

***7).* Подробное представление** – обозначается: символ с полосой для процесса или данных - указывает, что в этом комплекте документации в другом месте есть подробное описание символа.

**Символ с полосой** - любой символ из групп символов процесса или символов данных, внутри которого в верхней части горизонтальная линия, между которой помещен **идентификатор -** указывает имя схемы с подробным представлением данного символа. Первый и последний символы подробного представления д. б. **символы «Терминатор»**, в которых указывается идентификатор (как в символе с полосой):

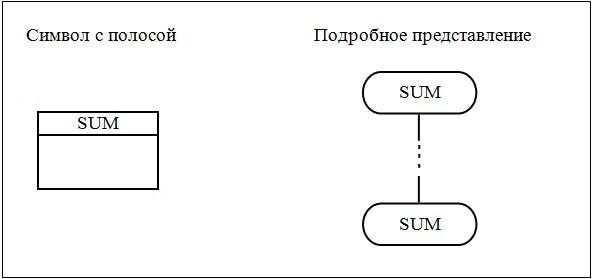


Рис 2.25 – Графическое представление символа с полосой и его

подробное представление

**Б. Правила выполнения соединений в схемах.**

1). Потоки данных и потоки управления в схемах показывают **линиями**. Направление потока слева направо и сверху вниз ***- стандарт.***

2). Д/большей ясности в схеме (например, при соединениях), на линиях используются **стрелки**. Если поток имеет направление, отличное **от стандартного**, оно должно указываться стрелками.

3). избегать пересечения линий. Они не связаны между собой, поэтому изменения направления в точках пересечения нет.

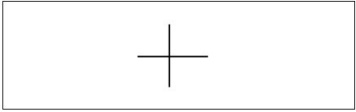


Рис 2.26 – Графическое представление пересечения линий

4). 2 и более входящие линии м объединять в 1 исходящую. места объединения линий должны быть смещены:

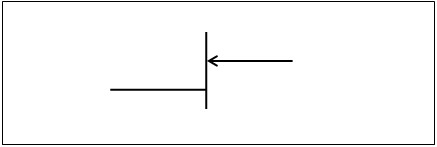


Рис 2.27 – Графическое представление объединения линий

5). Линии в схемах должны *подходить к символу* *слева или сверху*, а *исходить* *справа или снизу*. Линии д б направлены **к центру символа.**

6). Для избежания пересечений и длинных линий или схема из нескольких страниц, линии в схемах **разрывать.**

Соединитель в **начале разрыва называется *внешним соединителем***, а соединитель в **конце разрыва — *внутренним*** *соединителем*.

Для **межстраничных соединителей** совместно с символами соединителя используются символы комментария - указывают номера страниц, на которых расположен внутренний и внешний соединитель:



Рис 2.28 – Графическое представление межстраничных соединителей

**Специальные условные обозначения**

может иметь лишь символ «Решение»

1. Несколько линий от него к другим символам;
2. Одной линией от него, которая затем разветвляется в соответствующее число линий.

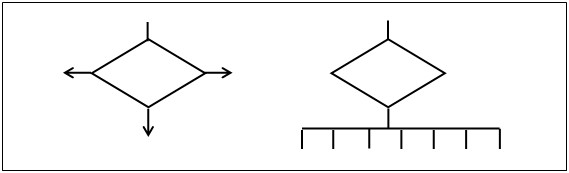


Рис 2.29 – Графическое представление нескольких выходов из символа

Каждый выход из символа д б со значениями условий, определяющими переход по данному логическому пути.

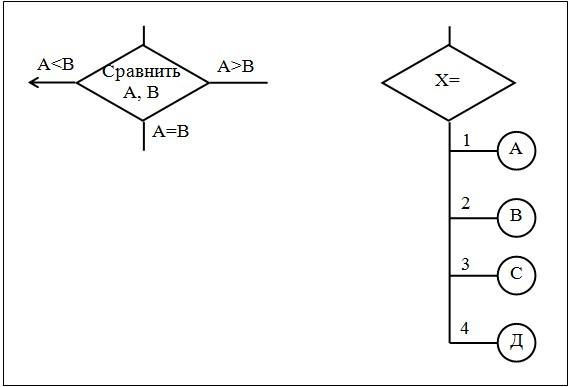


Рис 2.30 – Графическое представление нескольких выходов из символа   
с условиями переходов по конкретной ветви

2.2.3. Запись на алгоритмическом языке

Чтобы разработать алгоритм решения задачи, надо представить ее как последовательность правил.

**Понятия алгоритма и программы не четко разграничены.** программа, записанная на алгоритмическом языке – это окончательный вариант алгоритма решения задачи, ориентированный на конкретного исполнителя (компьютер или язык программирования).

**2.3. Разновидности структур алгоритмов**

*структуры алгоритмов*:

1. **линейные;**
2. **разветвляющиеся;**
3. **циклические.**

2.3.1. Линейный вычислительный процесс

***Линейный вычислительный процесс*** – процесс, в котором направление вычислений является (!)ым.

**Пример 2.2.**

Вычислить значение функции

2010-05-25_205423

Алгоритм вычисления данной функции является линейным: ход вычислительного процесса не зависит от условий.

Алгоритм м разработать с разной степенью детализации.

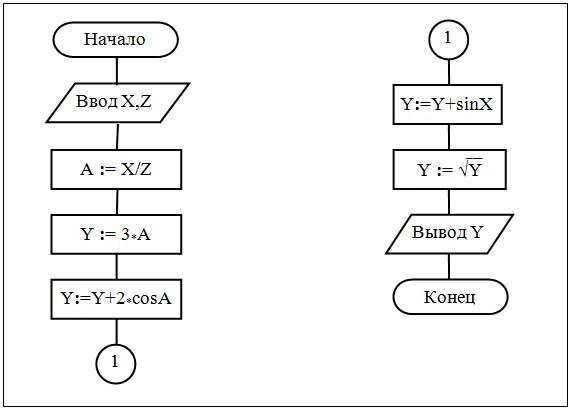


Рис 2.31 – Детализированная схема линейного алгоритма вычисления ф-ции ***Y***

Символом **\***в языках программирования и на схемах - операция умножения.

Рис 2.32 - укрупненная схема того же алгоритма.

Для повышения эффективности алгоритма: выражения, участвующие в вычислениях несколько раз, вычислялись 1 раз, а затем использовали уже вычисленные их значения (см. рис 2.31 и рис 2.32, где 1 раз вычислено значение ***X/Z***, полученное значение присвоено переменной ***A***, которая потом используется в вычислениях).

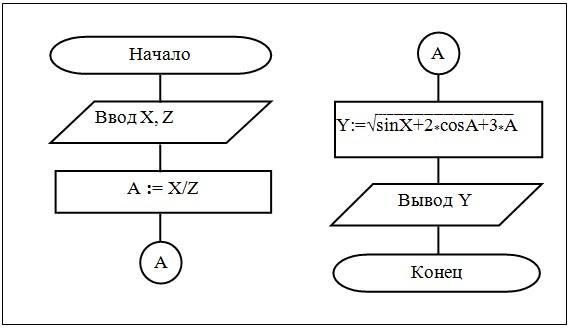
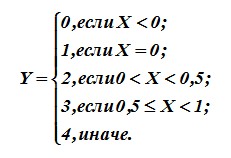


Рис 2.32 – Укрупнен схема линейного алгоритма вычисления ф-ции ***Y***

2.3.2. Разветвляющийся вычислительный процесс

– процесс, где направление вычислений определяется **условиями**.

**Пример 2.3**.

Вычислить значение функции 

используется набор вложенных символов «Решение» с двумя выходами.

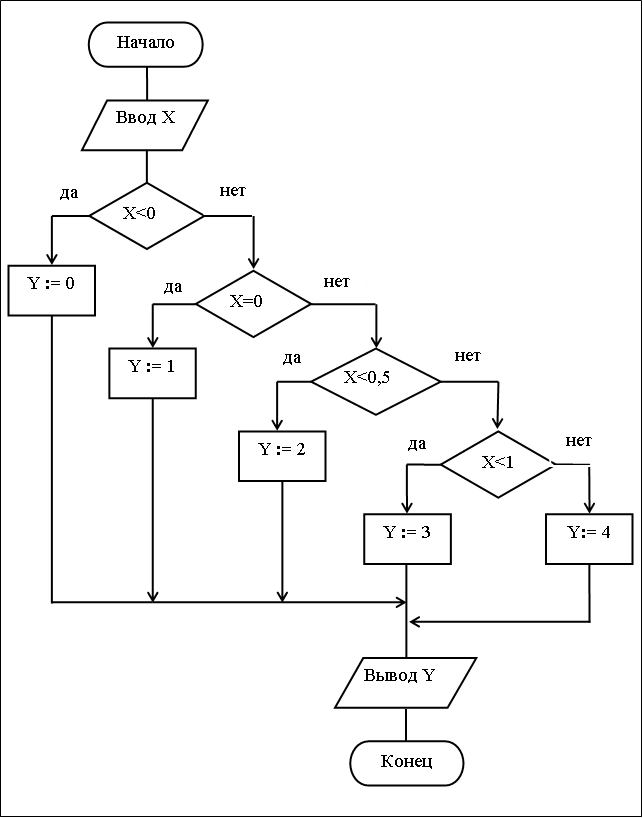


Рис 2.33 – Схема разветвляющегося алгоритма вычисления функции ***Y***

Этот же алгоритм можно представить (рис 2.34): 1 символ «Решение» со многими выходами.

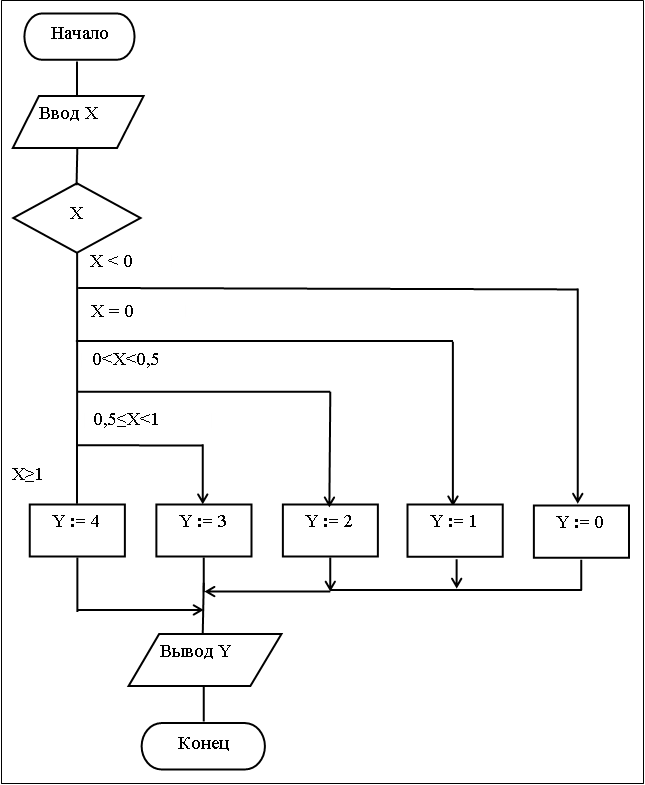


Рис 2.34 - Схема разветвляющегося алгоритма вычисления функции ***Y***   
(используется символ «Решение» с многими выходами)

2.3.3. Циклический вычислительный процесс

– процесс, где отдельные участки вычислений выполняются **многократно.**

**Цикл *-*** **многократно** повторяемый участок схемы в ходе вычислений. При повторениях используются новые значения исходных данных.

**Классификация циклов**

А). В зависимости от **взаимного расположения *циклов*** в теле программы или алгоритма циклы бывают:

1. **простые** – циклы, не содержащие внутри себя других циклов;

**сложные** – циклы, содержащие внутри себя другие циклы;

**вложенные (внутренние**) – циклы, входящие в состав других циклов (цикл в цикле);

4) **внешние** – циклы, не являющиеся составной частью других циклов, но содержащие в своем составе внутренние циклы.

Б). В зависимости от **местоположения** условия выполнения цикла:

1. **циклы с предусловием;**
2. **циклы с постусловием**.

В). В соответствии **с видом условия** выполнения:

1. **циклы с параметром;**
2. **итерационные циклы.**

**В.1) Циклы с параметром**

Для решения сколько раз нужно выполнять цикл, используется **анализ переменной** (или нескольких переменных), называемой **параметром цикла**.

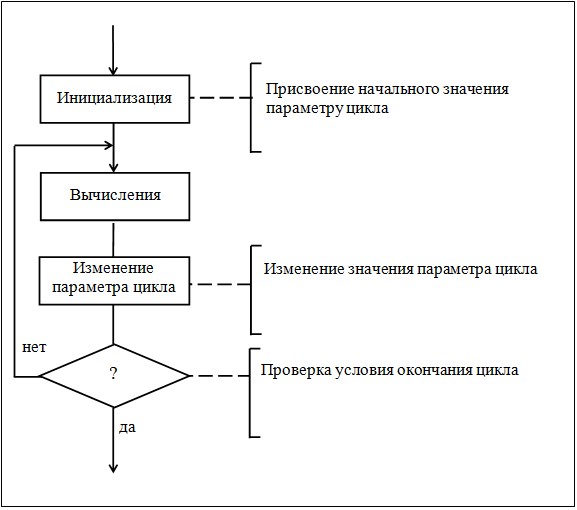


Рис 2.35 –схема алгоритма циклического процесса с параметром цикла с **постусловием «ДО». – Repeat … Until**

**цикл «До»** **(с постусл.) - натворила, а потом проверяешь** – в нем тело цикла выполняется **до** проверки условия вых из цикла (хоть 1 р).

Блоки, выполняющие вычисления и изменение значения параметра цикла, составляют **тело цикла** – **многократно** выполняемая последовательность действий**.**

*В* **цикле «Пока» (цикле с предусловием While Do)** проверка условия выполнения цикла производится **до** тела цикла:

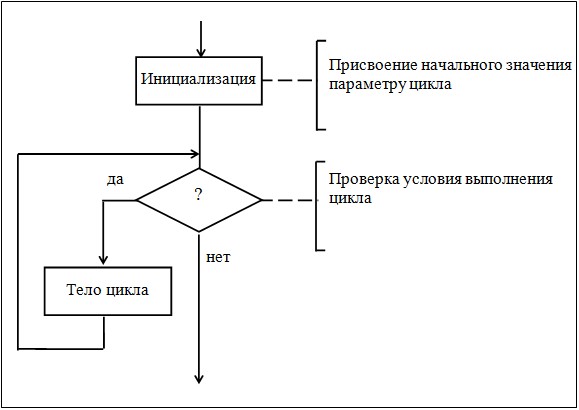


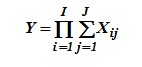
Рис 2.36 - –схема алгоритма циклического процесса с **предусловием While Do**

**Цикл со счетчиком** (**цикл с параметром цикла, FOR)** - циклический процесс, в котором число выполнений тела цикла (- **итераций**) заранее определено. Или: **циклический процесс с известным числом повторений**.

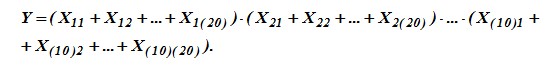
Например, при вычислении суммы 20ти элементов массива заранее известно, что число итераций = 20.

**Пример 2.4.**

Алгоритм со сложным циклом с заданным числом повторений. Вычислить значение функции:



Пусть ***I = 10, J = 20***. Тогда выражение для вычисления функции ***Y*** будет иметь вид:



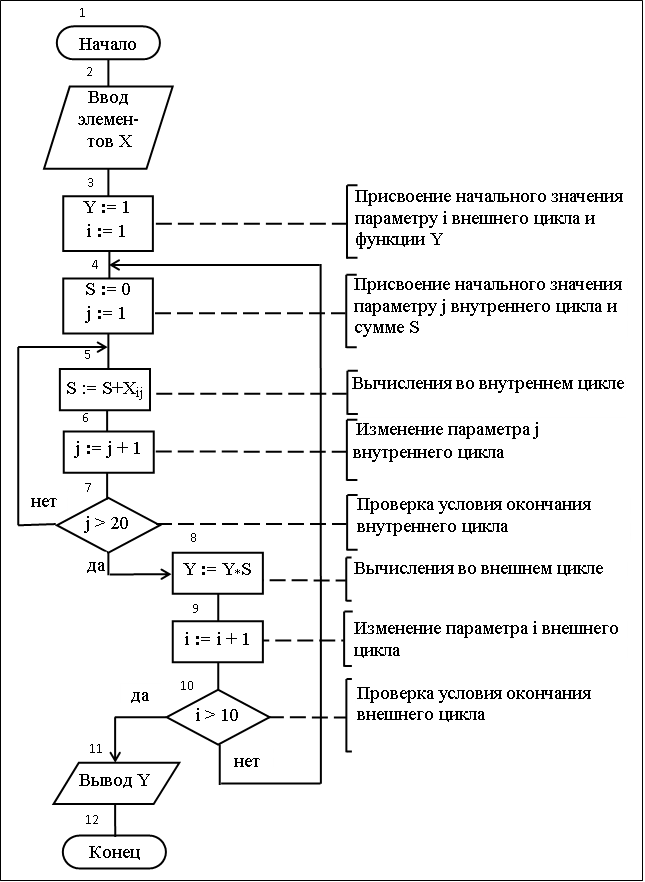
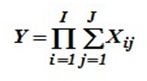


Рис 2.37 – Алгоритм вычисления функции 

Данный алгоритм содержит **внешний** цикл, в состав которого входит внутренний цикл. Параметр внешнего цикла - переменная ***i***, а внутреннего – ***j***. Блоки 5, 6 – **тело внутреннего цикла**, блоки 4 – 9 –внешнего.

Алгоритм реализован с использованием **циклов «До» (Repeat).**

**Переменные для накопления суммы*,*** *в исходном состоянии всегда устанавливаются в ноль* (в примере 2.4 это ***S = 0***). Д**ля произведения -**  *в единицу* (в примере 2.4 это ***Y = 1***).

**В.2) Итерационный цикл**

- процесс с заранее **неизвестным** количеством повторений (зависит от **результатов** вычислений)**.**

Типовой пример – вычисление суммы ряда **с погрешностью** вычислений, не превышающей заранее заданной. Вычисления продолжаются, пока, например, разность между значениями величин на соседних шагах цикла, не станет <= заданной величины – точности.

Для итерационных процессов характерно: значения на текущем шаге итерации - исходные данные для следующего шага итерации, что позволяет повысить эффективность алгоритма.

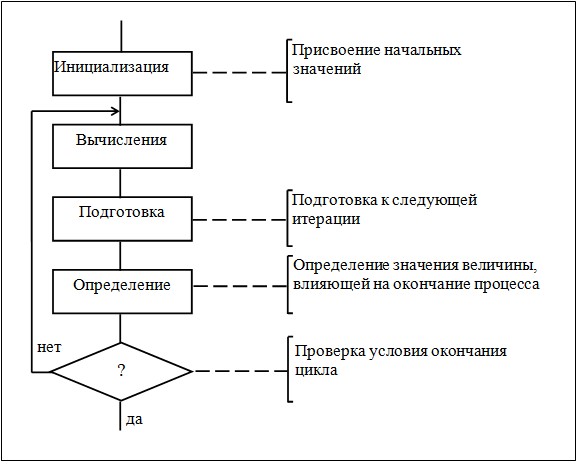


Рис. 2.38 –схема итерационного циклического процесса (с постусл. «До»)

**Пример 2.5.**

Алгоритм со сложным циклом с неизвестным числом повторений. Вычислить значение функции

2 через разложение функции в бесконечный ряд

2

Вычисления прекратить, когда разность между модулями двух соседних слагаемых станет меньше величины ***Eps = 0,0001***.

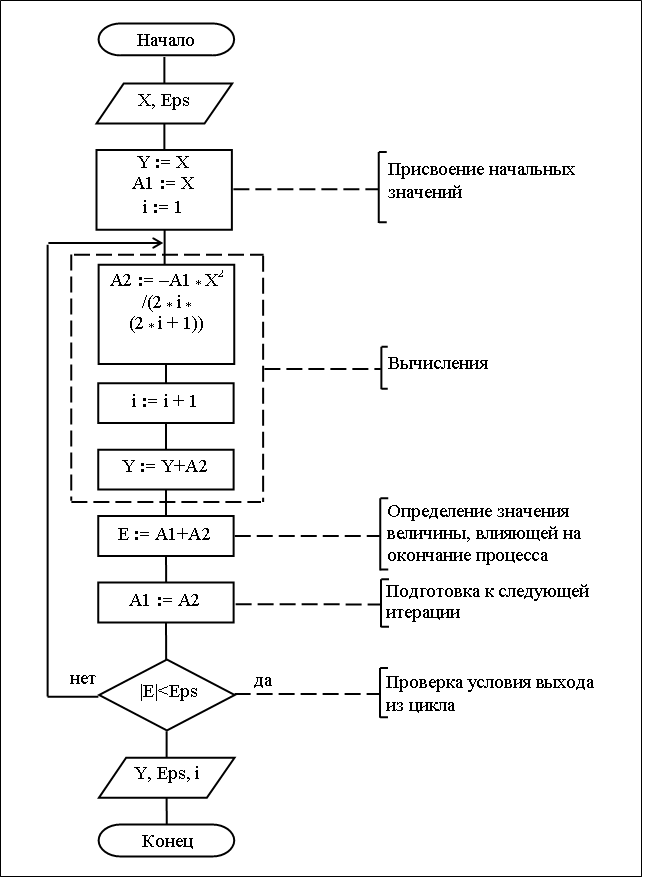


Рис 2.39 – Алгоритм вычисления функции ***Y = SinX – ВЕРНУТЬСЯ!!!***

В переменной ***Y*** накапливается значение суммы ряда.

В переменной ***А1*** хранится слагаемое, сформированное **на предыдущей** итерации – как **исходная величина** д/вычисления слагаемого на данной итерации.

В ***А2*** формируется слагаемое на данной итерации.

Переменная ***i*** – служебная переменная для формирования **знаменателя** слагаемых + определения выполненного числа итераций.

В ***Е -*** разность между соседними слагаемыми (с учетом их разнозначности). После окончания текущей итерации надо запомнить значение слагаемого в ***А2***, иначе оно потеряется на следующей итерации. Поэтому, значение ***А2*** присваиваем переменной ***А1***.

Многие алгоритмы имеют смешанный характер: линейные участки, разветвления, циклы с известным числом повторений и итерационные. ==> составление алгоритмов сразу в законченной форме затруднено.

==> для составления сложных алгоритмов использовать **нисходящее проектирование***программ* (1. метод пошаговой детализации, 2.метод последовательных уточнений).

**Суть** *нисходящего проектирования*: вначале продумывается общая структура алгоритма, без детальной проработки. Далее прорабатываются отдельные блоки (каждый блок - ∑ть блоков более низкого уровня). Матрешка.

Процесс прекращается при достижении нужной детализации блоков – которая позволяет выполнять непосредственное кодирование блоков. Таким образом, на каждом шаге разработки уточняется реализация фрагмента алгоритма, т.е. решается более простая задача.

**Раздел 3. Структурное программирование**

В 1970-х годах с появлением ЭВМ третьего поколения. Одним из первых инициаторов структурного программирования был профессор **Э.Дейкстра**. В 1965 он высказал предположение, что оператор **GoTo** **(оператор безусловного перехода)** м исключить из языков прогр-ия: «квалификация программиста обратно пропорциональна числу операторов GoTo в его программах».

*« + »* структурного программирования по сравнению с неструктурным:

1. Уменьшение трудностей **тестирования** программ.

2. Более высокая **производительность** программистов.

3. Ясность и **читаемость программ,** что упрощает их сопровождение.

4.  **Эффективность** программ.

**3.1. Теория структурного программирования**

К*онцепции* структурного программирования: 1. отказ от использования оператора (***GoTo***), 2. замена его рядом других структурированных операторов и 3. использование идей **нисходящего проектирования** программ.

Основное *назначение* *нисходящего проектирования* – служить средством **разбиения большой задачи на меньшие** подзадачи так, чтобы каждую подзадачу рассматривать **независимо.**

***требование*** структурного программирования **-** каждый модуль алгоритма (программы) проектировался *с* ***(!)*** *входом и* ***(!)*** *выходом.* Программа - **множество *вложенных* модулей**, каждый из которых имеет 1 вход и 1 выход.

**База** для реализации структурированных программ - ***принцип Бома и Джакопини***: логическая структура программы может быть выражена комбинациями 3х ***базовых структур***:

**1. Функциональный блок;**

**2. Конструкция принятия двоичного (дихотомического) решения.**

**3. Конструкции обобщенного цикла.**

***1.* Функциональный блок** – это отдельный вычислительный **оператор** (или другая последовательность вычислений) с (!) входом и (!) выходом. Изображается с помощью символа **«Процесс»:**

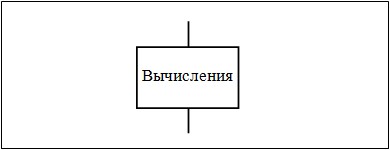


Рис 3.1 – Графическое представление функционального блока

**2. Конструкция принятия двоичного (дихотомического) решения** обычно называется элементом ***If-Then-Else*** **(если-то-иначе),** разветвлением или ветвлением (да/нет) – структура выбора между 2я альтернативными путями вычислительного процесса в зависимости от выполнения некоторого условия. Изображается с помощью символов «Решение» и «Процесс»:

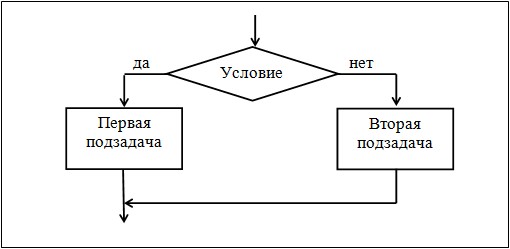


Рис 3.2 – Графическое представление конструкции принятия 2чного решения

**3. Конструкция обобщенного цикла***:* базовая конструкция структурного программирования – цикл с **пред**условием (**«Пока»** ***Do-While***). Изображается с помощью символов «Решение» и «Процесс»

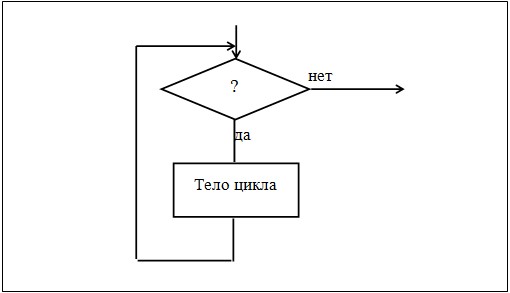


Рис 3.3 – Графическое представление конструкции обобщенного цикла

Рисунок 3.2 и 3.3 показывают, что обе логические конструкции (принятия двоичного решения и обобщенного цикла) имеют 1 вход и 1 выход. Поэтому они могут рассматриваться как **функциональные блоки.**

C учётом этого вводится **преобразование логических блоков в функциональный блок.**

последовательность функциональных элементов, называемая **конструкцией следования** (рис 3.4), также может быть приведена к 1 функциональному блоку.

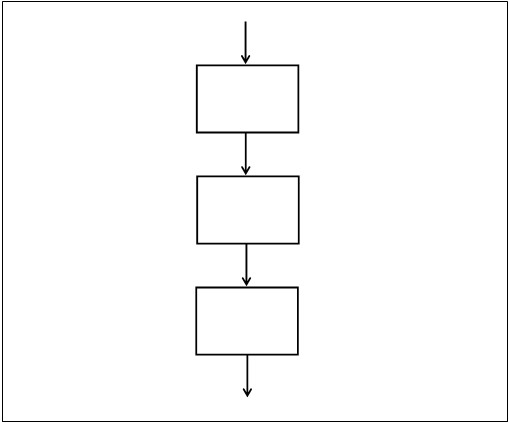


Рис 3.4 – Графическое представление конструкции следования

Данные преобразования называются ***преобразованиями Бома-Джакопини***. Их основу составляет ***принцип “чёрного ящика”*** (что-то с одним входом и одним выходом).

