

МИНИСТЕРСТВО ОБРАЗОВАНИЯ³ И НАУКИ РОССИЙСКОЙ
ФЕДЕРАЦИИ

Белгородский государственный технологический университет
им. В.Г. Шухова

В.С. Брусенцева

**АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ ПАСКАЛЬ**

3-е издание, стереотипное

**Рекомендуется Учебно-методическим объединением вузов РФ
по образованию в области автоматiki, электроники,
микроэлектроники и радиотехники для межвузовского
использования**

Белгород 2004

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	6
1. ЭТАПЫ РЕШЕНИЯ ЗАДАЧ НА ЭВМ.....	6
2. АЛГОРИТМЫ.....	7
2.1. Свойства алгоритмов.....	7
2.2. Способы записи алгоритмов.....	7
3. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ.....	11
3.1. Принципы структурного программирования.....	12
4. ЯЗЫКИ ПРОГРАММИРОВАНИЯ.....	15
4.1. Свойства языков программирования, характеризующие качество программ.....	16
5. ЯЗЫК ПАСКАЛЬ.....	16
5.1. Алфавит языка Паскаль.....	17
5.2. Способы описания синтаксиса.....	17
5.3. Идентификаторы.....	19
5.4. Структура программы на Паскале.....	19
6. ДАННЫЕ.....	20
6.1. Тип данных.....	20
6.2. Типы данных языка Паскаль.....	21
6.3. Константы.....	21
6.4. Переменные.....	23
7. ЧИСЛОВЫЕ ТИПЫ И АРИФМЕТИЧЕСКИЕ ВЫРАЖЕНИЯ.....	23
7.1. Целые типы ТР.....	23
7.2. Вещественные типы ТР.....	23
7.3. Арифметические операции, определенные над числовыми типами.....	24
8. ОПЕРАТОР ПРИСВАИВАНИЯ.....	26
9. СИМВОЛЬНЫЙ ТИП.....	27
10. ЛОГИЧЕСКИЙ ТИП.....	28
10.1. Логические операции.....	28
11. ПРОСТЫЕ ТИПЫ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ.....	29
11.1. Интервальный тип (тип диапазон).....	29
11.2. Перечисляемый тип.....	30
12. СТАНДАРТНЫЙ ВВОД.....	31
12.1. Ввод числовых данных.....	31
12.2. Ввод символьных данных.....	32
12.3. Процедура readLn.....	32
13. СТАНДАРТНЫЙ ВЫВОД.....	34
13.1. Форматный вывод.....	34
13.2. Вывод значений типа real.....	35
13.3. Пример программы на Паскале.....	35
14. ОПЕРАТОР БЕЗУСЛОВНОГО ПЕРЕХОДА.....	36

15. ПУСТОЙ	
ОПЕРАТОР.....	37
16 СТРУКТУРИРОВАННЫЕ ОПЕРАТОРЫ.....	37
16.1. Составной оператор.....	37
16.2. Выбирающий оператор.....	37
16.3. Оператор цикла.....	40
17. СТИЛЬ ЗАПИСИ ПРОГРАММЫ.....	43
17.1 Комментарии.....	43
18. ОТЛАДКА ПРОГРАММ.....	44
18.1. Виды ошибок и способы их устранения.....	44
18.2. Ручная проверка.....	45
18.3. Машинное тестирование.....	45
18.4. Проверка правильности данных.....	46
18.5. Исправление ошибок.....	46
19 РЕГУЛЯРНЫЙ ТИП (МАССИВ).....	47
19.1. Одномерные массивы.....	47
19.2. Упакованные массивы.....	49
19.3. Многомерные массивы.....	49
19.4. Еще один способ получения многомерных массивов.....	50
19.5. Строковый тип в стандартном Паскале.....	51
19.6. Строковый тип в ТР.....	52
20. ПОДПРОГРАММЫ.....	55
20.1. Область действия описаний.....	55
20.2. Параметры подпрограмм.....	55
20.3. Процедуры.....	57
20.4. Обращение к процедурам (вызов процедур).....	58
20.5. Функции.....	59
20.6. Обращение к функциям (вызов функций).....	60
20.7. Побочный эффект функций.....	61
20.8. Рекурсивные подпрограммы.....	61
20.9. Взаимно рекурсивные подпрограммы.....	62
21. КОМБИНИРОВАННЫЙ ТИП (ЗАПИСЬ).....	63
21.1 Оператор присоединения.....	64
21.2. Записи с вариантами.....	65
22. ПОБИТОВЫЕ ОПЕРАЦИИ.....	67
23. ТИПИЗОВАННЫЕ КОНСТАНТЫ В ТР.....	69
24. МНОЖЕСТВО.....	70
24.1. Машинное представление множества.....	71
24.2. Операции над множествами.....	71
25. ФАЙЛЫ.....	72
25.1. Файлы в Паскале.....	73
25.2. Текстовые файлы.....	74

25.3. Типизованные файлы.....	76
25.4. Нетипизованные файлы.....	77
25.5. Прямой доступ к нетекстовым файлам ТР.....	77
26. ДИНАМИЧЕСКИЕ ПЕРЕМЕННЫЕ.....	79
26.1. Ссылочный тип данных.....	79
26.2. Подпрограммы динамического распределения памяти.....	80
26.3. Операции над указателями.....	81
26.4. Работа с динамическими переменными.....	81
26.5. Создание структур большого размера.....	82
26.6. Длинные строки в ТР.....	83
26.7. Динамические структуры данных.....	83
27. ПРЕОБРАЗОВАНИЕ ТИПОВ.....	85
27.1. Приведение типов выражений.....	86
27.2. Приведение типов переменных.....	86
27.3. Обработка одномерных массивов разных размеров с фиксированным базовым типом.....	87
27.4. Нетипизованные параметры подпрограмм.....	87
28. ПРОЦЕДУРНЫЕ И ФУНКЦИОНАЛЬНЫЕ ТИПЫ.....	88
29. МОДУЛИ В ТР.....	90
29.1. Основные модули ТР.....	91
29.2. Создание собственных модулей.....	91
СПИСОК ЛИТЕРАТУРЫ.....	94

ВВЕДЕНИЕ

Со времени возникновения ЭВМ технология программирования претерпела существенные изменения. Первоначально программы составлялись в машинных кодах. И создание даже сравнительно простых программ требовало больших затрат времени и усилий. Затем появились алгоритмические языки, значительно повысившие производительность труда программиста. Расширение масштабов решаемых задач потребовало создания особой технологии программирования, которую называли структурным программированием. Основой этой технологии являются структурированные алгоритмические языки, такие, как Паскаль, Си.

Настоящее пособие посвящено основам алгоритмизации и программирования на языке Паскаль и охватывает программу курса программирования для специальностей 220400 и 071900.

Сейчас получают широкое распространение новые технологии программирования, такие, как объектно-ориентированное и визуальное программирование. Несмотря на это, владение основами алгоритмического программирования столь же необходимо программисту-профессионалу, как и раньше.

1. ЭТАПЫ РЕШЕНИЯ ЗАДАЧ НА ЭВМ

1. *Постановка задачи.* Если задача поставлена в общем виде, не сформулирована точно, то необходимо понять сущность задачи, правильно ее сформулировать, выделить объект исследования.

2. *Разработка математической модели.* Разработка математической модели заключается в формализации исследуемого объекта. Математическая модель – приближенное описание некоторого класса явлений внешнего мира, выраженное с помощью математической символики, то есть конкретное явление переводится на язык математических формул и дальше рассматривается как математическая проблема.

3. *Выбор способа решения задачи и спецификация алгоритма.* Должны быть установлены выходные и входные данные алгоритма, выделены основные отношения между ними. Необходимо определить, может ли задача быть решена и при каких исходных данных. Результат этапа - спецификация алгоритма, то есть формулировка в общем виде того, что должен делать алгоритм, чтобы переработать входные данные в выходные. Причем должен быть описан класс входных данных со всеми ограничениями.

4. *Разработка алгоритма и его запись.* При разработке алгоритма нужно ориентироваться на конкретную ЭВМ, учитывать ее возможности.

Нужно проанализировать: достаточен ли объём памяти и точность вычислений, хватит ли быстродействия для получения результата в приемлемые сроки. От разработки алгоритма зависит качество программы.

5. *Кодирование*. Кодирование – запись алгоритма на языке программирования. При тщательной разработке алгоритма кодирование, как правило, не вызывает затруднений.

6. *Отладка*. Отладка заключается в поиске и исправлении ошибок. Существует два метода отладки: ручная проверка, не требующая применения ЭВМ, и тестирование. Тестирование заключается в прогонах полностью или частично завершённой программы на ЭВМ. Оба метода требуют тестовых данных. Тестовые данные – наборы исходных данных и ожидаемых результатов. Подбор тестовых данных – важная часть работы программиста. Тесты рекомендуется готовить параллельно с разработкой алгоритма.

7. *Получение и анализ результатов*. Организация прогона программы на ЭВМ зависит от конкретного типа ЭВМ.

2. АЛГОРИТМЫ

Алгоритм является центральным понятием программирования. *Алгоритм* – это конечный набор команд, который определяет последовательность действий для переработки входных данных в выходные.

2.1. Свойства алгоритмов

1. *Детерминированность (определённость)* – ориентированность на определённого исполнителя, исключающая неоднозначность понимания.
2. *Массовость* – пригодность для решения задач определённого класса при любых допустимых значениях исходных данных.
3. *Дискретность* – пошаговый характер получения результата.
4. *Результативность* – свойство алгоритма приводить к результату за конечное время.

2.2. Способы записи алгоритмов

В практике программирования наибольшее распространение получили следующие способы описания алгоритмов:

1. Словесно-формульный.
2. Графический.

- а) блок-схемы;
 - б) структурограммы (диаграммы Насси – Шнейдермана).
3. Псевдокод.
 4. Программный (запись на языке программирования).

Выбор способа записи алгоритма зависит от цели его описания. Алгоритмы могут быть описаны с разной степенью детализации и формализации.

Словесно-формульная запись алгоритма

Алгоритм в словесно-формульном виде представляет собой перенумерованную последовательность действий, описанных обычным языком с использованием математической символики.

Пример 1. Описание алгоритма решения линейного уравнения $ax=b$.

1. Ввод коэффициентов уравнения a и b .
2. Если $a \neq 0$, перейти к п.8.
3. Если $b \neq 0$, перейти к п.6.
4. Вывод: «Любое x является корнем уравнения.».
5. Перейти к п. 10.
6. Вывод: «Уравнение не имеет корней.».
7. Перейти к п. 10.
8. $x := -b/a$.
9. Вывод x .
10. Конец.




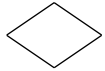

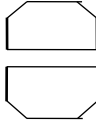

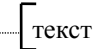
Блок-схемы алгоритмов

Графическая форма записи алгоритма более наглядна, позволяет отчетливо представить все логические связи между частями алгоритма.

Блок-схема алгоритма представляет собой набор геометрических фигур (блоков), соединенных линиями или линиями со стрелками для указания направления перехода от блока к блоку. Движение от блока к блоку сверху вниз или слева направо считается стандартным. В этом случае стрелки можно не указывать. Если же направление отлично от стандартного, то стрелки обязательны.

Необходимая для выполнения очередного действия информация помещается в блок в виде текста или математических обозначений. Перечень блоков, их форма и отображаемые функции установлены ГОСТ 19.701-90 ЕСПД. В таблице приведены основные блоки.

Форма	Название	Назначение
-------	----------	------------

	Терминатор	10 Отображает вход, выход, пуск, останов
	Процесс	Отображает процесс обработки данных любого вида: вычисление значений, изменение формы или размещения информации
	Предопределенный процесс	Отображает процесс, определенный в другом месте (в подпрограмме или в модуле)
	Решение	Отображает решение или функцию переключательного типа, имеющую один вход и несколько альтернативных выходов, из которых только один может быть активизирован
	Данные	Изображает данные. Носитель данных не определен и должен быть указан в блоке
	Границы цикла	Блок состоит из двух частей. Отображает начало и конец цикла. Обе части помечаются одним и тем же уникальным обозначением. В одной из частей помещается условие завершения цикла
	Соединитель	Отображает выход из части схемы и вход в другую часть этой схемы, используется для обрыва линии и продолжения ее в другом месте. Соответствующие соединители помечаются одним и тем же уникальным обозначением
	Комментарий	Используется для пояснительных записей

Структурограммы

Структурограммы изображают последовательность действий не с помощью линий перехода от блока к блоку, а в виде вложенных друг в друга

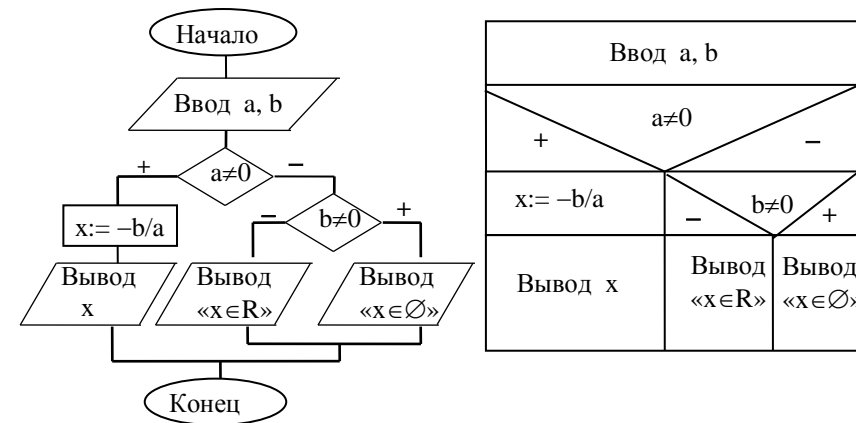
фигур. Каждый блок структурограммы имеет прямоугольную форму и может быть вложен в любой внутренний прямоугольник другого блока.

Основные блоки структурограмм:

Блок	Название	Назначение
	Обработка	Аналогичен блоку «процесс» в блок-схемах. Прямоугольник может быть внутри прямоугольника
	Следование	Последовательное выполнение процессов обработки
	Решение	Бинарное ветвление (выбор из двух альтернатив)
	Расширение блока решения	Множественное ветвление (выбор из нескольких альтернатив)
	Цикл с предуслови-ем	Повторять: если условие выполнено, выполнить тело цикла
	Цикл с постуслови-ем	Повторять: выполнить тело цикла, если условие выполнено, выйти из цикла

Преимущество структурограмм – в их компактности.

Пример 2. Блок-схема и структурограмма алгоритма примера 1.



Псевдокод

Псевдокод занимает промежуточное место между естественным языком и языком программирования. Он позволяет описывать логику программы на естественном языке, но включать типовые конструкции языка программирования, не заботясь о синтаксических тонкостях. Псевдокод удобно использовать при разработке программы.

3. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

Трудность этапа разработки алгоритма заключается в отсутствии формального аппарата для этой работы. Разработка алгоритма зависит от опыта и интуиции программиста. Создание программ на заре программирования в некоторой степени можно отнести к искусству. Увеличение масштабы решаемых на ЭВМ задач приводит к возрастанию сложности программ и программных комплексов, которые разрабатываются целыми коллективами. Готовые программы отчуждаются от их создателей и передаются для эксплуатации в другие коллективы. Ошибки в таких программах могут дорого стоить. В связи с этим появилась необходимость в создании научно обоснованной методологии разработки алгоритмов и программ для получения надежных программ. Эта методология должна касаться анализа задачи, разделения ее на достаточно самостоятельные части и программирования этих частей по возможности независимо друг от друга. Такой методологией, зародившейся в начале 70-х годов, явилось структурное программирование.

1. *Разработка алгоритма «сверху вниз» (метод пошаговой детализации).*

Начиная со спецификации, полученной в результате анализа задачи, выделяют небольшое число достаточно самостоятельных подзадач и описывают спецификации для каждой. Это будет первый шаг детализации. С каждой из выделенных подзадач поступают так же (второй шаг детализации) и т.д. Таким образом получается последовательность все более детальных спецификаций, приближающаяся к окончательной версии программы.

2. *Модульность.*

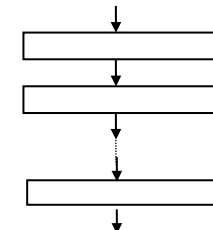
Метод пошаговой детализации дает возможность разбить алгоритм на части (модули), каждая из которых решает самостоятельную подзадачу. Размеры модулей должны быть небольшими, а инструкции, входящие в состав модуля, должны давать исчерпывающее представление о действиях, выполняемых модулем. Связи по управлению между модулями осуществляются посредством обращений к ним, а обмен информацией – через параметры и глобальные переменные.

3. *Каждый модуль должен иметь один вход и один выход.*

Это позволяет упростить стыковку модулей в сложной программе.

4. *Логика алгоритма должна опираться на небольшое число достаточно простых базовых управляющих структур:*

1. Следование



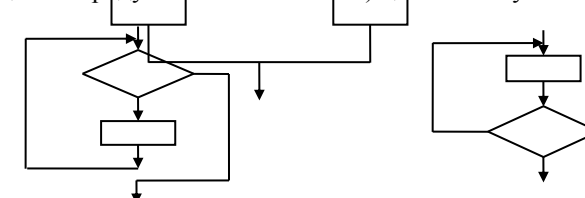
2. Развилка



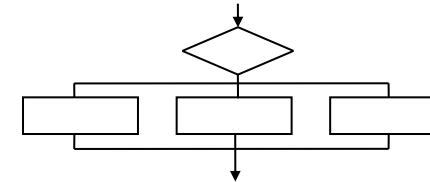
3. Цикл:

а) цикл с предусловием

б) цикл с постусловием



4. Выбор из нескольких альтернатив (переключатель)



Фундаментом структурного программирования является **теорема о структурировании**. Она утверждает, что *как бы ни сложна была задача, схема алгоритма может быть представлена с использованием ограниченного числа элементарных управляющих структур*.

Теорема о полноте. Базовые элементарные структуры: *следование, разветвление и цикл* – обладают функциональной полнотой, то есть любой алгоритм может быть реализован в виде композиции этих конструкций.

Пример. Применим принципы структурного программирования для разработки и описания алгоритма решения следующей небольшой задачи.

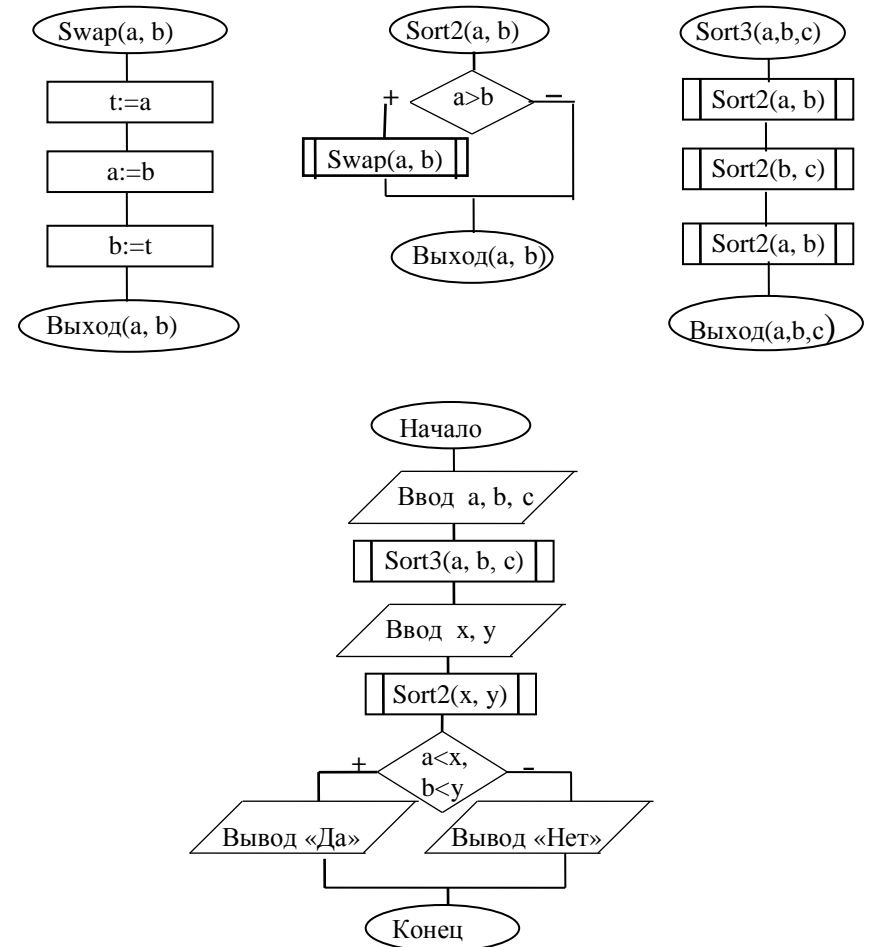
Требуется определить, пройдет ли кирпич с ребрами **a**, **b** и **c** в прямоугольное окно со сторонами **x**, **y**. Грани кирпича должны быть параллельны или перпендикулярны сторонам окна.

