Вопросы для проведения собеседования в магистратуру

010400.68 Прикладная математика и информатика

по дисциплинам «**Математический анализ» и «Информатика**»

Оглавление.

[1. Предел последовательности. Бесконечно малая и бесконечно большая величины. Арифметические действия с переменными, имеющими предел. 3](#_Toc393445825)

[2. Теорема о монотонной последовательности. Второй замечательный предел. 5](#_Toc393445826)

[3. Фундаментальная последовательность. Критерий Коши сходимости последовательности. 6](#_Toc393445827)

[4. Определения предела функции. Первый замечательный предел. 7](#_Toc393445828)

[5. Непрерывность функции. Разрывы первого и второго рода. 8](#_Toc393445829)

[6. Операции над непрерывными функциями. 10](#_Toc393445830)

[7. Теоремы Вейерштрасса о функции непрерывной на отрезке. 11](#_Toc393445831)

[8. Определение производной функции. Геометрический и физический смысл производной функции. 13](#_Toc393445832)

[9. Дифференциал функции. Геометрический смысл. 15](#_Toc393445833)

[10. Производная и дифференциал высшего порядка. 18](#_Toc393445834)

[11. Теорема Коши о среднем. 20](#_Toc393445835)

[12. Теорема Лагранжа о среднем. 21](#_Toc393445836)

[13. Формула Тейлора для функции с остаточным членом в форме Лагранжа. 22](#_Toc393445837)

[14. Локальный экстремум функции одной переменной. Необходимые условия экстремума функции. 23](#_Toc393445838)

[15. Достаточные условия экстремума функции одной переменной. 24](#_Toc393445839)

[16. Исследование функции. Построение графика функции. 25](#_Toc393445840)

[17. Первообразная и неопределенный интеграл. 26](#_Toc393445841)

[18. Таблица основных интегралов. 27](#_Toc393445842)

[19. Простейшие правила интегрирования. Интегрирование путем замены переменной. 28](#_Toc393445843)

[20. Интегрирование по частям. 29](#_Toc393445844)

[21. Понятие определенного интеграла, его геометрический смысл. 30](#_Toc393445845)

[22. Свойства определенных интегралов. 31](#_Toc393445846)

[23. Интеграл как функция верхнего предела. Формула Ньютона−Лейбница. 32](#_Toc393445847)

[24. Суммы Дарбу, условия существования интеграла. 33](#_Toc393445848)

[25. Интегрируемость непрерывных и монотонных функций. 34](#_Toc393445849)

[26. Несобственные интегралы. 35](#_Toc393445850)

[27. Несобственные интегралы от неотрицательных функций. Признаки сходимости. 36](#_Toc393445851)

[28. Частная производная функции многих переменных. 37](#_Toc393445852)

[29. Производная по направлению функции многих переменных. 38](#_Toc393445853)

[30. Градиент функции многих переменных, его свойства 39](#_Toc393445854)

[31. Полный дифференциал функции многих переменных. 40](#_Toc393445855)

[32. Дифференциалы высших порядков. 41](#_Toc393445856)

[33. Экстремумы функции многих переменных. Необходимые условия. 42](#_Toc393445857)

[34. Достаточные условия экстремума функции многих переменных. 43](#_Toc393445858)

[35. Понятие условного экстремума. 45](#_Toc393445859)

[36. Метод множителей Лагранжа. 46](#_Toc393445860)

[37. Числовые ряды, основные понятия, сходимость ряда. 47](#_Toc393445861)

[38. Абсолютная и условная сходимость рядов. 48](#_Toc393445862)

[39. Ряды с неотрицательными членами. Признаки сходимости положительных рядов. 49](#_Toc393445863)

[40. Степенные ряды. Промежуток сходимости степенного ряда. 50](#_Toc393445864)

[41. Понятие типа данных в различных языках программирования. Основные типы (на примере языка Паскаль или др.). 51](#_Toc393445865)

[42. Конструирование новых типов данных. 53](#_Toc393445866)

[43. Структурные типы данных в языках программирования. Особенности их представления в памяти ЭВМ. 54](#_Toc393445867)

[44. Файловые типы данных в языках программирования. Операции с файлами. 56](#_Toc393445868)

[45. Понятие формата файла. Примеры известных форматов файлов. 61](#_Toc393445869)

[46. Работа со списками в языках программирования. Представление списков с использованием динамической (управляемой) памяти (на примере языка Паскаль или др.). 62](#_Toc393445870)

[47. Операции со списками. 64](#_Toc393445871)

[48. Работа с бинарными деревьями в языках программирования. Представление бинарных деревьев с использованием динамической (управляемой) памяти (на примере языка Паскаль или др.). 68](#_Toc393445872)

[49. Операции с бинарными деревьями. 69](#_Toc393445873)

[50. Алгоритмы и задачи работы с множествами в языках программирования. 72](#_Toc393445874)

[51. Программное представление графов. 80](#_Toc393445875)

[52. Реализация операций над графами. 81](#_Toc393445876)

[53. Понятие рекурсии в языках программирования. Рекурсивные процедуры и функции. 84](#_Toc393445877)

[54. Механизм выполнения рекурсивных процедур. 86](#_Toc393445878)

[55. Методы и алгоритмы сортировки одномерных массивов. 88](#_Toc393445879)

[56. Трансляторы языков программирования. Назначение основных блоков транслятора, схема их взаимодействия и используемые ими структуры данных. 90](#_Toc393445880)

[57. Компиляторы. Основные функции компилятора. 92](#_Toc393445881)

[58. Описание языков программирования с помощью БНФ. 93](#_Toc393445882)

[59. Описание языков программирования с помощью синтаксических диаграмм. 94](#_Toc393445883)

[60. Восходящие методы грамматического разбора. 96](#_Toc393445884)

[61. Нисходящие методы грамматического разбора. 100](#_Toc393445885)

[62. Основы архитектуры ЭВМ. Номенклатура сегментных регистров. 105](#_Toc393445886)

[63. Варианты задания исполнительного адреса в архитектурах Intel или др. 107](#_Toc393445887)

[64. Особенности языков ассемблерного типа. 109](#_Toc393445888)

[65. Макросредства языка Ассемблер (на примере архитектуры Intel или др.) 110](#_Toc393445889)

[66. Операционные системы, их назначение и функции. Виды операционных систем, их характеристики. 113](#_Toc393445890)

[67. Управление процессами. Понятия процесса, состояния процесса. 117](#_Toc393445891)

[68. Операции над процессами. Асинхронные параллельные процессы. 119](#_Toc393445892)

[69. Взаимоисключение. 122](#_Toc393445893)

[70. Объектно-ориентированное программирование. Классы и объекты. Наследование. 123](#_Toc393445894)

[71. Объектно-ориентированное программирование. Полиморфизм. 124](#_Toc393445895)

[72. Технологии программирования. Основные технологические подходы. Жизненный цикл программного продукта. Модели ЖЦ ПП. 125](#_Toc393445896)

[73. Компьютерные сети. Типы каналов связи. Эталонная модель OSI. Понятия: протокол и интерфейс. 131](#_Toc393445897)

[74. Маршрутизация в компьютерных сетях: способы и методы. 134](#_Toc393445898)

[75. Глобальная компьютерная сеть Интернет. Основные принципы построения и управления сетью. 137](#_Toc393445899)

[76. Адресация в Интернет. 140](#_Toc393445900)

[77. Основные службы Интернет. 142](#_Toc393445901)

[78. Инструментальные средства разработки Web-сайтов. Языки HTML, XML. 145](#_Toc393445902)

[79. Основные понятия и проблемы защиты информации (угрозы, требования, критерии, способы, средства). 148](#_Toc393445903)

[80. Методы построения систем защиты информации (аппаратные, программные, организационные и др. аспекты). 151](#_Toc393445904)

# Предел последовательности. Бесконечно малая и бесконечно большая величины. Арифметические действия с переменными, имеющими предел.

Последовательностью называется любой счётный набор действительных чисел ***а***1, ***а***2, ***а***3,…, ***аn***,….

Обозначается .

Число ***а*** называется пределом последовательности , если для любого положительного числа ε существует такое натуральное число ***N*** (зависящее от ε), что для членов последовательности с номерами ***n***>***N*** выполняется неравенство | ***an*** - ***a*** |<ε.

Обозначения: ; ;  при .

Если  при , то говорят также, что последовательность  сходится к числу ***а***.

Функция ***f***(***x***) называется бесконечно малой при ***х***→***a***, если .

БМ функции принято обозначать греческими буквами:α(***х***), β(***х***) и т.д, так и будем делать. Перевод определения на язык ε-δ:

α(***х***) - БМ при ***х***→***a*** ⇔ {∀ε>0 ∃δ: 0<| ***x***-***a*** |<δ⇒|α(***х***)|<ε}.