Таким образом, всякая программа (или алгоритм), состоящая из 1.функциональных блоков, 2. конструкций цикла и 3.элементов If-Then-Else, поддаётся последовательному преобразованию к (!) функциональному блоку.

Эту последовательность преобразований м использовать как 1.средство понимания программы, 2.***доказательство ее правильности и 3.структурированности.***

Обратная последовательность преобразований может быть использована в процессе проектирования алгоритма (программы) по *методу* *нисходящего проектирования* – алгоритм (программа) разрабатывается, исходя из единственного функционального блока, который раскрывается в сложную структуру основных элементов.

**3.2. Реализация структурного проектирования в современных языках программирования**

- базируется на следующих *правилах.*

1. Все операции в проге д б: или выражения (исполняются в линейном порядке), или 1 из ***управляющих конструкций***:

***2. вызовы подпрограмм*** – любое допустимое на языке программирования обращение к замкнутой подпрограмме с1 входом и1 выходом;

1. вложенные на произвольную глубину *операторы If-Then-Else*;
2. *циклические операторы* (цикл с предусловием = цикл «Пока»).

Этих средств достаточно для составления структурированных программ.

Но иногда допускаются их ***расширения***:

***1). дополнительные*** *конструкции организации цикла*:

*1.1) цикл* ***с параметром*** - как вариант цикла с **пред**условием;

*1.2) цикл с* ***пост****условием Repeat* - циклом «До»: тело цикла выполняется **до** проверки условия выхода из цикла;

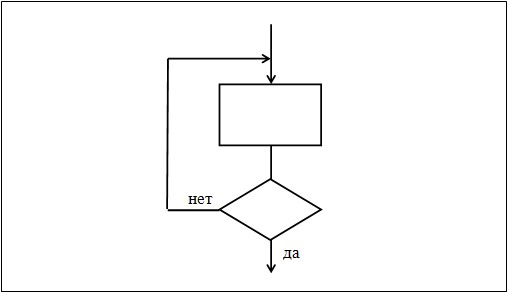


Рис 3.5 – Графическое представление цикла **с пост**условием - цикл «До»:

***2). подпрограммы с несколькими*** *входами и несколькими выходами* (например, 1 выход нормальный, 2й – по ошибке);

*3).. применение оператора* ***GoTo*** *с жёсткими ограничениями* (например, передача управления не далее, чем на 10 операторов, или только вперёд по программе);

*4). использование оператора* ***Case*** как расширения If-Then-Else:

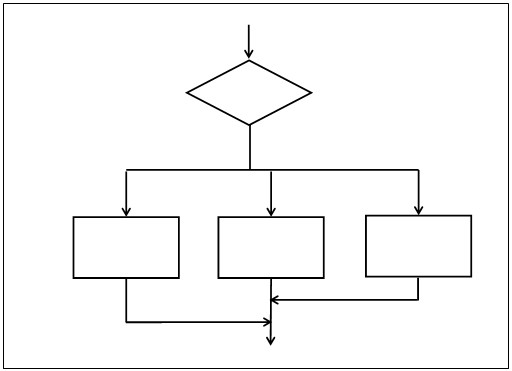


Рис 3.6 – Графическое представление оператора **Case** расческа

**3.3. Преобразование неструктурированных программ в структурированные**

преобразование возможно при использовании 3х *методов*:

* **дублирование кодов программы; 3.3.1**
* **введение переменной состояния; 3.3.2**
* **метод булевых признаков.3.3.3**

3.3.1. Метод дублирования кодов программы

Рассмотрим его применение на примере неструктурированной программы типа «решетка».

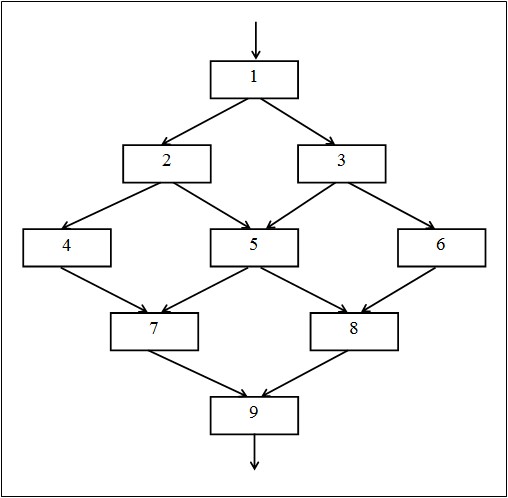


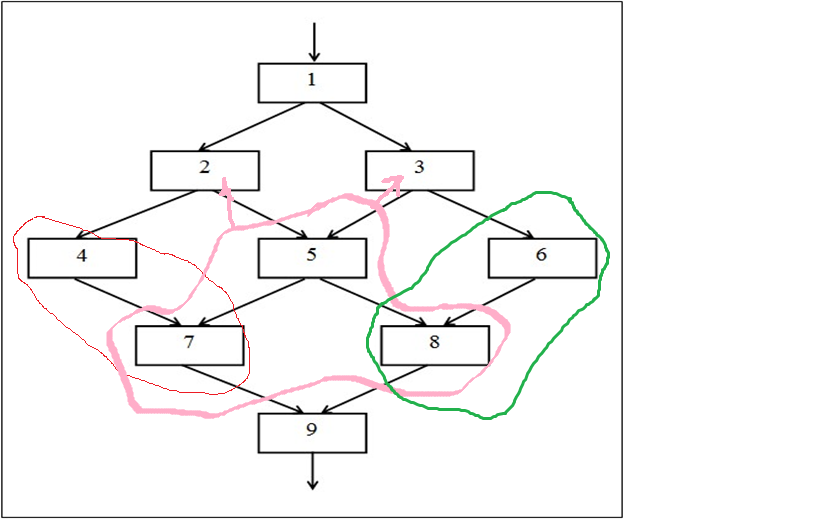
Рис 3.7 – Алгоритм неструктурированной программы типа «решетка»

Стрелки соединения блоков - операторы перехода **(Goto?)**.

Программа с данным алгоритмом не структурирована: не удовлетворяет условию «1 вход – 1 выход».

**Суть *метода дублирования кодов*:** дублируются те модули исходного алгоритма / проги, в которые можно войти из нескольких мест (кроме последнего блока).

В соответствии с этим в исходной схеме надо дублировать модули 5 (в него можно войти из модулей 2 и 3), 7 (в него - из 4 и 5) и 8 (- из 5 и 6). Это приводит исходную схему к виду:



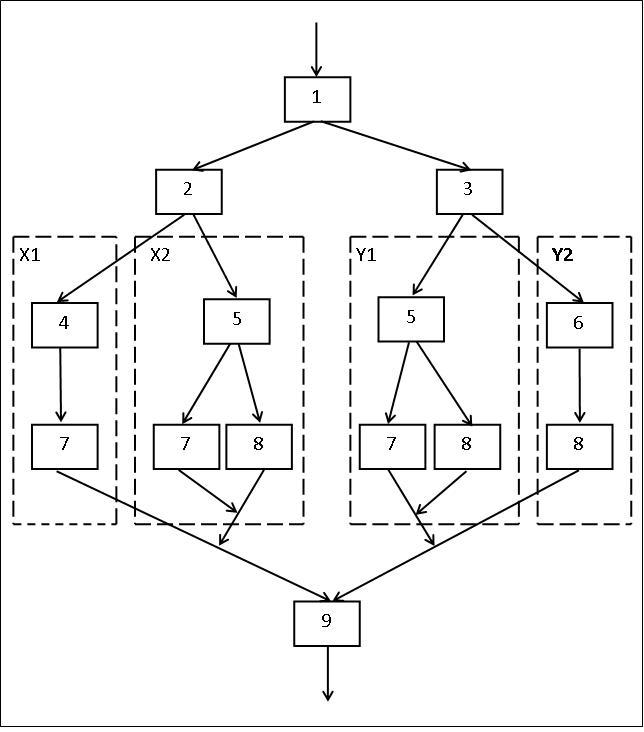


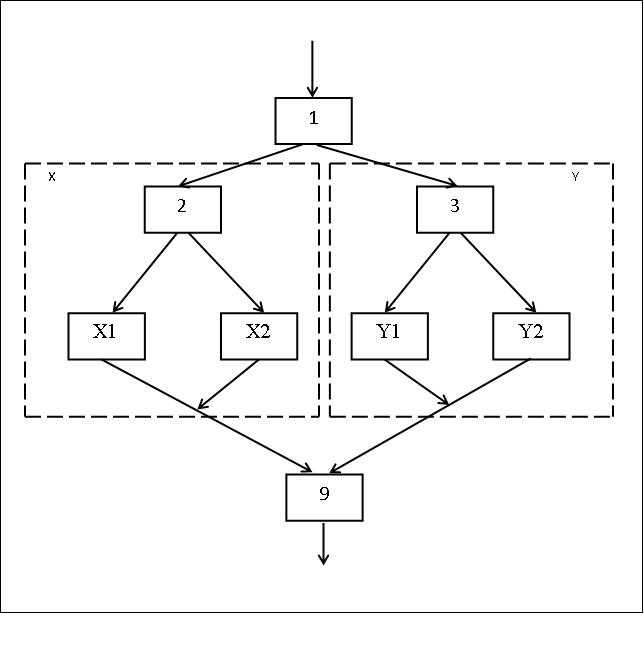
Рис 3.8 – Преобразованная по методу дублирования кодов

Полученная схема алгоритма является структурированной.

Чтобы доказать это, используем преобразования Бома-Джакопини, т.е. последовательно преобразовать схему к 1у функциональному блоку с 1 вх и1 вых. Для этого необходимо выполнить несколько шагов:

***Шаг 1: преобразования***

На рис 3.8 модули 4 и 7 **- конструкции следования** – преобразов Бо-Джак (принцип **черного** ящика: 1вх/1вых). Согласно преобразованиям Бома-Джак-и их м свести к 1 функциональному блоку ***Х1***. Модули 5, 7, 8 – конср-ция **If-Then-Else** с 1 вх и1 вых. Их тоже м преобразовать к 1 функциональному блоку (***Х2*** и ***Y1***). Аналогично для конструкции следования из модулей 6, 8 – она сводится к блоку ***Y2***. В результате предыдущая схема принимает вид:

**Рис. 3.9 – Схема алгоритма после 1го шага преобразования Бома-Дж-и

***Шаг 2: преобразования***

В полученной схеме группы модулей (2, ***X1, X2)*** и (3, ***Y1, Y2) -*** конструкции **If-Then-Else** с 1 вх и 1 вых. Согласно преобразованиям Бома-Джак-и их можно представить в виде функц-ных блоков ( ***Х*** и ***Y*** ):

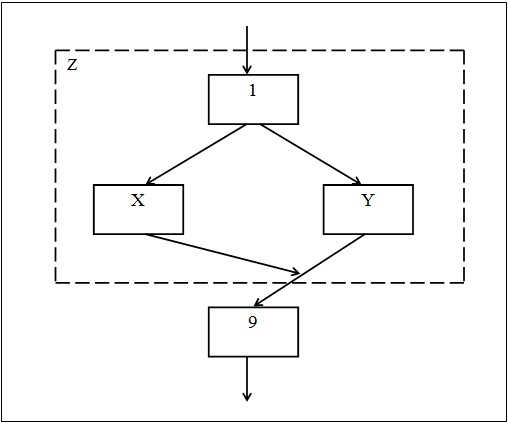
**

Рис. 3.10 – Схема алгоритма после 2го шага преобразов Бома-Дж-ни

***Шаг 3: преобразования***

В полученной схеме группа модулей 1, ***X, Y*** - конструкция **If-Then-Else** c 1 вх и1 вых. Согласно преобразованиям Бома-Джак-и ее можно представить в виде функционального блока ***Z*** (рис 3.11).

***Шаг 4: преобразования***

Согласно преобразованиям Бома-Джак-и конструкцию следования можно представить в виде функционального блока ***R***:

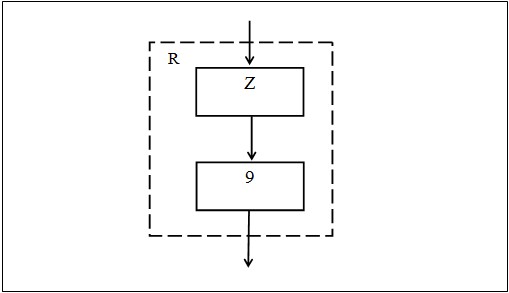


Рис. 3.11 − Схема алгоритма после 3го шага преобразов Бома-Дж-ни

Таким образом, с помощью 4х шагов преобразования доказано, что полученная в результате применения **метода дублирования кодов** схема алгоритма (см. рисунок 3.8) является структурированной.

*«+»* метода дублирования кодов: 1. удобно использовать при нисходящем проектировании программ. Исходную задачу укрупненно можно представить в виде 1 функционального блока, а затем разукрупнять через промежуточные схемы алгоритма к результирующей структурированной схеме. В этом алгоритме - 4 шага преобразования в обратном порядке.

*«-»*:1) неприменимость к **программам с циклами**;2) дополнительные **затраты памяти** для хранения дублируемых модулей.

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

3.3.2 Метод введения переменной состояния

Данный метод был впервые предложен **Ашкрофтом и Манной.**

Рассмотрим его на примере неструктур-ной проги, алгоритм - на рис 3.12.

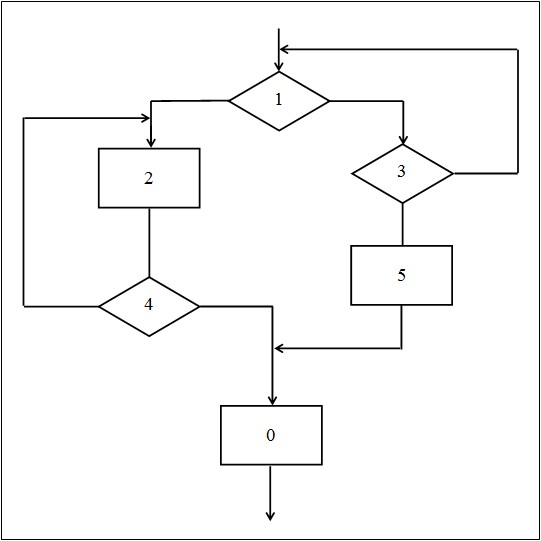


Рис 3.12 – Исходная схема неструктурированного алгоритма

Данная схема не является структурированной, так как из цикла, состоящего из блоков 1 и 3 есть 2 выхода - нарушено условие «1 вх – 1 вых» структурированных схем.

Процесс преобразования программы в структур-ную состоит из **5 шагов**:

1. Каждому блоку неструктурированной схемы приписывают №. первому блоку присваивается № 1, последнему – № 0.
2. В программу вводится д/п-ая переменная целого типа (например, ***J***) - **переменная состояния**.
3. Функциональные блоки исходной схемы заменяют блоками, преобразующие переменную ***J*** (помимо основных функций): переменной ***J :=*** значение, = № блока-приёмника в исходной схеме.
4. Аналогично преобразуют логические блоки: если в логическом блоке условие истинно, то это соответствует 1у значению ***J***, если ложно – другому.
5. Исходная схема приводится к виду Ашкрофтом-Манной (рис 3.13).

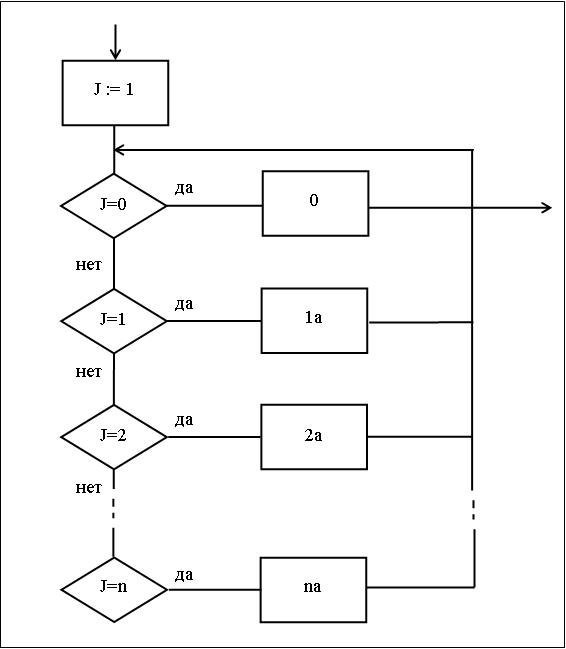


Рис 3.13 – **Обобщенный** вид схемы алгоритма Ашкрофта-Манныю. Это просто схема типичная для Манки.

Здесь блоки **1а– nа** - аналоги соответствующих блоков исходной схемы и, помимо этого, присваивают значение переменной ***J***.

В результате преобразований Ашкрофта-Манны исходная неструктурир-ная схема (см. рис 3.12) принимает структурир-ный вид ( рис 3.14):

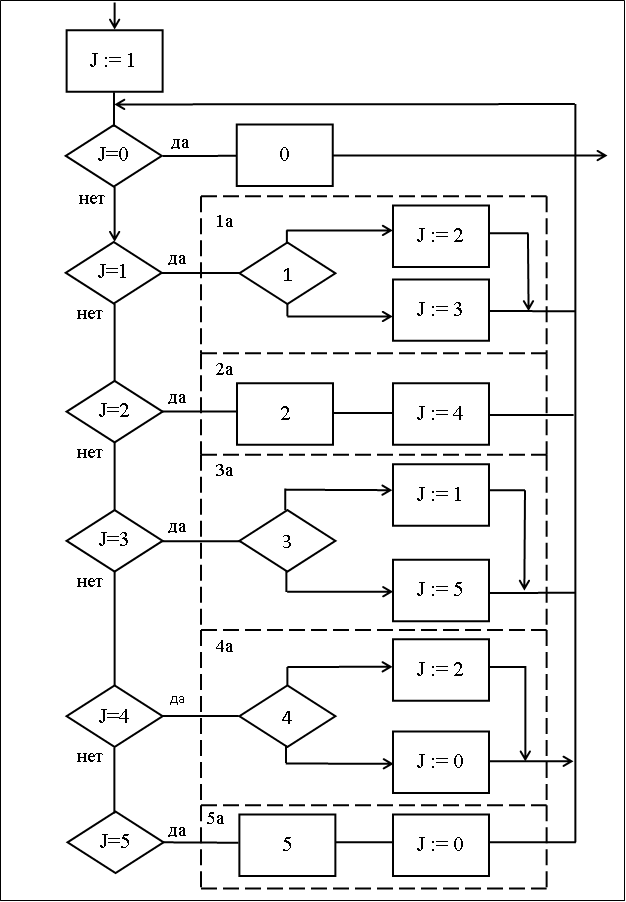
**

Рис 3.14 – Структурированная форма исходной схемы

При выполнении алгоритма по методу Ашкрофта-Манны, переменная состояния ***J*** устанавливается в начальное значение = № первого блока непреобразованной схемы (= 1). Затем последовательный **опрос** переменной ***J***, начиная с 0 и заканчивая максимальным № блока исходной схемы (здесь = 5). Выполняется блок исходной схемы, № которого = текущему значению ***J***. + в ***J*** заносится значение = № того блока исходной схемы, который должен выполняться за текущим блоком. Когда значение ***J*** станет =0, выполняется последний блок непреобразованной схемы (блок с №0) и выход из алгоритма.

Полученная по методу Ашкрофта-Манны схема алгоритма - структурированная. Для доказательства достаточно последовательно её преобразовать к 1 функциональному блоку.

***Шаг 1 преобразования.***

Конструкции **1а, 3а** и **4а** - конструкции If-Then-Else с 1 вх и 1 вых, конструкции **2а, 5а** - конструкции следования (см. рис. 3.14). Следовательно, они м б преобразованы к функциональным блокам:

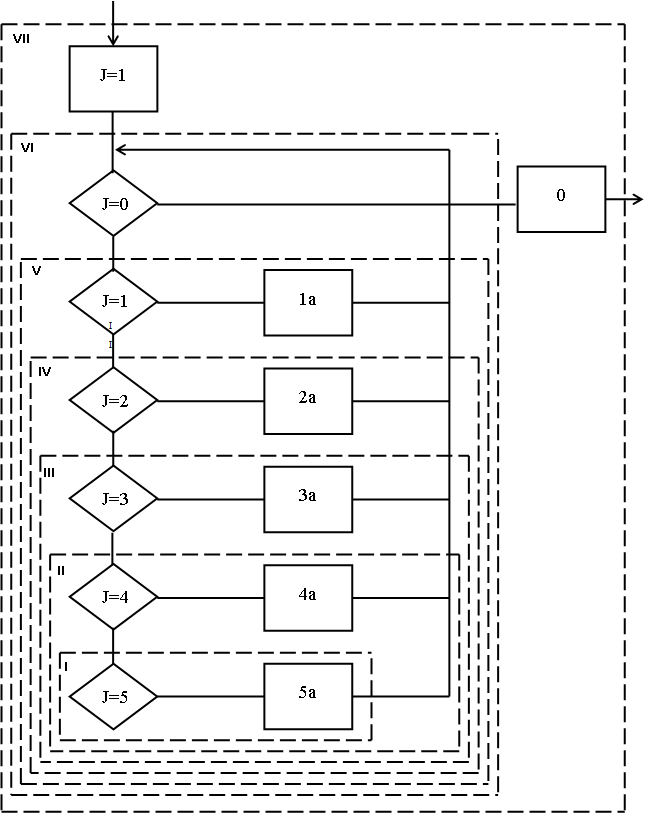


Рис 3.15 – Шаги преобразований Бо-Джакопини для структурир схемы

шаги преобразований надо проводить **снизу вверх** (см. рис 3.15).

***Шаг 2: преобразования.***

Символ “Решение” с проверкой условия ***J = 5*** и блок **5а** = конструкция **If-Then-else** (с 1 ветвью) с 1 вх и 1 вых. => данный символ «решение» и блок **5а** заменим функциональным блоком **I.**

2я ветвь данного символа «Решение» используем для повышения надежности программы и на данном рисунке не показана (данная ветвь обеспечивает возможность контроля непопадания значений ***J*** в диапазон ***0 – n***, где ***n*** – максимальный № блока в исходной схеме).

По сути, без проверки условия ***J = 5*** в алгоритме можно обойтись, перейдя на выполнение блока **5а** по ветви «нет» проверки условия ***J = 4***.

***Шаг 3: преобразования.***

Символ “Решение” с проверкой условия ***J = 4*** и блоки **I** и **=** конструкция If-Then-Еlse с 1 вх и 1 вых. => заменяются функц-ым блоком **II** (см. рис 3.15).

***Шаг 4 преобразования.***

Символ “Решение” с проверкой условия ***J = 3*** и блоки **II** и **3а** = If-Then-Else с 1 вх и 1 вых. => заменяются функциональным блоком **III**

***Шаг 5 преобразования****.*

Символ “решение” с проверкой условия ***J = 2*** и блоки **III** и **2а** = If-Then-Else с 1 вх и 1 вых. => заменяются функциональным блоком **IV**

***Шаг 6 преобразования.***

Символ “решение” с проверкой условия ***J = 1*** и блоки **IV** и **1а** = If-Then-Else с 1 вх и 1 вых. => заменяются функциональным блоком **V**

***Шаг 7 преобразования.***

Символ “решение” с проверкой условия ***J = 0*** и блок **V** - конструкция обобщенного цикла с 1 вх и 1 вых. => заменяются функцион-ым блоком **VI**

***Шаг 8 преобразования.***

Функциональный блок начальной установки ***J = 1***, блок **VI** и блок **0** – конструкция следования с 1 вх и 1 вых. => заменяются функц-ым блоком **VII**

Т. о., за 8 шагов преобразований Бома-Джакопини исходная схема, построенная по методу Ашкрофта-Манны, преобразована в один функциональный блок с 1 вх и 1 вых. Т.е. она - структурированная.

***«+»*** метода введения переменной состояния:

1). преобразование программы - наглядный и чёткий;

2). любому блоку исходной схемы соответствует определённое состояние программы, что помогает выполнять тестирование и отладку программы;

3). применим к программам любой структуры (разветвляющимся и **циклическим)**;

4). возможно автоматическое применение данного метода.

***«-»*** метода:

1. структурированная форма схемы алгоритма сильно отличается от исходной, что затрудняет ее понимание;

2. дополнительные затраты времени на анализ и установку значений переменной состояния;

3. громоздкость результирующей схемы.

3.3.3. Метод булевого признака (boolean)

*Суть:* В программу с циклами вводится **некоторый признак.** Его начальное значение задаётся **до цикла**. Цикл выполняется, пока признак сохраняет своё исходное значение. Значение признака изменяется при наличии условий внутри цикла.

Рассмотрим этот метод на неструктурированной схеме:

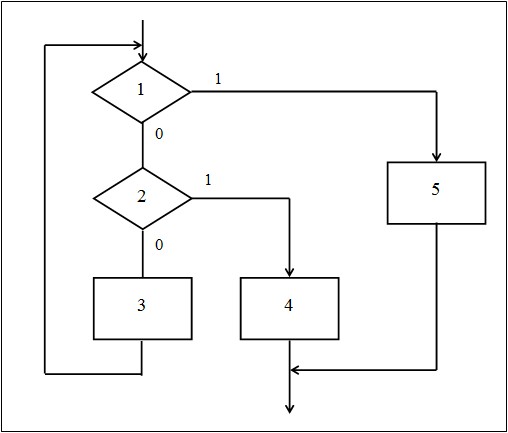


Рис 3.16 – Исходная неструктурированная схема

Схема алгоритма (см. рис 3.16) не является структурированной: входящий в нее цикл (блоки 1, 2, 3) содержит 1 вх и 2 вых. В блоках 1 и 2 записаны условия. Значения 1 и 0 = «да» и «нет».

Согласно методу в исходную схему **вводят признак** (***J*)**. Схема получает структурированный вид и реализуется конструкциями **обобщенного цикла** и принятия двоичного решения (рис 3.17).

доказательство что данная схема – структурир-ная, подробно рассмотрен в пп. 3.3.1–3.3.2. Рис 3.17 - шаги последовательности преобразований Бома-Джакопини:

1. блок 4, функциональный блок ***J := 1*** → блок **I**;
2. блоки 2, 3, **I** → блок **II**;
3. блок 5, функциональный блок ***J := 1*** → блок **III**;
4. блоки 1, **II, III** → блок **IV**;
5. условный блок (решение) ***J = 1***, блок **IV** → блок **V**;
6. блоки ***J := 0***, **V** → блок **VI**.

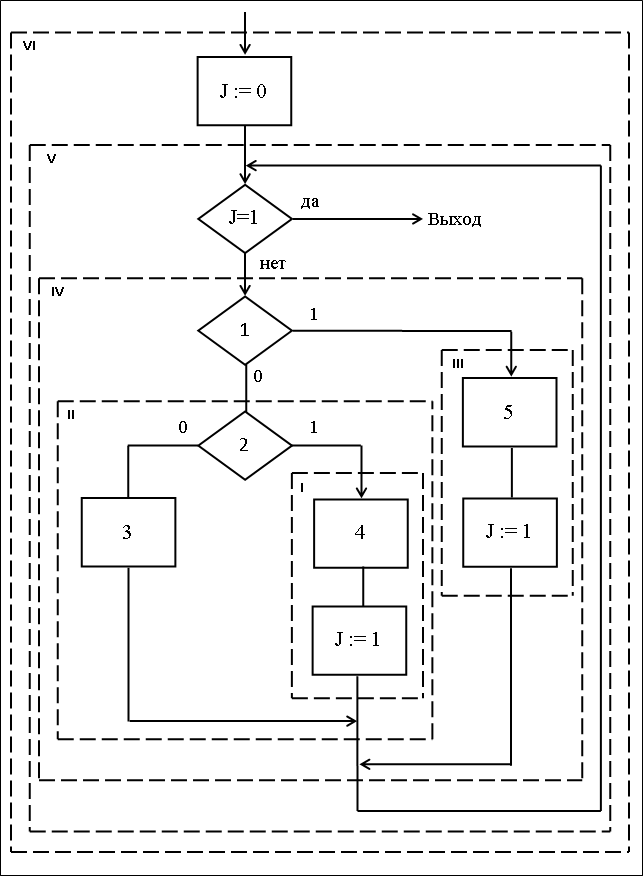


Рис 3.17 − Структурированная форма исходной схемы, преобразованная по методу булевого признака

Каждый из шагов преобразований показан пунктирным функциональным блоком.