Если бы стороны окна и ребра кирпича были упорядочены, то для решения задачи достаточно сравнить меньшую сторону с меньшим ребром и большую сторону со средним ребром.

Выделяем подзадачи:

1. Упорядочение пары чисел по неубыванию.
2. Упорядочение тройки чисел по неубыванию.
3. Определение, пройдет ли кирпич с ребрами $a \leq b \leq c$ в прямоугольное окно со сторонами $x \leq y$.

Алгоритмы решения задач 1 и 2 опишем, как алгоритмы для подпрограмм, в дальнейшем на них можно ссылаться в определенных блоках.



Спецификация алгоритма Sort2.

Sort2(a, b) сортирует пару вещественных чисел **a** и **b** по неубыванию.

Входные параметры: **a, b** – вещественные числа.

Выходные параметры: **a, b**, удовлетворяющие условию $a \leq b$.

Аналогично описывается спецификация алгоритма Sort3.

Решение задачи сортировки пары заключается в обмене значениями переменных **a** и **b**, если **a > b**. Обмен значениями пары переменных – еще одна подзадача (подзадача подзадачи). Назовем эту подзадачу Swap.

4. ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Язык программирования – это строгий набор правил, символов и конструкций, которые позволяют в формульно-словесной форме описывать алгоритмы для обработки данных на ЭВМ.

Каждая ЭВМ имеет свою систему команд, и программа в машинных кодах может быть выполнена сразу. Вначале язык машинных кодов был единственным языком программирования. Но использование этого языка – очень трудоемкий процесс, программы получаются громоздкими, их трудно отлаживать, модифицировать, практически невозможно перенести на другую ЭВМ. Возникла необходимость создания новых способов и средств записи программ. Стали появляться новые языки программирования. Их создание шло по двум направлениям: машинно ориентированные и машинно независимые (алгоритмические) языки. Машинно ориентированные языки (ассемблер, автокод) – языки низкого уровня, требующие указания мелких деталей процесса обработки данных. Алгоритмических языков – языков высокого уровня – в настоящее время более 500. Каждый язык имеет свои особенности, но есть ряд общих черт, отличающих алгоритмические языки от языков низкого уровня:

1. Алгоритмические языки обладают большими выразительными возможностями, имея широкий алфавит, что повышает наглядность текста программ.
2. Набор операций не зависит от машинных операций, а выбирается для удовлетворения потребностей конкретной прикладной области.
3. Операции задаются в удобном виде.
4. Одним предложением можно задать значительный этап обработки данных.
5. Программным объектам, над которыми выполняются действия, присваиваются имена, и обращение к ним происходит по имени.
6. В алгоритмических языках предусмотрен широкий набор типов данных.

При использовании алгоритмических языков мы идем на издержки. Во-первых, программа должна быть оттранслирована (переведена на язык машинных кодов программой-транслятором), на это требуется время. Во-вторых, полученная после трансляции программа может быть менее эффективной, чем составленная опытным программистом на языке

низкого уровня с учетом специфики системы команд и организации памяти. Но эти недостатки окупаются удобствами для программиста.

4.1 Свойства языков программирования, характеризующие качество программ

1. Простота.
2. Надежность (некоторая мера отсутствия ошибок в программе).
3. Быстрота трансляции.
4. Эффективность (характеризует быстродействие и объем используемой памяти).
5. Удобочитаемость.
6. Модульность (возможность независимой обработки отдельных частей программы и последующего их связывания в единую систему).

Некоторые языки программирования (например, Си, Паскаль) стандартизированы на международном уровне, на них имеются эталонные описания. Но на практике мы, как правило, пользуемся некоторой версией, то есть диалектом языка, который может быть подмножеством или надмножеством стандарта.

5. ЯЗЫК ПАСКАЛЬ

Язык Паскаль создан в начале 70-х годов одним из соавторов методологии структурного программирования, швейцарским профессором Никлаусом Виртом. Создавался этот язык как учебный, но вскоре приобрел популярность среди программистов как язык для решения различных серьезных задач. Программы на Паскале удобочитаемы, они транслируются в эффективные машинные коды. Паскаль способствует внедрению структурного программирования. Наиболее распространенной версией Паскаля является Турбо Паскаль, используемый в интегрированной интерактивной системе Турбо Паскаль. Ниже рассматривается язык Турбо Паскаль (TP) и его отличия от стандартного Паскаля.

Любой язык, в том числе и язык программирования, определяется алфавитом, синтаксисом и семантикой.

Алфавит – это фиксированный набор символов, из которых состоит текст на данном языке. Текст на языке программирования – программа.

Синтаксис языка программирования определяет правила построения из символов алфавита специальных конструкций, с помощью которых можно составлять алгоритмы решения задач.

Семантика – система правил истолкования конструкций языка.

5.1. Алфавит языка Паскаль

Алфавит Паскаля составляют буквы, цифры и специальные символы.

Буквами в стандартном Паскале являются прописные латинские буквы от A до Z. Во многих реализациях буквами являются также и строчные латинские буквы, причем прописные и соответствующие им строчные буквы не различаются. Кроме этого, как букву можно использовать знак подчеркивания «_».

Цифры – 0, 1, 2, ... , 9.

Специальные символы – это знаки операций, разделители и ключевые (служебные) слова.

Знаки операций :

-	+	*	/	mod	div	>	<	>=	<=	=	<>	^
---	---	---	---	-----	-----	---	---	----	----	---	----	---

В ТР знаком операции, кроме перечисленных, является @.

Разделители :

,	;	:	()	[]	{	}	'	=	..	.
---	---	---	---	---	---	---	---	---	---	---	----	---

В ТР разделителями являются символы # и \$.

Примеры *ключевых слов*: program, for, if, then, else. В Паскале служебных слов более 40. Они будут вводиться в рассмотрение при изучении конструкций языка.

5.2. Способы описания синтаксиса

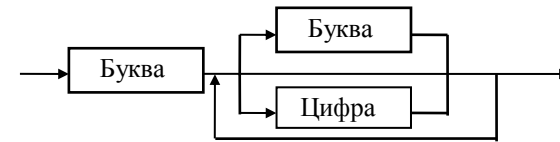
Для описания синтаксиса широко применяются:

- 1) синтаксические диаграммы;
- 2) металингвистические формулы Бэкуса – Науэра.

Синтаксические диаграммы

Синтаксические диаграммы напоминают структурные схемы, но с их помощью описываются не процессы обработки, а структуры данных. В синтаксических диаграммах используются блоки двух видов: овальные и прямоугольные. В овальные блоки помещают символы, которые без изменений входят в описываемую конструкцию, а в прямоугольные помещают понятия, требующие определения или определенные ранее. Блоки соединяются стрелками. Чтобы получить правильные грамматические конструкции, нужно идти по путям, указанным стрелками, от одного блока к другому, пока не придем к выходу. Если предусмотрено более одного направления движения, можно выбрать любое.

В качестве примера приведем синтаксическую диаграмму одного из важных понятий программирования – идентификатора.



Словесно это определение можно записать следующим образом.

Идентификатором является последовательность букв и цифр, начинающаяся с буквы.

В стандартном Паскале допускается длина идентификатора не более 8 символов, а в ТР – до 63 символов включительно.

Металингвистические формулы Бэкуса – Науэра

Метод описания синтаксиса с помощью формул **Бэкуса – Науэра** заключается в использовании специальных обозначений :

::= – читается: «по определению есть»;

| (вертикальная черта) – выбор, альтернатива;

{ } (фигурные скобки) – возможность повторения, что соответствует замкнутому циклу в синтаксических диаграммах;

[] (квадратные скобки) – необязательная часть синтаксической конструкции;

(..|..) (круглые скобки с вертикальной чертой) – альтернатива внутри определения;

< > в угловые скобки заключают понятия, требующие определения (метаварiable), которые в синтаксических диаграммах записывают в прямоугольных блоках.

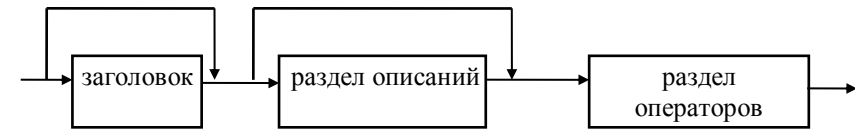
На языке формул Бэкуса – Науэра (ЯБНФ) определение идентификатора имеет вид $\langle \text{буква} \rangle \{ \langle \text{буква} \rangle | \langle \text{цифра} \rangle \}$

5.3. Идентификаторы

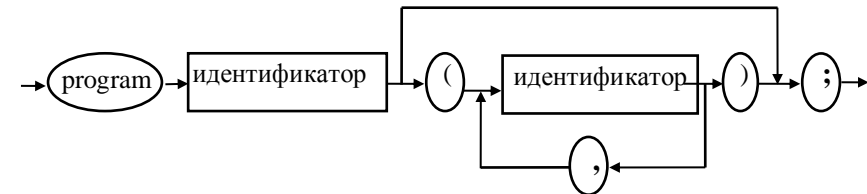
Идентификатор – имя программного объекта. Программными объектами являются программы, переменные, константы, процедуры, функции, типы и метки. Каждый идентификатор является либо стандартным, либо определяемым программистом. Стандартные – это встроенные в язык идентификаторы и идентификаторы, описанные в библиотеках Паскаля. Смысл этих идентификаторов уже определен. Например, *integer* – имя целого типа, *read* – имя процедуры ввода, *cos* – имя функции, возвращающей косинус своего аргумента.

5.4. Структура программы на

Паскале



Раздел описаний вместе с разделом операторов называют *блоком*.

Заголовок

Идентификатор после ключевого слова **program** – имя программы. В стандартном Паскале заголовок обязателен, а также обязателен список идентификаторов в круглых скобках.

Раздел описаний

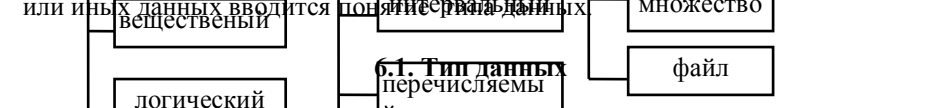
Все программные объекты, которые вводятся в рассмотрение программистом, должны быть описаны в разделе описаний. Нельзя использовать программный объект до его описания. В стандартном Паскале программные объекты должны описываться в следующей последовательности: метки, константы, типы, переменные, подпрограммы. В TP они могут описываться в произвольном порядке. Правила описаний для каждого вида программных объектов будут рассмотрены ниже.

Раздел операторов

В разделе операторов записывается алгоритм решения задачи в виде последовательности операторов. Оператор – законченная фраза языка, которая определяет некоторый этап обработки данных. Операторы можно разделить на две группы: простые и производные. В Паскале 4 вида простых операторов: оператор присваивания, оператор процедуры, оператор перехода, пустой оператор, а также 4 вида производных операторов: составной оператор, выбирающий оператор, оператор цикла, оператор присоединения.

6. ДАННЫЕ

Информация, предназначенная для передачи и обработки, хранится в формализованном виде в виде данных. Например, в памяти ЭВМ данные представляют собой последовательности из 0 и 1 (битов). Для удобства обработки данных биты объединяют в байты (8 битов составляет 1 байт), а байты — в слова. Данные могут быть переменными или константами. Конечная цель обработки данных — получение новой информации. Одну и ту же последовательность битов можно интерпретировать и обрабатывать по-разному. Например, байт 01000001 можно рассматривать как символ 'А', и как целое число 65. Для правильной интерпретации и обработки или иных данных вводятся понятия интерпретации данных.



Тип данных определяет:

1. Множество значений, которые могут принимать данные этого типа. Это множество определяется формой представления значения в оперативной памяти и объемом памяти, выделяемой значению;
2. Множество операций, которые разрешены над данными этого типа.
3. Каждая константа и переменная относятся к определенному типу по их виду или описанию.
4. Каждая операция требует операндов определенного типа и формирует тип результата, в соответствии с правилами языка.
5. Каждая функция требует аргументов определенного типа и возвращает результат определенного типа.

6.2. Типы данных языка Паскаль

К *простым типам* относятся данные, значения которых являются неделимыми. Данные *структурированных типов* представляют собой совокупности компонентов ранее определенных типов.

6.3. Константы

Константа – это программный объект, не изменяющий своего значения. Каждая константа является либо *литералом*, либо *именованной константой*.

Литералы – те константы, тип которых определяется по их виду. Литералами являются целые и вещественные числа, символы, символьные строки.

Целые константы записываются в общепринятом виде, как целое без знака (например, 368) или как целое со знаком (например, –95). ТР позволяет записывать целые константы в шестнадцатеричном виде. Признаком шестнадцатеричной константы является префикс \$. Цифры дополняются начальными буквами латинского алфавита: A(10), B(11), C(12), D(13), E(14), F(15). Регистр букв не имеет значения. Например, \$5F=95.

Вещественная константа может быть представлена в форме с фиксированной или с плавающей точкой.

Форма с фиксированной точкой (на ЯБНФ):

< целое >.[< целое без знака >]

В записи вещественной константы в форме с фиксированной точкой обязательно должна быть точка. Например, 12.37; 0.063; -4.; 5.09.

Форма с плавающей точкой:

(<целое> | <веществ. с фикс.точкой>)(E | e)[+ | -]<цифра>{<цифра>}

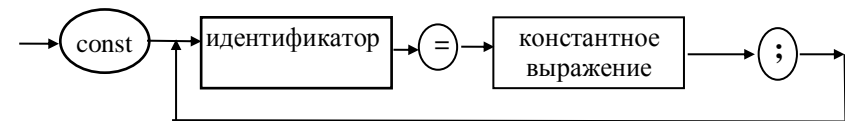
Например, -1.012E+01 (= -10,12), 1E2 (=100).

Символьная константа – это символ из определенного упорядоченного набора символов, заключенный в апострофы. Например, '1', 'w', '+'. В упорядоченном наборе каждый символ имеет номер (код). В ТР символ можно представить в виде #<код>. Так, символ A, имеющий код 65, можно записать как 'A' или #65.

Символьная строка – это последовательность символов, заключенная в апострофы. Например, 'Это символьная строка'.

Именованная константа – это фиксированное значение, которому в разделе описаний присваивается имя.

Описание констант:



Идентификатор – имя константы. Имена констант должны быть осмысленными. Использование именованных констант делает программу удобной для понимания и внесения изменений.

Константным выражением является выражение, не содержащее переменных. В константных выражениях допускается использование некоторых стандартных функций: Abs, Chr, Hi, Lo, Length, Odd, Ord, Pred, Ptr, Round, Trunc, Succ. Тип константного выражения определяет тип описываемой константы.

Например:

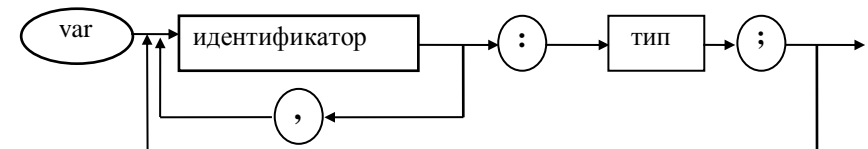
```

const MAX = 100;           {целый тип}
      LEN = 1.5 * MAX;      {вещественный тип}
      SYMBOL = 'z';         {символьный тип}
      FLAG = SYMBOL = 'z';  {логический тип}
  
```

6.4. Переменные

Переменные – объекты, способные изменять свои значения.

Описание переменных:



Идентификатор – имя переменной. Каждая переменная в Паскале имеет тип, определяемый при описании, в соответствии с которым этой переменной выделяется определенный объем памяти. Секцией в разделе описания переменных называется список перечисленных через запятую переменных с указанием их общего типа.

Например:

```
var i, j : integer;
    f : real;
```

В приведенном описании две секции. В первой определены две переменные *i* и *j* целого типа, во второй – вещественная переменная *f*.

7. ЧИСЛОВЫЕ ТИПЫ И АРИФМЕТИЧЕСКИЕ ВЫРАЖЕНИЯ

7.1. Целые типы ТР

- 1) shortint – короткое целое $\in [-128; 127]$, занимает 1 байт;
 - 2) byte – целое без знака $\in [0; 255]$, занимает 1 байт;
 - 3) integer – целое со знаком, занимает машинное слово;
 - 4) word – целое, без знака, занимает машинное слово;
 - 5) longint – целое со знаком размера больше или равного integer.
- Все целые типы являются *упорядоченными* типами.

7.2. Вещественные типы ТР

Вещественное число, не равное нулю, можно представить в виде $r = a \cdot 10^n$, где *a* – мантисса, удовлетворяющая условию $1 \leq a < 10$ при $r \neq 0$, а *n* – порядок ($n \in \mathbb{Z}$). Значения вещественных типов хранятся в памяти в виде мантиссы и порядка.

Имя типа	Название	Объем памяти	Цифр в мантиссе	Порядок ($n \in \mathbb{Z}$)
single	вещественный одинарной точности	4	7–8	$n \in [-39; 38]$
real	вещественный	6	11–12	$n \in [-39; 38]$
double	вещественный двойной точности	8	15–16	$n \in [-324; 308]$

extended	вещественный расширенный	10	19–20	$n \in [-4932; 4932]$
comp	сложный	8	19–20	$n \in [0; 18]$

Вещественный тип, ввиду приближённого представления чисел, *не является в Паскале упорядоченным*.

Операция сравнения на равенство данных вещественного типа считается некорректной, равенство $r1=r2$ будем заменять неравенством $|r1-r2| < \varepsilon$, где ε – точность.

В стандартном Паскале определен один вещественный тип *real*. В TP типы *single*, *double* и *extended*, *comp* можно использовать только при наличии сопроцессора.

7.3. Арифметические операции, определенные над числовыми типами

Название операции	Знак операции	Запись на Паскале	Типы операндов	Тип результата
унарный плюс	+	+a	целый	целый
			вещественный	вещественный
унарный минус	–	–a	целый	целый
			вещественный	вещественный
сложение	+	a+b	оба целые	целый
			хотя бы один вещественный	вещественный
вычитание	–	a–b	как при сложении	как при сложении
умножение	*	a*b	как при сложении	как при сложении
Название операции	Знак операции	Запись на Паскале	Типы операндов	Тип результата
деление	/	a/b	любые числовые	вещественный
целочисленное деление	div	a div b	целые	целые
нахождение остатка от деления	mod	a mod b	целые	целые

Правило выполнения операции div и mod:

$$a \text{ div } b = \text{sign}(a*b) * \lfloor a/b \rfloor \quad ([x] - \text{целая часть } x);$$

$$a \text{ mod } b = a - (a \text{ div } b) * b.$$

Например, $-15 \text{ div } 7 = -2$; $-15 \text{ mod } 7 = -1$.

Выражение определяет правила вычисления значения, оно представляет собой последовательность, состоящую из констант, переменных, вызовов функций, соединенных знаками операций и, возможно, круглыми скобками. При вычислении значений выражений вначале выполняются операции более высокого приоритета. Операции одинакового приоритета выполняются слева направо по порядку. Для изменения естественного порядка действий используются круглые скобки.

Выражения, типы которых числовые, будем называть *арифметическими*.

На Паскале выражения записываются в строку. Например, выражение

$$\frac{a+b}{a-b}$$

на Паскале имеет вид $(a+b)/(a-b)$. В выражениях не допускаются подряд два знака операций. Например, недопустима запись $a*-b$; правильное выражение $a*(-b)$.

Приоритеты операций в арифметических выражениях (в порядке убывания):

1. Унарные (+ и -).
2. Умножение, деление, деление нацело, нахождение остатка от деления.
3. Сложение, вычитание.

Некоторые стандартные функции Паскаля:

Обращение	Математическое обозначение	Тип аргумента	Тип результата
sin(x)	sin x	числовой	вещественный
cos(x)	cos x	числовой	вещественный
arctan(x)	Arctg x	числовой	вещественный
ln(x)	ln x	числовой	вещественный
exp(x)	e ^x	числовой	вещественный
sqrt(x)	Вставить формулу	числовой	вещественный

abs(x)	x	целый вещественный	целый вещественный
sqr(x)	x*x	как для abs(x)	как для abs(x)
Pi	π	нет	вещественный
round(x)	округление до ближайшего целого	вещественный	целый
trunc(x)	отбрасывание цифр после точки	вещественный	целый

Так как в Паскале нет операции возведения в степень, для возведения в степень x для $x \notin \mathbb{Z}$ воспользуемся основным логарифмическим тождеством $a^x = e^{\ln a^x}$ при $a > 0$, получим $\exp(x * \ln(a))$.