БМ обладают всеми свойствами функций, имеющих предел. В этом разделе мы изучим специфические свойства БМ.

**Теор. 1.1** Произведение БМ на ограниченную функцию - БМ функция.

**Док-во**. Пусть α(***х***) - БМ при ***х***→***a***, ***f***(***x***) ограничена в окрестности точки ***a***. Требуется доказать, что α(***х***) ***f***(***x***) - БМ при ***х***→***a***. ∃С>0: | ***f***(***x***) |<C; ∀ε>0 ∃δ: 0<| x-a |<δ⇒|α(***х***)|<ε/C⇒

| α(***х***) ***f***(***x***)|<ε, т.е. α(***х***) ***f***(***x***) действительно БМ при ***х***→***a***.

**Теор. 1.2** Алгебраическая сумма конечного числа БМ функций - БМ функция.

**Док-во**. Пусть α(***х***), β(***х***) - БМ при ***х***→***a***. Требуется доказать, что α(***х***) ±β(***х***) - БМ при ***х***→***a***. ∀ε>0 ∃δ1: 0<| x-a |<δ1⇒|α(***х***)|<ε/2; ∃δ2: 0<| x-a |<δ2⇒|β(***х***)|<ε/2. Если взять 0<| x-a |<min{δ2,δ1}=δ, то | α(***х***) ±β(***х***)|≤ | α(***х***) |+ | β(***х***)|< ε/2+ε/2=ε, т.е. α(***х***) ±β(***х***) действительно БМ при ***х***→***a***.

**Следствие**: Линейная комбинация БМ функций - БМ функция.

Докажем теорему, которой придётся часто пользоваться и которая, в основном, объясняет причину выделения БМ функций в отдельный класс:

**Теор. 1.3** (о связи функции с её пределом). Для того, чтобы функция ***f***(***x***) имела предел, равный ***b***, при ***х***→***a***, необходимо и достаточно, чтобы ***f***(***x***) представлялась в виде ***f***(***x***)= ***b***+α(***х***) , где α(***х***) - БМ при при ***х***→***a***.

**Док-во**. **Необходимость**. Пусть ∃. Обозначим α(***х***)= ***f***(***x***) - ***b***, докажем, что α(***х***) - БМ при при ***х***→***a***. По определению предела ∀ε>0 ∃δ: 0<| ***x***-***a*** |<δ⇒| ***f***(***x***) - ***b*** |=|α(***х***)|<ε, т.е. α(***х***) удовлетворяет определению БМ.

**Достаточность**. Для доказательства достаточно прочитать доказательство необходимости в противоположном порядке.

Функция ***f(x***) называется бесконечно большой при ***х***→***а***, если .

Обозначение: .

Функция ***f(x***) называется положительной бесконечно большой при ***х***→***а***, если .

Функция ***f(x***) называется отрицательной бесконечно большой при ***х***→***а***, если .

Такие же определения даются для случаев ***х***→***а***+0, ***х***→***а***-0, ***х***→+∞, ***х***→-∞.

**Теорема 1.4** Пусть функции ***f***(***x***), ***g***(***x***) имеют предел при ***х***→***a***, С=const. Тогда имеют пределы функции С ***f***(***x***), ***f***(***x***)±***g***(***x***), ***f***(***x***)***g***(***x***),  ( если ), и

1. ;

2. ;

3. ;

4. .

**Док-во** основано на **теореме 1.3** о связи функции с её пределом. Пусть , ⇒ ***f***(***x***)=***b***1+α(***х***), ***g***(***x***)=***b***2+β(***х***), где α(***х***), β(***х***) - БМ. Тогда:

1. С***f***(***x***)=С***b***1+Сα(***х***); Сα(***х***) - БМ по **теор. 1.1**⇒∃.

2. ; α(***х***)±β(***х***) - БМ⇒

∃.

3. . Выражение в квадратных скобках - БМ ⇒∃.

4. Оценим: . В числителе стоит БМ, функция - ограничена при  (почему?) ⇒∃.

С двумя функциями можно произвести ещё следующие действия: возвести ***f***(***x***) в степень ***g***(***x***) и взять их суперпозицию. Для степени ***f***(***x***)***g***(***x***) оказывается, что если существуют конечные , , то существует , это следствие непрерывности показательной и логарифмической функций; и этот вопрос будет рассмотрен ниже. Для суперпозиции функций оказывается, что существование пределов внешней и внутренней функций недостаточно для существования предела сложной функции. Более точно, если ***х***=***g***(***t***) имеет предел ***а*** при ***t***→ ***t0***, функция ***y***=***f***(***x***) имеет предел при ***x*** → ***а***, то  может не существовать. Пример: пусть . Очевидно, ∃. Пусть . ∃. Для последовательности точек  ; если выбрать последовательность , не попадающую в эти точки, то . Две последовательности дают разные пределы⇒ не существует. Дальше мы увидим, что существование предела сложной функции обеспечивает непрерывность внешней функции.

# Теорема о монотонной последовательности. Второй замечательный предел.

Теорема Вейерштрасса. (Основная теорема теории последовательностей).

1. Если последовательность возрастает и ограничена сверху, то она сходится (т.е. имеет предел).

2. Если последовательность убывает и ограничена снизу, то она сходится.

Данную теорему можно сформулировать немного иначе - Любая монотонная и [ограниченная последовательность](http://www.webmath.ru/poleznoe/formules_7_2.php)  имеет предел.

Док-во. Докажем утверждение 1. (2 доказывается аналогично). Так как множество чисел  ограничено сверху, оно имеет точную верхнюю грань . По свойствам точной верхней грани 1. an a; 2. для ∀ε>0 существует элемент множества aN такой, что aN>a-ε. Если n> N, то a-ε< aN an(вследствие монотонного возрастания) a<a+ε. Итак, для ∀ε>0 мы нашли такое N, что при n> N имеет место a-ε<an<a+ε, т.е. доказали, что ∃.

Изучая пределы последовательностей, мы доказали, что ∃. Распространим это доказательство на случай действительной переменной, докажем, что . Пусть n=E(x), тогда n≤ x <n+1. Если x →+∞, то и n→∞, поэтому можем считать n >1. Из неравенства  вследствие монотонного возрастания степенной функции с аргументом и степенью >1, получим . Предел правого члена при n→∞ равен числу е, предел левого  тоже равен числу е. По теореме о пределе промежуточной функции ∃, и он тоже равен числу е. Далее, , и снова применяя теорему о пределе промежуточной функции, получаем, что  существует и равен числу е.

Пусть теперь x →-∞. Введём новую переменную y=-x-1,тогда x=-y-1, и y→+∞ при x →-∞. . Доказано, что односторонние пределы при x →±∞ существуют и равны⇒ ∃.

Эквивалентная форма второго замечательного предела:  (сводится к предыдущему случаю заменой ).

# Фундаментальная последовательность. Критерий Коши сходимости последовательности.

Из определения сходимости последовательности  к точке a вытекает, что для любого \varepsilon > 0 интервалом длиной 2\varepsilon можно накрыть всю эту последовательность, исключением может быть конечное число ее элементов, если середину интервала поместить в точке a. Справедливо и обратное : если последовательность  такова, что для любого \varepsilon > 0 можно накрыть всю эту последовательность, исключая может быть конечное число ее элементов, поместив центр интервала в некоторую точку, то она сходится. Сформулируем это утверждение более точно.

Определение. Подпоследовательность x_{n_k} называется последовательностью Коши или фундаментальной, если \forall \varepsilon > 0 \exist N ( \varepsilon ) \forall n, m > N ( \varepsilon ) : \mid x_n - x_m \mid <\varepsilon  

Теорема (Критерий Коши). Для того, чтобы последовательность \{{ x_n \}} сходилась, необходимо и достаточно чтобы она была фундаментальной.

Доказательство:

Необходимость. Пусть \{{ x_n \}} сходится. \lim_{n \to \infty} x_n = x

\forall \varepsilon > 0 \exist N ( \varepsilon ) \forall n, m > N ( \varepsilon ) : \mid x_n - x_m \mid  < \varepsilon 

\forall p \in N \mid x_{n+p} - x_n \mid \le \mid x_{n+p} - x \mid < 2\varepsilon  \forall n > N(\varepsilon)

Достаточность. Пусть \{{ x_n \}} - фундаментальная последовательность. Докажем, что она ограничена и \bar x = \underline x.

Так как последовательность фундаментальна, то \forall \epsilon > 0 \exist x_N, в \varepsilon-окрестности которой существуют все элементы x_1, x_2, x_3, ..., x_{N-1} .

Предположим, A = max \{{ |x_1|, |x_2|, | x_3|, ..., |x_{N-1}|, |x_n - \varepsilon|, |x_n + \varepsilon| \}}.