*«+»* *метода булевого признака*:

1. компактность, экономичность;

2. топология исходной схемы изменяется незначительно.

«-»: метод предназначен **только** для **использования в циклах.**

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

Например, эту исходную схему можно представить на рис3.18:

условие «(1 и 2) = 0» означает одновременное равенство 0 условий, записанных в блоках 1 и 2 исходной неструктур-ой схемы (см. рис 3.16). => тело цикла 3 будет выполнено, если оба условия 1 и 2 не выполняются.

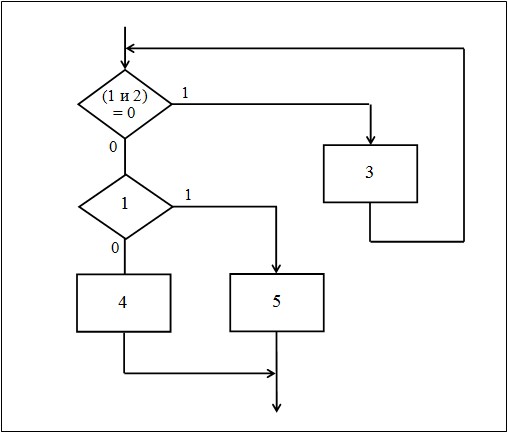
**

Рис 3.18 − Структурированная форма исходной схемы, преобразованная без использования дополнительного признака

Результирующая схема (см. рисунок 3.18) содержит обобщенный цикл с 1вх и 1 вых + конструкцию **If-Then-Else**, т е структурированная.

преобразование схемы к структурированному виду **без дополнительного** признака возможно только при **небольшом количестве условий**.

**3.4. Способы графического представления структурированных схем алгоритмов**

графическое представление алгоритмов в соответствии с *ГОСТ 19.701-90* (*ISO 5807-85*, см. [п. 2.2.2](#_2.2.2._Графическое_описание)) скрывают структуру структурированной программы. ==> предложены **специальные методы графических обозначений** – метод Дамке и диаграммы Н**а**сси-Шнейдермана.

3.4.1. Метод Дамке

предложил **специальные обозначения.**

1. **основные конструкции Стр Прог** изображаются:

***1). Функциональный блок***по-прежнему обозначается прямоугольником:

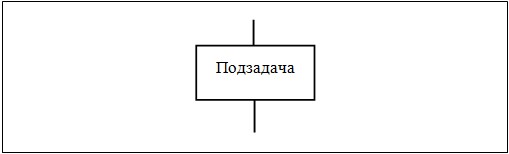


Рис 3.19 – Представление функционального блока по методу Дамке

***2). Конструкция If-Then-Else*** изображается:

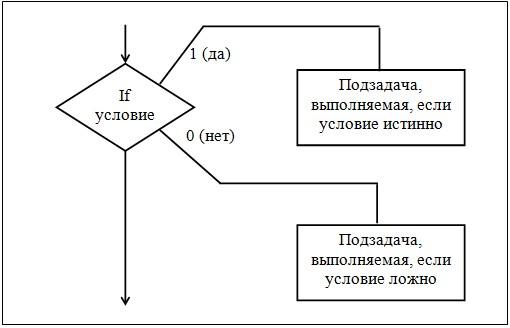


Рис 3.20 – Представление конструкции If-Then-Else по методу Дамке

Элементы с действиями - **справа** от символа «Решение». Вход и выход из конструкции - сверху и снизу символа «Решение».

***3). Конструкция While-Do (цикл* с пред*условием “Пока”)*** - рис 3.21.

Тело цикла выполняется, пока условие истинно. Условие проверяется первым: 6-угольник **над** выполняемым телом.

Вх и вых-ы из всех конструкций метода Дамке - в левой части (сверху и снизу) графического представления конструкций. Расширения конструкций в правой части представления выходов не имеют.

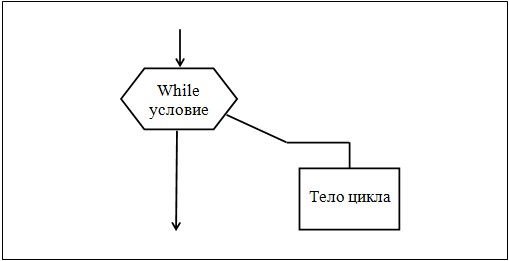


Рис 3.21 – Представление конструкции Do-While по методу Дамке

+ в Стр Прог-ии есть **д/п конструкции** в методе Дамке:

4). ***Конструкция Repeat-Until (цикл с постуслв “*До*”)*** - рис 3.22.

Если **условие истинно - выход** из цикла. Тело цикла выполняется **до** проверки условия. положение 6-угольника - **под** телом.

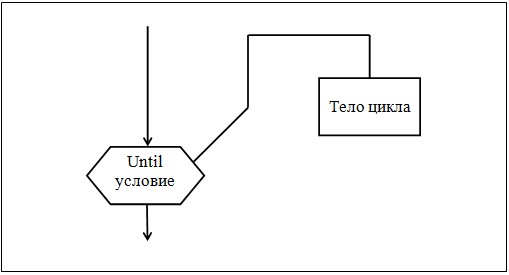


Рис 3.22 – Представление конструкции **Repeat-Until** по методу Дамке

***5). Конструкция цикла с параметром (FOR DO)*** *-* рис 3.23: (как While)

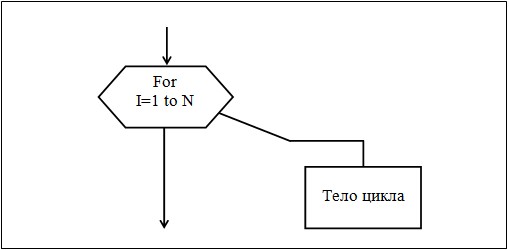


Рис 3.23 – Представление конструкции цикла с параметром по м-ду Дамке

***6). Конструкция Case*** - рис 3.24.

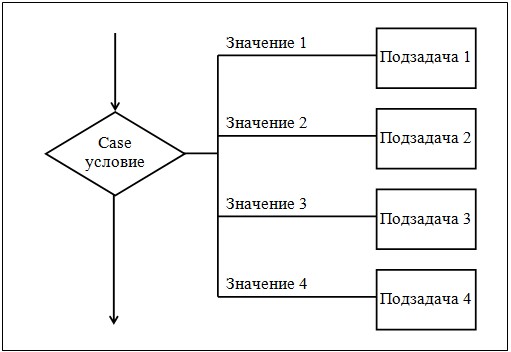


Рис 3.24 − Представление конструкции Case по методу Дамке

**Основной принцип по методу Дамке** (для структ-ых схем) - **принцип декомпозиции**: любой элемент (представляющий собой з**адачу)** можно разделить на несколько элементов (**подзадачи)**.

Элементы в самой **левой части - укрупнённая структура алгоритма и расширяются вправо** по мере разделения каждого элемента на подзадачи.

Чтобы исследовать любую подзадачу, надо анализировать элементы и управляющие структуры справа.

**Пример 3.1.**

Дан массив ***А***, состоящий из ***N*** элементов. Найти наибольший из элементов массива (***Amax***) и его № (***Imax***) по методу Дамке.

Инициализация - присвоение????

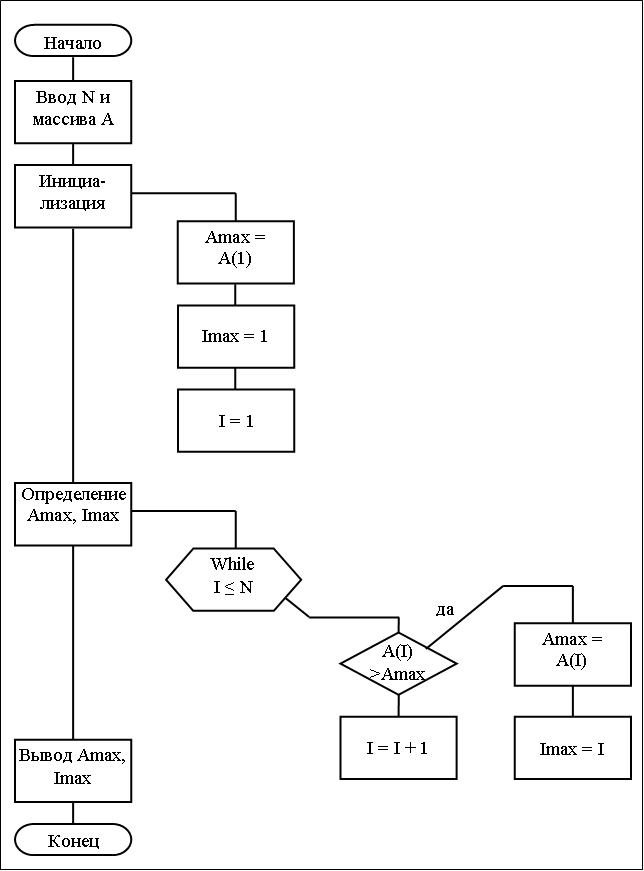


Рис 3.25 – Схема алгоритма поиска максимального элемента массива и его № по методу Дамке

*«+» метода Дамке*:

1. нагляднее, чем классическая, особенно для больших программ;
2. удобно при разработке алгоритма по методу нисходящего проектирования;
3. удобен при коллективной разработке программ (независимо разрабатывать отдельные подзадачи).

3.4.2 Схемы Насси-Шнейдермана

– **структурограммы.** Иллюстрируют структуру **передач управления внутри модуля с помощью вложенных** **блоков** друг в друга. Д/изображения **структурированных схем:** **уменьшить громоздкость** схем за счёт отсутствия линий перехода.

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

Изображение основных элементов в схемах Насси-Шне: Каждый блок имеет форму **прямоугольника** и может быть вписан в любой внутренний прямоугольник другого блока. Информация в блоках записывается как и в структурированных схемах (на естественном языке или языке математических формул).

Изображения конструкций структурированных алгоритмов в схемах Насси-Шнейдермана:

***1). Функциональный блок (блок обработки)***

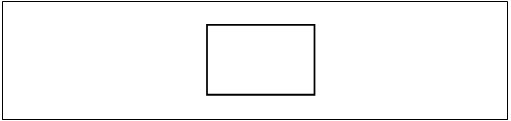
**

Рис 3.26 –функциональный блока в схемах Насси-Шнейдермана

Каждый символ схем Насси-Шнейдермана + каждый прямоугольник внутри любого символа – блок обработки

***2).* Блок следования**объединяет ряд друг за другом процессов обработки.

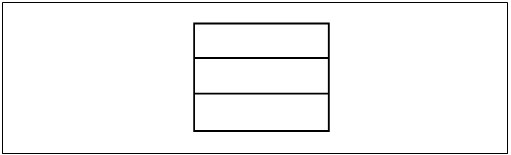
**

Рис 3.27 – блок следования в схемах Насси-Шнейдермана

***3). Блок решения***

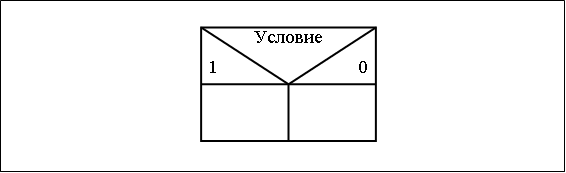


Рис 3.28 –блок решения в схемах Насси-Шнейдермана **If Then Else**

- д/**If-Then-Else**. Условие записывается в центральном треугольнике, варианты исполнения условия – в боковых треугольниках (варианты исполнения: *1, 0; да, нет; +, −*). Процессы обработки обозначают прямоугольниками.

***4). Блок Case***

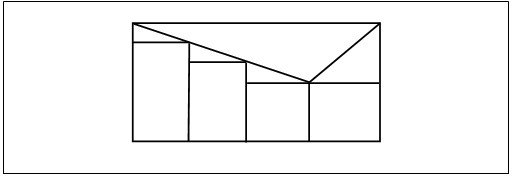


Рис 3.29 − блок **Case** в схемах Насси-Шнейдермана

- расширение блока решения If. Варианты выхода из этого блока, которые можно точно сформулировать, размещаются слева от нижней вершины центрального треугольника. Остальные выходы объединяются в 1 - **выход по несоблюдению условий** и расположен справа.

Если можно перечислить все случаи, м б/**правой** части, а выходы разместить по обе стороны центрального треугольника.

условие записывается в центральном треугольнике, варианты исполнения условия – в боковых треугольниках. Процессы обработки обозначаются прямоугольниками.

***5). Цикл “Пока” While Do***.

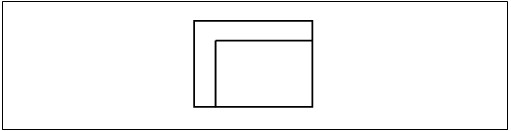


Рис 3.30 − **цикл “Пока” (цикл While**) в схемах Насси-Шнейдермана

Условие выполнения цикла - в верхней полосе.

***6). Цикл “До”*** Repeat Until

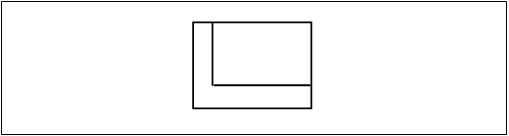


Рис 3.31**− цикл “До”** (цикл **Repeat-Until**) в схемах Н-Шнейд-на

Условие - в нижней полосе.

Рис 3.32 – 3.34 (см ниже) – примеры структурированных схем алгоритмов (слева) и соответствующие им схемы Насси-Шнейдермана (справа).

***Bi*** обозначает ***i***-ое **условие**, ***Di*** – ***i***-ое **действие** (Do).

схемы Насси-Шнейдермана - более компактные, чем аналогичные (в соответствии с требованиями *ГОСТ 19.701-90)*.

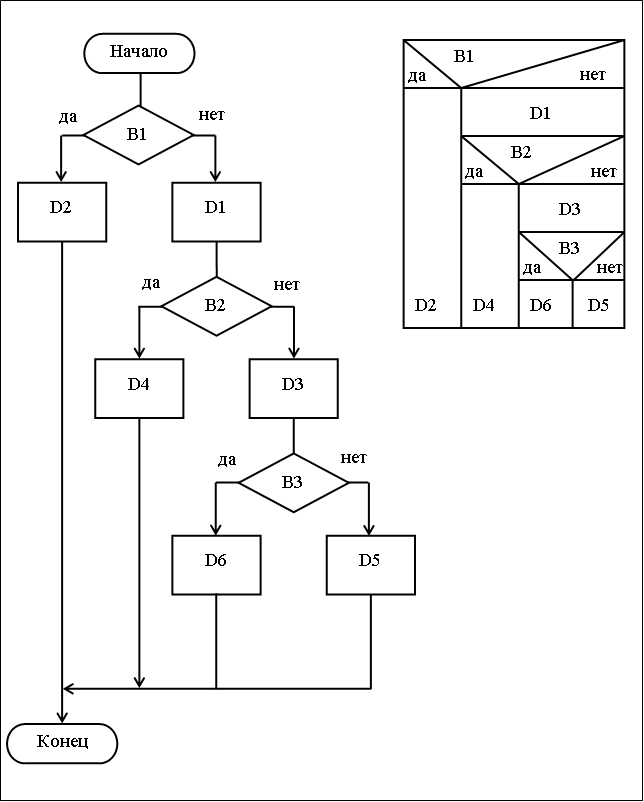
****

Рис 3.32 – Пример схемы Насси-Шнейд-на для алгоритма свложенными **разветвлениями.**

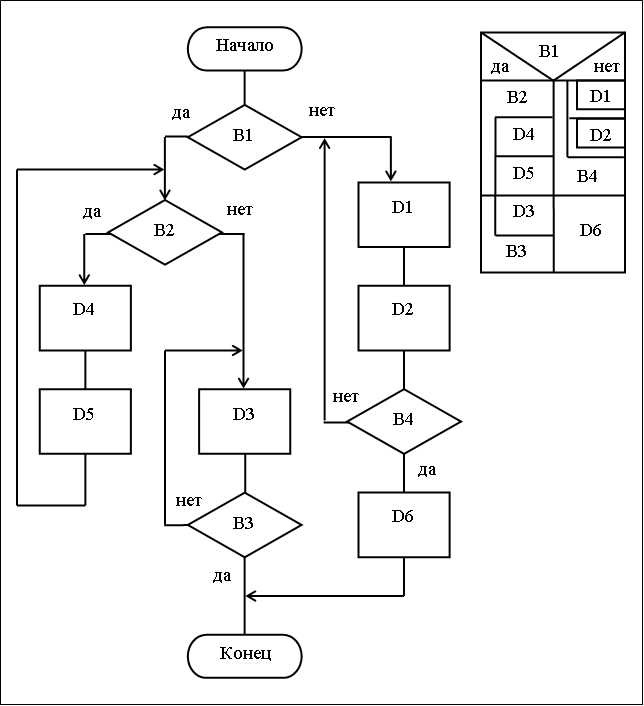


Рисунок 3.33 – Пример схемы Насси-Шнейдермана алгоритма с последовательностью **циклов**

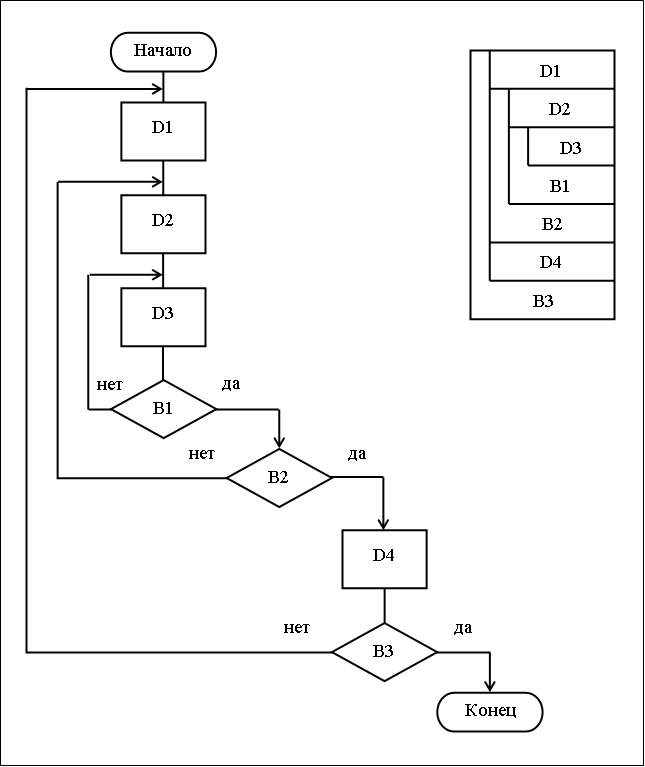


Рис 3.34 – Пример схемы Насси-Шнейд-на для алгоритма с вложенными циклами

**Пример 3.2.**

**555555555555555555**

Дан массив ***А***, состоящий из ***N*** элементов. Найти наибольший из элементов массива (***Amax***) и его номер (***Imax***). Нарисовать укрупненную и детализированную схемы Насси-Шнейдермана.

Схема алгоритма решения данной задачи - приведена в п. 3.4.1.

Схемы Насси-Шнейдермана для решения данной задачи:

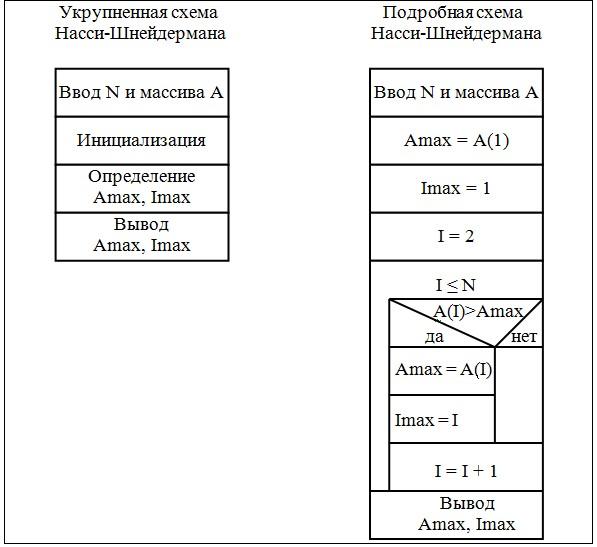


Рис 3.35 – Пример диаграммы Насси-Шнейд-на для алгоритма поиска максимального элемента массива и его номера

**Раздел 4. Алгоритмический язык программирования Паскаль**

**4.1. Общая характеристика Паскаль**

Язык Паскаль относится к **процедурно-ориентированным** языкам высокого уровня. Разработал - американский ученый **Никлас Вирт, 1971 г.** язык для обучения программированию. База при разработке - язык **Алгол.**

*«+» языка Паскаль*:

1. простота (с целью обучения программированию);
2. идеология Паскаль близка к современным методикам 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;

5). наличие средств повышения надежности программ: контроль правильности использования данных различных типов и программных элементов на этапах трансляции, редактирования и выполнения.

***Грамматическое описание*** любого языка программирования включает:

1. *алфавит* – набор символов для записи текста программы; 4.2.

2. *синтаксис* – **правила** построения фраз;

3. *семантику* – **смысловое** значение фраз (языка). А С С

**4.2. Алфавит Паскаль**

- делится на ***три*** ***группы***: \_

**1. Буквы** - прописные (заглавные) буквы латинского алфавита от ***A*** до ***Z***, строчные буквы латинского алфавита от ***a*** до ***z*** и символ подчеркивания­­\_.­ В программах строчные латинские буквы эквивалентны прописным везде, за исключением **литералов (строковых констант**). Например, **неразличимы записи**

***cost COST Cost ,***

**но литералы** *‘cost’ ‘COST’ ‘Cost’ -* разные.

**2. Цифры -** 10 десятичных цифр от ***0*** до ***9***.При записи программы на бумаге перечеркивают цифру ***0***, чтобы отличить ее от буквы ***O***.

**3. Специальные символы.** Классификация специальных символов:

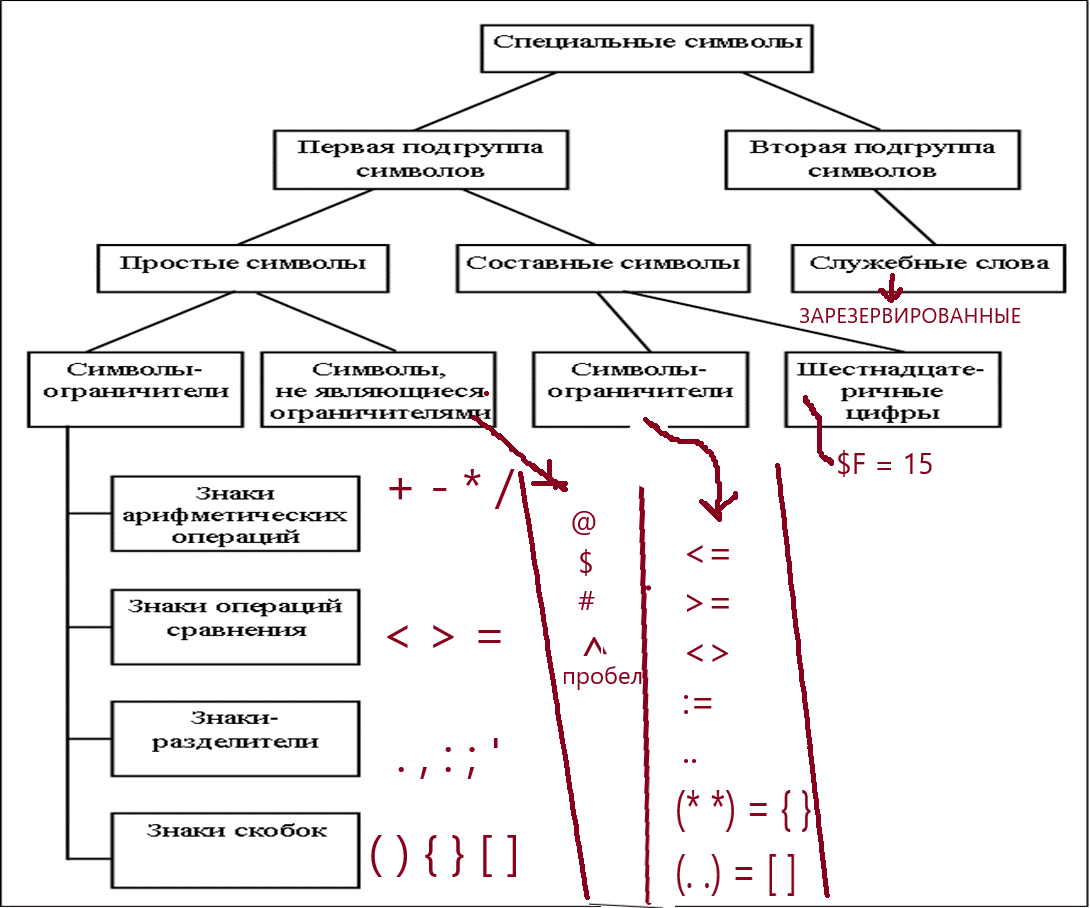


Рис 4.1 – Классификация специальных символов

Специальные символы делятся на 2 подгруппы (см рис 4.1).

**Первая подгруппа -** **простые и составные** спец символы.

**Простые символы**:

**1)****Символы-ограничители:**

1.1) знаки арифметических операций:

**+** плюс (сложение, объединение множеств, сцепление строк);

**–** минус (вычитание, разность множеств);

**\*** звездочка (умножение, пересечение множеств);

**/** наклонная черта, слэш (знак деления, результат всегда имеет вещественный тип);

1.2) знаки операций сравнения:

**<**  меньше;

**>** больше;

**=** равно;

1.3) знаки-разделители:

**.** точка (десятичная точка в вещественных константах, разделитель полей при обращении к записи, признак конца модуля (блока));

**,** запятая (разделитель в перечислениях)

**:** двоеточие (отделяет объявляемый элемент от его определения, и метку от оператора);

**;** точка с запятой (разделитель операторов программы);

**′** апостроф (ограничитель строковых констант);

1.4) знаки скобок:

**( )** левая и правая скобки (выделения подвыражений);

**[ ]** левая и правая квадратные скобки (индексы массивов);

**{ }** левая и правая фигурные скобки (комментарии);

**2)****Символы, не входящие в подгр символов-ограничителей:**

**@** коммерческое ‘at’ (операция взятия адреса элемента);

**#** номер (номер элемента в коде **ASCII**);

**$** знак доллара (признак **16-**ричных констант);

**^** знак карата (указателей и динамических переменных);

пробел (символ, не имеющий обозначения).

**Cоставные** специальные **символы (тоже** *символы-ограничители)* **–**  имеют специальное значение:

**<=** меньше или равно;

**>=** больше или равно;

**<>** не равно;

**:=** операция присваивания;

**..** обозначение диапазона.

Кроме того, имеются составные символы, являющиеся **аналогами** специальных символов скобок:

**(\* \*) аналоги { };**

**(. .) аналоги [ ].**

**16-ричная цифра** *=* 10-чные цифры + ***A, B, C, D, E, F*** (или ***a, b, c, d, e, f***). Обозначения 16-ричных констант начинаются со знака ***$*** (***$F*** – = число 15 в 10-чной системе счисления) – тоже составные символы.