Для вычисления $\log_a x$ используем формулу перехода к основанию e . Функции Arcsin(x) и Arccos(x) выражаем через Arctg(x).

8. ОПЕРАТОР ПРИСВАИВАНИЯ



Действие оператора присваивания заключается в вычислении значения выражения и присваивании этого значения переменной. Например, после выполнения операторов $i:=5$; $i:=i+3$ значение переменной i будет равно 8.

Тип переменной в левой части оператора присваивания должен быть совместим по присваиванию с типом выражения в правой части. Для разных типов требования совместимости по присваиванию различны. Любой целый тип совместим по присваиванию только с целым. Вещественный тип совместим по присваиванию и с целым, и с вещественным. После описания `var j:integer;` оператор $j:=6/2$ недопустим, так как операция деления формирует результат вещественного типа, который не может быть присвоен целому.

Оператор присваивания используется как для инициализации переменных, так и для изменения их значений.

9. СИМВОЛЬНЫЙ ТИП

Имя символьного типа – *char*. Значениями символьного типа являются символы из определенного набора символов, который зависит от конкретной реализации языка. В этот набор входят некоторые символы алфавита и, возможно, другие символы. Среди них могут быть символы, не имеющие графических изображений (например, символ перехода к новой

строке). Все символы пронумерованы. Их номера - коды. Множество значений символьного типа является упорядоченным множеством (чем больше код, тем больше символ). В различных реализациях символы могут быть упорядочены по-разному, но обязательно выполнение следующих условий:

1. Коды цифр – последовательные числа и '0' < '1' < '2' < ... < '9'.
2. Прописные и строчные буквы должны быть упорядочены по алфавиту, но не требуется, чтобы коды были последовательными числами.

В ТР множеством значений символьного типа является множество символов из расширенной таблицы ASCII, состоящей из 256 символов. Первая половина таблицы (с кодами от 0 до 127) – неизменяемая, вторая (с кодами от 128 до 255) – альтернативная. Символы с кодами меньше 32 являются управляющими, пробел имеет код 32. Цифры, латинские буквы, некоторые знаки операций, разделители находятся в первой половине таблицы. Важная особенность: строчные латинские и прописные латинские буквы имеют последовательные коды. Альтернативная часть таблицы содержит буквы русского алфавита, символы псевдографики.

Над данными символьного типа не определены никакие операции, кроме операций сравнения. Поэтому выражениями символьного типа являются константы, переменные и функции, возвращающие значения типа *char*.

Стандартные функции, используемые при работе с символами:

chr(i) возвращает символ по его коду **i**

ord(c) возвращает код символа **c**

Примеры:

1. Значением выражения `ord('7') – ord('0')` является число 7.
2. Значением выражения `chr(ord('A')+5)` является шестая буква латинского алфавита.

10. ЛОГИЧЕСКИЙ ТИП

Имя логического типа – `boolean`. Множество значений: `false` и `true`. Значение логического типа занимает 1 байт. `False` (ложь) и `true` (истина) хранятся в памяти как 0 и 1 соответственно. Тип упорядочен: `false < true`.

10.1. Логические операции

Название	Знак операции	Запись на Паскале	Приоритет операции
Отрицание	<code>not</code>	<code>not a</code>	1

Конъюнкция	and	a and b	2
Дизъюнкция	or	a or b	3
Исключающее или (сложение по модулю 2)	xor	a xor b	3

Типы операндов и тип результата – boolean.

Результат логической операции определяется в соответствии с таблицей истинности (1 – true, 0 – false).

Таблица истинности

x	y	not x	x and y	x or y	x xor y
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Два выражения, соединенные одним из знаков сравнения, образуют логическое выражение, которое называется *отношением*. Все шесть операций сравнения (>, <, >=, <=, =, <>) определены над числовыми, символьным, логическим, перечисляемым, строковым типами.

Если операндами логических операций являются отношения, то в Паскале они должны быть заключены в круглые скобки, так как операции сравнения имеют более низкий приоритет, чем логические операции.

Примеры логических выражений на Паскале:

1. Неравенство $-2 \leq x < 5$ имеет вид $(-2 \leq x)$ and $(x < 5)$.
2. Условие $x \notin [-2; 5]$ имеет вид $(x < -2)$ or $(x >= 5)$.
3. Высказывание «с – латинская буква» имеет вид $(c >= 'A')$ and $(c <= 'Z')$ or $(c >= 'a')$ and $(c <= 'z')$.

В ТР по умолчанию, то есть если не указано иначе, при вычислении логического выражения операнды вычисляются до тех пор, пока значение логического выражения не станет определенным. Например, если значение первого операнда дизъюнкции – истина, то значение второго операнда вычисляться не будет. Если значение первого операнда конъюнкции – ложь, то значение второго операнда вычисляться не будет. С учетом сказанного, несмотря на то, что бинарные логические операции коммутативны, следует обращать внимание на последовательность операндов в логических выражениях.

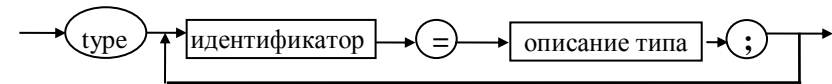
Значение выражения $(b < > 0)$ and $(a/b > 10)$ определено при любых b, при перестановке операндов этого выражения значение не может быть

вычислено при $b=0$, и при выполнении программы произойдет аварийный останов.

11. ПРОСТЫЕ ТИПЫ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ

Любой тип, вводимый в рассмотрение пользователем, можно описать непосредственно при описании переменных или этому типу можно присвоить имя в разделе описания типов, а затем использовать имя типа при описании переменных. Для того чтобы отличать имена типов от имен других программных объектов, можно начинать имя типа с префикса `t_`.

Раздел описания типов:

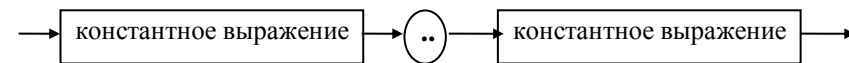


Здесь идентификатор – имя типа. Описание типа определяется правилами языка для каждого вида типов.

Из простых типов, определяемых пользователем, рассмотрим интервальные и перечисляемые. Указательный тип будет рассмотрен ниже.

11.1. Интервальный тип (тип диапазон)

Описание интервального типа:



Константные выражения (границы диапазона) должны принадлежать одному и тому же упорядоченному типу, который является *базовым* для вводимого типа. Значение первого константного выражения не может превышать значения второго константного выражения. Символ алфавита «`..`» называется индикатором диапазона. Множество значений интервального типа составляют значения базового типа, принадлежащие диапазону. Интервальные типы можно использовать всюду, где допустим соответствующий базовый тип.

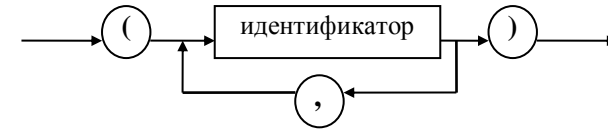
Примеры:

```

type t_month = 1..12;      {Интервальный тип на базе целого}
   t_letter = 'A'..'Z';    {Интервальный тип на базе символьного}
var m1, m2: t_month;      {Переменные интервального типа}
   digit: '0'..'9';       {Переменная интервального типа на базе символьного}
  
```

11.2. Перечисляемый тип

Описание перечисляемого типа:



Каждый идентификатор в описании перечисляемого типа является константой определяемого типа. Например,

```
Type t_season = (Winter, Spring, Summer, Autumn);
```

Определен перечисляемый тип из четырех значений.

Перечисляемый тип является упорядоченным, порядок определяется перечислением. В памяти значения представлены кодами, нумерация начинается с нуля. Данные перечисляемых типов, как и всех рассмотренных выше типов, можно сравнивать. Никакие другие операции над данными перечисляемых типов не определены. Для ввода и вывода данных перечисляемых типов, определяемых программистом, нельзя использовать стандартные процедуры **read** и **write**. Ввод и вывод таких данных осуществляется программно.

Использование перечисляемых типов улучшает смысловую читаемость программы.

В стандартной библиотеке есть ряд функций для работы с любыми упорядоченными типами, в том числе и перечисляемыми. Рассмотрим некоторые из них:

ord(n) – возвращает код значения аргумента **n** (для целочисленных типов код значения – само число).

pred(n) и **succ(n)** – возвращают соответственно значение, предшествующее аргументу и следующее за аргументом в упорядоченной последовательности значений типа аргумента. Если соответствующего элемента в последовательности не окажется, то произойдет ошибка времени выполнения.

Примеры:

```
ord(Winter) = 0;
```

```
ord(Summer) = 2.
```

```
pred(Summer) = spring;
```

```
succ('a') = 'b',
```

```
pred(Winter) – ошибка,
```

```
succ(Autumn) – ошибка.
```

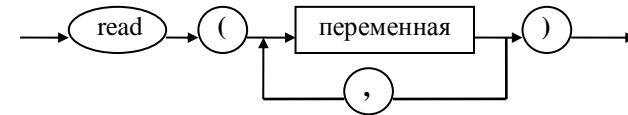
Будем называть *перечисляемыми* типами упорядоченные типы, нумерация элементов которых начинается с нуля. Согласно такому определению, стандартные типы `boolean`, `char`, `byte`, `word` являются перечисляемыми.

12. СТАНДАРТНЫЙ ВВОД

Связь программы с внешним миром осуществляется с помощью операторов ввода и вывода. Эти операторы представляют собой вызовы стандартных процедур ввода и вывода.

Процедуры ввода `read` и `readLn` позволяют инициализировать переменные значениями, вводимыми с клавиатуры.

Синтаксическая диаграмма процедуры `read`:



Процедура `read` позволяет ввести значения *числовых* типов, *символьного* и *строкового*. При выполнении процедуры ввода программа приостанавливает работу и ждет, пока с клавиатуры не будут введены значения для всех переменных, перечисленных в списке параметров. Набираемые на клавиатуре символы хранятся в некоторой области памяти, называемой буфером ввода. Считывание данных происходит из буфера ввода.

При вводе значений типы переменных должны быть совместимы по присваиванию с соответствующими им вводимыми значениями.

12.1. Ввод числовых данных

Перед набором числа допускается набор пустых символов (пробел, табуляция, переход к новой строке). После ввода числа должен быть введен хотя бы один пустой символ, в противном случае произойдет ошибка ввода. Разделителями при последовательном вводе чисел являются пустые символы. Пустые символы, предшествующие числу, считываются из буфера, но игнорируются. Затем считывается число, и его значение присваивается соответствующей переменной. Пустой символ после числа служит только признаком конца считывания, но он остается в буфере ввода, и с него начнется следующее считывание.

Пример 1.

```
var  n : word;
     r : real;
begin
  read(n,r);
  ... {операторы}
end.
```

Набрать значения `n` и `r` можно в одной строке,

`27 -4.8E-4`,

или в разных строках,

`27`
`-4.8E-4`.

12.2. Ввод символьных данных

При вводе символьных данных считывается очередной символ из буфера и присваивается символьной переменной. Поэтому при последовательном вводе символов разделители не требуются. Следует обратить внимание на то, что после ввода числа в символьную переменную может быть введен только пустой символ.

Пример 2.

```
var c1,c2 : char;
begin
  read(c1,c2);
  {операторы}
end.
```

При вводе

a b

 (между a и b - пробел) получим $c1 = 'a'$, $c2 = \#32$ (пробел). Символ 'b' остается в буфере.

12.3. Процедура readLn

Выполнение процедуры readLn отличается от read только тем, что после считывания значений и присваивания их параметрам происходит переход к новой строке.

Пример 3.

```
var n, m : byte;
    r : real;
begin
  read(n,r);

  read(m);
  ... {операторы}
end.
```

Набрать значения n, r и m можно в одной строке,

27 -4.8E-4 131

, или в разных строках,

27
-4.8E-4 131

Результаты будут одинаковы:

n=27
r=-4.8E-4
m=131

Пример 4.

```
var n, m : byte;
    r : real;
begin
  readLn(n,r);

  read(m);
  ... {операторы}
end.
```

Если набрать значения n, r и m,

27 -4.8E-4 131

 или

27
-4.8E-4 131

,

то значение 131 переменной m не будет присвоено, так как ожидается ввод значения m с новой строки.

Правильный ввод:

27
-4.8E-4
131

 или

27 -4.8E-4
131

Процедуру readLn можно использовать для ввода символьных данных после числовых.

Пример 5.

```
var n, m : byte;
    c1, c2 : char;
begin
  readLn(n,r);
  read(c1, c2);
  ... {операторы}
end.
```

Допустимый ввод:

27
164
gf

 или

27	164
gf	

.

Символьные значения вводятся с новой строки.

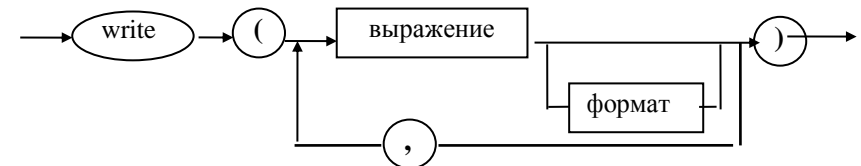
Процедуру readLn можно вызвать без параметров. В этом случае, если в буфере нет символа новой строки, выполнение программы приостанавливается до нажатия клавиши перехода к новой строке (Enter).

Ввод символьных строк будет рассмотрен в разделе «Строковый тип».

13. СТАНДАРТНЫЙ ВЫВОД

Вывод на дисплей выполняют процедуры write и writeLn.

Синтаксическая диаграмма вызова процедуры write:



Процедура write вычисляет и выводит на экран значения выражений, перечисленных в качестве параметров. Выражения могут быть *числовыми, символьными, логическими и строковыми*.

Вывод начинается с текущей позиции курсора на экране. После вывода значения курсор помещается за последним выведенным символом, и вывод следующего значения начнется с этой позиции. Никаких разделителей между выводимыми значениями не предусмотрено. Поэтому оператор write(12, 3, 'k') выведет строку

123k

. Для разделения значений можно предусмотреть вывод пробелов между ними. Так, оператор write(12, ' ', 3, ' ', 'k') выведет

12 3 k

.

Значения типа `real` выводятся в форме с плавающей точкой в следующем виде:

$(-|) < \text{цифра} > . < \text{цифра} > \{ < \text{цифра} > \} E (+|-) < \text{цифра} > < \text{цифра} >$ (1)

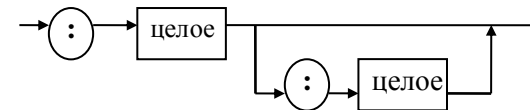
Количество цифр после точки – 10. Общее число позиций, занимаемых вещественным числом, при этом равно 17. Например, оператор `write(-123.45)` выведет `-1.2345000000E+02`.

Процедура `writeln` выполняется так же, как и `write`, но после вывода значений переводит курсор в начало новой строки. `WriteLn` можно использовать и без параметров для перехода к новой строке.

13.1. Форматный вывод

Необязательная часть конструкции оператора вывода – формат – предназначена для управления формой вывода.

Формат имеет вид



Первое целое определяет ширину поля вывода – количество выделяемых значению позиций. Второе целое в формате можно использовать только при выводе вещественных значений. Если значение занимает меньше позиций, чем ширина поля вывода, то оно прижимается к правому краю поля вывода, а свободные позиции слева заполняются пробелами.

Если для вывода целого, логического, символьного или строкового значения ширины поля вывода недостаточно, то поле вывода расширяется, и значение выводится без искажения.

13.2. Вывод значений типа `real`

1. Вывод в форме с плавающей точкой в виде (1) происходит по умолчанию (без указания формата) и в формате с одним целым. Максимальная длина выводимого значения – 17. Если в формате указано больше, то число будет выведено в 17 позициях, а свободные позиции слева заполняются пробелами.

Значение формата, меньшее 17, используется для уменьшения количества цифр после десятичной точки. Их минимальное число – 1, то есть минимальная длина выводимого значения – 8. Если целое в формате меньше 8, то считается, что оно равно 8.

Например, оператор `write(-126.45 : 4)` выведет `-1.3E+02`. Вывод происходит с округлением.

2. Вывод в форме с фиксированной точкой происходит при использовании формата с двумя целыми. Первое целое, по-прежнему, – ширина поля вывода. Второе целое определяет количество знаков после десятичной точки. Число выводится с округлением. Если при этом ширина поля окажется недостаточной для вывода целой части, то поле будет расширено. Оператор `write(-126.465:4:1)` выведет `-126.47`.

13.3. Пример программы на Паскале

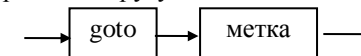
Операторы ввода, вывода и присваивания позволяют создавать линейные программы, то есть программы, в которых все операторы выполняются в порядке их описания. Следующая программа предназначена для нахождения суммы цифр трехзначного числа. В фигурных скобках записываются комментарии.

```
program digits_sum;
var n: 100 .. 999;           { для хранения трехзначного числа }
    a, b, c : 0 .. 9;       { для хранения цифр числа n }
begin
  write('Введите трехзначное число '); { подсказка пользователю }
  read(n);
  a := n mod 10               { a – младшая цифра n }
  b := n div 10 mod 10        { b – средняя цифра n }
  c := n div 100              { c – старшая цифра n };
  writeLn('Сумма цифр числа ', n, ' равна ', a+b+c);
end.
```

Для $n = 459$ будет выведено: «Сумма цифр числа 459 равна 18».

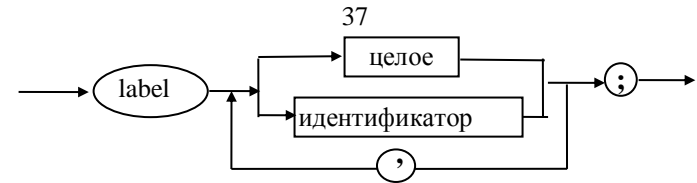
14. ОПЕРАТОР БЕЗУСЛОВНОГО ПЕРЕХОДА

Оператор безусловного перехода **goto** позволяет нарушить естественный порядок выполнения действий и передать управление из одной части программы в другую:



Выполнение этого оператора заключается в передаче управления оператору, помеченному меткой.

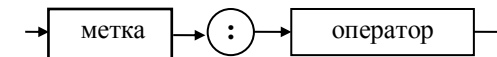
Метка – целое неотрицательное число от 0 до 9999, а в ТР метка может быть и идентификатором. Все метки описываются в разделе описания меток в виде



Порядок перечисления меток произвольный.

С помощью оператора **goto** нельзя передать управление из основной программы в подпрограмму и наоборот, нельзя передать управление в тело оператора цикла, оператору любой из ветвей условного оператора или переключателя.

Оператор, помеченный меткой, имеет вид



Оператор безусловного перехода в структурном программировании не используется.

15. ПУСТОЙ ОПЕРАТОР

Пустому оператору синтаксически не соответствуют никакие символы. Например,

```
read(x, y); ;a:=x+y;
```

Между оператором ввода и оператором присваивания находится пустой оператор.

Синтаксис не требует разделять точкой с запятой оператор и ключевое слово `end`; если она есть, это значит, что перед `end` стоит пустой оператор.

16. СТРУКТУРИРОВАННЫЕ ОПЕРАТОРЫ

Структурированные операторы – операторы, в состав которых входят другие операторы.

16.1. Составной оператор

В некоторых случаях синтаксис языка требует размещения одного оператора в какой-либо части конструкции, а по алгоритму там должно быть несколько операторов. Составной оператор позволяет рассматривать группу операторов как один оператор:

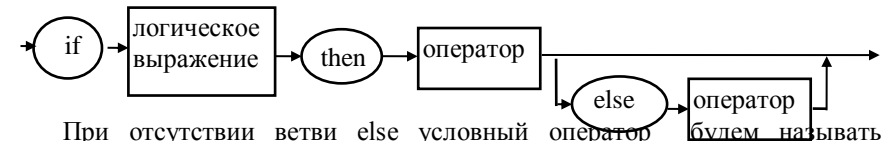


Ключевые слова **begin** и **end** называются *операторными скобками*.

16.2. Выбирающий оператор

В Паскале два вида выбирающего оператора: условный оператор и оператор переключатель.

1. **Условный оператор** используется при кодировании развилки, то есть для организации бинарного ветвления.



При отсутствии ветви else условный оператор будем называть *неполным*. После ключевых слов then и else синтаксис требует наличия только одного оператора. Если же по какой-либо ветви нужно выполнить несколько операторов, то следует использовать составной оператор. Перед else точка с запятой недопустима, так как в этом случае между then и else будут два оператора, один из которых пустой.