В отрезке [A, -A] содержатся все элементы последовательности, т.е. \{{ x_n \}} - ограничена.

Вследствие теоремы Больцано-Вейерштрасса (\bar x ; \underline x) < (x_n - \varepsilon; x_n + \varepsilon).

0 \le \bar x - \underline x < 2\varepsilon в силу произвольности \varepsilon

\bar x - \underline x = 0

\bar x = \underline x

# Определения предела функции. Первый замечательный предел.

Пусть а - предельная точка области определения Х функции f(x). Число b называется пределом функции при х, стремящемся к а, если для любого числа ε>0 существует такое число δ (вообще говоря, положительное и зависящее от ε), что если х∈Х принадлежит также проколотой δ-окрестности  точки а, то значение функции f(x) принадлежит ε-окрестности числа b.

Обозначения: ; f(x)→ b при x→ а; .

Краткая форма записи: .

0

***b***

***а***

***b***-ε

***b***+ε

***а***-δ

***а***+δ

Иллюстрация:

***x***

***y***

Неравенство  расписывается в виде двустороннего неравенства как  или . Аналогично неравенство  можно расписать как . Поэтому смысл определения предела таков: , если для любой наперед заданной степени близости значений f(x) к числу b мы в состоянии найти такую близость аргумента х к числу а, которая обеспечивает эту близость f(x) к b. Заметим, что в определении никак не участвует значение f(а) функции f(x) в точке а, в частности, f(а) не обязательно должно быть равным b; более того, f(x) может быть вообще не определена в точке а.

Так принято называть . Докажем, что он равен единице. 1. Докажем, что sin| x |≤.| x | (достаточно доказать это при х>0). Рассмотрим круг радиуса 1 с центром в точке О. В качестве переменной х будем брать центральный угол, отсчитываемый в радианах от радиуса ОА. Тогда длина дуги АВ =х, длина отрезка ВD =sin х, sin х< х (при х ≠0; перпендикуляр - кратчайшее расстояние от точки до прямой). 2. Сравним площади треугольников OBА, OCA и сектора OBA: S(тр.OBА)<S(сек.OBA)<S(тр.OCA). Выразим эти площади:  (CA=tg x). Делим это выражение на : . Мы получили эти неравенства в предположении х>0, но вследствие четности входящих в них выражений они верны при любом знаке х. 3. Переворачиваем эти неравенства: . cos x→1 при х→0, предел правой части тоже равен 1, по теореме о пределе промежуточной функции ∃.

Следствия: .

# Непрерывность функции. Разрывы первого и второго рода.

Опр. 5.1. Пусть функция y = f(x) определена в точке х0 и некоторой её окрестности. Функция f(x) называется непрерывной в точке х0, если:

1. существует ;
2. этот предел равен значению функции в точке х0: .

При определении предела подчёркивалось, что f(x) может быть неопределена в точке х0, а если она определена в этой точке, то значение f(х0) никак не участвует в определении предела. При определении непрерывности принципиально, что f(х0) существует, и это значение должно быть равно .

Опр. 5.2. Пусть функция y = f(x) определена в точке х0 и некоторой её окрестности. Функция f(x) называется непрерывной в точке х0, если для ∀ε>0 существует положительное число δ, такое что для всех х из δ-окрестности точки х0 (т.е. если |х- х0 |<δ) выполняется неравенство | f(x) - f(х0) |<ε.

Здесь учитывается, что значение предела должно быть равно f(х0), поэтому, по сравнению с определением предела, снято условие проколотости δ-окрестности 0<|х- х0 |.

Дадим ещё одно (равносильное предыдущим) определение в терминах приращений. Обозначим Δх=х - х0, эту величину будем называть приращением аргумента. Так как х → х0, то Δх→0, т.е. Δх - БМ величина. Обозначим Δу = f(x) - f(х0), эту величину будем называть приращением функции, так как |Δу| должно быть (при достаточно малых |Δх|) меньше произвольного числа ε>0, то Δу - тоже БМ величина, поэтому

Опр. 5.3. Пусть функция y=f(x) определена в точке х0 и некоторой её окрестности. Функция f(x) называется непрерывной в точке х0, если бесконечно малому приращению аргумента соответствует бесконечно малое приращение функции.

***f***(***х0***)

***х0***

Δ***х***

Δ***y***

Непрерывная функция

***f***(***х0***)

***х0***

Δ***х***

Δ***y***

Разрывная функция

Ещё одно равносильное определение на языке последовательностей:

Опр. 5.4. Функция f(x) называется непрерывной в точке х0, если для любой последовательности  точек области определения, сходящейся к х0, последовательность соответствующих значений функции  сходится к f(х0):

.

Опр. 5.5. Функция f(x) не являющаяся непрерывной в точке х0, называется разрывной в этой точке.

Опр. 5.6. Функция f(x) называется непрерывной на множестве Х, если она непрерывна в каждой точке этого множества.

Для того, чтобы функция f(x) была непрерывной во внутренней точке х0 области определения, необходимо и достаточно выполнение четырёх условий:

1. f(x) определена в точке х0 (т.е. ∃ f(х0)) и некоторой её окрестности;

2. ∃;

3. ∃;

4.Все эти три числа равны между собой: (в правом и левом концах области определения снимаются условия, относящиеся, соответственно, к пределам справа и слева).

Опр.5.7 Если хотя бы одно из перечисленных условий непрерывности функции в точке не выполняется, f(x) называется разрывной в точке х0, а сама точка х0 называется точкой разрыва функции f(x).

Рассмотрим возможные варианты:

Опр.5.8 Точка разрыва х0 называется точкой устранимого разрыва, если существуют односторонние пределы и они равны между собой (т.е. ∃).

Из этого определения следует, что точка разрыва х0 может быть точкой устранимого разрыва только в случае, когда значение f(x) в точке х0 либо не определено, либо не равно .

Пример:

10

0

10

0

1

***х***

***x***

sin***x***

. Эта функция не определена в точке х0 = 0, но ∃⇒ существуют односторонние пределы, и они равны. Следовательно, точка х0 = 0 - точка устранимого разрыва. Если доопределить функцию в этой точке:  то будет получена непрерывная в точке х0 = 0 функция, таким образом, разрыв будет "устранён".

Опр.5.10 Точка разрыва х0 называется точкой разрыва первого рода (иногда применяется термин "скачок"), если существуют односторонние пределы, но они не равны между собой.

Пример:

.

-1

1

10

0

10

0

sgn

(***x***)

***x***

 (сигнум, "знак-функция"). При х→+0 у(х)→1 (справа от точки 0 у(х)=const=1); при х→-0 у(х)→-1, у(х+0) и у(х-0) существуют и не равны⇒точка х0 = 0 - точка разрыва первого рода.

Опр.5.11 Точка разрыва х0 называется точкой разрыва второго рода, если хотя бы один из односторонних пределов  не существует ( в частности, он может быть бесконечным).

Для точек разрыва любого типа не требуется существования f(х0).

1

0

10

10

5

***х***

0

Пример: . Любая точка, кроме х0=0, принадлежит области определения, поэтому функция в ней непрерывна. При х→-0 1/x→-∞, поэтому 21/х→0, т.е. конечный предел слева существует. При х→+0 1/x→+∞, поэтому 21/х→∞, т.е. конечного предела справа не существует, следовательно, точка х0=0 - точка разрыва второго рода. Второй пример - функция .

Определение предела функции (график - слева; х≠0). Эта функция не имеет ни левого, ни правого односторонних пределов при х→0, т.е. х0=0 - точка разрыва второго рода.



# Операции над непрерывными функциями.

Теор.6.1 о непрерывности суммы, произведения, частного. Пусть функции f(x), g(x) непрерывны в точке х0. Тогда в этой точке непрерывны функции f(x)±.g(x), f(x)g(x),  (частное - в случае, когда g(х0)≠0).

Док-во непосредственно следует из теор.1.4 Для примера докажем непрерывность частного. Пусть f(x), g(x) непрерывны в точке х0, т.е. , , причём g(х0)≠0. По теор.1.4 существует , и этот предел равен , что означает непрерывность функции  в точке х0.

Теор.6.2 о переходе к пределу под знаком непрерывной функции. Пусть функция  определена в некоторой окрестности точки t0 и имеет , равный х0. Пусть точка  принадлежит области определения функции y = f(x), и f(x) непрерывна в точке х0. Тогда существует, и .

Док-во. Возьмём ∀ε>0. Так как f(x) непрерывна в точке х0, то ∃σ>0, такое что | х- х0|<σ⇒ ⇒ | f(x)- f(x0)|<ε. Так как существует = х0, то для σ ∃δ>0, такое что 0<| t- t0|<δ ⇒

⇒ |ϕ (t)- х0|<σ. Таким образом, для ∀ε>0 мы нашли такое δ>0, что из 0<| t- t0|<δ⇒

⇒ | f(x)- f(x0)|= | f(ϕ (t))- f()|<ε, что означает существование предела и равенство этого предела величине .