**Вторая подгруппа** специальных символов: **служебные слова** –определенные комбинации латинских букв, имеющие фиксированный смысл. Их нельзя применять для других целей - ***зарезервированные слова***:

**Absolute And Asm Array Begin Break Case Const Constructor Continue Destructor Div Do Downto**

**Else End Exit External File For Forward Function Goto If Implementation In Inline Interface Interrupt**

**Label Mod Nil Not Object Of Or**

**Packed Procedure Program Record Repeat Set Shl Shr String Then To Type Unit Until Uses Var Virtual While With Xor.**

**4.3. Основные понятия языка**

**Текст** проги состоит из: 1.лексем, 2. комментариев и 3.пробелов.

**Лексема** (**лексическая единица**) - неделимая последовательность знаков алфавита, имеющая в программе определенный **смысл.**

**типы лексем**:

1) простые и составные **спец. Символы** (см в предыдущем подразделе);

2) **идентификаторы** (**символические имена**); 4.3.1.

3) литералы (строковые и символьные **константы**);

4) числовые константы (**Pi**);

4.3.1. Идентификаторы

***-*** Последовательность символов **для обозначения** какого-о **элемента** программы / его атрибутов.

- для обозначения переменных или других элементов программы используются **символические имена.**

В Паскале идентификатор - любая последовательность букв, цифр и знаков подчеркивания, начинается буквой или знаком подчеркивания.

**Внутренние** идентификаторы - обозначают элементы внутри программы. *максимальная длина* = **до 126** символов, т.е. = длине программной строки. Иначе избыточные справа символы обрежутся. Значимыми являются только первые **63 символа.**

+ идентификаторы м использовать для связи между собой независимо транслированных частей программы и с операционной системой (ОС)- называются **внешними -** **до 8** символов (лишние справа - обрежутся).

Идентификаторы делят на **2 группы:**

1. предопределенные (**стандартные,** встроенные);
2. определенные программистом.

**1) Предопределенный (стандартный,** встроенный**) идентификатор** –идентификатор, имеющий **стандартный смысл** и входит в описание языка, не является служебным словом. Поэтому можно менять их смысл, используя описание. Но это нежелательно, чтобы избежать ошибок. Это - имена встроенных функций и атрибуты переменных. Примеры:

***Real*** – атрибут действительных чисел.

***Integer*** – атрибут целых чисел.

***Sin*** – встроенная функция синус.

**2) Идентификаторы, определенные прогером** – его смысл определен в самой программе. Они задают **имена элементов программы.** Например, ***Х*** – имя переменной. Примеры:

*Name X ALFA N\_18*

Хороший стиль программирования - использование **смысловых** идентиф-ров. Для разделения отдельных смысловых частей идентификатора м использовать знак подчеркивания (например, *SERGEY\_IVANOV*) или большие и малые букв (*NomerVarianta*).

Примеры **непра**вильных идентификаторов:

*7AR* – первый символ цифра;

*R.8* – содержит специальный символ – точку;

*NAME NINA* – содержит пробел.

4.3.2. Комментарии

***-*** для внесения пояснений в программу и может размещаться **везде, где допускается пробел**. М б любая последовательность символов: русские буквы, за исключением символов-ограничителей комментариев. ограничен специальными символами:

**{Комментарий}** или **(\*Комментарий\*)**

Вложенность комментариев не допускается. Но разного вида – можно:

**(\*...{...}...\*)**  или **{...(\*...\*)...}**

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

Основные **виды комментариев**:

**1. вводный комментарий** – записывается перед текстом программы, в котором даются общие сведения о программе: назначение программы, методы вычислений, длительность работы программы, необходимые ресурсы памяти, даты разработки и последнего обновления, автор, …;

**2.** комментарии**-заголовки** – записываются перед подпрограммами (или большими блоками программы), описывается: назначение подпрограмм (блоков), выполняемые действия, методы, положенные в основу их работы и т.д.; нормы: 4 – 5 строк на блок;

3. **строчные** комментарии– описывают мелкие фрагменты программ; ***норма*: один комментарий на каждые 2-3 строки** исходного текста программы.

**4.4. Структура простейшей программы**

Программа состоит из 1 или несколько независимо компилируемых **исходных модулей**. Текст программы состоит из 1. лексем, 2. комментариев и 3. Пробелов, 4. операторов.

**3. Пробел** – это символ, не имеющий графического изображения. Однако он имеет в машине **цифровой код,** могут использоваться вне (между) лексем, в комментариях и в литералах (как часть их значения).

Идентификаторы, служебные слова, константы должны отделяться друг от друга хотя бы 1 пробелом (допускается любое число пробелов), комментариями или специальными символами (из группы **символов-ограничителей**).

1. Каждая **лексема** должна целиком размещаться в одной строке программы. **Перенос лексем на следующую строку не разрешается!**

**2. Комментарии** записываются вне лексем и могут размещаться на нескольких строках.

**3. Оператор** – **команда**, указывающая, какие действия должны быть выполнены машиной. 2 группы: **основные (простые) и производные.**

**1. Основные операторы** – не имеют в своем составе других операторов. Например, оператор присваивания:

X := X + Y

**2. Производные операторы** – операторы, в состав которых входят другие операторы, например, операторы цикла, составной оператор.

Признак конца оператора - точка с запятой:

**;**

Программа состоит из 3 **основных разделов**:

1. заголовок программы; - 1 строка.
2. раздел описаний (**объявлений**); («Послушайте объявление: …»).
3. раздел операторов.

2 и 3 разделы = **тело программы (блок).** Признак конца тела - **точка.** Простейшая программа схематически изображена так:

***Program N ;*** — заголовок программы;

***Var A1 ; A2 ; ... ; Ak ;*** — раздел описаний;

***Begin P1 ; P2 ; ... ; Pn  end.*** — раздел операторов.

Здесь: ***N*** – имя программы (**внешний идентификатор**);

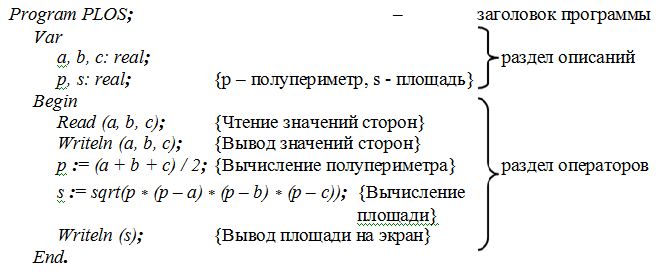
***Ai*** – описание ***i***-й группы переменных;

***Pj*** – ***j***-й **оператор.**

Для повышения наглядности и его соответствия принципам построения СтрПрог надо отражать вложенность управляющих структур друг в друга. Для этого **- отступы (табуляция):** служебные слова, которыми начинается и заканчивается оператор, записывают на одной вертикали; все вложенные в него операторы - с **отступом вправо.**

**Пример 4.1.**

программа вычисления площади треугольника со сторонам ***a****,* ***b****,* ***c*** с использованием формулы Герона. Данная программа является линейной. Поэтому операторы выполняются последовательно.



Заголовок начинается **служебным словом *Program*,** за которым указывается имя программы. Имя - **внешний идентификатор.**

Раздел описаний переменных открывается **служебным словом *Var*** (**variable – переменная**). здесь он содержит только 2 описания переменных: 1- описывает переменные ***a, b, c***, 2 – переменные ***p, s***.

**Предопред**еленный **идентификатор *real***  – указывает, что переменные ***a, b, c, p, s*** будут иметь атрибуты **вещественных чисел (с плавающей точкой).**

В Паскаль все определенные идентификаторы должны быть объявлены.

Раздел **операторов** задает **действия,** которые нужно **выполнять** в программе. Этот раздел всегда начинается **служебным словом *Begin* и заканчивается служебным словом *End* =** **операторные скобки ( ).** Внутри них записывается последовательность операторов, разделенных точкой с запятой (**;**). Так как слова *Begin* и *End* эквивалентны скобкам (по смыслу), то после *Begin* и перед *End* точка с запятой (**;**) не ставится:

*Begin P1****;*** *P2****;*** *...****;*** *Pn End* = = *(P1; P2; ...; Pn).*

1). *Read (a, b, c)* – прочитать значение переменных **a**, **b**, **c** из **вход**ного **файла** ***Input***. Что такое оператор?

2). *Writeln (a, b, c)*  – стандартная процедура: вывести в выходной **файл *Output*** значения ***a****,* ***b****,* ***c*** - **для контроля** правильности введенных данных.

**Файл** – **набор данных**, организованный специальным образом в памяти машины, который может отображаться на различные внешние носители (экран дисплея, бумагу устройства печати, магнитные диски и т.п.).

2 стандартных **текстовых файла** – ***Input* и *Output*.**

Имя ***Input*** имеет системный файл **из фйала в программу** читаются **исходные** данные. связан с клавиатурой.

Имя ***Output*** имеет системный файл **для вывода** результатов вычислений. отображается на дисплей.

3). выполняют **операторы присваивания**, в которых выполняются вычисления (в **p** заносится полупериметр треугольника, в ***s*** – площадь треугольника; ***sqrt*** – **встроен**ная функция квадратного корня).

4). вывод результатов вычислений - *Writeln (s)*. Признак конца - точка.

Литералы?????????

Числовые константы??????????

**4.5. Способы описания синтаксиса**

**Синтаксис** – **набор правил и соглашений** (договоренностей), описывающих правильные предложения языка. Для записи правил синтаксиса языков применяют формализованные **системы обозначений - метаязыки**.

1. метаязыка для описания синтаксиса Паскаль:

**1). язык металингвистических формул.** Одна из его разновидностей –расширенная форма Бэкуса-Наура (РБНФ);

**2). синтаксические диаграммы.**

4.5.1. Расширенная форма Бэкуса-Наура (РБНФ)

В метаязыках есть понятия: **1. метапеременная, 2. Метаконстанта (const), 3. синтаксическая 1ца и 4. Метасимвол (=, { }, [ ], | ).**

**1). Метапеременная** – **конструкция языка** (кроме основных символов). Для записи метапеременных - последовательности слов **русского языка** и служебных слов, между которыми - символ подчеркивания ( \_ ).

Метапеременные при записи заключаются в угловые скобки < ... \_ … >. Примеры записи метапеременных: *<Шестн.\_цифра> <Оператор\_While> <Объявление\_переменных>*

**2). Метаконстанта** – **лексема языка**. В программе метаконстанте соответствует она сама. В РБНФ метаконстанты заключают **в кавычки**.

Примеры метаконстант: *“End” “+” “For”*

***3).* Синтаксическая 1ца** – **строка,** описывающая **состав и порядок следования** элементов конструкций языка. Состоит: **метапеременные + метаконстанты + метасимволы.**

Метапеременная в **синтаксической диаграмме** означает: фрагмент диаграммы должен быть детализирован **подстановкой** синтаксической диаграммы с соответствующим именем данной метапеременной.

**4). Метасимволы** – спец. символы для описания синтаксиса яз:

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

б) **.** точка;обозначает конец определения;

в) **|** вертикальная черта; выбор, альтернатива («либо», «или»);

г) **{ }** фигурные скобки; возможность повторения заключенной в них конструкции 0, 1 или > раз; например, **{<Тело цикла>};**

д) **[ ]** квадратные скобки; - **необязательная** часть конструкции, т.е. возможность повторения заключенной в них конструкции 0 или =1 раз; например **[*“+”*]**- знак + перед числом м быть или нет;

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

**Пример 4.2.**

**Идентификатор – последовательность латинских букв, цифр и знаков подчеркивания, начинающаяся с буквы или символа 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> ::= “\_”.**

**Пример 4.3.**

Запись в РБНФ **синтаксиса комментария:**

**<Коммен-рий>::=“{”{<Любой\_символ\_кроме\_символов\_{\_}>}“}”**

Здесь:

внутренние { } − **часть** метапеременной;

средние **{ }** − метасимволы, означающие любое к-во повторений (заключенной в них) метапеременной;

наружные **“{” “}”** − основн символы Паскаль (**метаконстанты) –** часть его алфавита; заключены в РБНФ в кавычки, чтобы отличить их от метасимволов.

**альтернатива** символам **{ } - (\* \*)**, то запись в РБНФ:

**<Ком-ий> ::=**

**(“{” | “(\*”) {<Любой\_символ\_кроме\_симв-ов\_{\_}\_(\*\_\*)>}( “\*)” | “}”)**

4.5.2. Синтаксические диаграммы (СинДи)

- **граф (Калиостра)** с размеченными **ребрами** (**графически** изображает структуру **синтаксической 1цы).**

СинДи имеет **имя** - **метапеременная.**

Для разметки ребер используют мета**перемен**ные и метак**онста**нты. **Метасимволы не** используют. Поэтому **метаконстанты** в кавычки не берут. Символы **< >** и **::=** -единственные используемые метасимволы.

Метапеременные - в угловые скобки **< >**.

Для отделения **имени** от графа используют **::=.**

Метапеременная на размеченном ребре графа означает, что соответствующий фрагмент диаграммы должен быть детализирован подстановкой **синтаксической** диаграммы с именем, соответствующим данной метапеременной.

Представление в виде графов несколько **метасимволов** языка РБНФ:

*1. Выбор,* **альтернатива** (метасимволу **|** (Или)) соответствует разветвление в синтаксической диаграмме с последующим объединением.

Например, переменная может принимать значение ***А*** или ***В*** (***А, В*** – это **лексемы языка Паскаль, т.е. метаконстанты**).

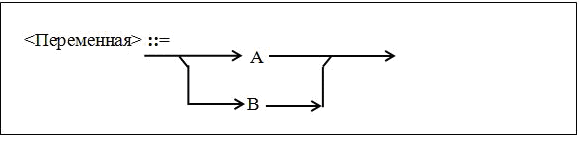


Рис 4.2 – СинДи, представляющая **выбор**

В **РБНФ** это будет записано так: **<Переменная> ::= “A”|”B**”

здесь <Переменная> - **имя СинДи.**

Вход СинДи - слева, выход – справа. Стрелки указывают возможных **преемников** каждого из элементов диаграммы.

Основное направление ребер графа – слева направо и сверху вниз.

*2.* **Необязательная часть конструкции** *(повторение 0 или 1 раз*, т.е. метасимволам **[ ]**) (рис 4.3):

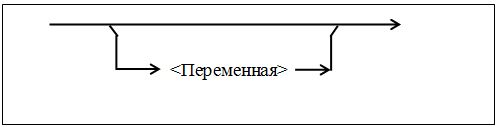


Рис 4.3 – СД **необязательной** ч конструкции в РБНФ:[<Переменная>]

*3.* Возможности **повторения** *констр-ий 0, 1 или > раз* (метасимвол **{ }**):

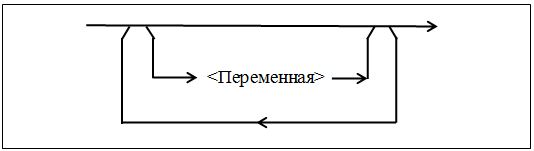


Рис 4.4 – СД возможность повторения конструкций 0, 1 или более раз

**Пример 4.4.**

**идентификатор** – последовательность букв, цифр и знаков подчеркивания, начинающаяся буквой, ... в виде СД:

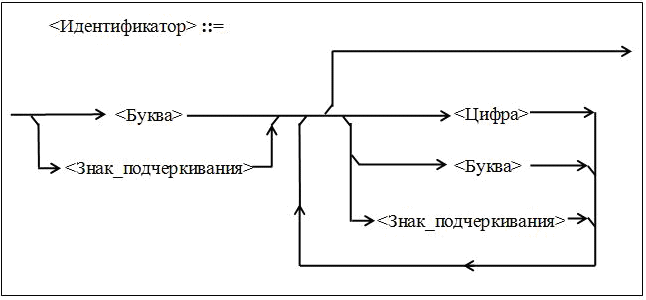
****

Рис 4.5 – СинДи определения «Идентификатор»

Если сравнивать между собой язык РБНФ и СинДи-ы, то выводы: Язык **РБНФ более** строг и точен, более удобен для представления синтаксиса в памяти машины, компактен, наглядны и просты для понимания, но громоздки.

**Раздел 5. Основные типы данных**

**5.1. Классификация данных**

**Данные *–*** любой **набор знаков**, над которыми выполняют определенные действия. Данные – 1). константы и 2). переменные.

**Константа** – элемент данных с **фиксированным значением**, определяемое записью константы в программе. Значение не меняется в ходе программы. Обозначение константы - она сама либо идентификатор при **объявлении.**

**Переменная** – элемент данных, который обозначается идентификатором и может **меняться.** Например, в операторе присваивания***X := X + 5;***

***X*** – Переменная, 5 – константа.

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

**тип значения** - число отдельных данных, входящих в структуру, тип каждого из этих данных и способ их объединения в структуру.

**Тип значений** определяет:

1.множество **значений,** которые могут принимать элементы этого типа;

2.их свойства;

3.операции над ними.

***2 группы*** типов значений:

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

**1) Основные типы** значений – состоят из единственного элемента данных (так называемые **тривиальные структуры данных**). Делят на:

**1.1) ссылочный тип;**

**1.2) скалярные типы.**

Скалярные типы в свою очередь делят на:

**1.2.1) стандартные типы** – зафиксированы в языке (предопределенные типы); например, типы **Integer, Real, Char**;

**1.2.2)** **описанные типы** – определены программистом; например, перечислимый тип.

Скалярные типы имеют **свойство перенумерованности (упорядоч-сти),** т е в элементах этого типа данных есть **линейный порядок** - относительно любых 2 элементов этого типа определено кто за кем идет.

**2) Производные (структурные) типы** – образуются на основе других типов (как основных, так и производных). состоит из нескольких элементов данных. Делят на:

2.1) **стандартные (предопределенные)** (типы String, Text, ...)

2.2) **описанные –** например, **массивы, записи** и др.

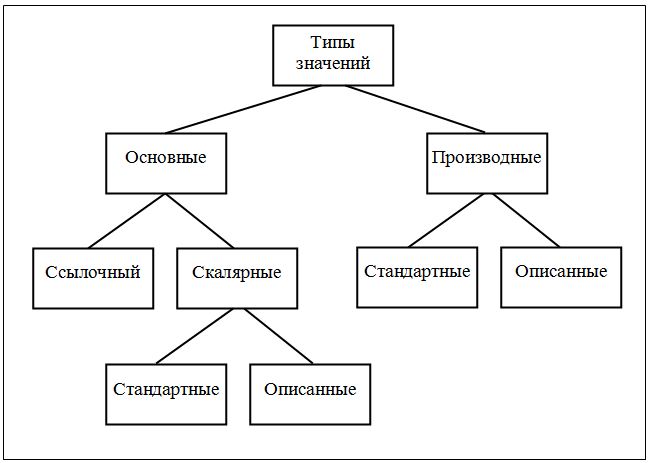


Рис 5.1 – Классификация типов Паскаль

**5.2.** Стандартные **скалярные типы данных**

5.2.1. Целочисленные типы

5 целочисленных типов:

1. **shortint** (короткое **целое**); (-128 ÷ 127)
2. **integer** (**целое**); (-32 768 ÷ 32 767)
3. **longint** (длинное **целое**); (-2 147 483 648 ÷ 2 147 483 647)
4. **byte** (байт); (0 ÷ 255)
5. **word** (слово). (0 ÷ 65 535)

Таблица 5.1 – **Характеристики целочисленных типов**

| **Тип** | **Диапазон** | Формат |
| --- | --- | --- |
| **shortint** | -27 ÷ 27 **– 1**  (-128 ÷ 127) | 1 байт  (8 битов со знаком) |
| **integer** | -215 ÷ 215 **– 1**  (-32768 ÷ 32767) | 2 байта  (16 битов со знаком) |
| **longint** | -231 ÷ 231 **– 1**  (-2147483648 ÷ 2147483647) | 4 байта  (32 бита со знаком) |
| **byte** | 0 ÷ 28 **- 1**  (0 ÷ 255) | 1 байт  (8 битов без знака) |
| **word** | 0 ÷ 216 **– 1**  (0 ÷ 65535) | 2 байта  (16 битов без знака) |

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

**5.2.1.1. Внутреннее представление целых чисел**

на базе **16-разрядных микропроцессоров Intel**. Разрядная сетка: **под** каждым разрядом указан его № (нумеровать разряды со стороны младших разрядов с 0), а над каждым разрядом – его **вес**.

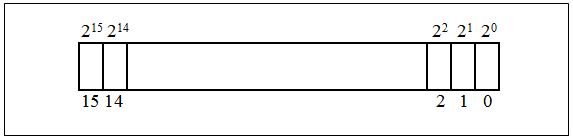


Рис 5.2 – Разрядная сетка 16-**разрядного** микропроцессора

Минимальное двоичное число, которое может быть записано в данной сетке –

f101

Максимальное число в 16 разрядах –

f101

Этот диапазон (0 ÷ 65535) соответствует формату ***Word*** Паскаля.

Для типа ***Byte*** используется 1 байт без знака. диапазон представления чисел:

f101

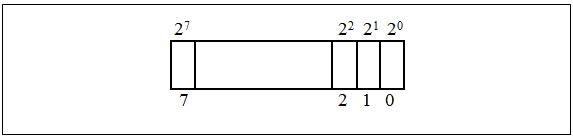


Рис 5.3 – **Представление типа Byte**

**Для чисел со знаком** старший бит (соответствующего поля памяти для хранения числа) - знаковый (рис 5.4). Если в нем 0 – число полож-ое (знак +), 1 – отрицательное (знак –).

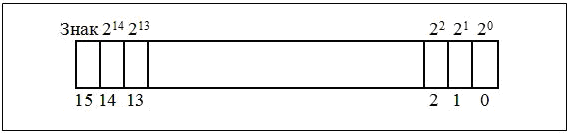


Рис 5.4 – Представление 16-ричных чисел **со знаком**

Максимальное (положительное) число в данной разрядной сетке:

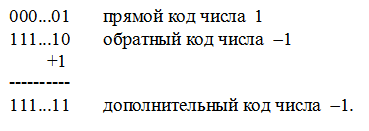
f102

Минимальное положительное число: f102

**Отрицательные числа** в памяти машины хранятся в **дополнительном коде.**

**Дополнительный код** (ДК) числа ***i*** образуется: инвертирование (замена значения на противоположное) всех значащих разрядов прямого кода числа + 1 к самому младшему разряду. Аналогичное преобразование из дополнительного кода в прямой код.

Например, пусть ***i = − 1***.



Дополнительный код числа i может также быть определен по формуле



***n*** – разрядность представления операндов (включая знак), | ***i*** |− модуль ***i***.

**Пример 5.1.**

Представление чисел для ***n = 3***:

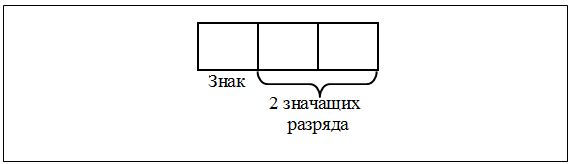


Рис 5.5 – Представление чисел для n = 3

Таб 5.2 поясняет правила получения д/п кода числа из его прямого кода в полученном диапазоне. Здесь: 10СС – десятичная система счисления, 2СС – двоичная система счисления.

Диапазон представления при ***n = 3***:

f103

Таб 5.2 – Правила получения д/п кода из прямого при ***n=3***

|  |  |  |  |
| --- | --- | --- | --- |
| Исходное число ***i***  в 10СС | числа ***i*** в 2СС | Правило получения ДК  ***2n – |i| = 8 − |i|*** | Примечания |
| –4  –3  –2  –1 | 100  101  110  111 | 8 – 4 = 4  8 – 3 = 5  8 – 2 = 6  8 – 1 = 7 | Формирование дополнительного кода для отрицательных чисел. В знаковом разряде - 1. |
| 0  1  2  3 | 000  001  010  011 | —  —  —  — | Сохранение прямого кода для положительных чисел. В знаковом разряде – 0. |

**Пример 5.2.**

Представление целых чисел для ***n = 16***. Формула для определения **дополнительного** кода имеет вид:



Таб. 5.3 – Представление целых чисел в **формате 2 байта** **со знаком**

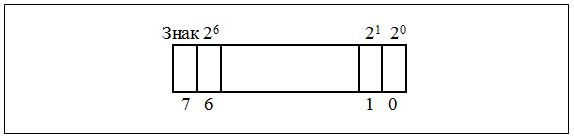
| Исходное число ***i***  в 10СС | Представление числа ***i*** в 2СС | Представл. числа ***i***  в 16СС | Примечание |
| --- | --- | --- | --- |
| –215  –215+1  –215+2  --------  −1 | 1000 0000 0000 0000  1000 0000 0000 0001  1000 0000 0000 0010  --------------------------  1111 1111 1111 1111 | 8000  8001  8002  -----  FFFF | Формирование **дополнительного** кода д/отрицательных чисел.  В знаковом разряде – 1. |
| 0  +1  +2  --------  +215 –1 | 0000 0000 0000 0000  0000 0000 0000 0001  0000 0000 0000 0010  --------------------------  0111 1111 1111 1111 | 0000  0001  0002  -----  7FFF | Сохранение **прямого** кода для положительных чисел.  В знаковом разряде – 0. |

Здесь 16СС – 16-ричная система счисления.

Т. о., диапазон чисел со знаком в 16 разрядах =:f105

что соответствует типу **Integer** в Паскале.

Аналогично представление для типа ***Shortint****:*



Диапазон чисел со знаком в 8 разрядах =: f105

Для типа **Longint** представление информации **32х** разрядных:

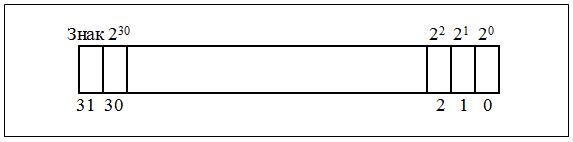


Рис 5.7 – Представление 32-**хразрядных** чисел со знаком,

диапазон представления: f105

**5.2.1.2.Целочисленные константы и переменные**

**Целые числа** используют как: 1). счетчики (для управления количеством повторений цикла) и 2). индексы. При обработке целых чисел используется арифметика с **ФТ**, округлений нет, вычисления выполняются точно.

Множество целочисленных значений является **перенумерованным –** порядковый № каждого значения этого типа – само это значение (**i =1 to n**): порядковый № 5 = 5, числа (-5) – (-5), **порядковый № 0 = 0.**

**Переменные** целочисл-ных типовобъявляют в разделе описания переменных с помощью **идентификатора типа**. Например,

*Var*

*a, b: integer;*

*c: shortint;*

*d: word;*

**Целые константы** делят на 2 типа – **10-чные и 16-ричные**:

Признак 16-ричной константы – знак ***$*** перед числом.

Примеры записи констант 10-ных: +16, 25, –48, 16-чных: $F, $9A0.

Целой константе присваивают целочисленный тип с min диапазоном, в котором есть значение этой константы.

* + - 1. **Операции над целочисленными данными**

В 2хместных 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. **Общий тип *-*** целочисленный тип с **наименьшим** диапазоном со всеми возможными значен-ми обоих типов.

Например, общий тип для целого (*Integer*) и длинного целого (*Longint*) - *Longint*. Общий тип для типов *Integer* и *Word -* *Longint*.

Действие выполняется в соответствии с точностью общего типа.

Для всех целочисленных типов **переполнения** при выполнении арифметических операций **нет.**

Таб. 5.4 – **Операции над целочисленными данными**

| Группа операций | Операция | Вид | Описание | Тип результата |
| --- | --- | --- | --- | --- |
| **Арифметические** | ***+***  ***–***  ***+***  ***–***  ***\****  ***/***  ***div***  ***mod*** | Одноместная  Одноместная  Двухместная  Двухместная  Двухместная  Двухместная  **Двухместная**  **Двухместная** | Сохранение знака  Отрицание знака  Сложение  Вычитание  Умножение  **Деление**  **Целочисленное деление**  **Остаток целочис-го деления** | Целый  Целый  Целый  Целый  Целый  **Вещественный**  Целый  Целый |
| Логические  **(Boolean)** | ***not***  ***and (&)***  ***or***  ***xor*** | Одноместная  2хместная  2хместная  2ухместная | Поразрядное **дополнение** целого  Поразрядн логич **умножение (И)**  Поразрядное логич **сложение (ИЛИ)**  Поразрядное логич **исключ-щее ИЛИ** | Целый |
| Операции **сдвига** | ***shl***  ***shr*** | left  2хместная  rite | ***i shl j*** – сдвиг **влево** значения ***i*** на ***j*** битов  ***i shr j*** – сдвиг **вправо** значения ***i*** на ***j*** битов | Тип ***i***  Тип ***i*** |
| Операц **сравнения** | **=**  **<>**  **<**  **>**  **<=**  **>=** | 2хместная | Равно  Не равно  Меньше  Больше  Меньше или равно  Больше или равно | Логический  (**boolean**) |

**Примеры к выполнению операций над целочисл-ыми данными**

**Пример 5.3.**

Выполнение операций ***div, mod*.**

Результат операции ***А* *div В =*** целой части результата деления ***А*** на ***В***. Знак результата - по общим правилам деления.