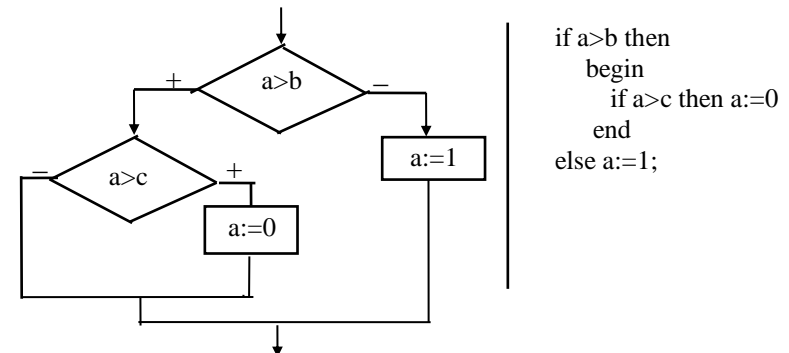
Если оператор, выполняемый по ветви then или else, является условным, то говорят о вложенных условных операторах. В случае вложенных условных операторов каждое else относится к ближайшему предшествующему then.

Примеры вложенных условных операторов (каждое else рекомендуется располагать под соответствующим then):

а) Оператор присваивает переменной max maximum{a, b, c}:

```
if a>b then if a>c then max:=a
              else max:=c
            else if b>c then max:=b
              else max:=c ;
```

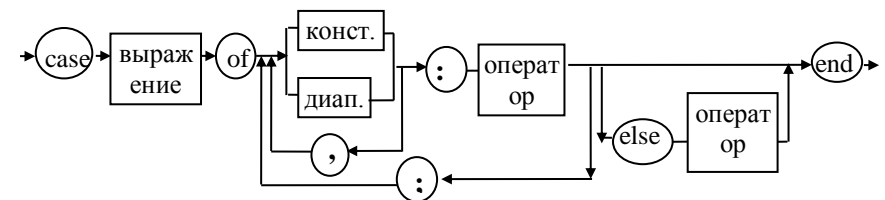
б) Оператор, эквивалентный приведенной слева развилке, имеет вид



В приведенном примере вложенный условный оператор – неполный, поэтому он заключен в операторные скобки. При отсутствии операторных скобок неполным будет внешний оператор.

2. **Оператор переключатель** предназначен для организации множественного ветвления.

Синтаксическая диаграмма переключателя:



Выражение после ключевого слова **case** называется *селектором*. Селектор должен иметь упорядоченный тип, кроме типа **longint**. Константы называются метками случаев. Они должны относиться к тому же типу, что и переключатель.

Работа оператора **case** заключается в следующем. Вычисляется значение селектора. Если это значение совпадает с одной из меток случаев, то выполняется оператор, записанный после нее. Если значение селектора не совпало ни с одной из меток случаев, то выполняется оператор, следующий за **else**. Если ветвь **else** отсутствует, то управление передается оператору, следующему за переключателем. Диапазоны и метки случаев не должны пересекаться.

В качестве примера рассмотрим программу для определения экзаменационной оценки по количеству набранных абитуриентом баллов. Оценка выставляется по правилам, приведенным в таблице:

Балл (b)	Оценка
$b \in [0; 4]$	2
$b \in [4.25; 7]$	3
$b \in [7.25; 9.25]$	4
$b \in [9.5; 10]$	5

Program ball_to_ mark;

```

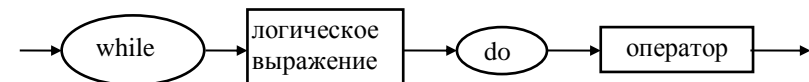
Var b: real;
    m: 2..5;
begin write ('Введите количество баллов ');
    read(b);
    case round (b*4) of {селектор должен иметь упорядоченный тип}
        0..16: m:=2; {границы диапазонов из таблицы умножены на 4}
        17..28: m:=3;
        29..37: m:=4
        else m:=5
    end;
    writeln(b, ' баллов – оценка ', m)
end.

```

16.3. Оператор цикла

В Паскале 3 вида оператора цикла: цикл с предусловием, цикл с постусловием и цикл с фиксированным числом шагов.

1. Цикл с предусловием:



Логическое выражение в заголовке цикла является условием возобновления цикла, то есть, пока оно истинно, выполняется оператор – тело цикла. Если при входе в цикл логическое выражение ложно, то тело цикла не выполнится ни разу.

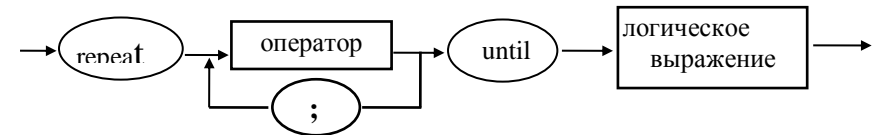
Опишем программу для нахождения наибольшего общего делителя двух чисел, используя алгоритм Евклида.

```

Program LCD;
var a, b, m, n, r : word;
begin
    write('Введите два натуральных числа. '); read(a, b);
    m := a; n:= b; {для сохранения исходных данных}
    r := a mod b;
    while r <> 0 do
        begin
            { Тело цикла – составной оператор, содержащий три оператора }
            a := b; b := r; r := a mod b
        end;
    writeln( 'НОД( ', m, ', ', n, ') = ', b)
end.

```

2. Цикл с постусловием:



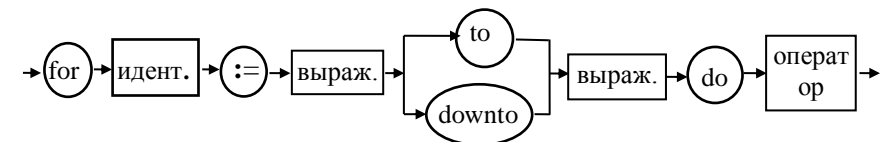
Телом цикла с постусловием является последовательность операторов между ключевыми словами **repeat** и **until**. Логическое выражение представляет собой условие выхода из цикла, то есть тело цикла выполняется до тех пор, пока значение логического выражения ложно. Как только оно станет истинным, происходит выход из цикла. Так как условие выхода из цикла проверяется после выполнения тела цикла, тело цикла с постусловием выполняется хотя бы один раз.

Опишем программу для решения следующей задачи. С клавиатуры вводится последовательность символов. Признак конца ввода – точка. Определить количество введенных цифр. Вводимую последовательность не хранить.

```

Program digits_sum;
Var ch: char; sum: word;
Begin writeln('Введите текст, оканчивающийся точкой. ');
      sum:=0;
      repeat
        read(ch);
        if (ch >='0') and (ch <= '9')
        then sum := sum + 1;
      until ch = '.';
      writeln(' Число цифр в тексте= ', sum)
end.
  
```

3. Цикл с фиксированным числом шагов:

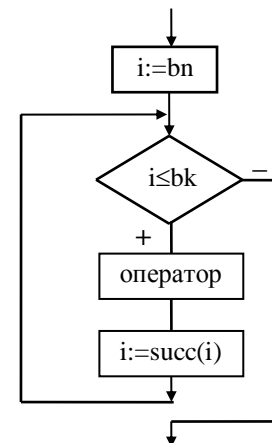


В приведенной диаграмме оператор представляет собой тело цикла. Перед телом цикла – заголовок цикла. Идентификатор после ключевого слова **for** называется *управляющей переменной* или *параметром цикла*. Управляющая переменная должна иметь упорядоченный тип, совместимый

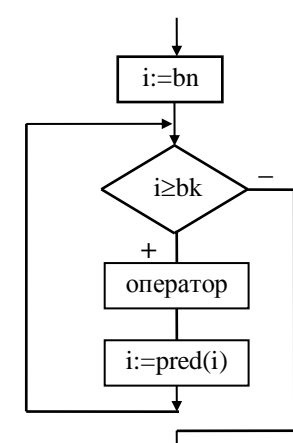
по присваиванию с первым (начальным) и вторым (конечным) выражениями в заголовке цикла. Следовательно, начальное и конечное выражения также должны быть упорядоченного типа.

Значения начального и конечного выражений вычисляются один раз при входе в цикл. Цикл **for** является циклом с предусловием. При входе в цикл управляющей переменной (назовем ее **i**) присваивается значение начального выражения (обозначим его **bn**). Затем **i** сравнивается со значением конечного выражения (обозначим его **bk**). Если $i \leq bk$ для цикла с ключевым словом **to** или $i \geq bk$ для цикла с ключевым словом **downto**, то выполняется тело цикла. После чего изменяется значение **i**: для цикла с ключевым словом **to** $i := \text{succ}(i)$, для цикла с ключевым словом **downto** $i := \text{pred}(i)$. И снова **i** сравнивается с **bk**. Если условие возобновления цикла не выполняется, то происходит выход из цикла. Алгоритм работы цикла **for** можно описать блок-схемами.

1) for i:= bn to bk do <оператор>



2) for i:= bn downto bk do <оператор>



В теле цикла **for** *нельзя* изменять параметр цикла. Так как начальное и конечное выражения вычисляются только один раз при входе в цикл, то изменение **bn** и **bk** на количество повторений не влияет. После выхода из цикла **for** значение параметра цикла считается неопределенным.

Пример 1. Последовательность Фибоначчи задается рекуррентно:

$$f_1=1, f_2=1,$$

$$f_i = f_{i-2} + f_{i-1} \quad \text{для } i > 2.$$

Определить n -й член последовательности.

Программа для нахождения f_n :

Program Fibonacci;


```

Var f1, f2, f3: longint; i, n: word;
Begin write('Введите n'); read(n);
  f1 := 1; f2 := 1;
  for i := 3 to n do
    begin
      f3 := f1+f2; f1 := f2; f2 := f3
    end;
  writeln(n, '-ое число Фибоначчи=', f2)
end.

```

17. СТИЛЬ ЗАПИСИ ПРОГРАММЫ

При создании понятных, удобочитаемых программ большое значение имеет стиль записи программы. Стандарта для стиля записи нет, но можно дать некоторые рекомендации:

1. Основные части программы разделять пустыми строками.
2. Служебные слова, которыми начинается и заканчивается оператор, располагать одно под другим, начиная с одной позиции по вертикали.
3. Операторы одного уровня вложенности начинать с одной позиции по вертикали.
4. Вложенные операторы записывать с некоторым смещением вправо.
5. Включать в программу комментарии.

17.1. Комментарии

В программе комментарии заключаются в фигурные скобки. Их можно помещать в любой части программы, где допустим хотя бы один пробел.

Комментарии компилятором не обрабатываются, на размер программного кода не влияют. Комментарии дают возможность применять метод пошаговой детализации. Во время разработки программы комментарий соответствует еще не определенным процессам. Среди комментариев выделяют: *вводный комментарий, комментарии-заголовки и построчные комментарии.*

Вводный комментарий помещается в начале текста программы и содержит следующую информацию:

- 1) назначение программы;
- 2) сведения об авторе;
- 3) название организации, в которой создана программа;
- 4) дату написания программы;
- 5) используемый метод решения;
- 6) указание по вводу-выводу;
- 7) для программ производственного характера:

- а) объем занимаемой памяти;
- б) время выполнения программы;
- в) указания оператору ЭВМ.

Комментарии-заголовки служат для выделения и указания назначения основных частей программы.

Построчные комментарии поясняют особенности отдельных фрагментов программы и назначение переменных, имена которых не являются мнемоническими.

Программа, содержащая такую информацию, называется *самодокументированной*.

Программы производственного характера считаются хорошими, если их текст содержит не менее 30% комментариев.

18. ОТЛАДКА ПРОГРАММ

Для успешного выполнения программы необходимо выполнение двух требований:

1. Программа должна быть *правильной*. Правильность программы определяется степенью соответствия формальным требованиям, то есть спецификации программы.

2. Входные данные должны быть правильными.

Процесс нахождения и устранения ошибок в программе называется *отладкой*.

Процесс поиска ошибок во входных данных называется *проверкой корректности входных данных*.

18.1. Виды ошибок и способы их устранения

Выделяют три типа ошибок: *ошибки компиляции*, *ошибки времени выполнения* и *логические ошибки*.

Ошибки компиляции являются синтаксическими ошибками. Они выдаются при компиляции программы. Программа с синтаксическими ошибками не может быть выполнена.

Ошибки времени выполнения связаны с невозможностью по какой-либо причине выполнить очередное действие, что приводит к аварийному останову программы. Такие ошибки возникают, например, при делении на 0, вычислении логарифма с отрицательным аргументом, переполнении стека при выполнении рекурсивной подпрограммы и т.п.

Логические ошибки возникают при неправильном проектировании алгоритма или по невнимательности при записи или наборе программы. Программа с логическими ошибками может быть выполнена до конца,

возможно даже, что при некоторых наборах исходных данных результаты будут правильными.

В процессе устранения логических ошибок можно выделить три этапа:

1. установление факта существования ошибки;
2. локализация ошибки;
3. устранение ошибки.

Существуют 2 метода обнаружения ошибок:

1. статическая (ручная) проверка, которая заключается в анализе программы без выполнения ее на ЭВМ;
2. тестирование – прогон на ЭВМ.

Оба метода требуют наборов тестовых данных, которые должны подбираться параллельно с разработкой алгоритма. В число наборов тестовых данных рекомендуется включить несколько типичных, среди них должны быть корректные и некорректные данные. Тестовые данные должны охватывать предельные случаи. Если в программе есть разветвления, то необходимы наборы тестовых данных для каждой ветви. Если в программе есть циклы, то данные нужно подобрать так, чтобы цикл выполнялся минимально и максимально возможное число раз.

При отладке программ для практического использования нужно подобрать дополнительные тестовые данные:

1. Получить реальные данные у потенциального пользователя.
2. Порождать случайным образом наборы тестовых данных.

18.2. Ручная проверка

Нужно руководствоваться правилом: чем раньше обнаружена ошибка, тем легче ее исправить. Поэтому при разработке алгоритма и записи программы необходимо следить за инициализацией всех переменных в программе, необходимо устанавливать правильное завершение циклов, избегать заикливания. Какой бы простой ни была программа, рекомендуется выполнить трассировку с записью результатов на каждом шаге.

Статистика утверждает, что 70% ошибок можно устранить на этапе ручного тестирования.

18.3. Машинное тестирование

Известный специалист в области программирования Дейкстра считает, что тестирование доказывает наличие ошибок, а не их отсутствие.

Выделяют два вида тестирования: разрушающее и диагностическое:

1. *Разрушающее тестирование* производится над программой, которая считается правильной, с целью заставить ее дать сбой.

2. *Диагностическое тестирование* выполняется с целью локализации ошибки, если известно о ее существовании.

В ТР существуют системные средства для отладки программ. Они позволяют выполнить трассировку с выводом значений интересующих переменных и выражений, устанавливать точки останова, точки прерывания (контрольные точки).

18.4. Проверка правильности данных

Правильно написанная программа должна:

1. при вводе некорректных данных выдавать сообщение, указывающее на некорректный элемент данных и сообщать о причине некорректности;
2. произвести как можно больше правильных вычислений;
3. встретив некорректный элемент данных, проверить остальные данные.

18.5. Исправление ошибок

1. Перед всяким устранением ошибки нужно подумать, к чему это приведет.
2. После того как в программу внесены изменения, нужно заново ее протестировать.

Устойчивая программа – это программа, которая порождает правильные результаты во всех случаях, когда это возможно, иначе – указывает причину, по которой программа не может быть выполнена. Всякая практически используемая программа меняется в течение своей жизни. Изменения включают в себя исправление логических ошибок, выявленных при эксплуатации программы, и модификацию программы в соответствии с новыми требованиями. На эту деятельность, которая называется *сопровождением*, программисты тратят больше времени, чем на создание программы. Правильная программа, после внесения в нее изменений, может перестать быть правильной. Поэтому программа должна быть предельно ясной и понятной тому, кто занимается сопровождением, структура программы должна быть такой, чтобы ее легко было модифицировать, не лишая устойчивости.

19. РЕГУЛЯРНЫЙ ТИП (МАССИВ)

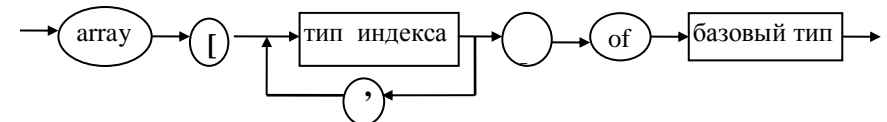
Чтобы хранить и обрабатывать сложные виды информации, нужно научиться строить нетривиальные структуры данных. Такие структуры

создают путем объединения данных, в конечном итоге, простых типов. Составные части структурированных типов называются их компонентами.

В TP существует ограничение на объем памяти, занимаемой значением любого структурированного типа. Этот объем не должен превышать 65520 байтов. BoFree Pascal такого ограничения нет.

Одним из структурированных типов во многих алгоритмических языках является *регулярный тип (тип массив)*. Массив представляет собой последовательность фиксированного числа компонентов одного типа. Тип компонентов массива называется *базовым* типом.

Описание массива:



Количество типов индекса, перечисленных в квадратных скобках, определяет *размерность* массива. *Размер* массива – это общее число его компонентов.

Тип индекса должен быть упорядоченным типом, он определяет число элементов по соответствующему измерению. Чаще всего для типа индекса используют интервальный тип, реже – перечисляемый или стандартный. Тип индекса не может быть четырехбайтовым.

Базовый тип может быть любым.

19.1. Одномерные массивы

Если указан только один тип индекса, массив является *одномерным*.

Примеры описаний одномерных массивов:

```
const MAXLEN=100; {константа удобна для изменения размера массива}
type  t_vect=array[1.. MAXLEN] of integer;
var   a : t_vect;      {для хранения целочисленных последовательностей}
       b : t_vect;
       c, d : array[1.. MAXLEN] of integer;
       point : array[(x, y, z)] of real;   {для хранения координат точки или
                                           вектора}
       flag : array['A'..'Z'] of boolean;
```

Массивы **a**, **b**, **c**, **d** имеют одинаковые размеры (=100). Размер массива **point** равен 3, **flag** – 26.

Компоненты (элементы) массива занимают последовательные ячейки памяти. Объем памяти, выделяемой массиву, равен произведению размера массива на объем, занимаемый элементом. Переменная **a** занимает $100*2=200$ байтов, **point** – $3*6=18$ байтов, **flag** – $26*1=26$ байтов.

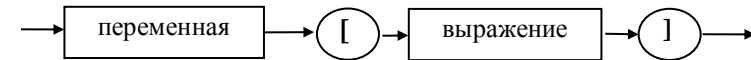
Следует обратить внимание на то, что типы переменных **a** и **c** компилятор считает разными. Важным понятием в Паскале является понятие *тождественности типов*. Переменные имеют *тождественные типы*, если они определены в одной секции или через одно и то же имя типа.

Согласно приведенным выше описаниям, типы переменных **a** и **b** тождественны, так как переменные описаны с использованием имени типа **t_vect**. Переменные **c** и **d** тоже имеют тождественные типы, так как описаны в одной секции.

Над массивами *любой размерности* как над единичными целыми не определены никакие операции. Разрешено присваивание переменной типа массив значения переменной тождественного типа, то есть *для совместимости массивов по присваиванию требуется тождественность типов*.

Если массивы **a** и **d** инициализированы, то допустимы операторы **b:=a** и **c:=d**.

Имя массива является общим именем совокупности компонентов (элементов) массива. Обращение к отдельным компонентам массива осуществляется с помощью переменных с индексами:



Выражение в квадратных скобках должно иметь тип индекса. Например, **a[1]**, **d[MAXLEN div 2]**, **point[x]**, **point[y]**, **flag['X']**, **flag[succ('G')]**.

Переменные с индексами можно использовать в любом месте программы, где допустим базовый тип. Ввод и вывод массива производится покомпонентно, обычно в цикле **for**.

Пример. Составим программу для определения максимального члена данной целочисленной последовательности, длиной не больше 100, и номера этого элемента:

```

Program max_elem;
const MAXLEN=100;
var v : array [1.. MAXLEN] of integer;
    max:integer;           { для максимального значения }
    n : 1.. MAXLEN;        { n - длина последовательности }
    imax, i : 1.. MAXLEN; { imax - для номера максимального члена }
Begin
  write('Введите длину последовательности ≤ ', MAXLEN);
  read (n);
  write('Введите члены последовательности ');
  for i:=1 to n do    read(v[i]);

```

```

{Поиск      максимального  элемента}
max := v[1]; imax := 1;           {инициализация переменных}
for i:=2 to n do
  if v[i] > max then
    begin                          {запоминаем большее значение}
      max := v[i];
      imax := i
    end;
  writeln(' Максим. член последовательности - v[', imax, '] = ', max)
End.

```

19.2. Упакованные массивы

Если в одном машинном слове можно разместить несколько элементов массива, то для экономии памяти можно использовать упакованный массив. Для этого при описании массива перед ключевым словом **array** следует указать ключевое слово **packed**.