Теор.6.3 о непрерывности суперпозиции непрерывных функций. Пусть функция  непрерывна в точке точке t0. Пусть точка  принадлежит области определения функции y = f(x), и f(x) непрерывна в точке х0. Тогда сложная функция  непрерывна в точке t0.

Док-во непосредственно следует из предыдущей теоремы. Так как ϕ (t) непрерывна в точке t0, то . Поэтому , что и означает непрерывность сложной функции  в точке t0.

# Теоремы Вейерштрасса о функции непрерывной на отрезке.

Теор.6.4 об обращении функции в нуль. Если функция f(х) непрерывна на отрезке [a,b] и принимает на концах этого отрезка значения разных знаков, то найдётся точка с∈[a,b], в которой функция обращается в нуль: f(с) =0, a<c<b.

*a1=a2=a3*

*a=a0*

*b=b0=b1*

*x*

*y*

*c*

*b2*

*b3*

*a4*

Док-во. Рассмотрим случай, когда f(а)>0, f (b)<0. Возьмём среднюю точку отрезка х1=(а+b)/2. Если f(х1)=0, то теорема доказана и с =х1; иначе на концах одного из отрезков [a, х1], [х1,b] функция опять принимает значения разных знаков (на рис. это второй отрезок), при том опять на левом конце f(х)>0, на правом - f(х) <0. Обозначим этот отрезок [a1, b1]. Снова поделим этот отрезок пополам: х2=(а1+b1)/2 и снова либо f(х2)=0, либо на концах одного из отрезков [a1, х2], [х2,b1] функция опять принимает значения разных знаков. Продолжая этот процесс, получим последовательность вложенных отрезков [an, bn], имеющую общую точку с. Для этой точки an≤ с ≤ bn, bn-аn→0 при n→∞, поэтому . В каждой точки аn справедливо f(аn)>0, в каждой точки bn f(bn)<0, переходя в неравенствах f(аn)>0, f(bn)<0 к пределам при n→∞ и пользуясь непрерывностью f(х), получим: , , одновременно это может иметь место, только если : f(с) =0. Теорема доказана.

Необходимость непрерывности: функция sgn x + 0.5 принимает на концах отрезка [-1,1] значения разных знаков, но нигде не обращается в нуль. Необходимость замкнутости множества: функция 1/x непрерывна в каждой точке множества [-1,0)∪(0,1], принимает значения разных знаков в его левом и правом концах, но нигде не обращается в нуль.

Теор.6.5 о промежуточном значении. Если функция f(х) непрерывна на отрезке, и в двух точках a и b (a < b) принимает неравные значения A= f(а)≠B= f(b), то для любого числа С, лежащего между A и B, найдётся точка с∈[a,b], в которой значение функции равно С: f(с) = С.

Док-во. Пусть A>B, тогда A>С>B. Рассмотрим функцию g(х)= f(х)-C. Эта функция непрерывна на [a,b] и принимает на его концах значения разных знаков: g(а)= f(а)-C=А-С>0, g(b)= f(b)-C=В-С<0. Тогда по теореме об обращении функции в нуль найдётся точка с∈[a,b], в которой функция обращается в нуль: g(с) =0⇒ g(с)= f(с)-C=0⇒ f(с) = C. Теорема доказана.

Теор.6.6 об ограниченности непрерывной функции на отрезке. Если функция f(х) непрерывна на отрезке, то она ограничена на этом отрезке.

Док-во от противного. Предположим, что f(х) неограничена на [a,b], тогда для любого числа n найдётся точка xn, в которой | f(х)|> n. Из последовательности { xn } можно выделить подпоследовательность , сходящуюся к точке x0∈[a,b]. Так как f(х) непрерывна на [a,b], то , но это невозможно, так как  при . Это противоречие и доказывает теорему.

Необходимость непрерывности: функция разрывна в единственной точке отрезка [-1,1], но неограничена на этом отрезке. Необходимость замкнутости множества: функция 1/x непрерывна в каждой точке множества [-1,0)∪(0,1], но неограничена на этом множестве.

Теор.6.7 о достижении минимального и максимального значений. Если функция f(х) непрерывна на отрезке, то она достигает на этом отрезке свои нижнюю и верхнюю грани.

Док-во. Пусть М\*=. Требуется доказать, что , в котором f(х1)=М\*. От противного: предположим, что для  f(х)<М\*. Рассмотрим функцию . Так как, по предположению, знаменатель не обращается в нуль, эта функция непрерывна, следовательно, по теореме об ограниченности непрерывной функции на отрезке ограничена: ⇔, т.е. число , меньшее М\*, оказывается верхней границей множества , что противоречит определению верхней грани. Аналогично доказывается, что , в котором f(х2)= М\*=.

Необходимость непрерывности и замкнутости множества демонстрируют те же примеры, что и для предыдущей теоремы.

Следствие всех предыдущих теорем: множество значений непрерывной на отрезке [a,b] функции заполняет весь отрезок [М\*, М\*]. В дальнейшем величину  будем обозначать просто М, величину  будем обозначать символом m.

Теор.6.8 о непрерывности обратной функции. Пусть функция у= f(x) непрерывна и строго возрастает (убывает) на отрезке [a,b]. Тогда на отрезке [m,М] существует обратная функция х = g(у), также монотонно возрастающая (убывающая) на [m,М] и непрерывная.

Док-во. Как доказано выше, множество значений непрерывной функции заполняет весь отрезок [m,М], т.е. для ∀у∈[m,М] ∃! х∈[a,b], такой что у= f(x). Таким образом, функция х = g(у) определена в каждой точке отрезка [m,М]. Эта функция имеет тоже направление монотонности, что и исходная у= f(x): если, например, f(x) возрастает, то из ⇒. Тогда, по смыслу функции х = g(у), из ⇒. Эта функция непрерывна.

# Определение производной функции. Геометрический и физический смысл производной функции.

Задачи, приводящие к понятию производной.



***tg***α=***v***ср

***t0***

***t1***

Δ***t***

Δ***s***

1. Вычисление скорости неравномерно движущегося тела. Пусть материальная точка неравномерно движется вдоль оси Ох. Известна зависимость пути s(t), пройденного к моменту времени t от времени, требуется найти значение скорости точки в момент t0. Если мы возьмём любое t1≠ t0 и найдём отношение  , то будет получено среднее значение  скорости на отрезке [t0, t1]. Чтобы получить мгновенное значение скорости в момент t0, мы должны устремить t1 к t0, т.е. найти предел , где Δt= t1- t0, Δs= s(t1)- s(t0).

2. Нахождение углового коэффициента касательной к графику функции.

Опр. 6.1.1.2. Касательной к графику функции y=f(x) в точке M0(x0,y0=f(x0)) называется предельное положение секущей M0M1 при M1→ M0.

α

***y1***

***M1***

***M0***

***х***

***у***

Δ***x***

Δ***y***

***x1***

***y0***

***x0***

***tg***α=Δ***y***/Δ***x***

Угловой коэффициент секущей равен . Чтобы получить угловой коэффициент касательной, в этом выражении надо перейти к пределу при M1→ M0, или, что тоже самое, при х1→ х0. Следовательно, , где . Величины Δх и Δу называются, соответственно, приращением аргумента и функции. Таким образом, при решении этих совершенно разных задач, как и множества других задач науки и техники, требуется находить предел отношения приращения функции к приращению аргумента. Это приводит к определению основного понятия дифференциального исчисления - понятия производной.

Определение производной.

Опр. Пусть функция y=f(x) определена в точке х и некоторой её окрестности. Придадим значению аргумента х приращение Δх (положительное или отрицательное, но не выводящее за пределы этой окрестности) и найдем соответствующее приращение функции Δу=f(x+Δх)- f(x). Передел отношения приращение функции Δу к приращению аргумента Δх при Δх →0 называется производной функции y=f(x) в точке х.

Производную обозначают разными способами. Наиболее распространённые обозначения - . Чаще мы будем применять первое из этих обозначений. Таким образом, . Операция нахождения производной называется дифференцированием.

Геометрический смысл производной. Уравнения касательной и нормали к графику гладкой функции.

Геометрический смысл производной у'(x0), как следует из вышеизложенного, - угловой коэффициент касательной к графику функции y=f(x) в точке (x0,y0=f(x0)). Не любая функция имеет касательную в каждой точке, так, невозможно построить касательную к графику функции |x| в точке (0,0). Чтобы в точке (x0,y0=f(x0)) существовала касательная, необходимо существование предела , т.е. существование производной. Функции, имеющие производную в каждой точке своей области определения (т.е. функции, графики которых имеют касательную в каждой точке), будем называть гладкими. Применяя известные формулы аналитической геометрии для прямой, проходящей через данную точку с данным угловым коэффициентом, получаем:

уравнение касательной в точке (x0,y0=f(x0)): ;

уравнение нормали к графику функции в точке (x0,y0=f(x0)):  (при условии, что у'(x0)≠0).