Результат операции ***А* *mod В***определяется:

f108

Например,

10 div 3 = 3: 10 – (10/3)\*3=10 – 3\*3

10 mod 3 = 1

10 div (−3) = −3

10 mod (−3) = 1

(−10) div 3 = −3 (−10) div (−3) = 3

(−10) mod 3 = −1 (−10) mod (−3) = −1

**Пример 5.4.**

Выполнение операций **сдвига *shl, shr*.**

Логические операции и операции сдвига делают **поразрядную обработку внутренних** представлений целочисленных значений. Для использования этих операций надо знать внутреннее представление целых чисел в машине и приводит к машинной зависимости программ.

Пусть числа имеют **тип *Byte*.**

**А) 2 shl 5**

Внутреннее представление числа 2:

f108

Сдвиг 2ичного представления числа 2 влево на **5 двоичных разрядов** =:

f108

Поэтому *2 sh****l*** *5 = 64*. = эквивалентен **умножению** данного числа на 25

f109

Сдвиг двоичного представления числа влево на 1 двоичный разряд =ен умножению на 2.

**В) 6 shr 2**

Внутреннее представление числа 6:

f109

Сдвиг двоичного представления числа 6 вправо на 2 двоичных разряда =:

f109

Поэтому *6 shr 2 = 1*. = целой части результата **деления** на 22: f109

сдвиг двоичного представления числа вправо на 1 двоичный разряд = **делению** на 2.

Внимание: при сдвиге **вправо** отрицательныхчисел слева **вдвигат 1-цы.** При сдвиге **влево – 000 (🡺 1, 🡸 0)**.

**Пример 5.5.**

Выполнение поразрядных **логических** операций ***and, or, xor, not.***

Таблица 5.5 – **Правила выполнения логических операций**

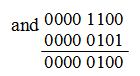
| And (И, **·**) | Or (Или, +) | Xor () | Not (Не) |
| --- | --- | --- | --- |
| 0 **·** 0 = 0  0 **·** 1 = 0  1 **·** 0 = 0  1 **·** 1 = 1 | 0 + 0 = 0  0 + 1 = 1  1 + 0 = 1  1 + 1 = 1 | 0  0 = 0  0  1 = 1  1  0 = 1  1  1 = 0 | 0**→**1  1→0 |

Пусть числа имеют тип ***Byt****e*.

**А) 12 and 5**

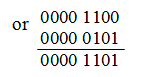
Внутреннее представление числа 12: f110

Внутреннее представление числа 5: f110



Результат операции = f110 Т. е.: **12 and 5 = 4.**

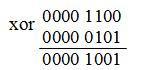
**В) 12 or 5**



**0001 0001**

Результат операции = f110 Т. е. **12 or 5 = 17**.

**С) 12 xor 5**



Результат операции = f111 т. е. : ***12 xor 5 = 9*.**

**D) not 12**



Результат операции = f111***Т.е.:* not 12 = 243**.

* + - 1. **Встроенные процедуры и функции (подпрограммы) над целочисленными данными**

– определены в **компиляторе языка** программирования. Имена встроенных процедур и функций - **предопределенные идентификаторы.**

Для получения значения встроенной процедуры (или функции), в программе записывают ее имя и аргументы так:

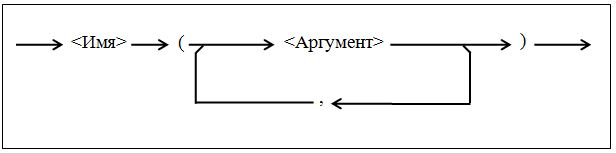


Рис 5.8 – Синтаксическая диаграмма **обращения к подпрограмме** встроенной процедуры или функции

**Определено 100 встроенных** процедур и функций – это кроме определенных в стандартных модулях!!!

Табл 5.6 – Процедуры и функции, определенные над целочисл-ми дан-и

| **Подпрограмма** | Вид | Описание | Тип рез-та |
| --- | --- | --- | --- |
| ***Chr(x)*** | Функция | Возвращает **символ кода** ASCII с заданным № (*x: byte*) | char |
| ***Ord(x)*** | Возвращает **порядковый №** скалярного аргумента x  (*Ord(5) = 5; Ord(0) = 0; Ord(-5) = −5*) | longint |
| Abs(x) | Абсолютное значение х | Тип x |
| Sqr(x) |  | Тип x |
| ***Dec(x, [n])\**** | Процедура | **Уменьшает** значение x на величину n (по умолчан на 1; *n:longint*) декремент | Тип x |
| ***Inc(x, [n])\**** | **Увелич-ет** значение ***x*** на величину ***n*** (по умолч-ю на1; *n:longint*) инкремент | Тип x |
| ***Odd(x)*** | Функция | Возвращает **True,** если аргумент **не**четный **1**, **False** – четный **0** (*x:longint*) | Boolean |
| ***Pred(x)*** | Возвращает **предыдущий** элемент в типе аргумента (для целочисленных данных возвращает ***x − 1***) | Тип x |
| ***Succ(x)*** | Функция | Возвращает **следующий** элемент в типе аргумента (для целочисленных данных ***x + 1***) | Тип x |
| ***Hi(x)*** | Возвращает **старший байт** своего аргумента (*x: integer* или *x: word*) | byte |
| ***Lo(x)*** | Возвращает **младший байт** своего аргумента (*x: integer* или *x: word*) | byte |
| ***Swap(x)*** | Возвращает значение, образованное сменой младшего и старшего байта своего аргумента (*x: integer* или *x: word*) | Тип x |
| Random(N) | Возвращает **случайное** число из диапазона ***0..N – 1*** (*N:word*) | Word |
| ***Sizeof(x)*** | Возвращает **число байт**, занимаемых своим аргументом (аргумент может быть именем переменной или именем типа) | Word |

*Прим*.\* - в квадратные скобки взята необязательная часть конструкции.

5.2.2. Вещественные типы

- числа с **плавающей точкой (ПТ**) в виде мантиссы и порядка.

Например,

f113

числа с ПТ представляются **неточно**, операции над ними выполняются по правилам действий над **приближенными числами**. Поэтому множества значений вещественных типов не имеют свойства **упорядоченности (= перенумерованности).**

**1). Вещественные константы**

*-* записаны в форме числа с **ФТ** или с **ПТ**.

***В форме с ФТ*** запись константы состоит из целой и дробной частей, разделенных точкой. **Знак необязателен**: 0.25, −2.48, +31.0.

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***):

***mEp*** или ***mep***

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

f114

Таб 5.7. Примеры вещественных чисел в формате с плавающей точкой

|  |  |
| --- | --- |
| Представление числа | Значение числа |
| 14.3E5 | 14,3\*105 |
| 681E-2 | 681\*10–2 |
| -5.16E-3 | -5,16\*10–3 |

2). **Вещественные типы и их представление в компьютере**

5 вещественных типов для представления **переменных:**

***1. Real* (вещественный);**

***2. Single* (с одинарной точностью);**

***3. Double* (с двойной точностью);**

***4. Extended* (с повышенной (**расширенной**) точнос);** временный

***5. Comp* (сложный тип).**

4. Тип **Extended** еще называется ***временным*,** т.к. при вычислениях в сопроцессоре все остальные типы **преобразуются к нему**. Данные типы различаются диапазоном и точностью представления связанных с ними значений:

Таб. 5.8 – Диапазоны и точности представления вещественных данных

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

*Примечание*. \*) – включая скрытый (неявный) бит ***F0*** (старший бит мантиссы, которого при передачах чисел и хранении их в памяти нет, т.к. его значение в нормализованных числах =1 – для всех чисел, кроме 0). В значениях типа *Extended* ***F0*** хранится явно.

**Нормализованное число *-*** его **мантисса** попадает в диапазон

f115

где ***q*** – основание системы счисления. Для 2СС f115

то есть **нормализованная мантисса - правильная дробь** с 1цей в старшем 2ичном разряде. Например, нормализованные мантиссы могут принимать значения:

0.**1**101 0.**1**000 0.**1**111:

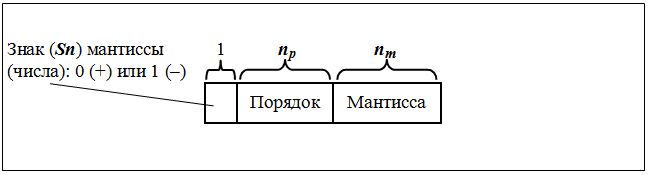


Рис 5.9. − Представление вещественных чисел в памяти компьютера



Значение числа = :

***mдв*** – значение 2ичного кода мантиссы, ***р*** – значен 2ичного кода порядка.

В различных микропроцессорах **поля** мантиссы и порядка могут меняться местами. Но это не имеет значения.

**1. Вещественные типы** различают **разрядностью: 1). поля мантиссы (*nm*) и 2). поля порядка (*np*).**

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

f116

**2.** Например, для типа **Single смещение** =127, для типа *Double* – 1023, для типа *Extended* – 16383. Характер-ка определяется выражением:

***Характеристика = порядок + смещение.***

Т. о., **формат вещественного числа** в битах (занимаемая им область памяти) определяется выражением:

f116

**Пример 5.6.**

Представление числа ***Х = –0,375*** типа *Single* в памяти машины.

f117

Представление числа Х в виде мантиссы и порядка:

f117

**Таким образом, *m = –0.011, p = 0*.**

**Порядок представляется в виде характеристики: 0 + 127 = 127.**

Представление числа ***Х*** в виде мантиссы и характеристики:

f117

После **нормализации** мантиссы: f117

с учетом представления характеристики в двоичном коде, а так же скрытого бита мантиссы число –0,375 типа *Single* в памяти машины:

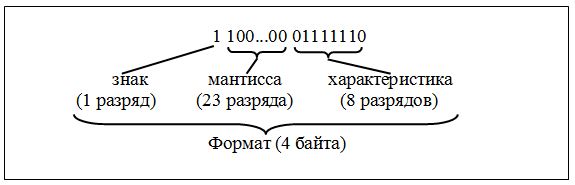


Рис 5.10 – Внутреннее представление числа −0,375

***5. сложный тип Comp -*** **не вещественный** тип с ПТ, а **длинный целый тип двойной точности**. В памяти машины он занимает формат 8 байт (64 бита) и обеспечивает диапазон представления чисел

f117

Тип *Comp* отнесен к вещественным типам т.к. использование вещественных типов *Single, Double, Extended, Comp* (т.е. всех, кроме *Real*) возможно только при **наличии** сопроцессора при включенной опции компилятора **{$N+}** (сопроцессор математических вычислений; глобальная опция) или при включенной опции ***{$E+}*** (полная программная эмуляция работы сопроцессора в случае его отсутствия; локальная опция). Последний режим очень медленный, поэтому его не используют.

**По умолчанию** - состояния ***{$N–}*** и ***{$E–}.***

Если сопроцессора нет, все вычисления с вещественными числами выполняются программно. Поэтому из-за соображений **скорости** допускаются только действия над переменными типа **Real**.

**А). Объявление переменных вещественных типов**

*-* объявляются в разделе описаний переменных. Например:

*Var*

*a, b, c: real;*

*d, e: single;*

**Б). Операции над вещественными данными**

1. ***Арифметические*:**

1.1) одноместные

+ (сохранение знака);

– (изменение знака);

1.2) двухместные

+ (сложение);

– (вычитание);

\* (умножение);

/ (деление).

Все арифметические операции дают тип **результата *Real*** (if нет сопроцессора) и **Extended** – при наличии сопроцессора.

1. ***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.

**В). Встроенные функции над вещественными данными**

Таб 5.9 – Встроенные функции, определенные над веществ-и данными

| **Функция** | Описание | Тип результата |
| --- | --- | --- |
| ***Round(х)*** | **Округление**, результат – ближайшее к ***х*** целое | LongInt  LongInt |
| ***Trunc(х)*** | **Целая часть** числа ***х*** независимо от знака (**по модулю**) как **div** |
| ***Int(х)*** | Возвращает **целую** часть ***х (div)*** | Real (или Extended)  Real (Extended)  Real (Extended)  Real (Extended)  Real (Extended)  Real (Extended) |
| ***Frac(х)*** | Возвр-ет **дробную** часть ***х (*mod*)*** |
| ***Abs(х)*** | Абсолютная величина ***х*** |
| Arctan(х) | ***arctg(х)***. Результат в радианах. |
| Cos(х) | ***cos(х)***. ***х*** – в радианах |
| Sin(х) | ***sin(х)***. ***х*** – в радианах |
| Exp(х) | (экспонента; ***е*** в степени ***х***) | Real (Extended)  Real (Extended)  Real (Extended)  Real (Extended) |
| Ln(х) | ***ln(х)*** |
| Sqr(х) |  |
| Sqrt(х) | (квадратный корень из ***x***) |
| ***Sizeof(х)*** | **Количество байтов** для представления вещественного значения | Word |

Стандартная вещественная константа: ***Pi = 3.1415926536E+00*** - тип *Real* или *Extended* (в зависимости от опции компилятора ***{$N})***. Обращение к ней в программе - по имени ***Pi***.

5.2.3. Символьный тип (литерный, Char)

тип ***Char*** – **скалярный тип.** Его значения – **элементы расширенного набора символов (литер**) кода **ASCII (американский (USA) стандартный код обмена информацией).** Для представления значений - 1 *байт памяти*.

Элементы множества значений типа *Char* считаются **перенумерованными ( = упорядоченными) =**  каждому значению типа *Char* поставлен в соответствие свой порядковый №, который = его коду ASCII.

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

**1) наименьшие** порядковые номера имеют: а) управляющие символы и б) специальные символы алфавита языка (но они не совсем упорядочены, они встречаются в разных местах кодовой таблицы);

2) десятичные цифры (упорядочены по возрастанию);

**3)** заглавные латинские буквы (по алфавиту);

4) маленькие латинские буквы (по алфавиту);

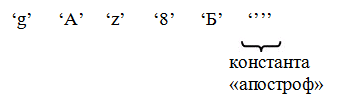
1. псевдографика (в основном варианте кодовой таблицы; в альтернативном же варианте - следует после русских букв);
2. заглавные русские буквы (по алфавиту);
3. **наибольшие** порядковые номера - малые русские буквы (по алфавиту).

***Переменные*** **Char** объявляют в разделе описания переменных.

***Var***

***a, b, c, x: char;***

**Константа** типа ***Char*** - 1 из допустимых символов **в апострофах**. Если значением константы является сам апостроф, то он записывается 2жды:



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

**=**  ; **<>**  ; **>=**  ; **>**  ; **<=**  ; **<**  .

Тип результата – **Boolean.** Сравнивают внутренние коды символов (в коде ASCII), т.е. порядковые №№ символов во множестве Char (с учетом упорядоченности).

**Встроенные функции** над значениями типа Char**:**

Таб 5.10 – Встроенные функции над символьными данными Char

| Функция | Описание | Тип рез-та |
| --- | --- | --- |
| **Ord(х)** | Преобразует ***х*** к целочисленному типу (возвращает порядковый № символа ***х*** во множестве значений типа *Char* в коде ASCII) | Longint |
| **Pred(х)** | Возвращает символ: его порядковый № **на 1 меньше** порядкового № ***х*** в коде ASCII | Char  Char  Char |
| **Succ(х)** | Возвращает символ: порядковый № на 1 **больше** порядкового № ***х*** в коде ASCII |
| **Upcase(х)** | Возвращает **большую** латинскую **букву**, если ***х*** – маленькая латинская буква, иначе возвращает ***х*** |
| **Sizeof(х)** | Указывает **количество байтов** д/ представления значения типа *Char* (значение функции = 1) | Word |

5.2.4.Логический тип (Boolean)

- определяется как **скалярный тип**, его множество значений состоит из двух значений: ***False (ложь)*** и ***True (истина).***

Значения логического типа упорядочены: значение *False* имеет порядковый **№ 0**, значение *True* - **№ 1**. Занимают *1 байт памяти*.

**Логические переменные** объявляются в разделе описания переменных:

***Var***

***X, Y, Z: Boolean;***

**Логические константы *-*** **предопределенные идентификаторы**: True и ***False***. Например, можно записать:

***X := True;***

***Y := False;***

переменным ***X, Y*** логического типа присваиваются значения **констант** *True* и *False*.

Таб 5.11 – Операции над логическими данными

| Гр операц | Опера-ция | Описание | Тип рез-та |
| --- | --- | --- | --- |
| **Логические операции** | ***Not*** | Одноместная операция (НЕ), результат = *True*, если значение операнда *False*, иначе – *False*. | **Boolean** |
| ***And*** | Двухместная операция **(И),** результат = True, если значение обоих операндов *True*, иначе – *False* |
| ***Or*** | Двухместная операция **(ИЛИ**) результат = *True*, если хотя бы 1 из операндов = *True*, иначе – *False* |
| ***Xor*** | Двухместная операция (исключающее ИЛИ) результат = *True*, если только один операнд = *True*, иначе – *False* |
| **Операции сравнения** | **=**  **<>**  **>**  **>=**  **<**  **<=** | Равно  Не равно  Больше  Больше или равно  Меньше  Меньше или равно |

Таб 5.12 – **Правила** **логических операций** над логическими значениями по правилам алгебры логики (булевой алгебры), где 0 = *False* (ложь), 1 = *True* (истина).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Операция | A And B | A Or B | A Xor B | Not B |
| B  A | 0 1 | 0 1 | 0 1 | 0 1 |
| 0  1 | 0 0  0 1 | 0 1  1 1 | 0 1  1 0 | 1 0  – – |

Например, если *A = False (0); B = True (1)*, то

A And B = False =0

A Or B = True = 1

A Xor B = True =1

Not A = True =1

Not B = False = 0.

Таб 5.13 – **Встроенные функции** над логическими данными boolean

| **Ф-ция** | Описание | Тип рез-та |
| --- | --- | --- |
| **Ord(x)** | Возвращает 0, если *x = False (0)*,  1, если *x = True (1)* | LongInt |
| **Pred(X)** | Для ***x = True*** возвращает *False*, иначе не опред-о | Boolean  Boolean |
| **Succ(X)** | Для ***x = False*** возвращает *True*, иначе не опред-о |
| **Sizeof(X)** | Указывает **к-во байтов** для представления значения типа *Boolean* | Word |

**5.3. Выражения**

**Выражение** – формула для вычисления некоторого значения, состоящая из 1) операндов, 2) знаков операций и 3) круглых скобок ( ).

***1)* Операнды –** константы, переменные и указатели функций (вызовы функций, функции).

**2) Приоритет операций**

В порядке убывания ***приоритета -*** 4 ***группы операций****:*

*1. операции* ***Not, @*** *(взятие адреса объекта)*

*2. операции группы* ***умножения****:* ***\*, /, Div, Mod, And, Shl, Shr***

***3. сложения: +, –, Or, Xor***

***4. сравнения: In*, =, <>, >, >=, <, <=**

**5.3.1. Общие правила написания и вычисления выражений:**

1. Первая группа операций - самый высокий приоритет.

2. В каждой из групп - операции имеют одинаковый приоритет.

3. Операции одного приоритета - в порядке их следования в выражении (слева направо).

4. Операнд между 2я операциями с различными приоритетами, связывается с операцией **более высокого** приоритета.

Операнд между двумя операциями с равными приоритетами, связывается с операцией **слева** от него.

5. Для уточнения последовательности действий - круглые скобки ( ). Действия в скобках - в первую очередь ().

6. Последовательная запись знаков двух операций запрещена. Исключение – **операция Not**. Например, нельзя писать: ***X\* –Y***. Можно: ***X\*(–Y).***

Но можно писать: ***X\*Not Y***.

7. Многоэтажные формулы в выражении - в одну строку.

8. Знак умножения \* опускать нельзя!

**5.3.2. Типы выражений**

Каждое выражение определяет значение какого-то типа, поэтому в нем операнды тоже определенных типов. Над каждым типом значений определен свой набор операций. Поэтому выражения также делят на несколько типов.

Рассмотрим выражения **скалярных стандартных типов.**

**Скалярное** **выражение**– его операнды - скалярные величины. В результате вычисления скалярного выражения получают одно значение элемента данных.

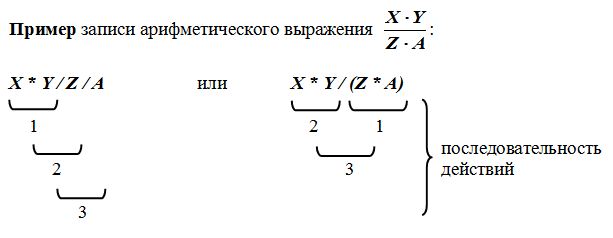
**Скаляр - это величина, которая описывается одним числом**

Стандартные скалярные выражения делятся на ***3 типа*:**

* 1. **арифметические;**
  2. **логические;**
  3. **символьные**. ***Char* (т.е. отдельные символы).**

Тип выражения определяется типом **результата** его вычисления.

1. **Арифметическое выражение** – его результат вычисления - **целое** или **вещественное** значение. Его операнды должны иметь арифметический тип (**целый/ вещественный)**. Операции - арифметические для этих типов операндов.



В Паскале **не** определена *операция* ***возведения числа******Х******в степень Y***. Но f126

Отсюда

f126

Но: вычисление  с помощью такого выражения приводит к погрешностям. Поэтому, если ***Y*** имеет целочисленный тип, то ***XY*** лучше вычислять путем умножения ***Х*** само на себя ***Y*** раз (в цикле).

**2. Логическое выражение** – его результат - логическое значение (***True*** / ***False***). Операнды должны иметь тип **Boolean** (переменные, константы, имена функций). Для логических выражений есть особый вид операнда – **отношение**. В РБНФ ***отношение*** записывается так:

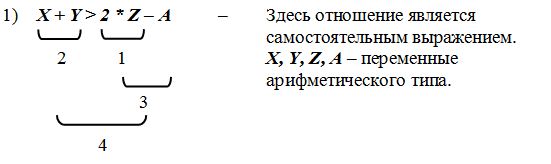
**<Отношение> ::= <Выражение> <Операция\_сравнения> <Выр-ие>**

Как выражение - м б выражения любого из упорядоченных типов и арифметические выражения. Оба выражения должны быть совместимых типов. Результат вычисления отношения - тип Boolean. Т о, в логическом выражении могут быть использованы все виды операций и все типы скалярных операндов.

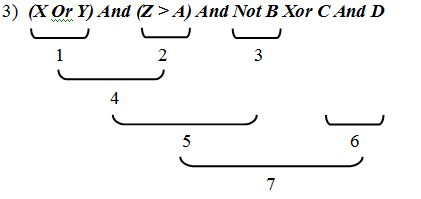
Если отношение - не самостоятельное выражение, а 1 из операндов логической операции, то оно д б заключено в круглые скобки (т.к. операции сравнения имеют наименьший приоритет).

**Пример 5.7.**

Примеры логических выражений.



2) ***Odd(X)*** – встроенная функция; возвращает значение *True*, если ***Х*** – **нечетное (1)**, *False* – ***Х*** **четное (0)**. Если ***Х*** имеет целочисленный тип, результат – *Boolean*.



Здесь отношение является операндом логической операции, поэтому в скобках. ***A, Z -*** значения арифметического/любого перечислимого типа (как операнды отношения). Результат вычисления отношения - тип *Boolean*. Все остальные операнды являются операндами логического выражения, а не операндами отношения. Поэтому они могут иметь только тип *Boolean*.

**3.Символьное выражение** – его результат - символьный тип данных ***Char* (т.е. отдельные символы).**

В Паскале нет операций с данные типа *Char*. Поэтому символьным выражением может быть: 1) **символьная константа**, 2) **символьная переменная или 3) 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*).

**5.3.3. Модели выполнения операций Or и And (2 шт)**

– полное вычисление и вычисление **по короткой схеме (частичное вычисление).**

***Полное -*** вычисляется каждый операнд логического выражения, даже если результат всего выражения уже известен.

***По короткой схеме*** - слева направо (с учетом приоритетов) и прекращается, когда очевиден результат всего выражения. Это уменьшает время вычислений.

Схема вычисления задается с помощью опции компилятора ***{$B}***. По умолчанию опция **выключена *{$B–}*** – вычисления выполняются по короткой схеме. Для полной схемы вычислений надо установить ***{$B+}.***

**Компиля́тор** — **программа** или техническое средство, выполняющее компиляцию.

**Компиля́ция** — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду. Входной информацией для компилятора (исходный код) является описание алгоритма или программа на предметно-ориентированном языке, а на выходе компилятора — эквивалентное описание алгоритма на машинном.

Опция компилятора ***{$B}*** эквивалентна пункту меню среды Паскаля «Опции / Компилятор / Булевские вычисления».

**Пример 5.8.**

дано выражение: ***(I <= Length(S))*** and ***(S[I] <> ’a’)***

***S*** – строка (переменная типа *String*),

***S[I]*** – ***I***-ый элемент строки (символ),

**Length** – встроенная функция, вычисляет длину строки.

При вычислении по полной схеме такая запись выражения является некорректной, т.к. при значении первого отношения, равном *False*, значение второго отношения не определено. При вычислении по короткой схеме вычисление выражения прекращается, как только первое отношение равно *False* (второе отношение вычисляться не будет).

Если при записи длинных выражений не уверен в последовательности выполнения операций – используй круглые скобки, даже если они не нужны.

**5.4. Оператор присваивания**

Многие программы или их фрагменты – **линейные -** операторы выполняются строго последовательно (в порядке записи) и частый оператор здесь – оператор **присваивания.**

Он предписывает вычислить значение выражения в правой части и присвоить его переменной в левой части. К моменту вычисления все переменные должны быть **определены (иметь некоторые значения).**

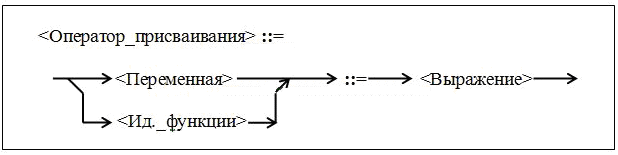


Рис 5.11 –Синтаксическая диаграмма оператора присваивания. <Ид.\_функции> − **идентификатор** (- **имя функции**).

Тип переменной в левой части оператора присваивания и тип выражения должны быть **совместимыми** по присваиванию.

3 типа скалярных стандартных оперов присваивания: **1. Арифметич-ий, 2. Логический, 3. Символьный.**

***1.* Арифметический оператор присваивания** *–* для присваивания значения переменной арифметического типа (**вещественного** или **целочисленного**). В правой части оператора должно быть записано арифметическое выражение:

**X := 0; Y := 2 \* a / b; Z := sin(c \* 2 + a \* a);**

Все переменные должны иметь **арифметический тип**.