Например, при описании

```
var a : packed array [1..10] of 0..7;
```

в двухбайтовом машинном слове размещается 5 элементов, для массива потребуется 2 слова, то есть 4 байта.

При работе с упакованными массивами усложняется доступ к отдельным элементам массива. В используемых реализациях массивы упаковываются автоматически, слово **packed** можно не указывать.

19.3. Многомерные массивы

Если количество типов индекса в описании массива равно **n**, то массив называют **n**-мерным. Формально размерность массива не ограничена, но фактически она зависит и от размера базового типа, и от ограничения на объем памяти структурированных типов.

Для представления матриц используются двумерные массивы. Тип первого индекса определяет число строк, а тип второго индекса – число столбцов матрицы. Например,

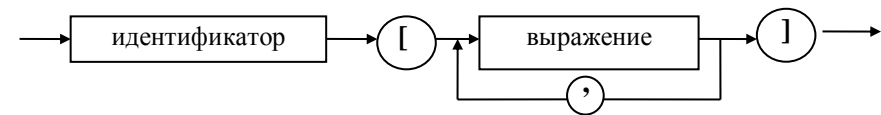
```

const ROW=20;
      COL=10;
type  t_matr=array[1..ROW, 1..COL] of word;
var   a : t_matr;
      d: array[1..2, 1..3, (x, y)] of real;

```

Тип `t_matr` – двумерный массив размером 20×10 . Его можно использовать для работы с матрицами порядка не больше чем 20×10 . Переменная `d` – трехмерный массив.

Обращение к элементам многомерных массивов:



Идентификатор – имя массива. Число выражений (индексов) в квадратных скобках должно быть равно размерности массива. Например, `a[3, 4]`, `d[2, 1, y]`.

В памяти двумерные массивы располагаются по строкам. В общем случае элементы многомерных массивов размещаются так, что чем правее индекс, тем быстрее он меняется. Так, элементы массива `d` размещаются в памяти в следующей последовательности:

`d[1, 1, x]`, `d[1, 1, y]`, `d[1, 2, x]`, `d[1, 2, y]`, `d[1, 3, x]`, `d[1, 3, y]`,
`d[2, 1, x]`, `d[2, 1, y]`, `d[2, 2, x]`, `d[2, 2, y]`, `d[2, 3, x]`, `d[2, 3, y]`.

Для ввода и вывода многомерных массивов используются вложенные циклы. В качестве примера приведем фрагмент программы вывода в виде таблицы описанного выше массива `a` (`i` и `j` – целочисленные переменные):

```
for i := 1 to ROW do      { i – номер строки }
begin
  writeln;                { переход к новой строке }
  for j := 1 to COL do    { j – номер столбца }
    write( a[i, j] : 7)
end;
```

19.4. Еще один способ получения многомерных массивов

В Паскале нет ограничений на базовый тип массива. Базовым типом одномерного массива может быть, в частности, одномерный массив. Например,

```
const MaxLen = 100;
type t_xyz = (x, y, z);
  t_point = array[t_xyz] of real; { тип для координат точки или вектора }
  t_points = array [1..MaxLen] of t_point; { тип для координат 100 точек }
var p: t_points;
```

С переменной `p` можно работать как с двумерным массивом. `p[i]` – одномерный массив координат `i`-й точки, содержащий 3 элемента: `p[i][x]`,

$p[i][y]$, $p[i][z]$ (i – целое от 1 до 100). Индексы в ТР можно записывать не в отдельных скобках, а перечислить через запятую: $p[i, x]$, $p[i, y]$, $p[i, z]$.

Использование многомерного массива, описанного как массив массивов меньшей размерности, удобно, если предполагается обрабатывать массивы меньшей размерности как единые целые. Например, в массиве точек **p** обмен **i**-й и **j**-й точек выполняет фрагмент программы

```
t := p[i]; p[i] := p[j]; p[j] := t; (t – переменная типа t_point).
```

При описании `var p : array[1..MaxLen, t_xyz] of real;` аналогичный

фрагмент имеет вид:

```
t := p[i, x]; p[i, x] := p[j, x]; p[j, x] := t;
t := p[i, y]; p[i, y] := p[j, y]; p[j, y] := t;
t := p[i, z]; p[i, z] := p[j, z]; p[j, z] := t;
```

19.5. Строковый тип в стандартном Паскале

Строкой называется последовательность символов. В стандартном Паскале нет специального типа для работы со строками переменной длины. Для работы со строками можно использовать упакованные символьные массивы с типом индекса $1..n$, где n – константа. Например,

```
var s1 : packed array [1..10] of char; {В ТР packed не обязательно}
    s2 : packed array [1..10] of char;
```

Использование упакованных символьных массивов дает некоторые преимущества по сравнению с обычными одномерными массивами:

1. Для совместимости по присваиванию не требуется тождественность типов, достаточно равенства размеров массивов.
2. Строковой переменной можно присвоить строковую константу такой же длины.
3. Определены операции сравнения строк одинаковой длины.

Из двух строк равной длины та больше, у которой первый из неравных символов больше.

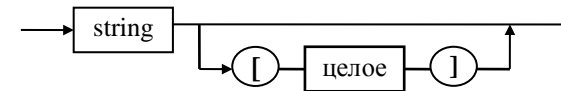
Таким образом, для переменных $s1$ и $s2$ допустимы присваивания

```
s1 := 'студент'; s2 := s1; s2[8] := 'ы';
```

В первом операторе строковая константа дополнена тремя пробелами до длины 10. После выполнения этих операторов $s2 = \text{'студенты'}$ (два последних символа – пробелы), и выражение $s1 < s2$ имеет значение true, так как первый из неравных символов строки $s1$ (пробел) меньше соответствующего символа ('ы') строки $s2$.

19.6. Строковый тип в ТР

В TP есть специальный тип для работы со строками переменной длины. Описание этого типа:



Целое определяет максимальную длину строки. Оно должно принадлежать промежутку [1; 255]. Например,

```
const n=10;
var s : string[n];
```

Переменной *s*, согласно описанию, выделяется *n* + 1 байт. Переменную *s* можно рассматривать, как символьный массив с типом индекса **0..n** и использовать для работы со строками длиной не более *n*. Собственно строка начинается с символа *s*[1]. Нулевой элемент массива резервируется для хранения информации о динамической длине строки. В нем хранится символ с кодом, равным динамической длине строки. Если *s*[0] есть символ с кодом 0 (#0), то строка *s* не содержит символов. Будем называть такую строку *пустой*. Динамическая длина определяется выражением *ord(s[0])*. Кроме этого, динамическую длину строки возвращает стандартная функция *length(s:string): byte*.

Тип **string** (без указания размера) эквивалентен типу **string[255]**. Длина строки при этом может изменяться от 0 до 255.

Ввод и вывод строк осуществляется с помощью процедур *read[ln]* и *write[ln]*. Следует обратить внимание на то, что при вводе в строку заносятся последовательно все символы из буфера ввода до символа перехода к новой строке. Сам символ перехода к новой строке остается в буфере, и с него начнется следующее считывание. Поэтому одним оператором *read* несколько строковых переменных ввести нельзя. При выполнении оператора *read(s1, s2)* строка *s2* всегда будет пустой. Последовательный ввод строк можно выполнить, обращаясь к процедуре *readln* с одним параметром:

```
readln(s1); readln(s2); readln(s3);
```

Строковый тип TP совместим по присваиванию:

- 1) с любым типом *string*;
- 2) с упакованным символьным массивом (равенство длин не требуется);
- 3) с символьным типом.

Если строковой переменной присваивается значение, превышающее длину строковой переменной, то перед присваиванием происходит усечение присваиваемого значения.

Оператор инициализации строки *s* пустой строкой: *s:='';*. В правой части оператора присваивания два апострофа.

В TP определена бинарная операция *конкатенация* (соединение строк). Знак этой операции '+'. Операндами могут быть символы, упакованные символьные массивы и строки. Результатом является строка, полученная дописыванием в конец первого операнда второго операнда.

Пример 1.

```
var ch : char;
    ch_arr : array[1..3] of char;
    s1 : string;
    s2 : string[8];
begin
    ch := 'W'; ch_arr := 'ord';
    s1 := '_and_byte';
    s2 := ch+ch_arr+s1;
    writeln(ch+ch_arr+s1);
    writeln(s2)
end.
```

В результате выполнения программы примера 1 будет выведено

```
Word_and_byte
Word_and
```

Первая строка вывода – результат конкатенации, вторая – значение s2, которое получено в результате конкатенации с последующим усечением до 8 символов.

Над строковым типом определены все операции сравнения, причем если один из операндов имеет строковый тип, то другой может быть строковым, символьным или упакованным символьным массивом. Примеры истинных отношений: 'abcwt' < 'abd', 'abc' > 'ab'.

Пример 2. Словом будем называть последовательность символов, не содержащую пустых символов. К пустым символам отнесем пробел (#32), символ табуляции (#9) и переход к новой строке (#10). Можно считать, что слово не содержит не только пустых, но и любых управляющих символов, то есть символов, меньших #33. Опишем программу для определения количества слов в данной строке:

```
Program words_number;
var str : string;
    k, i : byte; {k – счетчик числа слов, i – параметр цикла}
begin
    writeln('Введите строку'); ReadLn(str);
```

```

{инициализация счетчика}
if str[1]>#32 then k:=1 {первый символ – начало слова}
                else k:=0; {строка начинается пустым символом}
for i:=2 to length(str) do
  if (str[i-1]<#33) and (str[i]>#32) {если начало слова }
    then k:=k+1;
writeln ('Число слов строки ',str,' равно ',k);
end.

```

Если, работая со строкой, как с символьным массивом, приходится изменять длину строки, необходимо позаботиться об изменении значения нулевого байта.

Пример 3. Программа удаления из строки символов, равных данному:

```

Program symb_del;
var str : string;
    ch : char;
    len, i, j : byte;
begin
  writeln('Введите строку'); ReadLn(str);
  writeln('Введите символ'); ReadLn(ch);
  len := length(str); {Запоминаем длину исходной строки}
  j := 0; {j - число сохраненных символов в преобразованной строке}
  for i:=1 to len do {Просматриваем все символы строки}
    if str[i]<>ch {Символ остается в строке}
    then
      begin j := j+1; {Увеличиваем число сохраненных символов}
          str[j] := str[i] {Сохраняем символ}
        end;
  str[0] := chr(j); {Устанавливаем новую длину строки}
  write (' Преобразованная строка: ', str);
end.

```

20. ПОДПРОГРАММЫ

Один из принципов структурного программирования заключается в разбиении задачи на достаточно самостоятельные подзадачи. Для решения каждой подзадачи можно разработать свой алгоритм и записать его в виде самостоятельного программного блока, называемого *подпрограммой*. Подпрограммы имеют имена, и обращаться к ним можно по именам.

Подпрограммы позволяют избегать многократной записи одной и той же последовательности действий. В языке Паскаль существует два вида подпрограмм: *процедуры* и *функции*.

Программа, содержащая подпрограммы, проще для создания, легче для понимания, удобнее для использования.

При разработке программ, содержащих подпрограммы, необходимо уметь описывать подпрограммы и обращаться к ним.

Подпрограммы, как и другие программные объекты, должны быть описаны в разделе описаний. Структура описания подпрограммы такая же, как и структура программы, но в описании подпрограммы заголовок обязателен и описание подпрограммы заканчивается символом ';', а не точкой.

20.1. Область действия описаний

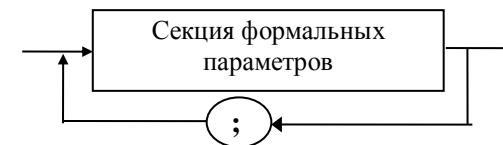
Область действия описания программного объекта – часть программы, где можно обращаться к этому программному объекту. Область действия описания распространяется от точки описания до конца блока, в котором выполнено это описание.

Программные объекты, описанные в блоке, содержащем подпрограмму до ее описания, доступны подпрограмме и являются по отношению к ней *глобальными*. Программные объекты, описанные в теле подпрограммы, называются *локальными*. Имена локальных программных объектов могут совпадать с именами глобальных, такие программные объекты в подпрограмме имеют смысл, соответствующий локальному описанию.

20.2. Параметры подпрограмм

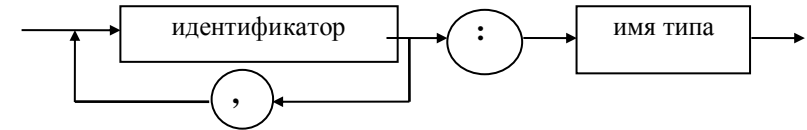
Обмен информацией между подпрограммой и вызывающей ее программой осуществляется через параметры и глобальные переменные. Но в подпрограммах не рекомендуется использовать глобальные переменные. В этом случае изменение программы может привести к неожиданным последствиям при выполнении подпрограммы, и наоборот. Программы и подпрограммы будут максимально независимыми, если обмен информацией происходит в четко определенных границах, только через параметры.

Передаваемые подпрограмме параметры описываются в списке формальных параметров заголовка подпрограммы, который имеет вид

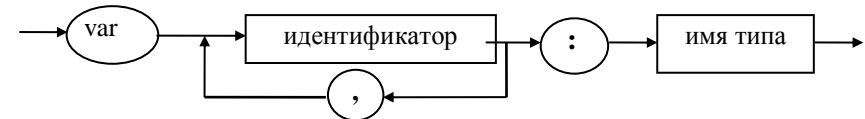


В стандартном Паскале существует 2 вида формальных параметров: параметры-значения и параметры-переменные. В ТР, кроме этого, есть нетипизованные параметры-переменные и параметры-константы.

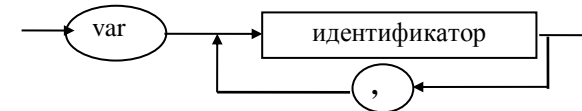
Описание секции параметров-значений:



Описание секции параметров-переменных:



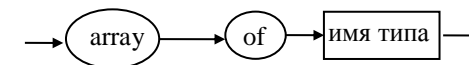
Описание секции нетипизованных параметров-переменных:



Описание секции параметров-констант:



Следует обратить внимание на то, что при описании формальных параметров указывается *имя типа*. В последних версиях ТР, кроме имени типа, можно использовать конструкцию вида



Формальные параметры такого типа называются *открытыми массивами*. Описание типа здесь *недопустимо*. Если предполагается передавать параметры нестандартных типов, необходимо предварительно дать типам имена.

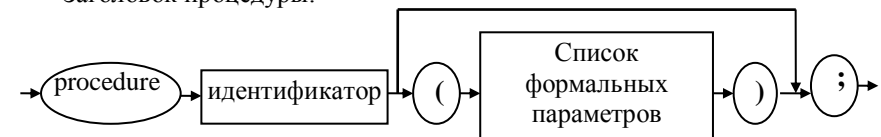
Параметры могут передаваться подпрограмме для *чтения*, *записи* или *чтения и изменения*. В список формальных параметров включаются параметры, через которые подпрограмме будут переданы исходные данные (это параметры для чтения), и параметры, в которые будут записаны результаты (это параметры для записи). Иногда подпрограмма должна использовать значение, передаваемое ей через параметр, а затем в этот же

параметр записать результат (это параметр для чтения и изменения). В Паскале параметры любого вида доступны подпрограмме для чтения. Для записи доступны параметры-переменные (в том числе и нетипизованные). Как бы мы ни изменяли параметр-значение в теле подпрограммы, после выхода из подпрограммы он будет иметь первоначальное значение. Поэтому параметры-значения после использования переданной через них информации можно использовать как локальные переменные. Значения параметров-констант изменять нельзя. Значения, присвоенные параметрам-переменным в подпрограмме, сохраняются после выхода из подпрограммы.

20.3. Процедуры

Процедура – это подпрограмма, предназначенная для решения подзадачи, необязательно связанной с вычислением значений. Процедуры можно использовать для ввода или вывода данных структурированных типов, вычисления нескольких значений, преобразования данных и т.п.

Заголовок процедуры:



Идентификатор после ключевого слова **procedure** – имя процедуры.

Пример 1. Описание процедуры для решения квадратного уравнения $ax^2 + bx + c = 0$:

```

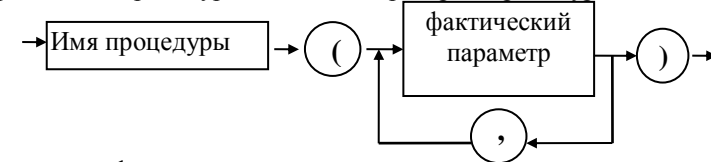
Procedure sqr_equation(a,b,c : real; var x1,x2 : real; var flag:boolean);
  {a, b, c – исходные данные (коэффициенты квадратного уравнения),
  параметры-значения; flag – параметр для записи: true – есть корни, false
  – нет корней; x1 и x2 – параметры-переменные для записи корней}
  var d :real; { дискриминант}
  begin
    d := b*b-4*a*c;
    flag := d>=0
    if d>0 then
      begin
        d := sqrt(d);
        x1 := (-b+d)/(2*a); x2 := (-b-d)/(2*a)
      end
    else if abs(d)<1E-10 then
      begin
        x1:=-b/(2*a); x2 := x1
      end
  end

```

end
end; {Конец описания процедуры.}

20.4. Обращение к процедурам (вызов процедур)

Обращение к процедуре является оператором процедуры и имеет вид:



Количество фактических параметров должно равняться количеству формальных параметров. Формальные параметры-значения должны быть совместимы по присваиванию с соответствующими им фактическими параметрами. Последние представляют собой выражения, значения которых присваиваются формальным параметрам при вызове процедуры. Фактический параметр, соответствующий формальному параметру-переменной, представляет собой переменную, тип которой тождественен типу формального параметра. Фактические параметры в списке разделяются запятыми.

Пример 2. Программа, предназначенная для решения квадратного уравнения вида $(p+q)x^2 + (p-q)x + 1 = 0$, где p и q – вещественные числа:

```
{Здесь должно быть описание процедуры sqr_equation примера 1}
var p, q, x1, x2 : real;
    roots_are : boolean;
begin
    writeln('Введите p и q'); read(p, q);
    sqr_equation(p+q, p-q, 1, x1, x2, roots_are);
    if roots_are then writeln('x1=', x1, ' x2=', x2)
        else writeln('корней нет')
    end.
end.
```

Если параметр должен быть доступен подпрограмме для чтения, то он может быть описан и как параметр-переменная, и как параметр-значение, но использование параметров-значений простых типов имеет преимущества:

- 1) менее жесткое ограничение на типы – требуется только совместимость по присваиванию, а не тождественность;
- 2) фактический параметр является выражением, необязательно переменной;
- 3) информация не может быть испорчена через этот параметр, то есть даже если этот параметр является переменной, то его значение при выходе из подпрограммы остается тем же, что и при входе;

4) параметры-значения в теле подпрограммы можно использовать как локальные переменные.

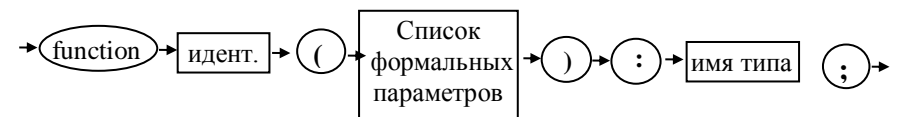
Во время обращения к подпрограммам с фактических параметров-значений снимаются копии, они помещаются в некоторую область памяти, называемую *стеком*, и подпрограмма работает с этими копиями. К параметрам-переменным и параметрам-константам подпрограмма обращается по адресам, в стек помещаются адреса параметров. Поэтому, если подпрограмме нужен *только для чтения* параметр структурированного типа, занимающий большой объем памяти, то рекомендуется его передавать как параметр-константу.

Все изменения параметров-переменных в теле подпрограммы сохраняются при возвращении в вызывающую программу.

20.5. Функции

Функция – это подпрограмма, в результате выполнения которой происходит вычисление значения скалярного или строкового типа.

Заголовок функции имеет вид



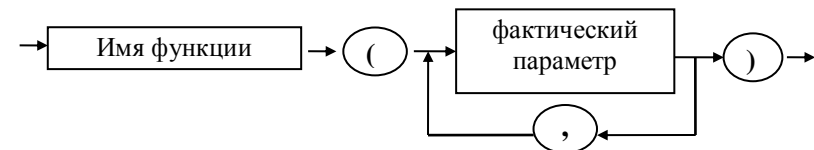
В разделе операторов описания функции должен быть хотя бы один оператор присваивания:



Значение последнего выражения, присвоенного имени функции, будет возвращено в точку вызова функции. Но использовать имя функции в качестве переменной, которой присвоено значение, недопустимо.