Физический смысл производной.

Если точка движется вдоль оси х и ее координата изменяется по закону  x(t), то мгновенная скорость точки:

http://yaklass-shkola.s3-eu-west-1.amazonaws.com/goods/ymk/algebra/work8/theory/17/7.gif

# Дифференциал функции. Геометрический смысл.

9.1 Определение дифференцируемости и дифференциала. Пусть функция y = f(x) определена в точке х и некоторой окрестности этой точки и непрерывна в точке х. Тогда приращению Δх аргумента соответствует приращение Δу = f(x+Δх)- f(x), бесконечно малое при Δх→0. В особый класс дифференцируемых функций выделяются функции, для которых Δу с точностью до бесконечно малой высшего порядка по сравнению с Δх линейна по Δх. Более точно:

Опр.9.2. Функция y = f(x) называется дифференцируемой в точке х, если её приращение Δу в этой точке можно представить в виде , где А - не зависящая от Δх величина, α(Δх) - БМ высшего порядка по сравнению с Δх:  при Δх→0.

В более краткой записи для дифференцируемой в точке х функции .

Опр.9.3. Главная часть приращения Δу дифференцируемой функции, линейная относительно приращения Δх аргумента (т.е. ), называется дифференциалом функции и обозначается dy (или df(x)).

Связь между дифференцированием и дифференцируемостью даёт

Теор.9.4. Для того, чтобы функция y = f(x) имела в точке х конечную производную

y' = f'(x), необходимо и достаточно, чтобы она была дифференцируемой в этой точке.

Док-во. Необходимость. Пусть в точке х существует конечная производная y'. По теор. о приращении функции, имеющей производную, Δу= у'(x) Δх + α(Δх) Δх, где α(Δх) - бесконечно малая функция при Δх →0. Сравнивая это выражение с определением 9.2, делаем вывод: А= у'(x), БМ α(Δх) Δх имеет более высокий порядок по сравнению с Δх, т.е. f(x) действительно дифференцируема в точке х.

Достаточность. Пусть f(x) дифференцируема в точке х, т.е. её приращение Δу можно представить в виде , где А - не зависящая от Δх величина, α(Δх) - БМ высшего порядка по сравнению с Δх:  при Δх→0. Тогда . Следовательно, существует предел отношения приращения функции к приращению аргумента, т.е. ∃ у'(x), и у'(x)=А.

Таким образом, для функции одной переменной существование производной и дифференцируемость - эквивалентные свойства. При этом коэффициент А всегда равен у'(x), и выражение для дифференциала приобретает вид dy = у'(x) Δх. Для независимой переменной х принимают dх= =Δх (формально это можно обосновать так: если у=х, то у'(x)=1, и dy = dх = Δх). Итак, окончательное выражение для дифференциала имеет вид .

α

***M1***

***M***

***х***

***у***

Δ***x=dx***

Δ***y***

***x+dx***

***y***

***x***

***y+***Δ***y***

***dy***

***M2***

***y+dy***

***А***

***В***

Важно осознать, что в этом выражении не обязательно понимать dх как бесконечно малую, dх - произвольное не зависящее от х приращение аргумента (но именно при dх→0 и dу→0, и призведение у'(x)dх = dy становится главной частью приращения функции). Так как у'(x)=tg(α) - угловой коэффициент касательной, то геометрически дифференциал dy - это приращение ординаты касательной при смещении абсциссы на dх =Δх. Значение dy может значительно отличаться от приращения функции Δу, но при достаточно малых Δх (в окрестности точки касания) они близки (участок АВ графика функции).

9.5 Инвариантность формы первого дифференциала. Здесь мы рассмотрим одно важное свойство дифференциала, следующее из формулы для производной сложной функции : если функции и  имеют в соответствующих точках производные и , то производная сложной функции  равна .

Если х - независимая переменная, то формула для дифференциала: . Если , то . Таким образом, независимо от того, является ли х независимой переменной, или сама эта переменная х является функцией другой переменной t, формула для нахождения дифференциала первого порядка одна и та же. Это свойство и называется инвариантностью формы первого дифференциала, и часто применяется в теории и решении задач.

9.6 Правила для вычисления дифференциала. Примеры вычисления дифференциала. Правила для вычисления дифференциала - прямое следствие правил дифференцирования (раздел 6.5):

;

;

;

.

Докажем, для примера, формулу 3: .

При нахождении дифференциала можно вычислить производную и затем применить формулу :

, поэтому ;

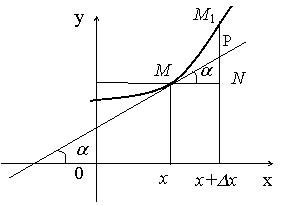
но более квалифицированным является прямое применение правил вычисления дифференциала:





9.7 Применение дифференциала в приближённых вычислениях. Именно близость исходной функции и её касательной в окрестности точки касания служит источником многочисленных приближённых формул для вычисления значений функций. Δу= у'(x) Δх + α(Δх) Δх, где α(Δх) - БМ при Δх →0; с учётом того, что у'(x) Δх = у'(x)dх = dy, пренебрегая бесконечно малым слагаемым высшего порядка по сравнению с Δх, получим Δу ≅ dу. Так как Δу=у(x+Δх)- у(x), то формула для приближённого значения у(x+Δх) будет иметь вид у(x+Δх) ≅ у(x)+ у'(x) Δх. На практике этой формулой пользуются так. Пусть требуется вычислить значение функции в точке х1. Подбирают близкую к точке х1 точку x, в которой легко вычислить точное значение у(x) и у'(x), тогда Δх = х1- х и у(x+Δх) ≅ у(x)+ у'(x) Δх. Примеры:

Вычислить . В этом случае , функция и производная легко вычисляется в близкой точке х=32, у(х)=2, у'(х)=1/(5\*24)=1/80, х1=30, Δх =30-32= -2, и

 ≅2-2/80 = 1.975 (более точное значение 1.97435).

Вычислить sin(0.5). y(x)=sinx, y'(x)=cosx, в качестве х примем x = π/6≅0.524, х1=0.5,

Δх =0.5-0.524= -0.024, y(x)=0.5, y'(x)=≅0.866, y(х1) ≅ 0.5 - 0.024\*0.866≅0.5-0.021=0.479 (более точное значение 0.47943).

На графике функции http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image003.gif возьмем произвольную точку http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image005.gif и дадим аргументу http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image007.gif приращение http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image009.gif. При этом функция получит приращение http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image011.gif(на рисунке отрезок http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image013.gif).

Проведем касательную к кривой http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image003.gif в точке http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image016.gif и обозначим угол ее наклона к оси http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image018.gif через http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image020.gif,тогда http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image022.gif.Из треугольника http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image024.gif находим http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image026.gif http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image028.gif, т.е. http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image030.gif.

Таким образом, дифференциал функции численно равен приращению ординаты касательной, проведенной к графику функции http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image003.gif в данной точке, когда аргумент http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image007.gif получает приращение http://ios.sseu.ru/public/eresmat/metod/met1/razdmet1_4/parmet1_4_2.files/image009.gif.

# Производная и дифференциал высшего порядка.

10.1. Производные высших порядков. Формула Лейбница. Пусть функция  имеет производную y'(x) в каждой точке интервала (а,b). Функция y'(x) тоже может иметь производную в некоторых точках этого интервала. Производная функции y'(x) называется второй производной (или производной второго порядка) функции и обозначается . Функция y''(x) тоже может иметь производную, которая называется третьей производной (или производной третьего порядка) функции и обозначается . Вообще n-ой производной (или производной n-ого порядка) функции называется производная от производной n-1-го порядка (обозначения: ).

Производные высших порядков последовательно вычисляются по уже известным формулам и правилам. Пусть, например, . Тогда , , ,  и т.д. В некоторых случаях можно получить общее выражение для n-ой производной функции: пусть . Тогда , , , и вообще . Аналогичную формулу можно получить для косинуса. Другой пример: . Если представить эту функцию в виде , то ,, и вообще .

Для высших производных произведения функций справедлива формула Лейбница:

. Эта формула внешне похожа на формулу бинома Ньютона и, также как формула бинома Ньютона, может быть доказана методом математической индукции. Для низших производных:

; ; .