***2.* Логический оператор присваивания *–*** в левой части указана переменная типа **Boolean.** В правой части - логическое выражение:

**A := False; B := G > L; C := (E <> F)** Or **Odd(X); D := Y = Z;**

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

***3.* Символьный оператор присваивания *–*** в левой части **переменная типа *Char***. В правой части - символьное выражение:

**A := ‘A’; B := C; D := Pred(B);**

***A, B, C, D*** – все переменные - типа *Char*.

**Стандартные типы:**



**Раздел 6. Структура программы**

Прога состоит из 1/неск-ких независимо компилируемых **модулей.**

2 вида***модулей*:**

* **программный модуль; 6.1.**
* **модуль *Unit*.**

**Программный модуль -** прога, которая получает управление от операционной системы (ОС).

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

Каждый модуль состоит из 1.заголовка, 2.*Uses*-части (предложения использования) и 3.**тела модуля (блока).** Признак конца модуля - **точка.**

**6.1. Программный модуль**

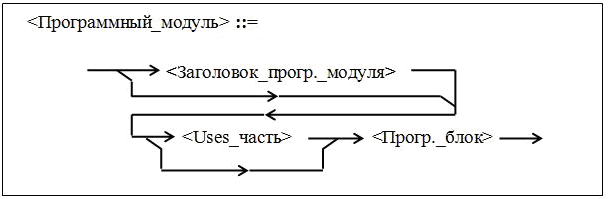


Рис 6.1 – Синтаксическая диаграмма **структуры программного модуля**

***1).* Заголовок** программного модуля начинается служебным словом ***Program*,** за которым – **имя** модуля.

Заголовок программы является **декоративным**, **для совместимости** с другими компиляторами с языка Паскаль. Заголовок компилятор**ом** **игнорируется.** Но желательно его писать и присваивать программе **смысловое** имя. Например: *Program Symma;*

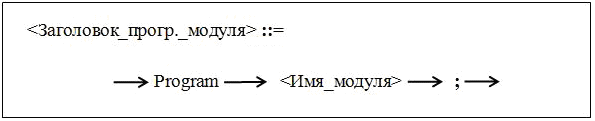


Рис 6.2 – Синтаксическая диаграмма заголовка программного модуля

Заголовок отделяется от Uses\_части **точкой с запятой** (**;**).

**Предложение использования *Uses*** определяет все модули ***Unit***, которые будут использовать в программе (как непосредственно используемые модули, так и модули, используемые этими модулями):



Рис 6.3 – Синтаксическая диаграмма **предложения использования**

**Модули *Unit***м б стандартными или созданы прогером.

***8 стандартных модулей Unit*:** **System, Dos, Crt, Printer, Graph, Overlay, Turbo3, Graph3.**

**Unit System** подключается автоматически, в предложении **Uses** не указывается. содержит встроенные процедуры и функции 1.ввода-вывода, 2.обработки строк, 3.арифметические, 4.динамического управления памятью, 5.поддержку арифметики с ПТ и т.д.

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

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

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

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

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

**Unit Turbo3, Unit Graph3** обеспечивают **совместимость** с программами, написанными для Паскаля 3.0.

Пример предложения использования: Uses **Dos, Crt, Graph; .**

***2). Тело программного модуля*** (**программный блок)** состоит из раздела 1.описаний (объявлений) и 2.раздела операторов. **Формат** программного блока:

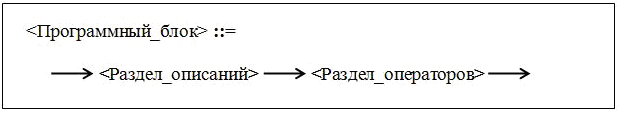


Рис 6.4 – СД **тела** программного модуля

**1.Раздел описаний (CONST, TYPE, VAR, …) –** вначале тела модуля. В Паскаль есть 5 видов **объявлений (описаний):**



В списке - стандартная последовательность разделов: следовать **строго!** друг за другом. Современные версии Паскаль допускают произвольный порядок. Отдельных видов объявлений и соответствующих разделов в модуле м не быть (пустые) – если нет соответствующих элементов. Каждый раздел м повторяться неск-ко раз.

**6.2. Раздел меток Label**

**Метка** – **идентификатор** или **целочисленная константа** в диапазоне 0 ÷ 9999 (10 000), стоящая перед оператором и отделенная от него **двоеточием** «**:**» (для выделения оператора) и играет роль **имени** данного оператора. Оператор с меткой называется **помеченным.**

Метки используют чтобы **перейти** к выполнению определенного оператора, **нарушая** последовательный порядок выполнения операторов.

Все метки операторов должны быть различными, не повторяться.

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

**Пример 6.1.**

Пример фрагмента программы:

**M1:***X* ***:=*** *X + Y****;***

*...*

**5:** *Z* ***:=*** *sin(X)****;***

*...*

*Goto M1****;***

*...*

*Goto 5;*

Все метки в разделе операторов модуля, должны быть **описаны** в разделе меток – начинается служебным словом **Label** и имеет вид:

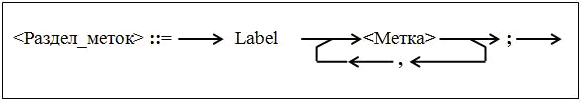


Рис 6.5 – Синтаксическая диаграмма раздела меток

Например: **Label M1, 5;**

Раздел закрывается точкой с запятой (**;**). может быть пустым (отсутствовать), если в модуле нет меток. Метки м. перечислять в **произвольном порядке**: **Label 5, M1;**

**6.3. Раздел констант**

Если в программе константы используют многократно, то м присвоить им имя: **e := 2.71;**

Такое присвоение удобно сделать 1 раз и далее обращаться к константе **по имени**. Раздел начинается служебным словом ***Const*:**



Рис 6.6 – Синтаксическая диаграмма раздела констант

В разделе констант могут быть объявлены **обычные (нетипизованные) константы и типизованные** константы.

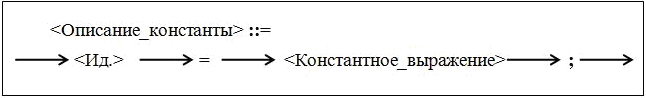


Рис 6.7 – Синтаксическая диаграмма описания **не**типизованной конст-ы

Описание константы вводит **идентификатор (<Ид.>) -** некоторое константное **выражение**. Тип константы определяется типом данного выражения.

**Константное выражение** – выражение, вычисляемое на этапе **компиляции.** не может содержать 1.имен переменных, 2.типизированных констант, 3.оператора получения адреса @, 4.включать обращение к функциям, определенным пользователем. В них можно обращаться к *стандартным функциям:* ***Abs, Chr, Hi, Lo, Length, Odd, Ord, Pred, Ptr, Round, Sizeof, Succ, Swap, Trunc*.**

В константных выражениях можно использовать только **имена констант.**

Если константа используется в программе 1 раз, то ей м не присваивать имя (в разделе констант): значение константы будет записано в выражениях **раздела операторов.** Т. о., раздел констант может быть пустым (отсутствовать).

**Пример 6.2.**

Объявление **не**типизованных констант.

**Const**

**E = 2.7115;**

**I = 10000;**

**Eps1 = 0.0001;**

**Eps2 = Eps1 / 100.0;**

**Pi1 = -2 \* Pi;**

**Типизованные константы** аналогичны инициализированным **переменным.** Это м б: 1.конст-массив, 2.Конст-запись, 3.Конст-множество, 4.конст-указатель.

**Инициализированные переменные** – им при входе в блок присваивают **начальные значения** (**инициатива)**.

Есть **скалярные и структурные** типи**зов**анные константы.

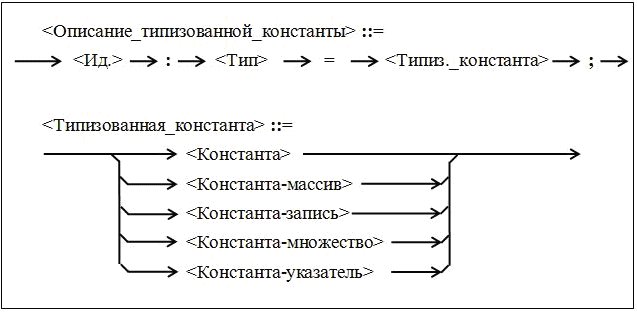


Рис 6.8 – Синтаксическая диаграмма описания **типизованных констант**

<Ид.> - идентификатор типизованной константы.

<Константа> - **скалярная типизованная** *константа*.

Остальные виды констант – **структурные**типизованные константы (будут представлены дальше).

**Типизованные константы** можно использовать как переменные того же типа (**значения можно изменять** в процессе вычислений, их м использоваться **в левой** части оператора присваивания :=).

Типизованные константы инициализируют 1 раз – **в начале** выполнения программы/подпрогр-мы (при каждом новом входе в подпрогу их заново **не** инициализируют).

Типизованные константы из всех подпрограмм **располагают в сегменте данных**. Поэтому их общий объем ограничен 64К байтами.

**Пример 6.3.**

Объявление **скалярных (просто конст) типизованных констант**.

**Const**

**MaxI: Integer = 10000;**

**MinI: Integer = 0;**

**Eps: Real = 0.0001;**

**Pi2: Real = 4 \* Pi;**

Типизованные константы нельзя использовать в объявлении других констант или типов.

Скалярные типизированные константы – просто константы.

**6.4. Раздел типов**

Можно наряду со стандартными типами значений вводить свои типы данных, которые надо описать и присвоить имя в разделе типов.

Раздел начинается служебным словом **Type** и имеет вид:

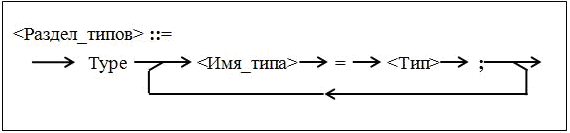


Рис 6.9 – Синтаксическая диаграмма раздела типов

После **Type** идет описания типов (разделены точкой с запятой ) – задает **имя** типа и присваиваемый ему **тип**. Тип определяется:



Рис 6.10 – Синтаксическая диаграмма описания типа

Т о, в описании типа м использовать имя **стандартного** типа, описанного ранее (верхнее ребро диаграммы) или **нестандартного** типа данных (нижнее ребро).

**Пример 6.4.** Пример **описания типов.**

**Type**

**L = Boolean; (1)**

**Int = Integer; (2)**

**Ned** = **(**Pn, Vt, Sr, Ch, Pt, Sb, Vs**);**

Первое и 2е описания - стандартные типы **Boolean** и **Integer** (для описания типа используется <Имя\_типа>, см. рис 6.10).

3-е описание - **нестандартный** тип данных – **перечислимый тип** (для описания типа используется <Задание\_типа> (см. рис 6.10). Ему присвоено имя **Ned** (Неделя). При вычислениях переменные типа *Ned* смогут принимать 1 из значений, перечисленных в скобках. Типы в разделе **Type** м использовать далее при описании **переменных.**

**6.5. Раздел переменных var**

Каждая переменная должна быть **объявлена (описана**): присвоено имя и указан тип значений. Раздел начинается служебным словом ***Var*** и имеет вид:

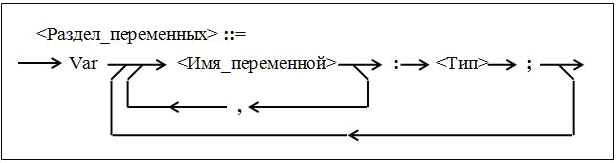
****

Рис 6.11 – Синтаксическая диаграмма раздела переменных

***2 способа*** задать Тип переменной:

**1)*.*** Тип описан в разделе типов или является стандартным. Тогда в разделе переменных указывается <Имя\_типа>.

**Пример 6.5.**

Использ **имени типа** при объявлении переменных (лучше способ)

***Var***

***X: L;***

***I: Int;***

***Day: Nеd;***

***A, B, C: Real;***

*L, Int, Ned* – описанные типы в разделе **Type**, *Real* – стандартный тип.

См пример 6.4:

**Type**

**L = Boolean; (1)**

**Int = Integer; (2)**

**Ned** = **(**Pn, Vt, Sr, Ch, Pt, Sb, Vs**);**

**2)*.*** **явное задание типа** в описании переменной - одновременно описывается переменная и задается новый **тип без имени**.

**Пример 6.6.**

Использование **задания типа** при объявлении переменных.

***Var***

**Day: (Pon, Vt, Sr, Cht, Pyat, Sub, Vosk);**

Лучше - первый способ.

Каждая переменная должна быть описана обязательно и ровно 1 раз.

***Автоматические переменные*** – для которых память выделяется автоматически при входе в блок, **где они** **объявлены** (локализованы). создаются при вызове подпрограммы и уничтожаются при ее завершении. Для описания используют Раздел переменных.

**6.6. Раздел операторов**

- основной раздел программы: описывают действия для решения конкретной задачи.

- последовательность операторов программы, разделенных точкой с зап. (**;**), заключен в операторные скобки ***Begin … End*.**

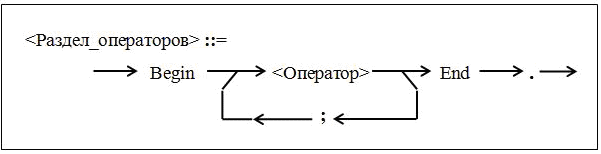


Рис 6.12 – Синтаксическая диаграмма **раздела операторов**

**Раздел 7. Операторы (оперы** и балета**)**

**7.1. Составной оператор**

- **производный** опер: в его состав входят другие оперы.

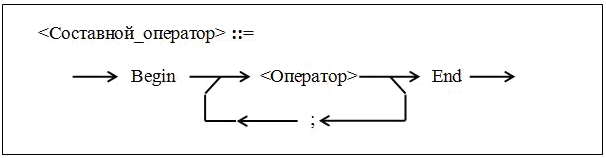


Рис 7.1 – Синтаксическая диаграмма **составного** оператора

**Составной опер** – **объединения** нескольких операторов**,** которые должны выполняться последовательно в операторных скобках. Символ (**;**) - для разделения операторов, входящих в состав составного.

В составной оператор могут входить **основные и производные** операторы, в том числе **составные**.

Раздел операторов = **составной оператор.** Компилятор воспринимает его как 1 оператор. Поэтому его можно использовать везде, где можно использовать основной оператор.

**Пример 7.1.**

Примеры составных операторов.

1) **Begin X := 0 End.**

Здесь **составной** оператор содержит **1 основной** оператор **:=**.

2) **Begin**

**X := A / B;**

**Y := 2 \* X**

**End.** – составной оператор содержит **2 основных** оператора.

3) **Begin**

**A := Sin(X); -- основной**

**Begin I := 0; -- составной в составе составного.**

**J :=0**

**End;**

**End.**

Здесь с**оставной** оператор содержит 2 оператора

**7.2. Программирование линейных   
и разветвляющихся структур алгоритмов**

**Линейная программа** - операторы выполняются как записаны (по порядку).

**Разветвляющаяся программа** – состоит из **линейных участков и точек ветвления** (**выбор** линейного участка): порядок выполнения операторов может не совпадать с порядком их записи в проге. Для управления этим порядком – 1.операторы перехода (Goto) и 2.выбирающие операторы.

7.2.1. Оператор (***безусловного***) перехода Goto

– для безусловной передачи управления помеченному оператору.

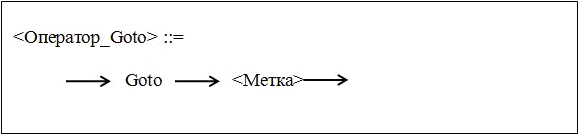


Рис 7.2 – Синтаксическая диаграмма оператора Goto

Опер *Goto* указывает какой следующий оператор должен выполняться.

О***граничения*** 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. запрещен переход из одного выбора в другой в выбирающих операторах (***If, Case*);**
3. с *Goto* нельзя входить в **подпрограмму** или выходить из нее.

**Пример 7.2.**

Пример абстрактной программы, использующей оператор *Goto*.

*--------------------------------*

**Label**  {- раздел меток}

**L1, L2, L3, L4;**

**--------------------------------**

**Begin**

**Goto L3;**{**неправильно**: переход внутрь составного оператора}

**Begin**

**L3: Goto L4;**{правильно}

**Goto L3 {правильно}**

**End;**

**L4: If A > B Then**

**L1: Goto L2** {**неправ-но:** переход между альтернативами If}

**Else**

**L2: Goto L1**{**неправ-но:** переход между альтернативами If}

*End;*

Наличие оператора *Goto* делает программу **ненаглядной**, трудночитаемой. Программа с *Goto* - не структурированная. использовать операторы *Goto* минимально, а лучше не использовать.

7.2.2. Условный оператор If ***(оператором условного перехода)***

- производный оператор. Относится к группе ***выбирающих*** *операторов*. в **разветвляющихся** программах **для выбора** участка вычислений в зависимости от условия.

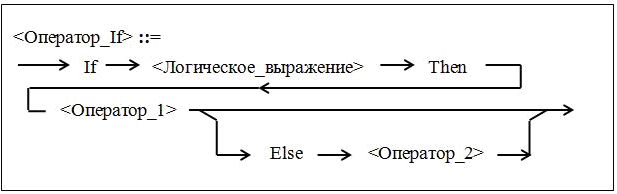


Рис 7.3 – Синтаксическая диаграмма оператора If

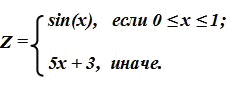
если выражение истинно, выполняется<Оператор\_1>, иначе следующий оператор. Конструкция *Else <Оператор\_2>* необязательна. Конструкция *If* без *Else* называется ***сокращенной формой оператора If***, с Else – **полной формой**.

Смысл оператора: если логическое выражение истинно (***True***), то выполняется <Оператор\_1>, иначе – <Оператор\_2>.

После выполнения любого из <Операторов\_1 или\_2> следующим выполняется оператор, записанный за оператором *If*.

В операторе *If* после *Then* и *Else* м записать **только 1 оператор**. Если надо записать группу операторов, используют операторные скобки *Begin … End*, т.е. **составной оператор.**

**Пример 7.3.** Вычислить



Решение:

**If (X >= 0) And (X <= 1) Then**

**Z := sin(X)**

**Else**

**Z := 5 \* X + 3;**

**Writeln (Z);**

Независимо от того, выполняется ветвь *Then* или *Else*, следующим будет выполняться оператор *Writeln (*в этом примере*)*.

Следующий выполняемый оператор может быть указан и в самом операторе *If* с помощью оператора *Goto*:

**Пример 7.4.**

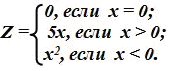
Указание в операторе If следующего выполняемого оператора.

**If X \* 2 + Y \* 2 > sin(Z)**

**Then Goto M1**

**Else Goto M2;**

Можно использовать *вложеный оператор If -*  записанный после Then или Else, также может быть **условным.**

**Пример 7.5.**

Вычислить:

Решение:

**If X = 0 Then Z := 0**

**Else**

**If X > 0 Then**

**Z := 5 \* X**

**Else Z := X \* X;**

Здесь 2 полных формы оператора *If*.

Если при записи вложенного оператора **If** используется его сокращенная форма, то служебное слово ***Else* относится к *последнему***оператору***If*:**

**Пример 7.6.**

Использование сокращенной формы оператора **If** во вложенном операторе.

*If A <> B*

*Then*

***If***  *C <> D Then*

*E := False*

***Else*** *E := True;*

эта конструкция эквивалентна записи:

*If A <> B Then*

*Begin*

*If C <> D Then*

*E := False*

*Else E := True*

*End;*

чтобы избежать ошибок использования сокращенной формы во вложенном операторе *If*, надо выделять форму *If* с помощью ***Begin End*:**

**Пример 7.7.** Как противоположность примеру 7.6.

**If A <> B Then**

**Begin**

**If C <> D Then**

**E := False**

**End;**

**Else**

**E := True;**

Здесь ветвь **Else** уже относится к первому оператору **If.**

**Пример 7.8.**

Ввести переменные ***А1, А2, А3*** и упорядочить их по возрастанию.

**Program Primer;**

**Var**

**A1, A2, A3: Real;**

**Vs: Real;** {*Vs – вспомогательная переменная*}

*Begin*

*Readln (A1, A2, A3);* {*Чтение исходных значений*}

**Writeln (A1, A2, 3);**

**If A1 > A2 Then**

**Begin**

**Vs := A1;**{*В А1 помещается наименьший из А1, А2}*

**A1 := A2;**

**A2 := Vs** {Vs – для перемены А1 и А2 местами. 3 стакана}

**End;**

**If A1 > A3 Then**

**Begin**

**Vs := A1;**  *{В А1 помещается наименьший из А1, А3}*

**A1 := A3;**

**A3 := Vs**

**End;**

**If A2 > A3 Then**

**Begin**

**Vs:= A2;**{ В А2 помещается наименьший из А3, А2}

**A2 := A3;**

**A3 := Vs**

**End;**

**Writeln (A1, A2, A3)**{Вывод результата}

*End****.***

7.2.3. Оператор варианта (выбора) Case

- производный оператор. Из группы ***выбирающих операторов*.**

Для разветвляющихся программах, если процесс нужно разветвить более чем по **2м** (возможным) направлениям. Является **обобщением** опера ***If*.**

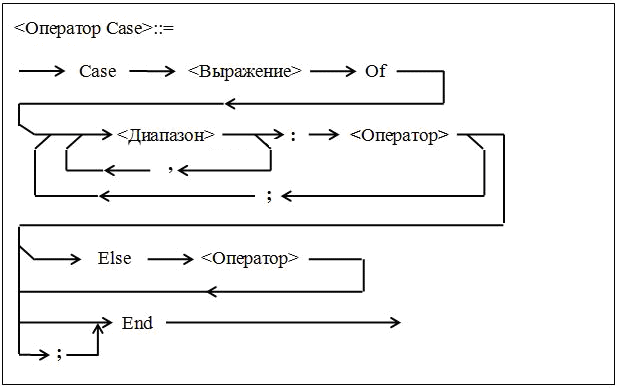


Рис 7.4 – Синтаксическая диаграмма оператора Case

Опер *Case* выбирает для исполнения 1 из операторов, входящих в него (в зависимости от значения **выражения** - **селектора *(переключателя)***, расположенного после слова *Case)*. Селектор может иметь любой перенумерованный тип, кроме типов ***Longint, Word*** и строкового типа String.

Каждый опер помечен **списком: 1) констант** выбора и/или 2) **диапазонов констант** выбора(от .. до: 1.. 10):

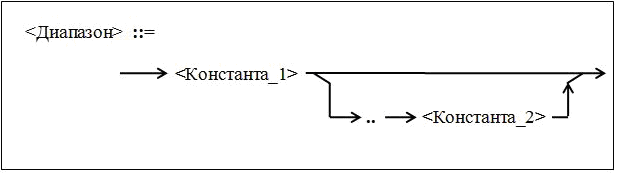


Рис 7.5 – Синтаксическая диаграмма **диапазона**

Последний оператор может быть помечен **словом *Else***. Если несколько констант выбора относятся к 1 и тому же оперу, то они разделяются запятой (,). Const-ы выбора должны иметь тот же тип, что и селектор.

Оператор *Case* выполняется так:

1) вычисляется значение селектора.

2) выполняется тот оператор, для которого значение константы выбора совпадает со значением селектора. Если нет совпадений, то выполнится опер, помеченный словом *Else*.

3) Конец оператора *Case (End)*.

Если значению селектора не соответствует ни один из вариантов (ветвь *Else* отсутствует), то выполняется оператор, следующий за *Case*.

**Пример 7.9.**

Примеры использования **селекторов** разных типов.

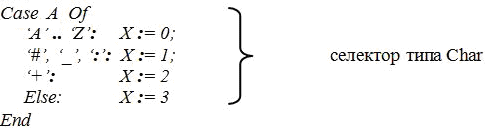
**Case 2 \* I – K Of {Cелектор целочисленного типа}**

**–100 .. –1: Writeln(‘<0’);**

**0: Writeln (‘=0’);**

**1 .. 100: Writeln(‘>0’)**

**End**



**Константы выбора** **метками** не являются. Поэтому они не описываются в разделе меток и не могут использоваться в **операторе перехода.** Например, в примере 7.9 нельзя написать *Goto 1*.

7.2.4. Пустой оператор

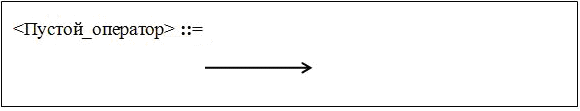
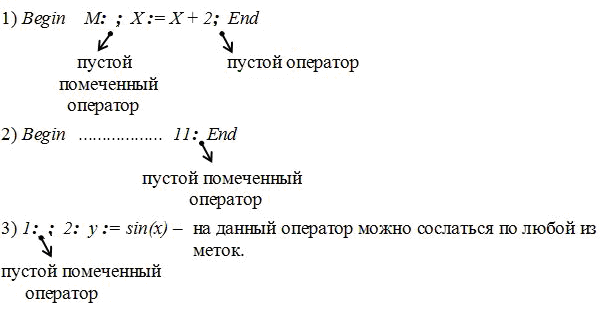


Рис 7.6 – Формат пустого оператора

- не задает **никаких действий** и не оказывает **никакого влияния** на программу. может быть использован везде, где по синтаксису используется понятие <Оператор>.

Используется для: **1)** упростить работу программиста с синтаксисом: позволяет свободнее оперировать с разделителем операторов ( **;** ); 2) **размещения метки в программе**, не соотнося ее с некоторым оператором; 3) организации обращения к 1 оператору по нескольким меткам; 4) в операторе ***If*** для устранения двусмысленности, связанной с использованием его вложенной сокращенной формы.

**Пример 7.10.** Примеры использования пустого оператора.



**Пример 7.11.** использование пустого оператора в операторе *If*.



**7.3. Программирование циклических структур алгоритмов**

**Циклическая программа** – частный случай разветвляющейся программы, в которой 1 из ветвей многократно повторяется. повторяющиеся ветви + управляющие повторениями оперы - **тело цикла**

М сделать циклы операторами ***If*** и ***Goto*.** Но это неэффективно. операторы ***For, Repeat, While*** *–*  для организации циклов.

7.3.1. Оператор цикла с параметром (оператор For)

- для циклов **с известным** числом повторений.

***Параметр цикла*** – переменная проверки условия цикла.

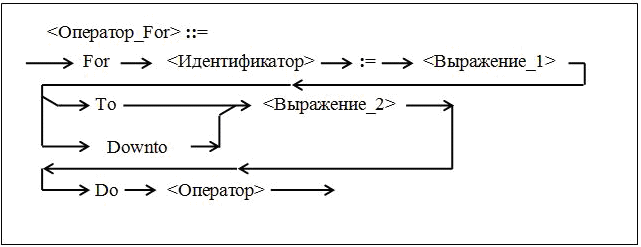


Рис 7.7 – Синтаксическая диаграмма оператора For

***For*** – для, ***To*** – до в сторону увеличения, ***Downto*** – до в сторону уменьшения, ***Do*** – делать.

<Идентификатор> – **имя** параметра цикла: переменная любого **скалярного** типа, кроме вещественного (т. е переменная **перенумерованного типа**).

<Выражения\_1 и 2> – выражения того же типа, что и параметр цикла. <Выражения> вычисляются один раз перед первым выполнением тела цикла.

<Оператор> - тело цикла. задается только 1 оператор. Если в цикле надо сделать последовательность действий, используют **составной опер**.

<Выражение\_1> - начальное значение параметра цикла.

<Выражение\_2> – конечное значение параметра цикла.

Если используют ***To*** – значение параметра возрастает автоматически при каждом прогоне цикла (***после*** выполнения тела цикла).

Возрастающее значение параметра вычисляется автоматически с помощью встроенной функции **Succ,** т.е. параметр принимает **следующее** значение типа параметра. Здесь для выполнения тела цикла надо, чтобы значение <Выражения\_1> было **<=** <Выражения\_2>. Иначе тело цикла не выполнится **ни разу**.

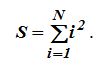
Если ***Downto* -** значение параметра **автоматически убывает** при каждом прогоне цикла (***после***) с помощью встроенной функции **Pred**, т.е. параметр цикла принимает **предыдущее** значение типа параметра. Здесь надо, чтобы значение <Выражения\_1> **>=** <Выражения\_2>. Иначе тело цикла не выполнится **ни разу.**

***Перед*** каждым выполнением тела цикла текущее значение параметра сравнивают с его **предельным** значением - <Выражение\_2>. Если текущее значение <= (в случае *To*) предельного или >= (*Downto*) предельного, тело цикла выполняется.