20.6. Обращение к функциям (вызов функций)

Синтаксически вызов функции не отличается от вызова процедуры:



Но вызов функции не является оператором. Обратиться к функции можно всюду, где допустимо использование значения типа, возвращаемого

функцией. Обращение к функции может быть операндом в выражении, фактическим параметром-значением.

Пример 3. Программа для проверки, упорядочена ли по невозрастанию данная целочисленная последовательность длиной не больше 100:

```
Program sort_check;
const MaxLen=100;
type t_range = 1.. MaxLen;
      t_vect= array[t_range] of integer;
function is_sort(a : t_vect; n : t_range): boolean;
{Логическая функция проверяет, упорядочена ли по невозрастанию
последовательность а длины n}
var i : t_range;
begin i:=2;
      while (i<=n) and (a[i-1]>=a[i]) do
        i := i+1;
      is_sort := i>n
end;
var i, n : t_range; a : t_vect;
begin write('Введите длину последовательности≤', MaxLen); read(n);
      writeln('Введите члены последовательности');
      for i := 1 to n do
        read(a[i]);
      if is_sort(n, a) then writeln(' Последовательность упорядочена')
        else writeln(' Последовательность неупорядочена')
end.
```

20.7. Побочный эффект функций

Основное назначение функции – возвращение значения в точку вызова, но функция так же, как и процедура, может изменять значения своих параметров-переменных. Такие функции называются функциями с *побочным эффектом*.

Пример 4. Описание логической функции поиска элемента, равного **x**, в целочисленном массиве **a** длиной **n**≤100. В качестве побочного эффекта параметру **i** присваивается номер первого из элементов, равных **x**. Используем типы, описанные в примере 3:

```
function el_search(a: t_vect; n: t_range; x: integer; var i: t_range): boolean;
begin i:=1;
      while (i<=n) and (x<>a[i]) do
        i := i+1;
      el_search:= i<=n
end;
```

20.8. Рекурсивные подпрограммы

В математике рекурсией называется способ описания функций или процессов через самих себя. Например,

$$n! = \begin{cases} 1 & \text{при } n = 0, \\ n \cdot (n-1)! & \text{при } n \in N. \end{cases}$$

В некоторых языках программирования, в том числе и в Паскале, допустимо, чтобы подпрограмма вызывала себя. Такие подпрограммы называются *рекурсивными*.

Рекурсивная подпрограмма обязательно удовлетворяет двум требованиям:

- 1) имеет нерекурсивный выход;
- 2) при каждом рекурсивном обращении задача упрощается, приближаясь к нерекурсивному решению.

При решении некоторых задач можно использовать как рекурсивный, так и итеративный алгоритм. Преимущество рекурсивных подпрограмм заключается в простоте написания, легкости понимания. Обычно это касается задач, связанных с процессами, рекурсивными по своей природе. Но рекурсивные алгоритмы, как правило, менее эффективны из-за затрат времени на организацию стека. При каждом обращении в стеке запоминаются значения всех локальных переменных, параметров и коды возврата. Глубина рекурсии (количество обращений к себе) ограничена объемом памяти стека. При глубокой вложенности рекурсии может произойти переполнение стека.

Пример 5. Опишем рекурсивную и итеративную функцию для вычисления $n!$:

{Итеративная}	{Рекурсивная}
<pre>function fact(n:byte):longint; var i:byte; f : longint; begin f:=1; for i:=n downto 2 do f:= f*i; fact:=f end;</pre>	<pre>function fact(n:byte):longint; begin if n=0 then fact:=1 else fact:=n*fact(n-1) end;</pre>

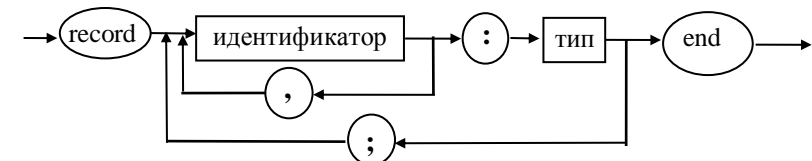
Две подпрограммы называются *взаимно рекурсивными*, если первая подпрограмма обращается ко второй, а вторая – к первой. Обычное описание таких подпрограмм невозможно, так как при этом вызов подпрограммы будет предшествовать ее описанию. Противоречие разрешается использованием *опережающего описания*. Описывается заголовок одной из подпрограмм, а тело ее заменяется ключевым словом **forward**. Затем описывается другая подпрограмма полностью, а после нее – неполный заголовок (без указания параметров) и тело первой подпрограммы:

```
Program pr1(x:real);
  forward;
procedure pr2(...);
  {описание тела с вызовом pr1}
procedure pr1;
  { описание тела с вызовом pr2 };
```

21. КОМБИНИРОВАННЫЙ ТИП (ЗАПИСЬ)

Комбинированный тип данных или тип запись – это структурированный тип, состоящий из компонентов, типы которых могут быть различны. Компоненты записей называются полями. Записи удобно использовать для хранения и обработки различных характеристик одного и того же объекта. Например, отдел кадров хранит и поддерживает в актуальном состоянии информацию о каждом сотруднике: фамилия, имя, отчество, пол, дата рождения, адрес, образование, оклад и т.п. Записи – основная структура данных, используемая в информационных системах.

Описание записи:



Идентификаторы – имена полей. Типы полей могут быть любыми.

Пример 1.

```
Const MaxLen = 25;
type t_range=1.. MaxLen;
   t_date=record      { Тип для работы с датами}
       day : 1..31;
```

```

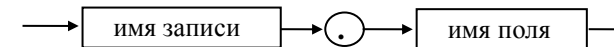
        month : 1..12;
        year : 0..9999
    end;          {end записано под соответствующим record}
t_student=record {Тип для хранения информации о студенте}
    name : string[20];
    birthday : t_date; {Поле-запись}
    group : string[5];
    marks : array[1..4] of 2..5
end;
t_group= array[t_range] of t_student;
var d1, d2 : t_date; {Переменные для хранения дат}
    group : t_group; {Переменная для хранения информации о студентах
                      группы}

```

Над записями, как едиными целыми, не определены никакие операции. Совместимость по присваиванию требует тождественности типов. Для описанных выше переменных допустимы присваивания:

```
d1:=d2; group[1]:=group[25].
```

Обращение к полю записи представляет собой составное имя:



Составное имя можно использовать везде, где допустим тип поля. Например, `d1.day:=5; read(group[1].name)`. Имена полей записей могут совпадать с именами других переменных, при этом путаницы не возникает, так как обращение к ним иное. Например, `group` – массив, `group[i]` – элемент массива, `group[i].group` – поле `i`-й записи.

Пример 2. Процедура для определения даты следующего дня невисокосного года:

```

Procedure next_date(d1: t_date; var d2: t_date);
    {d1 - данная дата, d2 - результат (дата следующего дня)}
    var max : 28..31; {число дней в месяце}
    begin d2:=d1;
        case d2.month of {Определение числа дней в заданном месяце}
            2: max:=28;
            4, 6, 9, 11: max:=30
            else max:=31
        end;
        if d2.day<max then d2.day:= d2.day+1
        else begin d2.day:= 1;
            if d2.month<12 then d2.month:= d2.month+1
            else begin d2.month:= 1; d2.year:= d2.year+1
            end
        end
    end
end

```

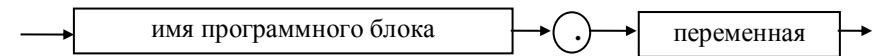
end;

21.1. Оператор присоединения

В Паскале есть возможность обращаться к полям записи без указания имени записи. Такую возможность дает оператор присоединения:



Перечисляемые в заголовке оператора присоединения идентификаторы являются именами записей. Оператор после заголовка – тело оператора присоединения. В теле оператора присоединения к полям записей, имена которых указаны в заголовке, можно обращаться непосредственно. Каждый идентификатор сравнивается с именами полей записей заголовка в порядке, обратном их перечислению. Если идентификатор совпадает с именем поля, то он рассматривается как поле соответствующей записи. Ясно, что для записей, содержащих одинаковые имена полей, важен порядок их перечисления в заголовке оператора присоединения. Если в теле оператора присоединения необходимо обратиться к переменной с именем, совпадающим с именем поля записи, указанной в заголовке, то используется составное имя:



Пример 3. Описание процедуры ввода информации о студентах группы.
 Procedure read_group(var gr : t_group; n : t_range); {n – число студентов}
 begin write('Введите число студентов≤', MaxLen); read(n);
 writeln('Введите. фам., даты рожд., группу и экз. оценки студентов.');

```

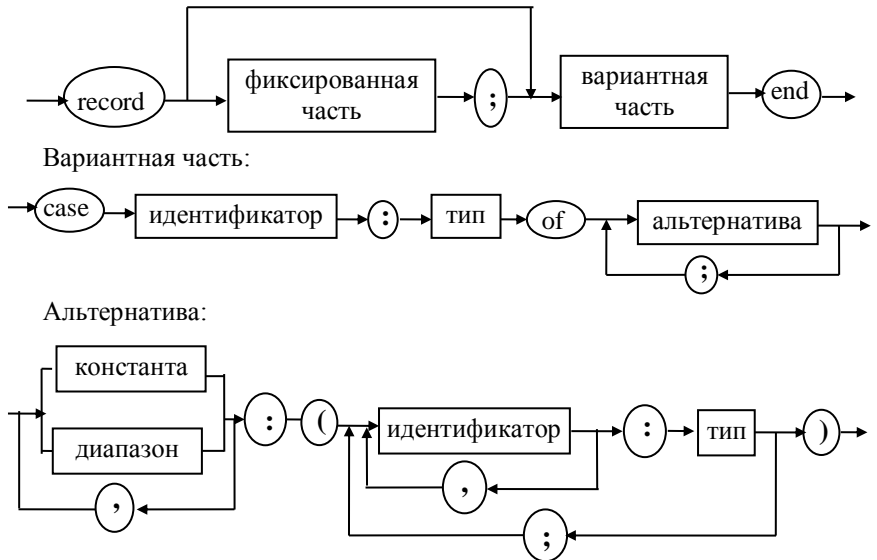
for i:=1 to n do
  with group[i], birthday do
    begin read(name); {вместо gr[i].name}
      readln(day, month, year); {вместо gr[i]. birthday. day, ... }
      read(group);
      for j:=1 to 4 do
        read(mark[j]); {вместо gr[i]. mark[j]}
      readln
    end
  end
end;
  
```

Оператор With group[i], birthday do <оператор>

равносилен оператору with group[i] do with birthday do<оператор>

21.2. Записи с вариантами

Часто необходимы записи с близкими, но не одинаковыми структурами. Для таких случаев используется тип-запись с вариантами:



В круглых скобках – список полей альтернативы (это ветвь вариантной части).

Для всех ветвей вариантной части выделяется одна и та же область памяти, где могут разместиться поля любой ветви. Идентификатор (дискриминант) определяет, какая из ветвей в данный момент активна. Но система не контролирует, к активным ли полям обращается программа. Ответственность за правильность использования полей ложится на программиста. Во избежание ошибок рекомендуется для обработки вариантной части использовать оператор-переключатель с селектором – значением дискриминантного поля.

Пример 4. Описание типа для хранения параметров любой из трех фигур: треугольника, прямоугольника или окружности:

```
type t_fig_type=(triangle, rectangle, circle);
t_figure = record
    s : real; { площадь }
    p : real; { периметр }
    case fig_type: t_fig_type of
```

```

triangle : (a, b, c: real); {стороны треугольника}
rectangle : (d1, d2: real); {стороны прямоугольника}
circle : (r : real)         {радиус окружности}
end;

```

Запись описанного типа содержит два фиксированных поля (s и p), полем-дискриминантом является вид фигуры (fig_type). Это поле может принимать одно из трех значений, поэтому вариантная часть содержит три ветви.

Пример 5. Описание процедуры инициализации переменной типа t_figure. Поля вариантной части вводятся с клавиатуры, а фиксированные поля вычисляются.

```

Procedure init_fig(var f:t_figure);
var n:byte;
begin write('Введите вид фигуры: 1–треуг., 2–прямоуг., 3–окружность');
  read(n);
  with f do
  begin
    fig_type:= t_fig_type (n-1);
    case fig_type of
      triangle : begin read(a, b, c );
                    p:=(a+b+c)/2;
                    s:=sqrt(p*(p-a)*(p-b)*(p-c));
                    p:=2*p
                  end;
      rectangle : begin read(d1, d2);
                    p:=(d1+d2)*2;
                    s:=d1*d2
                  end;
      circle : begin read(r);
                 p:=2*Pi*r;
                 s:=Pi*r*r
               end
    end
  end
end;

```

22. ПОБИТОВЫЕ ОПЕРАЦИИ

ТР позволяет выполнять побитовые операции над битами целых типов, которые используются при работе с двоичными представлениями целых чисел или на низком уровне, например, с видеопамятью.

Название операции	Запись на ТР	Приоритет
-------------------	--------------	-----------

Побитовое отрицание	not m	1
Побитовая конъюнкция	n and m	2
Побитовая дизъюнкция	n or m	3
Побитовое исключающее или	n xor m	3
Сдвиг влево	n shl i	2
Сдвиг вправо	n shr i	2

Побитовое отрицание инвертирует биты целого. not 1=0, not 128=127.

Для обнуления (сбрасывания) определенных битов целого числа выполняют *побитовую конъюнкцию* (побитовое умножение) данного числа с числом, называемым маской. Маска содержит нули только в тех битах, которые нужно сбросить, а в остальных битах – единицы. Например, при выполнении оператора `x:=x and $FF00` сбрасывается младший байт двухбайтового целого (\$FF00 – маска); при `n:=n and ($FF-8)` сбрасывается третий бит однобайтового целого. Биты нумеруются с нуля справа налево.

Побитовую дизъюнкцию можно использовать для установки единиц в определенных битах. При этом маска должна содержать единицы в устанавливаемых битах и нули – в остальных. Например, оператор

`n:=n or (32+8)`

установит 3-й и 5-й биты.

Побитовое исключающее или выполняет сложение соответствующих битов операндов по модулю 2 без переноса 1 в следующий разряд. Важное свойство операции **xor**: `n xor m xor m=n`.

При *сдвиге влево* биты левого операнда сдвигаются влево на число разрядов, равное значению правого операнда. Биты, выходящие за границы типа левого операнда пропадают, а освободившиеся разряды слева заполняются нулями. Сдвиг влево `n shl i` эквивалентен умножению `n` на 2^i , если старшие разряды не вышли за левую границу `n`.

При *сдвиге вправо* биты левого операнда сдвигаются вправо на число разрядов, равное значению правого операнда. Биты, выходящие за правую границу, пропадают. Освобождающиеся слева разряды беззнакового целого заполняются нулями, а знакового – значением старшего (знакового) бита.

Сдвиг вправо `n shr i` эквивалентен целочисленному делению `n` на 2^i в арифметике с положительными остатками. Например, `n shr 5=n div 32`.

Опишем процедуру вывода двоичного представления данного целого беззнакового числа:

```

Procedure WriteBinary(n:word);
var i:byte;
begin for i:=15 downto 0

```

```

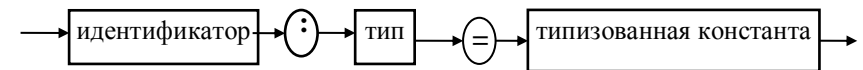
write(n shl(15-i) shr 15)
end;

```

Поменять местами старший и младший байты целого неотрицательного числа можно с помощью оператора $n := (n \text{ shl } 8) \text{ or } (n \text{ shr } 8)$.

23. ТИПИЗОВАННЫЕ КОНСТАНТЫ В ТР

ТР позволяет инициализировать переменные при их описании. Описание с инициализацией переменных производится в разделе описаний *констант*:



Идентификатор – имя переменной.

Переменные, инициализированные типованными константами, имеют особенность по сравнению с обычными переменными: инициализация выполняется только при первом вхождении в блок, в котором они описаны. Если этот блок является процедурой или функцией, то при последующих обращениях к этому блоку повторная инициализация не выполняется. Локальные переменные, инициализированные типованными константами, помещаются в сегмент данных, а не в стек.

Типованная константа может быть: константой простого типа, константой-массивом, константой-строкой, константой-записью, константой-множеством, пустой ссылкой **nil**.

Типованная константа простого или строкового типа представляет собой литерал или константное выражение. Константа-множество описывается обычным способом (см. главу 21). Например,

```

const n: integer=10;
c: char='A';
s: string[5]='YES';
f: real=1.3e-5;
m: set of 1..5=[1,3];

```

При описании типованной константы структурированного типа необходимо задать значения *всех* ее компонентов. Естественно, что компоненты таких констант не могут иметь файловый тип.

Значения элементов одномерного массива заключаются в круглые скобки и перечисляются через запятую. Например,

```

const a: array[1..3] of byte = (1,2,3);

```

Константа символьный массив может быть описана двумя способами: как константа-массив и как строковая константа. Например,

```

const digits1: array[0..9] of char = ('0','1','2','3','4','5','6','7','8','9');

```

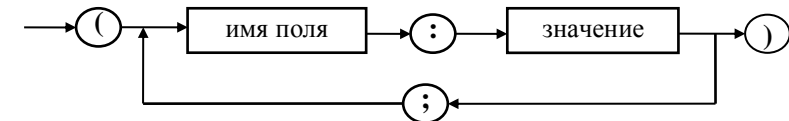
```
digits2 : array[0..9] of char = '0123456789';
```

При описании типизованной константы типа многомерный массив константы-массивы меньших размерностей заключаются в свои круглые скобки, которые разделяются запятыми. Необходимо помнить, что чем правее индекс, тем быстрее он изменяется. Например,

```
const b :array[1..2,1..3] of byte = ((1,2,3),(4,5,6));
```

(1,2,3) – первая строка, (4,5,6) – вторая строка матрицы.

Описание типизованной константы типа запись имеет вид



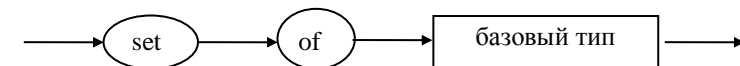
Обязательное условие: поля перечисляются в том же порядке, что и при описании типа. Например,

```
Type t_exam=record name: string[10];
                    marks: array[1..4] of 2..5
end;
```

```
Const exam: t_exam=( name: 'Иванов'; marks: (4, 3, 5, 4));
```

24. МНОЖЕСТВО

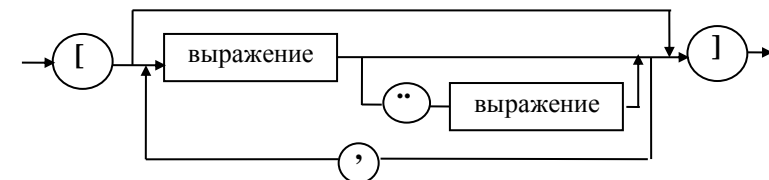
Описание типа множество:



Значениями типа множество являются все подмножества базового типа. Базовый тип играет роль универсального множества.

Число элементов базового типа множества ограничено. Ограничение определяется реализацией языка. В ТР число элементов базового типа не должно превышать 256. Базовый тип является перечисляемым типом или поддиапазоном перечисляемого типа. Кроме того, объем памяти, занимаемой значением базового типа, – один байт. Например, правильными описаниями являются: set of char, set of 10..100, set of 'a'..'z'. Описания set of 200..300 и set of – 10..10 недопустимы, так как в первом случае базовый тип не является однобайтовым, во втором – перечисляемым.

Конструкция



определяет значение типа множество. При этом значения всех перечисленных выражений принадлежат одному типу. Порядок перечисления выражений и диапазонов произвольный. Более того, в списке могут быть выражения с одинаковыми значениями. Например, [5, 0, 3..10] – числовое множество, ['0'..'9'] – символьное множество, ['a','a','c'], ['c','a'] и ['a','c'] – равные символьные множества.

24.1. Машинное представление множества

В памяти множества хранятся очень компактно: каждому элементу базового типа выделяется один бит. Таким образом, тип множество занимает $(n-1)\text{div}8+1$ байтов, где n – число элементов базового типа. Значением типа множество является подмножество тех элементов базового типа, которым соответствуют ненулевые биты.

24.2. Операции над множествами

Над множествами Паскаля определены общепринятые в математике операции: пересечение (*), объединение (+) и вычитание (–). В скобках указан знак операции. Кроме этого, определены следующие операции отношения:

- 1) равенство множеств (=);
- 2) неравенство множеств (<>);
- 3) левый операнд – подмножество правого операнда (<=);
- 4) правый операнд – подмножество левого операнда (>=);
- 5) принадлежность элемента множеству (in).

Логическое выражение «Символ ch – латинская буква» может быть компактно записано с помощью операции проверки принадлежности элемента множеству: $ch \text{ in } ['a'..'z', 'A'..'Z']$.