10.2. Дифференциалы высших порядков также определяются индуктивно: дифференциалом второго порядка (или вторым дифференциалом) функции называется дифференциал от её первого дифференциала; дифференциалом третьего порядка называется дифференциал от второго дифференциала; и вообще, дифференциалом n-го порядка функции называется дифференциал от её n-1-го дифференциала. При вычислении высших дифференциалов необходимо учитывать, что дифференциал независимой переменной - произвольная и независимая от х величина, которая при дифференцировании рассматривается как постоянная. Поэтому ; ; …., .

10.3. Неинвариантность формы старших дифференциалов относительно замены переменной. Независимо от того, является ли х независимой переменной, или сама эта переменная х является функцией другой переменной t, формула для нахождения дифференциала первого порядка одна и та же: dy = y'dx. Покажем, что уже второй дифференциал этим свойством не обладает. Если х - независимая переменная, то d 2y = y"dx2. Если x = ϕ(t), то d 2y = d(dу) = d(y'хdx) =

= d(y'х)dx + y'хd(dx). Для первого слагаемого вследствие инвариантности формы первого дифференциала d(y'х) = y"ххdx, для второго d(dx) = d 2x, поэтому окончательно d 2y = y"ххdx2+ y'хd 2x, что отличается от случая независимой переменной. Причина этого понятна: если х независимая переменная, то при нахождении второго дифференциала dx рассматривается как независимая от x константа; в случае x = ϕ(t) дифференциал dx определяется дифференциалом dt.

10.4. Старшие производные функции, заданной параметрически. Для первой производной функции

 была получена формула . Если применить эту формулу к функции

 то получим: ; аналогично, применяя ту же формулу ко второй производной , получим выражение для третьей производной, и т.д. Так, для функции  мы получили . Найдем вторую производную: .

10.5. Старшие производные функции, заданной неявно, находятся последовательно, в соответствии с определением старших производных. Так, для неявно заданной зависимости у от х  мы получили . Найдём вторую производную: . Дальше можно найти третью и т.д. производные.

# Теорема Коши о среднем.

Теорема: Если ф-ции f(x) и g(x) непрерывны на отрезке [a,b], дифференцируемы на интервале (a,b), причем на (a,b), то Ǝ такая что (1)

Д-во: Рассмотрим ф-цию .

Ф-ция F(x) удовл. dctv условиям т.Ролля:

1. F(x) – непрерывна на [a, b], по арифметическим свойствам непрерывной ф-ции.
2. F(x) – диф-ма на (a, b).
3. F(b)=0=F(a), вместо x подставляем b, a. ⇒ по т.Ролля⇒Ǝ : . разделив рав-во на получим:

, ч.т.д.

# Теорема Лагранжа о среднем.

Теорема: Если функция y=f(x) непрерывна на отрезке [a,b], дифференцируема на (a, b), то найдется такая точка ,что: (1).

Док-во: Рассм. вспомогательную ф-цию . Ф-ция F(x) удовл. условиям т.Ролля:

1. F(x) – непрерывна на [a, b], по арифметическим свойствам непрерывной ф-ции.
2. F(x) – диф-ма на (a, b).
3. F(b)=0=F(a), вместо x подставляем b, a. ⇒ по т.Ролля⇒Ǝ : , ч.т.д.

# Формула Тейлора для функции с остаточным членом в форме Лагранжа.

Формула Тейлора:

(Rn(x) - остаточный член формулы Тейлора).

Остаточный член формулы Тейлора в форме Лагранжа:

,

# Локальный экстремум функции одной переменной. Необходимые условия экстремума функции.

Опр: Точка называется точкой локального минимума(максимума) функции f(x), если Ǝ , .

Если нер-во нестрогое только при , то называется точкой широкого локального минимума (максимума) ф-ции . Все такие точки наз-ся точками локального экстремума.

Теорема(необходимое условие локального экстремума): Если ф-ция диф-ма на (a,b) и точка – точка локального экстремума, то в этой точке производная функции или не существует.

Док-во(следует из теоремы Ферма).

# Достаточные условия экстремума функции одной переменной.

Теорема 1(первое достаточное условие экстремума). Пусть точка c – является точкой возможного экстремума ф-ции , диф-мой в некоторой окрестности точки c. Если в левой полуокресности т. c, то т. c – явл-ся точкой локального минимума(максимума) ф-ции .

Теорема 2(Второе достаточное условие экстремума). Пусть функция f(x) дважды дифференцируема в точке – c , причем

= 0 , а Тогда точка c – есть точка локального минимума (локального максимума) функции f(x) .

# Исследование функции. Построение графика функции.

Полное исследование ф-ции можно проверить по следующей форме:

1. Находим область определения и область значения .
2. Исследование на четность, нечетность, периодичность.
3. Точки пересечения с осями координат. Нахождение промежутков знакопостоянства.
4. Нахождение асимптот графика.
5. Использование , нахождение промежутка монотонности и точек экстремума.
6. Использование , нахождение направления выпуклости, точек перегиба.
7. Построение графика.

# Первообразная и неопределенный интеграл.

Опр: Функция наз-ся первообразной для ф-ции на промежутке X, если .

Лемма1: Если на X, то .

Лемма2: Если – первообразная для на X, то любая другая первообразная для имеет вид , где .

Опр: Если – первообразная для ф-ции на X, то мн-во {f(x)+c}, c всех первообразных ф-ции f(x) на X наз-ся неопределенным интегралом от ф-ции f(x) и обозначается .

– подынтегральная ф-ция.

– подынтегральное выражение.

# Таблица основных интегралов.

1)

2) ,

3)

4)

5)

6)

7)

8)

9)

10)

11)

12)

13)

14)

15)

16)

17)

18)

# Простейшие правила интегрирования. Интегрирование путем замены переменной.

1. Вынесение функции из-под знака дифференциала.

2. Внесение функции под знак дифференциала.

, где , т.е. является первообразной .

3. Множитель-константу можно выносить за знак дифференциала и вносить под него (частный случай первого и второго правил).

.

4. Под знаком дифференциала можно прибавлять или отнимать любую константу (частный случай второго правила).

Метод замены переменной:

Теорема: если ф-ция определена на промежутке T и принимает значение на X, ф-я f(x) имеет первообразную на X, определена на T, то , . Применение формулы \* на практике слева-направо наз-ся методом внесения под знак диф-ла, справа-налево – метод подстановки.

Док-во: пусть – первообразная для => F(x)-f(x) => => в левой части (\*), а в правой .

Рассмотрим прием внесения под знак диф-ла. Используются след. формулы: , ,

# Интегрирование по частям.

Теорема: Пусть ф-ции u(x), v(x) –диф-мы на интервале X и сущ-ет первообразная для ф-ции Тогда сущ-ет первообразная для на X, причем (1).

Замечание: Формула (1) применяется в случаях, когда интеграл в правой части вычисляется значительно легче чем в левой части.

Замечание: Выражение наз-ся вне интегральным членом.

Т.к. , =>то (1) записывается след. образом: .

Док-во: Воспользуемся след. формулой: . Проинтегрируем обе части: .

# Понятие определенного интеграла, его геометрический смысл.

Пусть ф-я y=f(x) определена на отрезке [a,b]. Рассмотрим разбиение {xi}, i=0,n отрезка [a,b] точками xi, i=0,n: a=x0<x1<...<xi<xi+1<xn-1<xn=b. На каждом элементарном отрезке [xi,xi+1], i=0,n-1, выберем произвольную точку ρi, i=0,n-1. Обозначим . Произведение – площадь прямоугольника с основанием [xi,xi+1] высоты . Рассмотрим конкретную сумму – сумма площадей заштрихованных прямоугольников, которые наз-ся интегральной суммой для ф-ции f(x), отвечающей разбиению {xi} и набору {}. Обозначим, .

Опр: если сущ-ет конечный предел при n→∞, след-ти интегральных сумм , не зависящей от выбора разбиения {xi} набора {}, то его наз-ют определенным интегралом от ф-ции f(x) по отрезку [a,b] и обозначают: .

Геометрический смысл определенного интеграла: при график ступенчатой ф-ции стремится к графику ф-ции f(x) и в пределе определяет площадь криволинейной трапеции ABCD, площадь S криволинейной трапеции (фигуры ограниченной графиком непрерывной положительной на интервале [a;b] функции y=f(x), осью Ox и прямыми x=a и x=b) вычисляется по формуле: .

# Свойства определенных интегралов.

1. Линейность. Если функции y = f(x), y = g(x) интегрируемы по отрезку [a,b] , то по этому отрезку интегрируема их линейная комбинация a f(x) + b g(x) (a, b = const), и .
2. Аддитивность. Если y = f(x) интегрируема по отрезку [a,b] и точка c принадлежит этому отрезку, то .
3. Интеграл от единичной функции ( f(x) = 1). Если f(x) = 1, то .
4. Теорема об интегрировании неравенств. Если в любой точке  выполняется неравенство f(x)≤g(x) , и функции f(x), g(x) интегрируемы по отрезку [a,b], то  .
5. Теоремы об оценке интеграла. 1)если на отрезке [a,b] функция удовлетворяет неравенству m≤f(x)≤M, то . 2)если функция f(x) интегрируема по отрезку [a,b], то .
6. Теорема о среднем. Если f(x) непрерывна на отрезке [a,b], то существует точка , такая что .