***Операторы тела цикла не должны изменять значение параметра цикла.***

При завершении цикла значение параметра цикла = конечному значению, увеличенному на 1, иначе – не определено.

**Пример 7.12.** Цикл с известным количеством повторений.

Вычислить для ***N = 10*** 

**Program S1;**

**Const**

**N = 10;**

**Var**

**S, I: Integer;**

**Begin**

**S := 0;**

**For I := 1 To N Do**

**S := S + I \* I;**

**Writeln (S)** //S = 385

**End.**

**Пример 7.13.** как в примере 7.12, но с *Downto* в операторе *For*.

Program S2;

Const

N = 10;

Var

S, I: Integer;

Begin

S := 0;

**For I := N Downto 1 Do**

S := S + I \* I;

Writeln (S)

End**.**

7.3.2. Оператор цикла с постусловием

- для программирования цикла с заранее **неизвестным** числом повторений, окончание зависит от какого-либо условия.

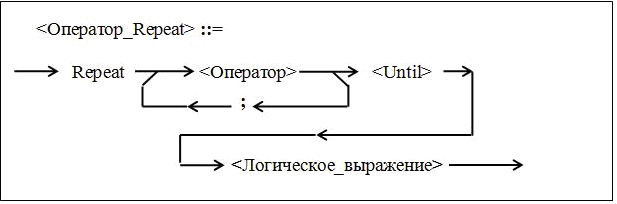


Рис 7.8 – Синтаксическая диаграмма оператора цикла **с пост**условием

*Repeat* – повторять,

*Until* – до.

Тело цикла - операторы между *Repeat* и *Until*.

*Тело цикла выполняется до тех пор, пока <Логическое\_выражение> принимает значение False*. Когда <Логическое\_выражение> станет равным True, выполнение цикла прекратится. Проверка условия - каждый раз ***после*** выполнения тела цикла. 🡺 тело цикла будет выполнено **хотя бы 1 раз.**

f155**Пример 7.14.**

Вычислить значение функции

через разложение функции в бесконечный ряд

f156

с точностью ***Eps = 0,0001***.

Алгоритм решения рассмотрен в [п. 2.3.3](#_2.3.3._Циклический_вычислительный_п) (см. пример 2.5).

**Program Sin1;**

**Const**

**eps = 0.0001;**

**Var**

**y, x: real;**

**n: integer;**  *{*вспомогат-ая перем-ая д/вычисления знаменателя}

**vs: real;** {вспомог-ая **ячейка** д/выч-ия очередного слагаемого}

**Begin**

**Readln (x);**

**y := x;**{Начальные установки}

**n := 2;**

**vs := x;**

**Repeat**

**vs := –vs \* x \* x / (2 \* n – 1) / (2 \* n - 2);** {Формирование слагаемого}

**n := n + 1;**

**y := y + vs**

**Until abs(vs) < eps;** { Цикл выполняется до первого выполнения условия *abs(vs) < eps*. }

**Writeln (x, y, eps)**

**End.**

С помощью оператора *Repeat* можно программировать и циклы с **заданным** числом повторений.

 **Пример 7.15.**

Вычислить для ***N = 10***

Program S3;

Const

N = 10;

Var

S, I: Integer;

Begin

S := 0;

I := 1; {Установка начального значения I}

**Repeat**

**S := S + I \* I;**

**I := I + 1** {Самостоятельное изменение параметра I}

***Until I > N;* {Выход из цикла по выполнению условия}**

*Writeln (S)*

*End.*

Сравните программу примера 7.15 с примерами 7.13, 7.14. В 7.15, по сравнению с использованием оператора *For*, самим нужно наращивать значение параметра *I* и устанавливать его в исходное значение.

7.3.3. Оператор цикла с предусловием (While)

- для программирования цикла с заранее **неизвестным** числом повторений когда тело цикла может не выполниться **ни разу.**

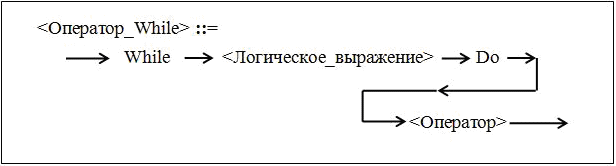


Рис 7.9 – Синтаксическая диаграмма оператора цикла с предусловием

*While* – пока, *Do* – делать.

<Оператор> определяет тело цикла.

***Перед*** каждым его выполнением вычисляется значение <Логического\_выражения>. *Если = True, то тело цикла выполняется и* прекращается, когда <Логическое\_выражение> впервые станет = *False*. Если к началу выполнения цикла значение < Логического выражения> = *False*, то тело цикла **не выполняется ни разу.**

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

Оператор *While* - более универсальный из операторов цикла. Его можно использовать вместо *For* и *Repeat*, но это не всегда удобно.

**Пример 7.16.** Вычислить значение функции

f158 через разложение функции в бесконечный ряд

f158

с точностью ***Eps = 0,0001***. Алгоритм – в п. [2.3.3](#_2.3.3._Циклический_вычислительный_п) (см. [пр 2.5](#Пример_2_5)).

Program Sin2;

Const

eps = 0.0001;

Var

y, x,: real; n: integer; vs: real;

Begin

Readln (x);

y := x;

n := 2;

vs := x;

**While abs(vs) >= eps Do** {Вход в цикл по выполнению условия}

**Begin**

vs := –vs \* x \* x / (2 \* n – 1) / (2 \* n – 2);

n := n + 1;

y := y + vs

**End;**

Writeln (x, y, eps);

End.

С помощью оператора *While* можно программировать и циклы с заданным числом повторений.

**Пример 7.17.**

Вычислить для ***N = 10*** f159

Program S4;

**Const**

**N = 10;**

**Var**

**S, I: Integer;**

**Begin**

**S := 0;**

**I := 1;**{Установка начального значения I}

***While I <= N Do*** {Вход в цикл по выполнению условия}

***Begin***

**S := S + I \* I;**

**I := I + 1**{Самостоятельное изменение параметра I}

***End****;*

*Writeln (S)*

*End.*

7.3.4. Операторы Continue и Brake

- для гибкого управления операторами циклов ***For, While, Repeat***. используется, если тело цикла или его часть **выполнять не надо.** Оператор **Continue** передает управление **на конец т**ела цикла, после чего выполняется анализ условия дальнейшего выполнения цикла.

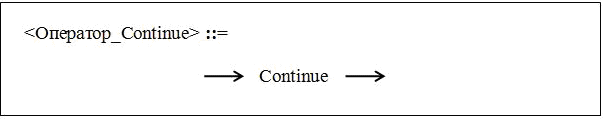
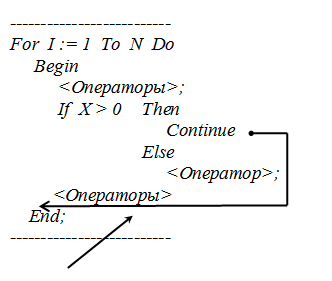


Рис 7.10 – Синтаксическая диаграмма оператора Continue

**Пример 7.18.**

Фрагмент программы, использующей оператор *Continue*.



По этой ветви - выход на конец тела цикла, затем изменяется значение параметра цикла ***I*** и выполнение тела цикла при следующем его значении (***I***).

**Оператор Break** *–*  безусловный **выход** из опер-ов *For, Repeat* и *While*

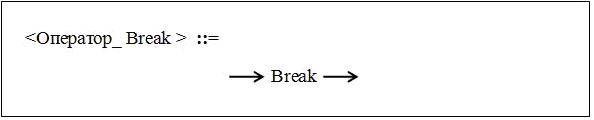


Рис 7.11 – Синтаксическая диаграмма оператора Break

Если в примере 7.18 вместо оператора ***Continue*** записать ***Break*,** то по отмеченной ветви будет **выход из цикла**: цикл **не будет** выполняться при следующих значениях параметра, a будет передача управления оператору, следующему за оператором цикла.

Использование оперов ***Continue*** и ***Break*** позволяет **обходиться без *Goto***.

**Раздел 8. Структурирование   
и оформление программ**

Паскаль - **структурированный язык**, т.е. позволяет писать структурированные программы.

*Структурирование программ* достигается за счет ***факторов***:

1. разбиения программы на **отдельные** **разделы** (заголовок, раздел описаний, раздел операторов);
2. наличия **операторов** языка, реализующих конструкции структурного программирования: :=, if, while

— оператор присваивания реализует функциональный блок;

— условный оператор реализует конструкцию принятия двоичного решения;

— оператор цикла с предусловием (*While*) реализует конструкцию обобщенного цикла «Пока»;

1. наличия составного оператора ***Begin ... End*** и синтаксиса условного оператора и оператора цикла (использование в них группы операторов за счет объединения их в операторные скобки), что позволяет реализовывать преобразования **Бома-Джакопини;**
2. дополнительных конструкций организации **цикла**: цикл **«До» (*Repeat ...*** *Until*); цикл с параметром (оператором **For**);
3. наличия **подпрограмм** (в виде процедур и функций)

Паскаль удобен для реализации *метода* ***нисходящего проектирования*** *программ* и, в частности, **метода пошаговой детализации (декомпозиции).**

Есть методологии **пошагового проектирования без** предварительной разработки схемы **алгоритма:**

1. на каждом этапе используют управляющие структуры и зарезервированные слова, а правила обработки данных не детализируют - описывают их в виде **комментариев.**

2. блоки **комментариев** частично детализируют, но сами комментарии не выбрасывают и т.д. В результате, после окончания проектирования, получается хорошо прокомментированная программа.

**Комментарии** делят на 3 *вида*:

1. **«вводные»** комментарии – помещают в **начале программы** и задают общую информацию о программе (1. назначение программы, 2.сведения об авторе, 3. дата написания, 4. метод решения, 5. время выполнения, 6. требуемый объем памяти и т.п.).
2. **«заголовки»** – объясняют назначения основных **блоков** программы на отдельных этапах пошаговой детализации;
3. **«построчные»** комментарии – описывают **мелкие фрагменты** программы;

Комментарии - эффективное средство облегчения понимания, тестирования, отладки и сопровождения программ. Отсутствие комментариев - дилетантский подход к программированию. + для наглядности программы, для отражения вложенности управляющих структур друг в друга - *расположение текста отдельных операторов ( табуляция программы (отступы))*.

***Общее правило записи программы***: служебные слова, которыми начинается и заканчивается оператор, записывать на одной вертикали; все **вложенные** в него операторы - с отступом вправо. Писать по одному оператору в строке.

**Пример 8.1.**

Здесь - упрощенный вариант **метода нисходящего проектирования** для задачи, рассмотренной при изучении операторов цикла *While* и *Repeat* (см. примеры 7.14, 7.16).

Вычислить значение функции

f163 через разложение функции в бесконечный ряд

f163

с точностью ***Eps = 0,0001***.

Алгоритм решения данной задачи рассмотрен в [п. 2.3.3](#_2.3.3._Циклический_вычислительный_п) (см. [пример 2.5](#Пример_2_5)).

**1-ый этап нисходящего проектирования** - вводный комментарий и укрупненная структура программы (в виде комментариев):

*{Вычисление значения y = sin(x) с заданной точностью с помощью разложения функции в ряд. Разработчик – Иванов А. А. 20.2.02 г.}*

*{Заголовок программы}*

*{Описания}*

*{Вычисления}*

**2-ой этап** - оставляем предыдущие комментарии и вводим служебные слова Паскаль + Вводим новые комментарии.

*{Вычисление значения y = sin(x) с заданной точностью с помощью разложения функции в ряд. Разработчик – Иванов А. А. 20.2.02 г.}*

*{Заголовок программы}*

*Program SN1;*

*{Описания}*

*{Вычисления}*

*Begin*

***{Чтение исходных данных}***

***{Присвоение начальных значений}***

*While* ***{Очередное слагаемое большей точности}***

*Do* ***{Вычисление слагаемого и суммы ряда}***

***{Печать результатов}***

*End.*

**3-ий этап -** Раздел «Описания» не изменяем. Детализируем **«Вычисления».** Вводим служебные слова + новые комментарии.

*{Вычисление значения y = sin(x) с заданной точностью с помощью разложения функции в ряд. Разработчик – Иванов А. А. 20.2.02 г.}*

*{Заголовок программы}*

*Program SN1;*

*{Описания}*

*{Вычисления}*

*Begin*

***{Чтение исходных данных}***

*Readln (X, Eps);*

***{Присвоение начальных значений}***

*Y := X; {Y – сумма ряда}*

*N := 2; {N – вспомогат-ная переменная д/вычис-ия знаменателя}*

*VS := X; {VS – очередное слагаемое}*

*While Abs(VS) >= Eps {Очередное слагаемое > точности}*

*Do {Вычисление слагаемого и суммы ряда}*

*Begin*

*VS := –VS\*X\*X/(2\*N–1)/(2\*N–2); {Вычисление слагаемого}*

*N := N + 1;*

*Y := Y + VS {Вычисление суммы}*

*End;*

*{Печать результатов}*

*Writeln (X, Y, Eps)*

*End.*

Комментарии-заголовки писать **перед** соответствующим блоком программы. Построчные комментарии - **за** оператором.

**4-ый этап -** детализируем раздел **«Описания».** Вводим описания элементов программы, используемых в разделе «Вычисления».

*{Вычисление значения y = sin(x) с заданной точностью с помощью разложения функции в ряд. Разработчик – Иванов А. А. 20.2.02г.}*

*{Заголовок программы}*

*Program SN1;*

*{Описания}*

*Var*

*X, Y, Eps, VS: Real;*

*N: Integer;*

*{Вычисления}*

*Begin*

*{Чтение исходных данных}*

*Readln (X, Eps);*

*{Присвоение начальных значений}*

*Y := X; {Y – сумма ряда}*

*N := 2; {N – вспомогат-ая переменная для выч-ия знаменателя}*

*VS := X; {VS – очередное слагаемое}*

*While Abs(VS) >= Eps {Очередное слагаемое > точности}*

*Do {Вычисление слагаемого и суммы ряда}*

*Begin*

*VS := –VS\*X\*X/(2\*N–1)/(2\*N–2); {Вычисление слагаемого}*

*N := N + 1;*

*Y := Y + VS {Вычисление суммы}*

*End;*

*{Печать результатов}*

*Writeln (X, Y, Eps)*

*End.*

Данный пример иллюстрирует проектирование программы с использованием **метода пошаговой детализации** **без** предварительной разработки схемы алгоритма, правила комментирования программы и правила взаимного расположения исходного текста программы.

**Раздел 9. Описанные скалярные типы**

**9.1. Перечислимый скалярный тип (ПСТ)**

- не является стандартным для Паскаль. должен быть описан программистом, поэтому относится к описанным скалярным типам. Перечислимый скалярный тип определяется заданием типа:

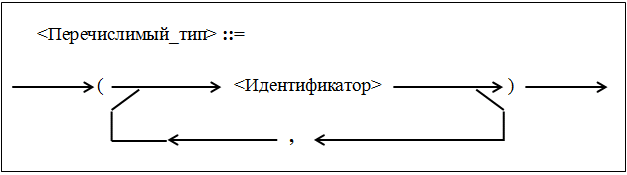


Рис 9.1 – Синтаксическая диаграмма задания перечислимого типа

ПСТ задается **указанием** всех значений, которые может принимать переменная этого типа. Каждое значение – идентификатор и считается **самоопределенной константой** (**константой определяемого типа**). Константы в списке перечисления считаются **перенумерованными** с 0, в порядке их перечисления: 0, 1, 2, 3…. (т.е. считаются **упорядоченными**: 1ое значение < 2го, 2е < 3го и т.д.).

2 определения ПСТ не должны содержать констант с одинаковыми именами в одной и той же области действия.

В памяти занимает **1 байт**.

Пример перечислимого типа – тип *Ned* (Неделя) (см. [пример 6.4](#Пример_6_4) в [подразд.6.4](#_6.4._Раздел_типов)). Данный тип может быть задан так:

**(Pn, Vt, Sr, Ch, Pt, Sb, Vs)**

2 *способа* задания типа данных: ***явно* в** разделе описания типов **Type** или ***неявно*** в разделе описания переменных **Var** (**см. подразд.** [**6.4**](#_6.4._Раздел_типов)**,** [**6.5**](#_6.5._Раздел_переменных)**).**

**Идентичные типы *-*** типы 2х переменных, если в их объявлении используют 1 идентификатор типа или используют разные идентификаторы ***Т1*** и ***Т2***, для которых объявление вида имеет вид:

**Type**

**T1 = T2;**

Идентичные типы - **совместимые**, т.е. переменные идентичных типов могут участвовать в одном выражении.

Если идентичные типы - не файловые типы, то они **совместимы по присваиванию**: значение одного типа может быть присвоено переменной другого типа.

**Пример 9.1.** Раздел типов (**явное задание типов**).

**Type**

**Ned = (Pn, Vt, Sr, Ch, Pt, Sb, Vs);**

**God = (Yan, Fev, Mar, Apr, May, Iyn, Iyl, Avg, Sen, Oct, Nob, Dec);**

**Citr = (Mandarin, Apelsin, Limon);**

**Fruct = Citr;**

Здесь ***Fruct*** и ***Citr*** – **идентичные типы**.

Одно и то же **имя константы** нельзя использовать как значение в разных типах. Поэтому в примере 9.1 нельзя написать:

*Fruct = (Mandarin, Apelsin, Limon);*

типы *Fruct* и *Citr* трактуются как **разные типы,** а в них нельзя использовать одни и те же имена констант.

+ нельзя объявить и такой тип:

*Desert = (Ananas, Vinograd, Grusa, Sliva, Apelsin);*

При объявлении типа *Desert* используется **константа** *Apelsin*. Однако она уже применена в другом типе (*Citr*).

Напомним: при **явном** задании типа в разделе ***Type*** переменные данного типа должны быть объявлены в разделе ***Var*** с помощью указания **имени** типа.

**Пример 9.2.**

Задание переменных при **явном** задании типов. Типы заданы в прим 9.1.

**Var**

**Den, Den1: Ned;**

**Mes, Mes1: God;**

**C1, C2: Citr;**

**F1, F2: Fruct;**

Type

**Ned** = (Pn, Vt, Sr, Ch, Pt, Sb, Vs);

**God** = (Yan, Fev, Mar, Apr, May, Iyn, Iyl, Avg, Sen, Oct, Nob, Dec);

Citr = (Mandarin, Apelsin, Limon);

Fruct = Citr;

**Пример 9.3.**

**Неявное** задание типов. Данные типы **в разделе Type не задаются**.

**Var**

*Den, Den1****:*** *(Pn, Vt, Sr, Ch, Pt, Sb, Vs);*

*Mes, Mes1****:*** *(Yan, Fev, Mar, Apr, May, Iyn, Iyl, Avg, Sen, Oct, Nob, Dec);*

*C1, C2, F1, F2****:*** *(Mandarin, Apelsin, Limon);*

При **неявном** задании типов **нельзя задать идентичные типы, так как тип не имеет имени.** Таким образом, на него уже невозможно сослаться в других местах программы. Поэтому, **предпочтение –** **явному** заданию типа.

Над значениями **перечислимого** типа определены только ***операции*** ***сравнения***. При сравнении учитывается его **перенумерованность**. Значения ПСТ м б операндами оператора присваивания :=.

**Пример 9.4.**

Выполнение присваивания и операций сравнения (к примерам 9.1, 9.2).

**Den := Sr;**

**Den1 := Den;**

**Den1 := Pon;**

**If Den > Den1**

**Then ...**

**Else ...**

Над данными ПСТ определены ***встроенные функции:***

***Succ (x)*** – **следующ**ее значение за ***x***;

***Pred (x)*** – **предыду**щее значение перед ***x***;

***Ord (x)*** – **порядковый №** ***x (ордер)***;

***Sizeof (x)*** – **размер** **памяти**, занимаемый аргументом; для перечислимого типа результат = 1;

Значения функции *Pred* для первого аргумента в списке перечисления (например, *Pred (Pon)*), и функции *Succ* для последнего аргумента (*Succ (Vos)*), **не определены.**

Таким образом, если в операторе присваивания **слева** := записана переменная **перечислимого** типа, то в качестве выражения может быть только **константа или переменная перечислимого** типа либо функция соответствующего аргумента.

Для ввода и вывода значений ПСТ нельзя использовать стандартные процедуры ввода-вывода *Read, Readln, Write, Writeln,* а использовать оператор варианта ***Case*.**

**Пример 9.5.**

Вв и вывод значений перечислимого типа (см объявления в прим 9.1, 9.2).

**Var**

i: 1..7;

-----------------------------------

**Begin**

Read (i);

**Case** i **Of**

1: Den:= Pn;

2: Den:= Vt;

3: Den:= Sr;

4: Den:= Ch;

5: Den:= Pt;

6: Den:= Sb;

7: Den:= Vs

**End**;

**Case** Den **Of**

Pn: Writeln ('ПОНЕДЕЛЬНИК');

.........................

Sb: Writeln ('СУББОТА');

Vs: Writeln ('ВОСКРЕСЕНЬЕ')

**End**;

**9.2. Тип диапазон**

- называют еще ***ограниченным*** или ***интервальным типом*** данных*.*

- задается путем накладывания **ограничений** на уже заданный или стандартный тип, называемый **базовым типом** – могут быть использованы **скалярные** типы, имеющие свойство перенумерованности (т.е. все скалярные типы, кроме вещественных). Тип диапазон – ***перенумерованный*.** Порядковый номер элемента совпадает с порядковым номером этого элемента *в* ***базовом типе***.

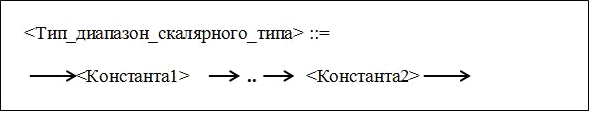


Рис 9.2 – Синтаксическая диаграмма задания типа диапазон

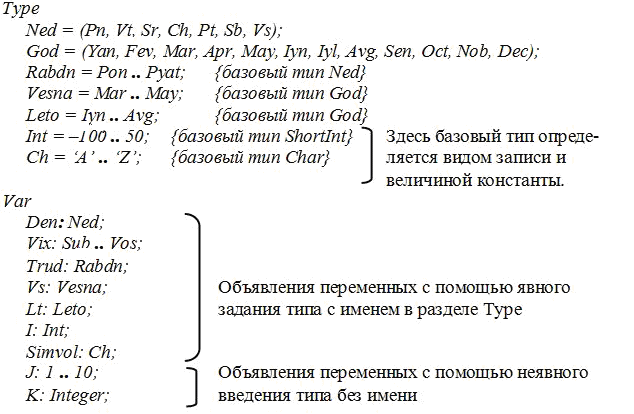
<Константа1>, <Константа2> – константы того базового типа, на основании которого вводится данный ограниченный тип;

<Константа1> задает **миним**альное значение типа диапазон. <Константа2> - **макс**имальное значение.

Значение <Константы1> д б <= значения <Константы2>.

Определяется: 1.в разделе типов, 2. описанием переменных в разделе *Var*.

**Пример 9.6. Объявление** типа диапазон (см пример 9.1).



Здесь типы *Rabdn, Vesna, Leto, Int, Ch* - типы диапазон. Переменные *Vix, Trud, Vs, Lt, I, Simvol, J, K* имеют тип диапазон.

Т о, множество значений **типа диапазон** принадлежит множеству значений **базового типа**. К значениям типа диапазон применимы **все операции и функции**, определенные над значениями **базового** типа.

Тип диапазон совместим со своим базовым типом, т.е. значения типа диапазон могут использоваться везде, где могут использоваться значения базового типа.

В одном и том же выражении могут быть использованы переменные как типа диапазон, так и **соответствующего ему базового типа.**

Тип диапазон **совместим с** базовым типом **и по присваиванию**, если значение, присваиваемое переменной ограниченного типа, принадлежит соответствующему диапазону.

Моё мнение о базовом типе: это всё поле возможных вариантов, а из него м б выделено несколько представителей.

**Пример 9.7.**

**Совместимость** типа диапазон с базовым (см предыдущий пример).

**Den := Sub;**

**Trud := Pred (Den);**{Trud = **Pyat**, это входит в диапазонRabdn}

**Vix := Sub;**

**K := Ord (Vix);**{К станет =5 (как в базовом типе)}

Но если исходное значение **Den := Vos**, то оператор

*Trud := Pred (Den) - не Pred, а Succ* вызовет ошибку.

Если *Trud := Pyat*, то оператор *Den := Succ (Trud*) б = *Sub*, что входит в диапазон значений переменной *Den*.

*«+» типа диапазон:*

1. транслятору **экономнее использовать память** при представлении значений переменных;

2. **контроль** как на этапе трансляции, так и во время выполнения программы **за корректностью присваиваний**: помогает исправлять ошибки в программе;

3. наглядная форма представления задачи.

Тип диапазон применяется в комплексе с **производным**и типами, в частности, в обработке массивов.

**Скалярный тип** – состоит из 1го элемента данных.

**Раздел 10. Структурные типы**

**10.1. Массивы**

10.1.1. Задание массивов

Все описанные в предыдущих разделах типы - **скалярные, т.е. состоящие из одного элемента данных.**

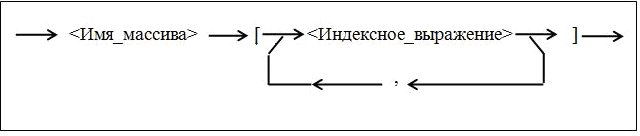
**Производные (структурные) типы *-*** состоят **из нескольких** элементов данных, Каждый из которых в свою очередь также может быть **структурой**.

Т о, значения **производных** типов имеют **иерархическую структуру:** на самом нижнем уровне - **отдельные** данные. Компонентам нижнего уровня м присваивать значения, и они могут присутствовать в выражениях, аналогично скалярным переменным.

**Массив – упорядоченная совокупность однотипных элементов, имеющих общее имя.** И это имя называется **полной переменной**, так как ее значение – весь массив.

**Базовый тип массива** = тип элементов массива.

Для выделения отдельных элементов массива - индексы (индексные выражения). ***Индекс*** задает правило вычисления **номера** элемента массива.

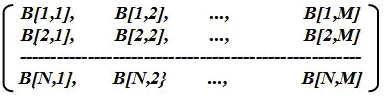
Рис 10.1 − Синтаксическая диаграмма указания элемента массива

**Индексное выражение** – выражение скалярного типа. Количество индексных выражений определяется количеством **измерений** массива.

1-мерный массив ***А*** - как **вектор**:

***A[1], A[2], A[3], ..., A[N],***

2мерный массив В – как **матрицу:**



**Количество** элементов массива, их упорядоченность и тип должны задаваться **явно** при описании массива до начала выполнения программы и **не могут изменяться** в процессе выполнения программы.

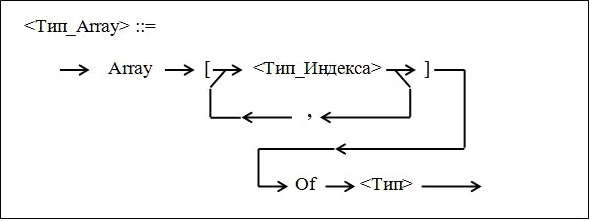


Рис 10.2 – Синтаксическая диаграмма задания типа массив

<Тип> – тип элементов массива (**базовый** тип). может быть задан как <Имя\_типа> или <Задание\_типа> (см. рис 6.10).

Множество значений <Типа\_индекса> д б **перенумерованным**. <Тип\_индекса> определяет **верхнюю и нижнюю** границу и значения, которые может принимать индексное выражение. <Типа\_индекса> могут б любые перенумерованные типы, кроме длинного целого и его поддиапазонов longint, shortint, integer.