Для совместимости типов множество в операциях достаточно совместимости базовых типов множеств. Это условие выполняется, если базовые типы одинаковы, или один из них – поддиапазон другого, или оба являются поддиапазонами одного и того же типа.

В операции проверки принадлежности элемента множеству левый операнд должен иметь тип, совместимый с базовым типом множества, которое является правым операндом.

Приоритеты операций в порядке убывания:

- 1) пересечение;
- 2) объединение и вычитание;
- 3) операции отношения.

Для совместимости множеств по присваиванию необходима совместимость типов и присваиваемое значение должно быть подмножеством базового типа переменной.

Пример. Программа для решения следующей задачи. С клавиатуры вводятся символы. Признак конца ввода – точка. Вывести те символы, которые встречались во входной последовательности по одному разу.

```
Program single_char;
var c : char;
    s, sd : set of char;
begin
    s:=[];      {Инициализация множества входных символов}
    sd:=[];     {Инициализация множества повторяющихся символов}
    repeat
        read(c);
        if c in s
            then sd:=sd+[c] {Накопление множества повторяющихся символов}
            else s:=s+[c];   {Накопление множества всех символов}
    until c='.';
    s:=s-sd-['.']; {Получение множества неповторяющихся символов}
    writeln(' Неповторяющиеся символы входной последовательности:');
    for c:=#33 to #255 do
        if (c in s) then write(c:3)
    end.
```

25. ФАЙЛЫ

До сих пор исходные данные при выполнении программы вводились с клавиатуры, а результаты выводились на дисплей, то есть использовался стандартный ввод-вывод. Язык Паскаль предоставляет возможность обмена информацией с файлами. На физическом уровне файлом является поименованная область памяти на внешнем носителе. С точки зрения программиста, файл – это информация, хранящаяся в этой области.

Ввод из файла удобен, если число исходных данных велико. Данные можно подготовить заранее, проверить. К данным, хранящимся в файле, могут обращаться разные программы. Данные, записанные в файл, сохраняются и после завершения работы программы. Файлы удобно использовать для временного хранения информации, если данных много и все они одновременно не могут быть размещены в оперативной памяти.

По способу хранения информации файлы подразделяются на два вида: текстовые и двоичные. Текстовые могут быть созданы с помощью текстового редактора, они представляют собой последовательность символов. Их можно выводить на дисплей и на принтер. Двоичные файлы предназначены для обработки компьютером. Устройства ввода и вывода рассматриваются как текстовые файлы, они, так же как и обычные файлы, имеют имена, например: con – консоль, rpt – принтер.

Файл состоит из последовательности записей. Записью называется порция данных. Разбиение файла на записи определяется способом его обработки и может меняться в зависимости от целей обращения к файлу.

По способу доступа к записям выделяют *последовательные* файлы и файлы *прямого доступа*.

Последовательный файл обрабатывается сначала. Нельзя обработать n -ю запись, не обработав $(n-1)$ -ю. Обработка представляет собой или только чтение или только запись.

Файл прямого доступа позволяет обращаться к записи по ее номеру.

25.1. Файлы в Паскале

В стандартном Паскале все файлы обрабатываются как последовательные.

Логический файл в Паскале – это последовательность компонентов, являющихся записями физического файла. В каждый момент времени доступна только одна запись. Длина последовательности заранее не определена.

В ТР связь между логическим и физическим файлами устанавливается процедурой `Assign(var f:<файл>, f_name:string)`.

Первым параметром является переменная одного из файловых типов, второй параметр – имя физического файла. Например,

```
var f:text;      {Описание файловой переменной}
...             {Другие описания}
begin Assign(f, 'A:\LAB\myfile.txt');
... {Операторы}
end.
```

После связи логического файла с физическим файл должен быть открыт одной из процедур: `Reset(f:<файл>)` или `Rewrite(f:<файл>)`.

При открытии файлов для чтения или для записи указатель файла устанавливается в начало файла. Файл, открываемый для чтения, должен существовать, в противном случае произойдет ошибка.

При открытии существующего файла для записи содержащаяся в нем информация стирается. Если для записи открывается несуществующий файл, то файл с таким именем создается.

Функция `Eof(f:<файл>):boolean` позволяет определить, достигнут ли конец файла. Она возвращает `true`, если указатель файла находится в конце файла, и `false` – в противном случае.

Завершается работа с файлом обращением к процедуре закрытия файла `Close(var f:<файл>)`.

После закрытия файла физический файл, связанный с логическим файлом f, можно переименовать или стереть.

Процедура ReName (f :<файл>; new_name : string) переименовывает физический файл, связанный с логическим файлом f. После ее выполнения именем файла будет значение переменной new_name.

Процедура Erase(f: файл) стирает физический файл, связанный с логическим файлом f.

После закрытия файла файловая переменная может быть связана с другим физическим файлом.

Логический файл в Паскале может быть *текстовым* или *типизованным*, в TP помимо этого – *нетипизованным*.

25.2. Текстовые файлы

Текстовый файл представляет собой последовательность символов, которые сформированы в строки. Признаком конца строки служит символ #13 (CR), он может быть объединен с символом перевода строки #10 (LF). Конец файла – символ #26 (^Z).

При открытии текстового файла создается буфер ввода или вывода. Информация накапливается в буфере, пока он не заполнится, а затем извлекается из него по мере необходимости. По умолчанию размер буфера равен 128 байтам. Он может быть изменен процедурой

SetTextBuf(var f:text; var Buf [: BufSize:word]).

Buf – переменная любого типа, которая будет использоваться как буфер. BufSize – размер буфера, который не должен превышать размера переменной Buf.

Обращение к этой процедуре должно быть после связи с физическим файлом, но до открытия файла. Увеличивать размер буфера рекомендуется для сокращения числа обращений к внешним устройствам.

Кроме описанных выше процедур открытия файла ReSet и ReWrite, существующий текстовый файл может быть открыт для записи в конец файла с сохранением содержащейся в нем информации процедурой

Append(var f:text).

Чтение и запись выполняются процедурами Read и Write или Readln, Writeln так же, как и при стандартном вводе-выводе, но в качестве первого параметра должна быть файловая переменная, например: read(f, i, r), f – файловая переменная.

При чтении и записи указатель файла смещается на число считанных или записанных символов.

Следует обратить внимание на то, что при чтении из текстового файла последовательность символов преобразуется в машинное представление значения соответствующего параметра, а при записи машинное представление - в последовательность символов.

Функция Eoln(var f:text):boolean позволяет определить, достигнут ли конец строки. Если указатель установлен на конец строки, функция возвращает **true**, иначе – **false**.

Функция SeekEoln(var f:text):boolean возвращает значение **true**, если между указателем файла и концом текущей строки находятся только пробелы и символы табуляции, иначе – **false**.

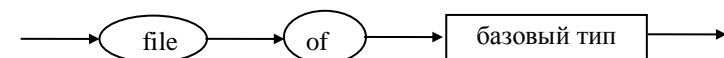
Функция SeekEof(var f:text):boolean возвращает значение **true**, если между указателем файла и концом файла находятся только пробелы, символы табуляции и символы перехода к новой строке, иначе – **false**.

Функции SeekEoln и SeekEof обычно используются при считывании из файла числовых данных.

<p>Пример 1. Функция определения количества строк в текстовом файле:</p> <pre>Function strings_number(s:string):byte; { s – имя физического файла } var f:text; n:byte; begin Assign(f,s); Reset(f); n:=0; while not Eof(f) do begin readln(f,s); n:=n+1; end; Close(f); strings_number:=n end;</pre>	<p>Пример 2. В каждой строке текстового файла s записана строка вещественной квадратной матрицы. Определить порядок матрицы можно с помощью следующей процедуры:</p> <pre>procedure Size(s:string: var n:byte); var f:text; r:real; begin Assign(f,s); Reset(f); n:=0; while not SeekEoln(f) do begin read(f,r); n:=n+1; end; Close(f) end;</pre>
--	--

25.3. Типизованные файлы

Типизованный файл – это последовательность данных одного типа (базового). Его описание:



Базовый тип может быть любым за исключением файлового и структурированного с компонентами файлового типа.

Данные в типизованных файлах хранятся в машинном представлении.

При работе с типизованными файлами ввод-вывод выполняется процедурами `read(f,<список переменных>)`, `write(f,<список переменных >)`. Все переменные в списках ввода и вывода должны иметь базовый тип файла.

Запись типизованного файла представляет собой машинное представление значения базового типа файла. При чтении и записи значения базового типа указатель файла смещается к следующей записи.

Пример 3. В файле записаны квадратные матрицы 3-го порядка. Сохранить в файле с таким же именем только первые строки каждой из матриц данного файла.

```
Type t_row=array[1..3] of real;
   t_matr=array[1..3] of t_row;
Var f_matr:file of t_matr;
    f_row:file of t_row;
    a:t_matr;
    s:string;
Begin write(' Введите имя файла матриц '); readln(s);
      Assign(f_matr, s);
      Assign(f_row, 'temp'); {temp – временный внешний файл}
      Reset(f_matr);
      Rewrite(f_row);
      while not Eof(f_matr) do
        begin
          read(f_matr, a);
          Rewrite(f_row, a[1])
        end;
      Close(f_marr);
      Close(f_row);
      Erase(f_matr);
      Rename(f_row, s)
End.
```

25.4. Нетипизованные файлы

Нетипизованные файлы используются для доступа к любым файлам независимо от типа и структуры.

Нетипизованный файл (файл с именем типа **file**) – последовательность записей фиксированного размера. Размер записей равен размеру буфера, определяемого при открытии файла процедурами `Reset(var f:file; BufSize:word)` или `Rewrite(var f:file; BufSize:word)`.

По умолчанию размер буфера равен 128. Чем больше размер буфера, тем быстрее происходит обмен данными между файлом и оперативной памятью.

Для чтения из нетипизованного файла используется процедура

`BlokRead (var f:file; var Buf; Count:word [;var Result:word]),`

а для записи –

`BlokWrite (var f:file; var Buf; Count:word [;var Result:word]).`

Параметр `Buf` указывает, откуда считывать или куда записывать данные.

`Count` – количество блоков, которые должны быть считаны или записаны.

При этом должно выполняться условие $\text{Count} * \text{BufSize} < 64\text{Kб}$.

При обращении к процедурам ввода-вывода при наличии необязательного параметра `Result` в него записывается фактическое количество считанных или записанных блоков. Если этот параметр не указан, то при невозможности считать или записать указанное число блоков произойдет ошибка выполнения программы.

Нетипизованные файлы используются для быстрого обмена данными.

25.5. Прямой доступ к нетекстовым файлам TP

Прямой доступ позволяет обращаться к записям файла по номерам. Нумерация записей начинается с нуля.

Следующие подпрограммы позволяют организовывать прямой доступ к записям типизованных и нетипизованных файлов. Всюду ниже `f` – файловая переменная.

1. Функция `FilePos(var f):longint` возвращает номер позиции указателя файла. После открытия файла эта функция возвращает 0.

2. Функция `FileSize(var f):longint` возвращает количество записей файла.

3. Процедура `Seek(var f;n:longint)` перемещает указатель файла `f` в позицию с номером `n`. Обращение `Seek(f; FileSize(f))` установит указатель в конец файла.

4. Процедура `Truncate(f)` усекает файл по текущей позиции указателя. Для удаления последней записи файла выполним последовательность операторов `Seek(f;FileSize(f) – 1); Truncate(f)`.

В TP существует предопределенная переменная `FileMode`, которая определяет режим для открываемых файлов. По умолчанию `FileMode=2`,

что соответствует режиму чтения и записи, FileMode=1 – только запись, FileMode=0 – только чтение.

Процедуры Reset и Rewrite открывают существующие файлы в соответствии с установленным режимом. Вновь создаваемый файл процедурой Rewrite открывается в режиме чтения и записи независимо от значения переменной FileMode.

Пример 4. Программа удаления четных чисел из файла целых чисел. Вспомогательный файл не используется. Порядок следования чисел не меняется.

```

Program del_even;
Var f:file of integer;
    i,j:longint; {i – указатель для чтения, j – указатель для записи}
    n:integer;
    f_name:string[40];
Begin
write('Введите имя файла '); readln(f_name);
Assign(f,f_name); Reset(f);
i:=0; j:=0;
while not Eof(f) do
begin read(f,n); i:=i+1;
    if odd(n) then
begin Seek(f,j);
        j:=j+1;
        write(f,n) {Записываем нечетное.}
    end;
    Seek(f,i)
end;
Seek(f,j);
Truncate(f);
Close(f)
end.

```

26. ДИНАМИЧЕСКИЕ ПЕРЕМЕННЫЕ

Переменные, структура и тип которых определяются в разделе описаний блока (программы или подпрограммы), существуют и не изменяют своей структуры в течение всего времени работы блока. Такие переменные и связанные с ними структуры называются *статическими*.

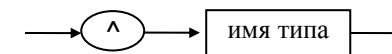
Использование статических переменных и структур удобно не всегда. Иногда требуется, чтобы переменные возникали уже при выполнении программы или меняли свои размеры и структуру. Такие переменные и структуры называются *динамическими*.

Например, при решении задачи требуется сохранить в оперативной памяти слова вводимого текста, удовлетворяющие некоторому условию. Таких слов может не оказаться вовсе или это будут все слова текста. При использовании статических структур придется зарезервировать объем памяти для хранения структуры максимального размера. При этом память используется нерационально.

26.1. Ссылочный тип данных

В Паскале есть возможность создания динамических переменных с помощью переменных ссылочного типа.

Описание ссылочного типа:



Значением типа указатель является адрес переменной базового типа. Тип указатель занимает 4 байта (сегмент и смещение).

Пример 1. Описание переменных ссылочных типов.

```

Type t_ptr_real = ^real;
      t_vect = array[1..100] of char;
Var p1, p2: t_ptr_real; {указатель на вещественное число}
    p_vect: ^t_vect;    {указатель на символьный массив}
  
```

Описанные переменные-указатели являются статическими, каждая из них занимает 4 байта. При описании переменные не инициализируются, поэтому их значения считаются неопределенными.

В TP predefined ссылочный тип **pointer** — нетипизованный указатель. Его значением является адрес ячейки памяти.

Начиная с версии TP7, в TP реализован тип `Pchar = ^Char`.

Инициализация переменной ссылочного типа может быть выполнена с помощью оператора присваивания или с помощью подпрограмм выделения памяти.

Ссылочные типы совместимы по присваиванию, если их базовые типы тождественны или один из них является нетипизованным указателем. Существует predefined константа **nil: pointer** (пустой указатель). После присваивания `p:=nil` указатель `p` не ссылается ни на какую область памяти. Если переменная `p1` уже инициализирована и `p1 ≠ nil`, то после присваивания `p2:=p1` обе переменные ссылаются на одну и ту же область памяти.

Тип `Pchar`, кроме того, совместим по присваиванию с символьным массивом и строковым типом.

26.2. Подпрограммы динамического распределения памяти

Создать динамическую переменную можно с помощью одной из следующих подпрограмм.

1. Процедура New(var p: <указатель>) выделяет место в динамически распределяемой области памяти, называемой «кучей», для размещения переменной базового типа указателя p и начальный адрес присваивает переменной p.

2. Функция New(<имя ссылочного типа>):pointer выделяет место в динамически распределяемой области памяти для размещения переменной указанного в качестве параметра типа и возвращает адрес выделенной области.

3. Процедура GetMem (var p:pointer; n:word) выделяет в «куче» блок памяти размером n, начальный адрес блока помещает в p. Если p типизованный указатель, то значением второго параметра должен быть размер базового типа указателя. Для его определения удобно использовать функцию SizeOf(<имя типа>|<выражение>):word, которая возвращает размер своего параметра в байтах.

Если по какой-либо причине место для размещения переменной не может быть выделено, то происходит аварийный останов.

Функция MaxAvail:LongInt возвращает размер максимального свободного блока в «куче». С ее помощью можно проверить, достаточно ли места для размещения динамической переменной. Функция MemAvail:LongInt возвращает суммарный размер всех свободных блоков.

После окончания работы с динамической переменной память, занимаемая этой переменной, должна быть освобождена одной из процедур:

Dispose(var p: <типизованный указатель>)

или

FreeMem(var p: pointer; size:word).

Dispose освобождает память, занимаемую переменной базового типа указателя p. FreeMem освобождает блок памяти размером size байтов с начальным адресом p. В качестве параметра size должен быть указан размер динамической переменной. Каждому обращению к подпрограмме выделения памяти должна соответствовать процедура освобождения памяти.

26.3. Операции над указателями

Над переменными ссылочных типов, кроме типа Pchar, не определены никакие операции, результатом которых является ссылочный тип.

Данные ссылочных типов можно сравнивать только на равенство и неравенство. Но и эти операции не всегда дают правильный результат, так

как сравниваются отдельно сегменты и смещения адресов, а один и тот же физический адрес может быть представлен в виде сегмента и смещения неоднозначно. Адреса, полученные при обращении к рассмотренным выше подпрограммам выделения памяти, всегда приводятся к смещению от 0 до 15. Поэтому сравнение таких адресов корректно. Сравнивать можно указатели с тождественными базовыми типами или указатель с нетипизованным указателем.

26.4. Работа с динамическими переменными

Обратиться к динамической переменной можно с помощью операции разыменования :



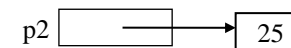
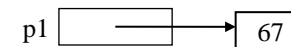
Это выражение является переменной базового типа указателя. Разыменование нетипизованных указателей не имеет смысла.

После размещения динамической переменной в «куче» значение ее не определено и требуется инициализация. С помощью указателей, описанных в примере 1, могут быть созданы и инициализированы динамические переменные, например, следующим образом:

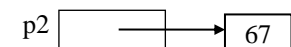
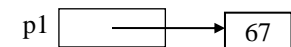
New(p1); Read(p1^); { Введем 67 }

New(p2); p2^:=25;

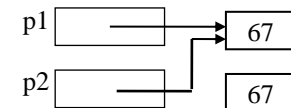
Изобразим схематично результат выполнения этих операторов:



После выполнения оператора $p2^ := p1^$ имеем



После присваивания $p2 := p1$ схема имеет вид



Теперь переменные p1 и p2 ссылаются на одну и ту же область памяти, адрес переменной $p2^$ до присваивания утерян. Этого не должно быть, так как «засоряется» динамическая память. Перед присваиванием нужно было освободить память, обратившись к процедуре, например, dispose(p2).

Пример 2. Создание и инициализация структурированной динамической переменной `p_vect`, описанной в примере 1:

```
New(p_vect);
for i:=1 to 100 do read(p_vect[i]);
p_vect имеет тип t_vect – массив, p_vect[i] – элемент массива.
```

26.5. Создание структур большого размера

Указатели позволяют обойти ограничение на объем памяти структурированных типов (65520 б). Например, TP не позволяет описать двумерный вещественный массив для обработки матриц 100 на 200. Для хранения такой матрицы требуется почти два сегмента данных.

Организуем структуру для работы с матрицами 100 на 200 и опишем процедуру создания и инициализации матрицы:

```
Type t_row=array[1..200] of real; {тип – строка матрицы}
    t_ptr_row=^t_row; {указатель на строку}
    t_ptr_matr=array[1..200] of t_ptr_row; {массив указателей на строки}
Procedure Creat_Matr(var a:t_ptr_matr);
var i,j:byte;
Begin
    for i:=1 to 200 do
        begin
            new(a[i]); {создание i-й строки}
            for j:=1 to 200 do {цикл ввода элементов i-й строки}
                read(a[i]^j) {a[i]^j – элемент матрицы}
            end
        end
    End;
```

26.6. Длинные строки в TP

Тип `string` позволяет работать со строками переменной длины, не превышающей 255. TP, начиная с версии 7, дает возможность использовать длинные строки (до 65534 символов). Такие строки, называемые ASCIIZ-строками, представляют собой последовательности символов, заканчивающиеся ноль-символом `#0`. Хранятся они в символьных массивах с типом индекса `0 .. n`. Для реализации операций над ASCIIZ-строками введен тип `Pchar=^Char`, который можно использовать при включенной директиве расширенного синтаксиса `{SX+}`. Переменные типа `Pchar` совместимы по присваиванию с ASCIIZ-строками, над ними можно выполнять следующие операции: сложение указателя с целым числом, вычитание из указателя целого, вычитание указателей, ссылающихся на

одну строку, и все 6 операций сравнения при условии, что оба указателя ссылаются на одну строку.

Пусть переменные $p1$ и $p2$ типа $Pchar$ указывают на символы одной и той же ASCIIZ-строки, а n – положительное целое.