# Интеграл как функция верхнего предела. Формула Ньютона−Лейбница.

Рассмотрим функцию f(x), связанную с функцией равенством: . Эту функцию называют: интеграл как функция верхнего предела.

Свойства этой функции:

Теорема 1: функция .док-во: пусть - произвольная точка. Докажем, что . Рассмотрим разность:. Так как , то f(x) ограничена на [a, b], т.е. . Следовательно, по свойству определённого интеграла И , ч.т.д.

Теорема 2: пусть и . Тогда и . Док-во: рассмотрим след-щее выражение: . Так как и по условию , то при Следовательно, при имеют место соотношения т.е. ч.т.д.

Формула Ньютона – Лейбница. Теорема: пусть . Тогда и, если f1(x) – любая первообразная для f(x) на [a, b], то справедливо равенство: (эта формула носит название формулы ньютона - лейбница). Док-во: рассмотрим функцию так как, по условию теоремы , то в силу теоремы 2 . Следовательно, f(x) – есть первообразная для f(x) на [a, b]. Заметим, что f(a) = 0 и . Пусть f1(x) – любая из первообразных для f(x). Как было доказано выше f1(x) = f(x) + c, где  - некоторая константа. Запишем разность .

# Суммы Дарбу, условия существования интеграла.

Пусть на отрезке [a,b]  задана ограниченная функция   . Введем разбиение: R: a=x0< x1<...< xn=b. ПустьНаряду с интегральными суммами рассмотрим суммы: , , которые называют нижней и верхней суммами Дарбу. Очевидно, что . Суммы дарбу не обязательно являются интегральными суммами. Однако, если f(x)  - непрерывная функция, то  и  являются, соответственно, наименьшей и наибольшей из интегральных сумм, отвечающих данному разбиению, так как по теореме вейерштрасса f(x)   достигает минимума и максимума в каждом [xi, xi+1] и поэтому можно выбрать точки  так, что и . Так как   и , то.            (1). При фиксированном разбиении   - постоянные числа, а интегральная сумма  остается переменной в силу произвольности чисел . Легко видеть, что за счет выбора точек   сумму  можно сделать как угодно близкой к , т. е. при данном разбиении  являются точной нижней и точной верхней гранями для интегральных сумм: . Пусть R1, R2, R3  - разбиения [a,b]. Если же точки R1 принадлежат R2, то будем писать R1R2 и говорить, что R2 есть продолжение R1. Если множество точек, из которых состоит R3, есть теоретико-множественная сумма множеств точек, из которых состоят R1 и R2, то будем писать R3= R1+ R2. Св-ва сумм дарбу: 1) если к имеющимся у разбиения R точкам деления добавить новые точки, то верхняя сумма дарбу  не возрастает, а нижняя  не убывает: , Таким образом, . 2) каждая нижняя сумма дарбу не больше каждой верхней суммы дарбу, хотя бы отвечающей другому разбиению промежутка: . Т е о р е м а   1 (существование интеграла). Для того, чтобы определенный интеграл ограниченной функции f(x) существовал, необходимо и достаточно, чтобы ,          (3), где число  называется колебанием функции f(x) на .

# Интегрируемость непрерывных и монотонных функций.

Теорема1.  Если функция f(x) непрерывна на [a,b] , то она интегрируема на [a,b].

Док-во. Так как функция f(x)  непрерывна на [a,b], то она равномерно непрерывна на [a,b] и, следовательно,   такое, что как только [a,b] разбит на части с  , то все колебания  . Отсюда . В силу произвольности заключаем, что  , и f(x) интегрируема.

Теорема2. Монотонная на отрезке функция интегрируема на этом отрезке.

Док-во. Будем считать, что f(a)<f(b), иначе функция постоянна и теорема тривиальна. Так как  то наша функция ограничена на [a,b]. Введем разбиение R отрезка [a,b] с . Так как в данном случае  , то Выберем теперь ; тогда , и f(x) интегрируема, ч.т.д.

Замечание1. Отметим, что монотонная функция может иметь счетное множество точек разрыва.

Замечание2. Если f(x)  интегрируема на [a,b] (a<b) , то  также интегрируем.

# Несобственные интегралы.

Определённый интеграл называется несобственным, если выполняется, по крайней мере, одно из следующих условий.

* Область интегрирования является бесконечной.
* Функция f(x) является неограниченной в окрестности некоторых точек области интегрирования.

Если интервал [a,b] конечный, и функция интегрируема по Риману, то значение несобственного интеграла совпадает с значением определённого интеграла.

Несобственные интеграла 1-го рода: пусть f(x) определена и непрерывна на множестве от [a,+∞] и . Тогда:

1. Если , то используется обозначение  и интеграл называется несобственным интегралом римана первого рода. В этом случае  называется сходящимся.
2. Если не существует конечного   , то интеграл    называется расходящимся к  , или просто расходящимся.

Пусть f(x)  определена и непрерывна на множестве от (-∞, b]  и  . Тогда:

1. Если , то используется обозначение   и интеграл называется несобственным интегралом Римана первого рода. В этом случае   называется сходящимся.
2. Если не существует конечного , то интеграл      называется расходящимся к , или просто расходящимся.

Если функция f(x) определена и непрерывна на всей числовой прямой, то может существовать несобственный интеграл данной функции с двумя бесконечными пределами интегрирования, определяющийся формулой: , где с — произвольное число. Геометрический смысл несобственного интеграла 1-го рода: несобственный интеграл первого рода выражает площадь бесконечно длинной криволинейной трапеции.

Несобственный интеграл 2-го рода: пусть f(x)  определена на (a,b] , терпит бесконечный разрыв в точке x=a и  . Тогда:

1. Если   , то используется обозначение   и интеграл называется несобственным интегралом Римана второго рода. В этом случае интеграл называется сходящимся.
2. Если  , то обозначение сохраняется, а      называется расходящимся к  , или просто расходящимся.

Пусть f(x)    определена на [a,b), терпит бесконечный разрыв при x=b . Тогда:

1. Если  , то используется обозначение   и интеграл называется несобственным интегралом Римана второго рода. В этом случае интеграл называется сходящимся.
2. Если  , то обозначение сохраняется, а   называется расходящимся к  , или просто расходящимся.

Если функция f(x) терпит разрыв во внутренней точке c отрезка [a,b] , то несобственный интеграл второго рода определяется формулой: . Геометрический смысл несобственных интегралов 2-го рода: несобственный интеграл второго рода выражает площадь бесконечно высокой криволинейной трапеции.

# Несобственные интегралы от неотрицательных функций. Признаки сходимости.

Установим признаки сходимости для несобственных интегралов от неотрицательных функций.

Лемма1. Если функция f неотрицательна на полуинтервале [a,b), то для сходимости интеграла необходимо и достаточно, чтобы множество всех интегралов , ,было ограничено сверху, т. е. чтобы существовала такая постоянная c > 0, что для всех  выполнялось бы неравенство (1). Положим (2). Если , то ибо в силу не отрицательности функции f имеет место неравенство > 0, т. Е. Функция ) возрастает на полуинтервале [a,b). Существование несобственного интеграла означает существование конечного предела что имеет место когда функция ограничена сверху, а это в силу (2) равносильно условию (1).

Теорема1 (признак сравнения). Пусть Тогда:

1) если интеграл  сходится, то сходится и интеграл .

2) если интеграл расходится, то расходится и интеграл . 

Следствие 1. Пусть функции f и g неотрицательны на промежутке [a,b), g(x)не равно0 при всех x  [a,b) и существует конечный или бесконечный предел Тогда:

1) если интеграл  сходится и 0 < k < +∞, то и интеграл  сходится;

2) если интеграл  расходится и 0 < k < +∞, то и интеграл расходится.   
    Следствие 2. Если функции f(x) и g(x) эквивалентны при x→b, т. e. f(x) = (x)g(x), a < x < b, то интегралы и одновременно сходятся или расходятся.

# Частная производная функции многих переменных.

Определение. Частной производной по x от функции z=f(x,y)  называется предел отношения частного приращения   по x к приращению   при стремлении   к нулю. Частная производная по x от функции z=f(x,y)   обозначается одним из символов Таким образом, по определению, Аналогично частная производная по у от функции z=f(x,y) определяется как предел отношения частного приращения функции   по у к приращению  при стремлении  к нулю. Частная производная по у обозначается одним из символов Таким образом, Заметив, что   вычисляется при неизменном у, а  при неизменном x мы можем определения частных производных сформулировать так: частной производной по x от функции z=f(x,y)  называется производная по x, вычисленная в предположении, что у — постоянная. Частной производной по у от функции z=f(x,y)   называется производная по у, вычисленная в предположении, что x - постоянная. Из этого определения ясно, что правила вычисления частных производных совпадают с правилами, указанными для функций одной переменной, и только требуется каждый раз помнить, по какой переменной ищется производная.