**Пример 10.1.** Использование **типа диапазон** как<Типа\_индекса>.

**Type**

**Diapazon = 1 .. 20;**

{Явное задание типа: 1-мерный массив из 20

элементов вещественного типа}

**Vector = Array [Diapazon] Of Real;**

**Matritca = Array [–5 ... 40, Diapazon] Of 0 .. 100;** {- Явное

задание типа. 2мерный массив из 46 строк и 20 столбцов,

содержащий элементы ограниченного типа 1…100}

**Var**

**I: Diapazon;**

**A, B: Vector;**

**E, F: Matritca;**

**C, D: Array [1 ... 20] Of Char;**{- **Нея**вное задание типа.

1-мерный массив из 20 элементов типа Char}

**Mes: Array [1 .. 4] Of (Pn, Vt, Sr, Ch, Pt, Sb, Vs);** {- **Нея**вное

задание типа. 1-мерный массив из 4 элементов

перечислимого типа}

**Пример 10.2.** Использ-е перечисл-го как <Тип\_индекса>.

**Type**

**Fam = (Ivanov, Petrov, Sidorov, Andreev, Sergeev);**

{**Явное** задание типа. 1-мерный массив из 5 элементов типа диапазон:}

**Vozrast = Array [Fam] Of 18 .. 70;**

{Или так:

**Vozrast = Array[(Ivanov, Petrov, Sidorov, Andreev, Sergeev)] Of 18 .. 70;}**

**Var**

**Voz: Vozrast;**

**F: Fam;**

{**Неяв**ное задание типа. 1-мерный массив из 5 элементов типа диапазон:}

**Godrozd: Array [Fam] Of 1920 .. 1980;**

Переменную ***F*** можно использовать как **индекс** для выделения отдельных элементов массивов *Voz, Godrozd*:

*Voz [F], Godrozd [F].*

В зависимости от значения F выбираются конкретные элементы массива, например: Voz [Ivanov] Voz [Petrov]

**Пример 10.3.**

Использование **идентификаторов** скалярного типа как <Тип\_индекса>.

**Type**

{1-мерный массив из 2 элементов (true, false) **вещественного** типа:}

**Mas1 = Array [Boolean] Of Real;**

{Одномерный массив из 256 элементов **целого** типа:}

**Mas2 = Array [Char] Of Integer;**

**Var**

**M1: Mas1;**

**M2: Mas2;**

**J, А, В: Boolean;**

**Simv : Char;**

**Пример 10.4.**

Примеры **обращений к элементам** массивов (см прим 10.1 ÷ 10.3).

*A [1]* – 1-ый элемент массива ***А***.

*E [–1, 18]* – обращение к элементу с номером строки –1 и номером столбца 18 матрицы ***Е***.

*C [2 \* I – 5]* – использование **индексного выражения**.

**Voz [Andreev]** – обращение к элементу с **индексом** *Andreev* массива *Voz*.

*Voz [F]* – обращение к элементу с индексом *Andreev* массива *Voz*. К этому

моменту значение переменной ***F*** **уже должно быть определено**.

*Voz [Pred(F)]* – использование индексного **выражения.**

*M1[False]* – обращение к элементу с индексом *False* массива ***М1*** (это первый элемент массива ***М1***).

*M1[J]* – обращение к элементу с номером ***J*** массива ***М1***. ***J*** должно иметь тип *Boolean*.

*M1 [A | B]* – использование **индексного выражения** (***A, B*** – логические

переменные).

*M2 [‘Z’]* – обращение к элементу с индексом ***‘Z’*** массива ***М2***.

*M2 [Simv]* – обращение к элементу с индексом *Simv* массива ***М2***(*Simv* –

переменная типа *Char*).

При использов-и **индексных выражений** надо, чтобы их значения не выходили за пределы <Типа\_индекса> в объявлении массива.

Синтаксическая диаграмма задания типа *Array* (см. рис 10.2) - **сокращенная форма *задания многомерных массивов***.

**Полная форма *задания многомерных массивов*** образуется заданием массивов **меньшей** размерности. Например, 2мерный массив – матрица из ***N*** строк и ***M*** столбцов – может быть представлен как 1мерный массив из ***N*** элементов, а каждый элемент этого массива в свою очередь является 1мерным массивом, состоящим из ***M*** элементов.

Т о, 2мерный массив ***А*** целых чисел из **10 строк** и 20 столбцов, может быть объявлен так:

**Пример 10.5.** Способы объявления многомерных массивов.

**1-й способ:**

*Type*

*Dvummas =* **Array [1 .. 10] Of Array [1 .. 20] Of Integer;**

*Var*

*A: Dvummas;*

**2-й способ**: полная форма

*Type*

**Odnmas = Array [1 .. 20] Of Integer; /**/к-во столбцов в строке

**Dvummas = Array [1 .. 10] Of Odnmas;**

*Var*

*A: Dvummas;*

Из полной формы объявления массива очевидно, что к **элементу** ***A [I, J]*** двумерного массива ***А*** можно обращаться: ***A [I] [J]***. Такая запись трактуется как ***J***-ый элемент ***I***-ого элемента массива ***А***, т.е. элемент на пересечении ***I***-ой строки и ***J***-ого столбца.

🡺 Паскаль допускает переменные, обозначающие ***подмассивы***.

Запись ***A[I]*** для двумерного массива ***А*** будет означать ***I***-ую строку массива ***А***.

Для ***М***-мерного массива обращение к его элементу в соответствии с полной формой объявления массивов выглядит так:

***A [I1] [I2]...[IМ]***

запись ***A[I1]*** означает *(М – 1)*-мерный подмассив массива ***А.***

В многомерных массивах тип индекса не обязательно должен быть одним и тем же. **Индексы по каждому измерению** могут иметь **разные** типы.

**Пример 10.6.** Использование индексов **разного типа.**

*Type*

*Den = 1* ***..*** *31;*

*Mes = (Yan, Feb, Mar, Apr, Mai, Iyn, Iyl, Avg, Sen, Okt, Noi, Dec);*

*God = 1900* ***..*** *2000;*

*Data =* ***Array*** *[Den, Mes, God] Of (Pn, Vt, Sr, Ch, Pt, Sb, Vs);*

Тип *Data* м задать и так:

*Type*

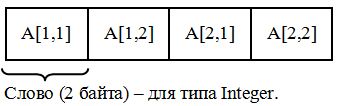
*Data = Array [1* ***..*** *31, (Yan, Feb, Mar, Apr, Mai, Iyn, Iyl, Avg, Sen, Okt, Noi, Dec), 1900* ***..*** *2000] Of (Pon, Vt, Sr, Cht, Pyat, Sub, Vos);*

Мерность (размерность) массива синтаксисом Паскаль ограничена только **объемом памяти**. *Максимально допустимый размер массива =* 65520 байт (размер сегмента **данные**).

Компилятор размещает массив в **сплошной** области памяти, отводя каждому элементу участок памяти в соответствии **с базовым** типом. Многомерные массивы располагаются **«по строкам»**, т.е. быстрее изменяется последний индекс. Например, двумерный массив ***А*** целых чисел (тип *Integer*):



Расположение данного двумерного массива в памяти:



10.1.2. Действия над элементами массивов

**Элементы массива** называются **индексированные переменные**. Для обращения к конкретному элементу массива - указать **имя** массива и **индекс** элемента в квадратных скобках [ ].

Элементы массива м использовать в выражениях как простые **переменные и вместе с** простыми переменными. Например,

***Sin (X [I + 5] ) \* X [I] + Y.***

***Y*** – простая переменная, ***X [I + 5], X [I]*** – элементы массива ***X***.

**Инициализация массива** – присвоение элементам массива **исходного** значения.

**Пример 10.7. Инициализация** двумерного массива.

*Type*

*Mas = Array [1* ***..*** *10, 1* ***..*** *20] Of Integer;*

*Var*

*J, I: Integer;*

*Mas1, Mas2: Mas;*

*----------------------------------*

*For I := 1 To 10 Do*

*For J := 1 To 20 Do*

*Mas1[I, J]* ***:= 0****;- все эл- ты =0*

**Пример 10.8.** ввод-вывод элементов массива (объявления – см пр 10.7).

----------------------------------

{**Ввод** исходных значений в массив:}

*For I := 1 To 10 Do*

*For J := 1 To 20 Do*

*Read (Mas1[I, J]);*

{**Вывод** значений элементов массива:}

*For I := 1 To 10 Do*

*For J := 1 To 20 Do*

*Writeln (Mas1[I, J]);*

В Паскале **нет** вв-выв всех элементов массива как 1 полной переменной**.**

**Пример 10.9.**

**Присвоение** значения **одного массива другому** м выполнить так:

**For I := 1 To 10 Do**

**For J := 1 To 20 Do**

**Mas2 [I, J] := Mas1 [I, J];**

Массив *Mas2* и *Mas1* должны быть объявлены (оба массива должны быть одного типа). Массив *Mas1* к этому моменту должен быть **определен (все элементы должны иметь значения**). Значения элементов *Mas1* после присваивания **не меняются**.

В 2х следующих пунктах описан другой **способ присвоения** значения 1го массива другому.

10.1.3. Действия над массивами

**Полные переменные** типа массив не могут использоваться как **операнды выражений**, их значения могут быть ***только*** в **операторах присваивания**, причем **полной переменной** может быть присвоено только значение полной переменной **того же типа**. Например,

***В := А;***

***А*** и ***В*** – массивы одного и того же типа.

К этому моменту массив ***А*** **уже** должен быть определен.

Это же и для **подмассивов**. Например, если ***А*** и ***В*** – двумерные массивы одного типа, то запись:

**B[I] := A[I];**

значит, что ***I***-ой строке массива ***В*** будет присвоено значение ***I***-ой строки массива ***А***.

10.1.4. Типизованные константы типа массив (ТКМ)

Для **инициализации** **м**ассива в Паскаль используют **типизованные константы** типа массив, которые объявляют в разделе констант.

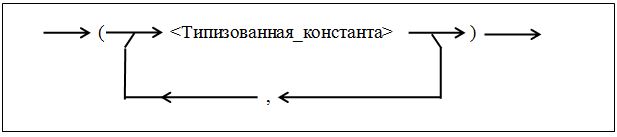


Рис 10.4 – Синтаксическая диаграмма задания типа массив

**Пример 10.10.** Объявление 1-мерной константы-массив.

**Type**

**Mas = Array [1..10] Of Integer;**

**Const**

{Для **присвоения** типа **типизованной константе-массив (ТКМ)** используют **имя** типа:}

**M:****Mas**= (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

{Для присваивания типа Т-ойКМ используют **задание** типа}

**Simv:****Array [1…5] Of** Char *= (‘A’, ‘B’, ‘C’, ‘D’, ‘E’);*

{или (эквивалентная форма записи):

**Simv:****Array [1…5] Of** Char *= (‘ABCDE’);*}

Здесь элементам М[1], M[2], **...**, M[10] константы-массива ***М*** присваивают значения 1, 2, **...**, 10 соответственно.

Аналогично элементам *Simv[1], Simv[2],* ***...****, Simv[10]* константы-массива *Simv* присваивают значения ‘A’, ‘B’, ‘C’, ‘D’, ‘E’.

При **объявлении** многомерной константы-массив учитывают, что задание типа

**Array [1..M, 1..N, 1..K] Of <Тип>**

(с учетом полной формы задания массивов) равносильно объявлению

***Array [1..M, 1..N] Of Array [1..K] Of <Тип>***

или

***Array [1..M] Of Array [1.. N] Of Array [1..K] Of <Тип>***

**Многомерный массив** состоит из массивов меньшей мерности.

Отсюда: при задании многомерной константы-массив как <Типизованной\_константы> (см. рис. 10.4) используют **константу-массив на единицу меньшей мерности**, чем текущая константа-массив. Массив каждой мерности заключают в круглые скобки ().

**Пример 10.11. Объявление многомерных** констант-массивов.

**Type**

**Mas3 = Array [1..2, 1..2, 1..2] Of Integer;**

**Const**

**M3: Mas3 = ( ((0,1), (2,3)), ((4,5), (6,7)) );**

Массиву ***М3*** будут присвоены начальные значения:

M3[1,1,1] = 0 M3[1,2,1] = 2 M3[2,1,1] = 4 M3[2,2,1] = 6

M3[1,1,2] =1 M3[1,2,2] = 3 M3[2,1,2] = 5 M3[2,2,2] = 7

Значения присваиваются в порядке размещения элементов массива в памяти машины. Наиболее быстро изменяется последний индекс.

Вывод значений структурной константы-массив – как все массивы – поэлементно в цикле. Вывод полной константы-массив сразу не предусмотрен.

С помощью **структурной** константы-массив м **инициализировать** массив.

**Пример 10.12.**

**Инициализация массива** с помощью структурной константы-массив

**Type**

**Mas = Array [1..10] Of Integer;**

**Mas3 = Array [1..2, 1..2, 1..2] Of Integer;**

**Const**

**M:****Mas**= (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

**M3: Mas3 = ( ((0,1), (2,3)), ((4,5), (6,7)) );**

**Var**

**M1: Mas;**

**MM: Mas3;**

**-----------------**

**Begin**

**M1 := M;**

**MM := M3;**

!: **типизованные константы** рассматривай как **инициализированные переменные**, т е они м **менять исходное значение** в ходе программы.

**10.2. Строковые данные String**

**Строка** – это последовательность **любых** символов кода обмена информацией (**код ASCII**).

Строка может содержать **русские буквы +** знаки алфавита языка Паскаль.

Строка хранится в поле **основной** памяти, состоящей из смежных байтов. Каждый байт содержит 1 символ. *Максимальная длина строки* – **255 байт** (допускаемая компилятором Паскаль)**.**

Различают строковые: константы и переменные.

10.2.1. Строковые константы ‘ …’

– последовательность любых символов, заключенная **в апострофы** (допускаемых для представления в компьютере). Например, **‘**TEKST**’**.

***Правила записи строковых констант*** в программе:

1). Если в строке нужен апостроф, то его повторяют дважды. При подсчете длины строки 2 рядом стоящих апострофа считаются одним символом. Например,

***‘****ПРОГРАММА* ***‘‘****ИНТЕГРАЛ****’’’*** *:* так должна быть написана строка ПРОГРАММА ‘ИНТЕГРАЛ’.

***‘****ЗНАК* ***‘‘****+****’’’*** *–* Так должна быть написана строка ЗНАК ‘+’.

2). При подсчете длины строки учитывают **пробелы**.

3). Допускаются **пустые символьные константы**, т.е. константы, не содержащие ни одного символа.

**‘’** – **пустая строка нулевой длины.**

4). м вставлять в строку символов управляющие символы. Символ **#** с целой константой без знака (от 0 до 255) означает соответствующий этому значению **символ в коде ASCII**. Между **#** и целой константой (и если несколько управляющих символов входят в строковую константу) – б/пробелов. Управляющие символы пишут **вне апострофа**. Например,

*‘TURBO’****#13#10****’TEKST’*

**- #13#10** – **управляющие символы**: **переход** к началу следующей строки

***#13#10*** *-* строковая конст-та состоит **только из управляющих** символов.

***#7#7****’MAKE’#7#7* **#7** – управляющий символ «Звонок».

10.2.2. Строковые переменные

**- 2** способа **задания строковых переменных**:

**1-ый способ:** Задает строковые переменные **постоянной длины**. Определяется как **1мерный массив символов (свойства массива)**:

**Array [1 .. N] Of Char**

<Тип\_индекса> может быть задан только с помощью **типа диапазон**, где ***N*** – длина строки (*N* **≥ 1**), - целое число без знака lenth.

Здесь строка имеет ***свойства массива****.* Например, строковой переменной можно присвоить значение другой строковой переменной **той же длины**; или обращаться к отдельным символам строковой переменной, используя индексные переменные.

**Пример 10.13.**

Работа состроковыми переменными постоянной длины. Использование **свойств строки** как **свойства массивов**.

---------------------------------------------------------

**Type**

**Stroka = Array [1 .. 9] Of Char;** {-**1мерный** массив символов

**- строка**}

{**2мерный** массив символов – **1-мерный массив строк**}

**Stranitca = Array [1 .. 30] Of Stroka;**

**Var**

**Stroka1, Stroka2: Stroka;**

**Stranitca1, Stranitca2: Stranitca;**

**I, K, J: 1..30;** *{I – строка, К – новый символ, кот присвоят*

*J-симв-у}*

**X, Y: Boolean;**

**Const**

**Literal = ‘Программа’;**

**Begin**

…………....... {K этому моменту Stroka2 должна быть определена :}

**Stroka1 := Stroka2;** {Одной строке м присвоить знач-ие др

строки **той же длины**.}

**Stranitca1[K] := Stroka2**; {К-ой строке страницы присваивается

значение строки}

………………………

{Обращение к отдельным символам строковой переменной:}

**Stroka1[I] := ‘A’;**

**Stroka1[J] := Stroka2[I];**

**Stranitca1[I, J] := Stroka1[K];** {J-ому **символу** I-ой строки

страницы присваивают значение К-ого **символа** строки}

***Особенности* строковых переменных *по сравнению с* массивами**:

**1)** Строковым переменным можно присвоить значения **строковых констант (литералов),** если длина строки = длине литерала.

**Пример 10.14.** Д/прим 10.13 м писать операторы присваивания:

**Stroka1 := Literal;**

**Stranitca[I] := Literal;**

**Stroka1 := ‘ПРОГРАММА’;**

**2)** Над значениями строковых переменных одинаковой длины можно выполнить операции сравнения (=, <>, >, <, >=, <=), которое производится посимвольно, начиная с левой стороны до первого несовпадающего символа. Считается **большей** та строка, в которой первый несовпадающий символ имеет больший № в **коде обмена информацией (ASCII).**

**Пример 10.15.**

См. пример 10.13: можно написать такие операторы присваивания:

**X := Stroka1 = Stroka2;**

Если **Stroka1** и **Stroka2** *=*ы, то ***X*** примет значение **True,** иначе – **False.**

**Y := Stranitca[I] >= Stroka2;**

**I-**ая с**трока** страни**цы** сравнивается со строкой (у них длины д. б. **=**ые).

Строковые переменные **разной длины** сравнивать **нельзя!**

**2-ой способ:**  использование типа **String:**

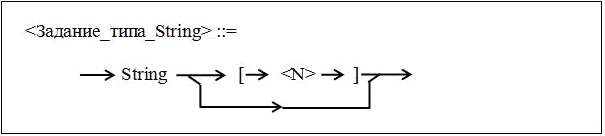


Рис 10.5 – Синтаксическая диаграмма задания типа String

С помощью типа *String* определяют строки **переменной длины**. ***N*** после слова *String =* **максимальная длина строки (255)**.

1≥ ***N*** <­­­­= 255 ***–*** положительное **цело**численное значение.

Если ***N*** в определении опущено (по умолчанию), то максимальная длина строки равна **255 = 28 – 1.**

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

Если длина присваиваемого значения > указанной в объявлении максимальной длины, то символы справа **отсекают**.

Переменной типа *String* выделяют памяти на **1** байт **>** максимальной длины, указанной в определении типа. В левом байте (с № 0) хранится текущая длина строки в двоичном коде.

Есть доступ к отдельным символам строки типа **String:** используют **индексные переменные.** Правила индексации аналогичны массиву символов с диапазоном индексов:

**1 … <Текущая\_длина\_строки>.**

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.

**Строковое выражение *-*** выражение, результатом вычисления которого является **строковое значение.** состоят из: 1) строковых **констант**, 2) строковых **переменных**, 3) имен **ф-ций** и 4) **знаков** операций.

Над данными типа *S***tring** определены **операции:** 1) **сравнения** (=, <>, >, <, >=, <=) Сравнение – в соответствии с **упорядочением символов** в коде **ASCII.** Сравниваются символы строк последовательно слева направо **до первого несовпадающего символа: б**ольшая та строка, у которой он имеет больший код в таблице **ASCII.** Если строки имеют разную длину и их символы совпадают в общей части, то **более короткая** строка - меньшая. Строки **равные**, если одинаковая длина и одинаковые символы

и 2) **конкатенации** (сцепления, обозначается символом **+**.) – имеет более высокий **приоритет** (чем сравнения): 2 строки соединяют в 1 результирующую, которая д. б. <= 255 символов.

**Пример 10.16.** Работа состроками переменной длины**.**

**………………………**

**Var**

**St1 : String [100];** {**макс**имальная длина = 100}

**St3, St2 : String;** {**по умолчанию** максимальная длина = **255**}

**L: Boolean;**

**Begin**

**St1 := ‘Ми’;** {текущая длина ST1= 3}

**St2 := ‘н’;**  {текущая длина ST2 =1}

**St3 := ‘ск’;** {текущая длина ST3 = 2}

**St3 := St1 + St2 + St3;** {**сцепление** (конкатенация) строк: в ST3 значение ‘Минск’; текущая длина =4}

**L := St3 > St1;** {в L запишется **True**}

**St3**[2] := ‘e’; {в ST3 значение **‘Менск’**}

10.2.3. Встроенные функции, определенные над данными типа String 5 шт

1. Copy (St, Poz, N) 1- стр, 2 –поза, 3 – длина нов стр
2. Concat ([St1, St2, ..., StN] )
3. Length (St)
4. Pos (St1, St2) ПОЗА
5. UpCase (Ch) (из малой в БОЛЬШУЮ)
6. **Copy (St, Poz, N) 1- стр, 2 –поза, 3 – длина нов стр**

Выделяет из строки ***St*** начиная с позиции ***Poz*** подстроку **длиной** ***N*** символов. ***St*** – выр-ие типа *String*; ***Poz, N*** – выражения целочисл-ого типа.

Если ***Poz*** больше длины строки, результат – пустая строка.

Если ***Poz + N >*** текущей длины ***St***, результат = последние символы ***St***, начиная с позиции ***Poz***.

Если ***Poz >*** **255** - ошибка выполнения.

**Пример 10.17.**

**Выделение подстроки**. Раздел описаний переменных –д/следующих прим-ов

**Var**

**St1, St2, St3, St4: String;**

**I: Word;**

**Begin**

**St1 := ‘Беларусь’;**

**St2 := Copy (St1, 1, 3); {в St2 значение ‘Бел’}**

**St3 := Copy (St1, 5, 8); {в St3 значение’русь’}**

1. **Concat ([St1, St2, ..., StN] )**

**- сцепление** строк **по** **порядку**, в каком они указаны в списке параметров:

***St1 + St2 + ... + StN***.

***St1 ÷ StN*** – выражения типа String. Длина итоговой строки - **<= 255** симв.

**Пример 10.18.** Продолжение примера 10.17. Сцепление строк.

**St4 := Concat (St2, ‘a’, St3); {в St4 значение’ Белaрусь’}**

1. **Length (St)**

- **Возвращает** текущую длину строки ***St***. Результат – целочисленный тип.

**Пример 10.19.**

См прим. 10.17, 10.18. Определение текущей длины строки.

**I := Length (St4); {I равно 8}**

1. **Pos (St1, St2) ПОЗА**

- Опрелеляет **первое появление** подстроки **St1** в строке **St2**.

Результат – целочисленный: = № позиции, где находится **первый** символ **St1**. Если в ***St2*** подстроки ***St1*** нет, то результат = 0.

**Пример 10.20.**  (прим. 10.17, 10.19). Обнаруж-ие подстроки в строке

**I := Pos (St3, St4); {I = 5}**

1. **UpCase (Ch) (из малой в БОЛЬШУЮ)**

- Преобразует **строчную** **латинскую** букву в **прописную**. В остальных случаях возвращает аргумент ***Ch***. Параметр и результат – тип ***Char***.

**Пример 10.21.**

(пр. 10.17 – 10.20) Преобразов-ие строчных **латинских** букв в Прописные

**St1 := ‘Минск is a capital of Belarus’;**

**For I := 1 To Length (St1) Do**

**St1[I] := UpCase (St1[I]);**

{в St1 знач: **‘Минск IS CAPITAL OF BELARUS’, Минск – не преобраз}**

10.2.4. Встроенные процедуры, определенные   
над данными типа String

1. Delete (St, Poz, N)
2. Insert (St1, St2, Poz)
3. Str (I, St) (число – в строку)
4. Val (St, I, Cod) (ошибка преобраз-ия строки в число)
5. **Delete (St, Poz, N)**

- Удаляет ***N*** символов строки ***St*** с позиции ***Poz***. Значение ***Poz*** д б <= 255. Если ***Poz >*** текущей длины строки, строка ***St*** не изменяется. Если ***Poz + N >*** текущей длины строки, то удаляется конец строки, начиная с ***Poz***.

**Пример 10.22.**

Удаление подстроки. (Раздел описаний переменных – с учетом следующих примеров.)

**Var**

**St1, St2, St3, St4: String;**

**I, I1, Cod, Cod1: Word;**

**Begin**

**St1 := ‘Минск’;**

**Delete (St1, 4, 2);**{в St1 значение ‘**Мин**’}

**…………………………………..**

1. **Insert (St1, St2, Poz)**

Вставляет строку ***St1*** **в** строку ***St2***, начиная с позиции ***Poz***. Если значение ***Poz*** больше текущей длины ***St2***, то результат = сцеплению строк ***St2 + St1***.

**Пример 10.23.** См. пример 10.22. Вставка подстроки.

**St1 := ‘Минск ’;**

**St2 := ‘is a capital of a republic’;**

**Insert (St1, St2, 1);**{в St2: ‘**Минск is a capital of a republic’**}

1. **Str (I, St) (число – в строку)**

- Преобразует **числовое** значение величины **I** и помещает результат в строку ***St***. Величина **I** д им целочисленный / вещественный тип.

**Пример 10.24. (**примеры 10.22, 10.23). Преобраз-ие **числа** в строку

**I := 4716;**

**Str (I, St3);**  {в St3 значение **‘4716’**}

*--------------------------------*

1. **Val (St, I, Cod)**

- Преобразует значение ***St*** в величину **целочисленного или вещественного типа** (в зависимости от типа параметра **I**) и помещает результат в **I**. Значение ***St*** может содержать незначащие пробелы в начале и **не** может в конце.

***Cod*** – целочисленная переменная. Если во время операции преобразования ошибки нет, то значение ***Cod =0***, иначе (например, символьное значение переводится в цифровое) ***Cod*** будет иметь № позиции первого **ошибочного** символа, а значение **I** будет **неопределено**.

**Пример 10.25.** (примеры 10.22 – 10.24). Преобраз-ие **строки** в Число

**St3 := ‘255’;**

**Val (St3, I, Cod);** {в I значение **255**, в Cod значение **0**}

**St4 := ‘48A6’;**

**Val (St4, I1, Cod1);**{значение I1 неопределено, в Cod1 значение **3**}

--------------------

. функции

1. Copy (St, Poz, N) 1- стр, 2 –поза, 3 – длина нов стр
2. Concat ([St1, St2, ..., StN] )
3. Length (St)
4. Pos (St1, St2) ПОЗА (определяет 1ое появление)
5. UpCase (Ch) (из малой в БОЛЬШУЮ)

процедуры

1. Delete (St, Poz, N)
2. Insert (St1, St2, Poz)
3. Str (I, St) (число – в строку)
4. Val (St, I, Cod) (ошибка преобраз-ия строки в число)

## ФСЁ!!!

Из инета:

# 1). Директива $N Pascal-Паскаль

### Описание

**$N**: Математический сопроцессор - Переключает между двумя различными моделями генерации объектного кода обработки чисел с плавающей точкой, обеспечиваемыми компилятором.

|  |  |
| --- | --- |
| **Синтаксис:** | {**$N+**} или {**$N-**} |
| **Значение по умолчанию:** | **{$N-}** |
| **Тип:** | Глобальная |
| **Команда меню:** | Options|Compiler|8087/80287 |

### Состояние $N-:

В состоянии **$N-**, компилятор генерирует код для выполнения всех вещественных вычислений программно, вызывая подпрограммы библиотеки поддержки.

### Состояние $N+:

В состоянии **$N+**, компилятор генерирует код для выполнения всех вещественных вычислений с использованием математического сопроцессора 80x87 и дает вам доступ к четырем дополнительным вещественным типам: Single, Double, Extended и Comp.

### ВНИМАНИЕ:

Вы можете также использовать директиву **$E+**, чтобы эмулировать 80x87. Это дает вам доступ к вещественным типам при отсутствии 80x87 сопроцессора.

* [**{$E}**](http://www.pascal.helpov.net/index/pascal_$E)