$p2 := p1 + n$ $p2$ – указатель на символ, полученный смещением $p1$ вправо на n символов

$p2 := p1 - n$ $p2$ – указатель на символ, полученный смещением $p1$ влево на n символов

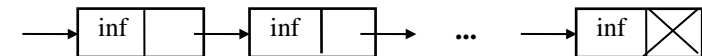
$n := p1 - p2$ n – целое число, равное отклонению $p2$ относительно $p1$.

Переменные типа $Pchar$ можно индексировать. Так, $p1[0] = p1^{\wedge}$, $p1[3] = (p1 + 3)^{\wedge}$.

Модуль `Strings` содержит подпрограммы для работы с ASCIIZ-строками.

26.7. Динамические структуры данных

Возможность выделения и освобождения области памяти позволяет создавать гибкие структуры, размеры и организация которых динамически изменяются. Простейшим примером такой структуры является линейный однонаправленный список. Он представляет собой набор следующих друг за другом однотипных звеньев. Каждое звено состоит из двух частей, информационной и указательной, и имеет тип записи. Информационная и указательная части являются полями этой записи. Информационная часть – некоторая структура данных, чаще всего тоже запись. Назовем это поле `inf`. Указательная часть (второе поле записи) – ссылка на следующее звено. Обычно это поле называют `next`. Последнее звено списка не ссылается ни на что. Указательная часть последнего звена имеет значение `Nil`:



( – обозначает `Nil`).

Пусть T – тип информационного поля звена. Поле `next` имеет тип *указатель на звено*. По общим правилам Паскаля тип *звено* не может быть описан до описания типа *указатель на звено*, так как *указатель на звено* – это тип одного из полей записи, и наоборот, тип *указатель на звено* не может быть описан, пока не описан тип *звено*. Для разрешения этого противоречия в языке Паскаль делается исключение: при описании типа указатель разрешается использовать имя еще не описанного базового типа указателя:

```

Type t_ptr_link = ^t_link;
t_link = record
    inf: T;
    next: t_ptr_link
  
```


end;

Типовыми задачами при работе со списками являются: создание и уничтожение списка, вставка и удаление звена, поиск звена в списке, просмотр списка и т.п.

Опишем подпрограммы для решения некоторых из этих задач. Для определенности в качестве типа информационной части выберем тип integer. Для единообразия выполнения операций вставки и удаления звена создадим список с заголовочным звеном: информационная часть первого звена не используется.

Удаление из списка звена, следующего за звеном с адресом p: <pre> procedure del_zv(p: t_ptr_link); var q: t_ptr_link; begin q:=p^.next; p^.next:=q^.next; dispose(q) end;</pre>	Вставка в список звена с информац. полем s после звена с адресом p: <pre> procedure inszv(p:t_ptr_link;s:integer); var q: t_ptr_link; begin new(q); q^.next:=p^.next; p^.next:=q; q^.inf:=S end;</pre>
Создание списка целых чисел с заголовочным звеном, 0 – признак конца вводимой последовательности: <pre> procedure creat_list(var list: t_ptr_link); var p: t_ptr_link; s: integer; begin new(list); p:=list; {заголовочное звено} read(s); while s<>0 do begin new(p^.next); p:=p^.next; p^.inf:=s; read(s) end; p^.next:=nil end;</pre>	

27. ПРЕОБРАЗОВАНИЕ ТИПОВ

В Паскале преобразование типов выполняется автоматически при условии совместимости типов при выполнении операций или

присваивания. Например, при присваивании целого вещественной переменной целое значение преобразуется в вещественное. При сложении длинного целого и целого целый тип преобразуется к длинному целому. В ТР при выполнении конкатенации оба операнда приводятся к типу `string`.

Одна и та же область памяти может рассматриваться как значение разных типов в следующих случаях:

- а) при использовании записей с вариантами;
- б) если типизованные указатели с разными базовыми типами содержат один и тот же адрес;
- в) при размещении переменных разных типов по одному и тому же абсолютному адресу. Это выполняется при описании переменных с использованием директивы ***absolute***:

absolute (<Абсолютный адрес> | <идентификатор ранее описанной переменной>).

<Абсолютный адрес>::=< сегмент>:< смещение>

Сегмент и смещение – двухбайтовые целые без знака.

Пример.

Var *a* : array[1..6] of byte;

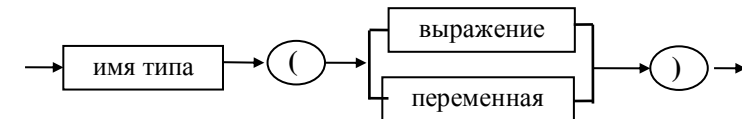
b : array[1..3] of word ***absolute a***;

r : real ***absolute a***;

z : longint ***absolute \$A12C : \$0000***;

Переменные *a b r* располагаются по одному адресу.

В ТР можно обойти ограничения на совместимость типов и с помощью операции приведения типов:



Эта конструкция имеет тип, указанный слева.

Приведение типов выражений и переменных выполняется по-разному.

27.1. Приведение типов выражений

Тип выражения и тип, к которому приводится выражение, должны быть скалярными упорядоченными типами или указателями.

При приведении типов выражений значение может быть расширено или усечено, если размер типа выражения не равен размеру типа, к которому оно приводится. При приведении к типу большего размера значение не искажается, числовые значения сохраняют знак. При приведении к типу меньшего размера отбрасываются старшие байты. Например, оператор `write(byte(2>3))` выведет 0. Значение выражения `byte($7A1F)` равно \$1F.

При приведении типа переменной размер переменной должен быть равен размеру типа, к которому она приводится.

Например, переменная типа LongInt может быть приведена к типу массив из четырех символов. В этом случае после приведения типа переменной можно указать индекс.

Операцию приведения типа переменной можно использовать как в правой, так и в левой части оператора присваивания в качестве фактического параметра подпрограммы.

Пример 1.

```
type t_season=(Winter, Spring, Summer, Autumn);
t_vect=array[1..2] of word;
var p:pointer; s:t_season;
begin write('Введите сезон: 0 – зима, 1 – весна, 2 – лето, 3 – осень');
read(byte(s));
...
getmem(p,1);
write('Сегмент: ', t_vect(p)[2], '; смещение:', t_vect(p)[1]);
...
end.
```

27. 3. Обработка одномерных массивов разных размеров с фиксированным базовым типом

Для обработки одномерных массивов с разными типами индексов, но с одним и тем же базовым типом, можно описывать подпрограммы, которым передается массив, как параметр-открытый массив. В таких подпрограммах параметр-массив приводится к типу массив с тем же базовым типом, но с типом индекса $0 \dots n-1$, где n – размер массива. Максимальное значение индекса элемента массива a , передаваемого как открытый массив, в подпрограмме возвращает функция $\text{High}(a)$.

Пример 2. Описание процедуры ввода вещественного массива a :
 Procedure read_vect(var a :array of real);

```
var i:word;
begin for i:=0 to High( $a$ ). do
      read( $a[i]$ )
end;
```

27.4. Нетипизованные параметры подпрограмм

Нетипизованные параметры- переменные позволяют создавать гибкие подпрограммы для обработки параметров любого типа. В теле подпрограммы нетипизованный параметр обязательно приводится к подходящему для его обработки типу как переменная, но равенство размеров типов не требуется.

Опишем логическую функцию сравнения двух переменных произвольного размера и типа. Для того чтобы переменные имели равные значения, достаточно, чтобы соответствующие байты этих переменных совпадали.

```
Function Equal(var a,b; size:word):boolean;
  type bytes=array[1..65520] of byte; {тип – массив байтов максимального
                                     размера}
  var i:word;
  begin i:=1;
    while (i<=size) and (bytes(a)[i]=bytes(b)[i]) do
      i:=i+1;
    Equal:=i>size
  end.
```

В подпрограммах обработки двумерных массивов (матриц) разных размеров, но с одним и тем же базовым типом нетипизованный параметр-матрица приводится к типу одномерный массив. Для этого нужно использовать формулу для определения k – номера элемента в одномерном массиве, который в двумерном массиве имеет индексы i и j (i – номер строки, j – номер столбца матрицы).

$k=(i-1)n+j$, где n – число элементов в строке.

Пример 3. Описание процедуры обмена строк с номерами $r1$ и $r2$ матрицы **a** (n – длина строки):

```
Procedure Rows_Swap(var a; n, r1, r2:word);
  var i, k1, k2:word;
  t:real;
  begin
    for i:=1 to n do
      begin
        k1:=(r1-1)*n+i;
        k2:=(r2-1)*n+i;
        t:=t_max_vect(a)[k1];
        t_max_vect(a)[k1]:=t_max_vect(a)[k2];
        t_max_vect(a)[k2]:=t
      end
    end
```

end;

28. ПРОЦЕДУРНЫЕ И ФУНКЦИОНАЛЬНЫЕ ТИПЫ

ТР позволяет рассматривать процедуры и функции как объекты, которые могут быть присвоены переменным или переданы в качестве параметра подпрограммам. Такие переменные должны иметь процедурный или функциональный тип. Далее будем называть любой из этих типов процедурным. Описание процедурного типа представляет собой описание заголовка подпрограммы, но без указания имени подпрограммы, например:

```

Type proc=procedure; {тип используется для процедур без параметров }
  swap_proc =procedure(var a,b:real);
  math_func =function(x:real):real;

```

Имена параметров играют только иллюстративную роль. Можно описывать переменные процедурных типов, компоненты структурированных типов могут иметь процедурные типы. Например,

```

Var f:math_func;
    f_arr: array[1..3] of math_func;
    psw: swap_proc;
    p_sort2: procedure (var x, y:real);

```

Для совместимости по присваиванию процедурных типов необходимо, чтобы подпрограммы имели одинаковое количество параметров, соответствующие параметры имели тождественные типы и типы возвращаемых функциями значений были одинаковыми. Процедурный тип по сути является ссылочным типом, его значениями являются адреса кодов подпрограмм. Переменной процедурного типа может быть присвоено значение уже инициализированной другой переменной или имя подпрограммы. Имя подпрограммы рассматривается как константа процедурного типа. Для обеспечения совместимости по присваиванию процедура или функция, если ее нужно присвоить переменной или передать в качестве параметра подпрограмме, должна удовлетворять следующим условиям:

- 1) компилироваться с дальним типом вызова ({ $\$F+$ });
- 2) не должна быть вложенной;
- 3) не должна быть стандартной;
- 4) не должна быть процедурой прерывания или типа inline.

Для обеспечения первого условия перед описанием подпрограммы или подпрограмм можно указать директиву компилятора { $\$F+$ }, а после описания – { $\$F-$ } (пример 1), или после заголовка подпрограммы указать директиву FAR (пример 2).

Если стандартную подпрограмму нужно передать в качестве параметра или присвоить переменной, то требуется создать некоторую «надстройку» (пример 2).

Пример 1.

```
{ $F+ }
function tg(x :real) :real;
begin
  tg:=sin(x)/cos(x)
end;
```

Пример 2.

```
function MySin(x :real) :real; FAR;
begin
  MySin:=sin(x)
end;
```

~~Пример 3. Программа нахождения корней трех уравнений на промежутке [a,b] с точностью ε :~~

```
Type t_math_func=function (x:real):real;
var f: array[1..3] of t_math_func;
a,b,eps:real;
i:byte;
function root(a, b, eps: real; f:t_math_func):real;
{функция нахождения корня уравнения f(x)=0 методом половинного
деления}
var c: real;
begin
  while abs(b-a)> eps do
  begin
    c:=(a+b)/2;
    if f(a)*f(c)<0 then b:=c else a:=c;
  end;
  root:=(a+b)/2
end;
{ $F+ }
function f1(x: real): real;
begin f1:=sin(x) -0.1
end;
function f2(x: real): real;
begin f2:= sin(x)/cos(x)+0.1
end;
function f3(x: real): real;
begin f3:= exp(x) -1/2
end;
{ $F- }
```

```

Begin write('Введите промежуток и точность');
read(a,b,eps);
f[1]:=f1; f[2]:=f2; f[3]:=f3;
for i:=1 to 3 do
  writeln('Уравнение ', i, ': x=', root(a, b, eps, f[i]))
end.

```

29. МОДУЛИ В TP

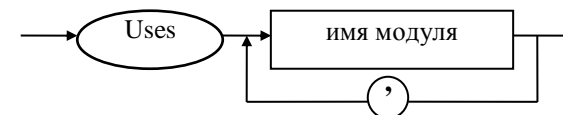
TP содержит библиотеку подпрограмм, которая хранится в файле **turbo.tpl**. Все подпрограммы разбиты на несколько групп, называемых модулями. В каждом модуле содержатся описания констант, типов, переменных, процедур и функций. Один из модулей, модуль **Graph**, находится в отдельном файле **Graph.tpu**.

29.1. Основные модули TP

1. **System.** Содержит арифметические функции, подпрограммы работы со строками, файлами, подпрограммы ввода-вывода, подпрограммы для работы с динамическими переменными.
2. **Crt.** В этом модуле собраны подпрограммы для работы с дисплеем и клавиатурой. Подпрограммы этого модуля позволяют изменять цвет, организовать работу с окнами и звуком.
3. **Dos.** Поддерживает различные функции MsDos.
4. **Printer.** В этом модуле описана файловая переменная **lst: text**, она связывается с устройством вывода rpn (принтером). Если подключен модуль **printer**, то для вывода на принтер достаточно указать в качестве первого параметра процедуры **write** переменную **lst**:

```
write( lst, <список параметров>).
```
5. **Overlay.** Содержит подпрограммы для работы с перекрытиями, с помощью которых можно организовывать большие программы.
6. **Graph.** Используется при работе в графическом режиме.
7. **Strings.** Содержит подпрограммы для работы с длинными строками.

Если в программе используются подпрограммы или другие программные объекты, описанные в каких-либо модулях, то эти модули должны быть подключены к программе. Подключение выполняется в начале программы, сразу после заголовка, и имеет такой вид:

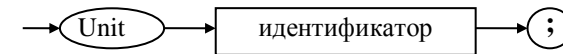


Модуль **Sistem** подключается ко всем программам автоматически.

29.2. Создание собственных модулей

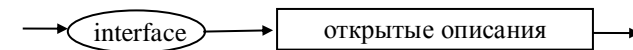
ТР позволяет создавать собственные модули. Модуль состоит из заголовка, секции интерфейса, секции реализации, секции инициализации. Секции реализации и инициализации не являются обязательными.

Заголовок модуля:



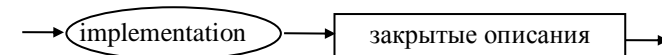
Идентификатор – имя модуля.

Секция интерфейса:



В секции интерфейса подключаются модули, описываются программные объекты, которые будут доступны программам и модулям, подключающим данный модуль. Можно описать константы, переменные, типы, *заголовки* процедур и функций, тела этих подпрограмм описываются в секции реализации.

Секция реализации:



В секции реализации могут подключаться модули, могут быть описаны константы, переменные, типы, подпрограммы, но к ним программы, подключающие этот модуль, доступа не имеют. Эти программные объекты используются при описании тех подпрограмм, заголовки которых включены в секцию интерфейса. При описании подпрограмм, перечисленных в секции интерфейса, в заголовках можно опускать списки параметров.

Секция инициализации:



В секции инициализации присваиваются начальные значения переменным. Выполнение программы, подключающей модуль с секцией инициализации, начинается с выполнения операторов секции инициализации модуля.

Порядок подключения модулей имеет значение. Если некоторый модуль А использует объекты, описанные в модуле В, то модуль В подключается раньше, чем модуль А.

Если в программе описан программный объект, имя которого совпадает с именем объекта, описанного в секции интерфейса модуля, то такой объект имеет смысл, соответствующий программному описанию. Если же надо обратиться к объекту, описанному в модуле, следует использовать составное имя: **<имя модуля >. <идентификатор>.**

Пример. Модуль для работы с комплексными числами:

```
Unit Complex;
interface
  type compl = record
    Re, Im :real
  end;
  procedure read_compl( var z:compl ); { Ввод комплексного числа }
  procedure write_compl( z:compl );   { Ввод комплексного числа }
  function mod_compl( z:compl ):real; { Определение модуля компл. числа }
  { ..... }
  {Продолжение описания заголовков подпрограмм для выполнения
  действий над комплексными числами}

implementation { описание подпрограмм }
  procedure read_compl;
  begin
    read( z.Re, z.Im);
  end;
  procedure write_compl;
  var s: char;
  begin
    if z.Im<0 then s:=' - ' else s:='+';
    write( z.Re:4:3, s, abs(z.Im), '*i');
  end;
  { ..... }
  {Продолжение описания подпрограмм, объявленных в секции
  интерфейса}
end.
```

Бывают случаи, когда модуль А использует модуль В и наоборот, В использует А. Такие циклические ссылки возможны только в случае, когда модули подключаются в секции реализации.

После того как модуль описан, его нужно записать в файл с именем <имя модуля>.pas. Если длина имени модуля превышает 8, то первые 8 символов

должны совпадать с именем файла. Далее надо компилировать модуль. Для компиляции используется пункт меню **Compile** или **<Alt>+<F9>**. В результате компиляции получим файл с расширением **.tru**. Все модули, созданные нами, можно собрать в одном каталоге и имя каталога указать в меню **Options \ directories \ Unit directories**. В одном модуле рекомендуется собирать константы, типы, подпрограммы для решения задач определенного класса.

При внесении изменений в модуль его нужно перекомпилировать. Это можно сделать, используя меню **Compile \ Make** или **<F9>**. В пункте **Primary file** меню **Compile** указывается имя файла с текстом основной программы. Меню **Compile \ Make** проверяет даты создания файлов ***.pas** и ***.tru**. Исходные файлы, в которые вносились изменения будут откомпилированы заново. Проверяется, вносились ли изменения в интерфейсы модулей, если да, то будут перекомпилированы все файлы, использующие этот модуль. Если изменения вносились только в секцию реализации какого-либо модуля, то вызывающие этот модуль модули и основная программа перекомпилироваться не будут. При использовании меню **Compile** пункт **Build** будет перекомпилирована вся программа и все модули, независимо от даты внесения изменений.

Если в пункте **Primary file** меню **Compile** не указано имя файла, то компилируется файл из текущего окна.

Модули позволяют разбить большую задачу на части и этим избежать ограничений, накладываемых на размер программы. Допустимый размер программы в ТР 64 кБ. При использовании модулей объем используемой памяти может быть увеличен до 640 кБ.

СПИСОК ЛИТЕРАТУРЫ

1. Абрамов В. Г., Трифонов Н. П., Трифонова Г. Н. Введение в язык Паскаль: Учеб. пособие. – М.: Наука. Гл. ред. физ.-мат. лит., 1988. – 320 с.
2. Абрамов С. А., Гнездилова Г. А. и др. Задачи по программированию. – М.: Наука. Гл. ред. физ.-мат. лит., 1988. – 243 с. (Библиотека программиста).
3. Абрамов С. А., Зима Е. В. Начала информатики. – М.: Наука. Гл. ред. физ.-мат. лит., 1989. – 256 с. (Библиотека программиста).
4. Андерсон Р. Доказательство правильности программ / Пер. с англ. – М.: Мир, 1982. – 168 с.
5. Вирт Н. Алгоритмы + структуры данных = программы / Пер. с англ. – М.: Мир, 1985. – 406 с. (Математическое обеспечение ЭВМ).
6. Вирт Н. Систематическое программирование. Введение / Пер. с англ. – М.: Мир, 1977.

7. Д а л У., Д е й к с т р а Э., Х о о р К. Структурное программирование / Пер. с англ. – М.: Мир, 1985. – 247 с. (Математическое обеспечение ЭВМ).
8. Д ж о н с т о н Г. Учиться программировать / Пер. с англ. – М.: Финансы и статистика, 1989. – 367 с.
9. Й о д а н Э. Структурное программирование и конструирование программ. – М.: Мир , 1989. – 416 с.
10. М а р ч е н к о А. И., М а р ч е н к о Л. А. Программирование в среде Турбо Паскаль 7.0. – М.: Бином Универсал, Киев: Юниор, 1997. –495 с
11. П и л ь щ и к о в В. Н. Сборник упражнений по языку Паскаль. – М.: Наука. Гл. ред. физ.-мат. лит., 1989. – 154 с.
12. П о л я к о в Д. Б., К р у г л о в И. Ю. Программирование в среде Турбо Паскаль. – М.: Изд-во МАИ, 1992. – 575 с.
13. Х ь ю з Дж., М и ч т о м. Дж. Структурный подход к программированию / Пер. с англ. – М.: Мир, 1980. – 278 с.
14. Ф а р о н о в В.В. Турбо Паскаль 7.0. Начальный курс. –М.: Нолидж, 1997. –612 с.
15. Ф о р с а й т Р. Паскаль для всех / Пер. с англ. – М.: Машиностроение, 1986. – 288 с.