# Производная по направлению функции многих переменных.

Рассмотрим [функцию двух переменных](http://ios.sseu.ru/public/eresmat/gloss/g144.htm)*n=2; z=f(x, y).* Под направлением мы будем понимать любой вектор  на плоскости.

***Определение 1.***Направляющими косинусами данного направления  называются косинусы углов, которые данное направление образуют с положительными направлениями осей координат. Направляющие косинусы данного направления - . Направляющие косинусы любого направления в любом пространстве обладают следующим свойством: сумма квадратов направляющих косинусов равна единице На плоскости имеем Если рассмотреть вектор , координатами которого являются направляющие косинусы данного направления, то этот вектор сонаправлен с вектором  и имеет единичную длину. Пусть даны точка   и направление . Переместим точку *М*0 вдоль направления  на величину D*l* в точку *М*1. Тогда функция и аргумент получат соответствующие приращения. Функция получит приращение, которое называется приращением функции в данном направлении:  Из треугольника М0М1А: . Из треугольника  М0М1В:.

***Определение 2.***Предел отношения приращения функции в данном направлении к приращению направления, когда приращение направления стремится к нулю, называется производной функции в данном направлении (если этот предел существует и конечен); Если направление  совпадает с направлением оси *ОХ*, то производная по направлению совпадает с [частной производной](http://ios.sseu.ru/public/eresmat/gloss/g147.htm) по переменной *х*. Аналогично производная по направлению оси *ОУ* совпадает с частной производной по переменной *у*.

***Теорема.*** Производная по направлению равна сумме попарных произведений частных производных в данной точке на направляющие косинусы данного направления.

# Градиент функции многих переменных, его свойства.

**Производная по направлению** любого **единичного вектора** выражается формулой:       (1),

(α, β, γ - углы, которые вектор  составляет с осями *x*, *y*, *z*).

Определение 1 : введем вектор  ,     (2) называемый **градиентом функции**  *f* в точке (*x*, *y*, *z*). Формула (1) говорит, что производная от *f*в точке (*x*, *y*, *z*) по направлению единичного вектора   равна проекции градиента в этой точке на направление        (3). Имеет место очевидное неравенство       (4) для любого вектора .  Если grad *f*  = 0, что обычно бывает только в исключительных точках, то   для любого вектора .  Если же grad *f*  ≠0 (одна из частных производных от*f* не равна нулю), то (4) есть строгое неравенство для всех единичных векторов , за исключением единичного вектора , направленного в сторону grad *f, ( )*. Таким образом, **, ,** . Из сказанного следует, что градиент функции *f* в точке (*x*, *y*, *z*) можно определить как вектор, обладающий следующими двумя свойствами:

1) длина его равна максимальной величине производной по направлению   в (*x*, *y*, *z*) (для дифференцируемой в (*x*, *y*, *z*) функции этот максимум существует и есть число неотрицательное);

2) если его длина не равна нулю, то он направлен в ту же сторону, что и вектор, вдоль которого производная   максимальна.

# Полный дифференциал функции многих переменных.

Опр: Главная линейная по ,… часть приращения ,…, – f(,…, )=A+…+A+ o() ф-ции f(x) в т.M(,…, ) отвечающего приращениям ,… аргументам ,…, наз диф-лом ф-ии U=f(x) в т.M и обознач: .

Если U,V- ф.м.п., то 1) , c=const;

2) ;

3) d(UV)=UdV+VdU;

Д-во: +

4) d() =

Инвариантность формы дифференциала:

Если U= f(,…, ), то для случая независимых перем имеет место:

Для случая зависимых:

() +…+()=

# Дифференциалы высших порядков.

Диф-лом 2-ого порядка ф-ии f(x,y) в т.M(x,y) наз выраж вида:

n-ого порядка наз выраж

= =

Неинвариантность: x=x(t),y=y(t)

; =

;

+2

# Экстремумы функции многих переменных. Необходимые условия.

Ф-я U= f(,…, ) определена в обл D, а т. - внутр т обл D

Опр: Т. наз точкой локального минимума(максимума) ф-ии U=f(x), если окрестность такая, что М f(M) ≥ f() (f(M) ≤ f())

Если равенство выполняется лишь при М= , то - точка строгого локального минимума(максимума)

Точка локального минимума(макс) наз т. локального экстремума

Теорема: Необходимое усл. локального экстремума

Если т. явл т. локального экстремума ф-ии f(,…, ) и частные производные i=1..m, то выполняется след равенство:

, i=1..m (1) – необходимое усл экстремума

Д-во: Пусть – т локального экстремума min, тогда М f(M) ≥ f(). Зафиксируем все координаты т кроме i-й, т.е. получим f() = g() – ф-я одной переменной причем (): g() ≥ g()

f (M) ≥ f ()

следовательно - точка лок. min ф-ии g(), производная которой() равна частной производной

по Т Ферма 0 = () усл(1) получено

Необходимое условие не является достаточным условием существования экстремума.

# Достаточные условия экстремума функции многих переменных.

Теорема: Пусть т - стационарная точка ф-ии U=f(,…, ) определенной в нек окр-ти т., причем в этой окрестности частные производные до второго порядка ф-ии f(x) непрерывны. Тогда если () ∀ , i=1..m , то т.-точка лок min(max) ф-ии f(x). А если может принимать значения различных знаков, то - не явл т экстремума.

Д-во: U=f(x,y), , dx, y=dy

= d+ 2dxdy+ d=(A) ; A=

По критерию Сильвестра получим:

A>0,

A<0,

Если для стационарной точки выполнимо нер-во и , то в т ф-я f(x,y) имеет строгий экстремум, причем при – min, – max,если <0 то экстремума нет.

По ф-ле Тейлора приращение ф-ии = = , где N()

Т.к. частные производные 2ого порядка ф-ии f(x,y) непрерывны в т то

*==*

=

Где - бесконечно малые

+ 2xy+ + 2xy+ + 2 x y+

x=

y=

(+ 2+ ) + (+ 2+ ))

+ 2+

1. > 0,

*> 0* знак совпадает со знаком

Если , то ф-я – непрер. ф-я по >0 по т Вейерштрасса

При достаточно малых <0 выполняется нер-во: <m знак совпадает со знаком со знаком т. min при > 0 max при < 0

1. < 0

a) , причем . Тогда выберем направление так, чтобы > > выражение при и при при , сохраняет знак нет экстремума

b)

При ,

При – решение ур = 0 ctg

, при малых

Экстремума нет.

# Понятие условного экстремума.

Общая постановка задачи на усл экстремум

U= f(,…, ;,…, ) → extr (1)

(2)

f(x,y) – целевая ф-я, (2) - условие связи

Опр: Говорят, что ф-я U=f(x,y) имеет в т удовлетворяющий усл (2) условный min(max), если ∀ М(,…, ;,…, ) из некоторой окрестности т удовл (2) выполнено нер-во: f(M) ≥ f() (f(M) ≤ f())

Предположим ф-ии непрерывно диф-мы в окрестности и

(3)

Тогда по Т о сист неявных ф-ий существует окрестность т.( в которой однозначно определены k ф-ий: (4), являющиеся решениями сист (2)

подставив ф-ции (4) в (1)

(1,2) – задача на усл экстремум

(4)(1) – задача на безусловный экстремум

U= f(,…, ; ) extr

U() = extr dU() = d+…+d= 0

d+…+d+d+…+d= 0 (5)

Дифференцируя ур-е связи получим:

*(6)* – линейная сист разрешимая однозначно относит неизвестных …, т.к. её определитель

Все – линейные комбинации величин

После подстановки их в (6) получаем рав-во вида:

, т.к. диф-лы могут выбираться независимо, то =0 i=1..m

Необх усл экстремума

# Метод множителей Лагранжа.

U= f(,…, ;,…, ) → extr (1)

(2)

Рассматривается ф-я Лагранжа L(x,y,) = f(x,y) + , где - некоторые неизвестные пока коэффициенты(множители Лагранжа)

Выберем их так что (k условий)

получим систему:

– линейная система из k уравнений с k неизвестными

её определитель это определитель Якобиан: - находятся однозначно dL = d + … +d+ d+…+ d+ d+…+d= 0

( = 0… = 0)

превращается в тождество: d+ … +d т.е. ∀ d… d получаем ещё m условий : =…= =0

Условия для условного экстремума:

Достаточное условие экстремума – знакоопределенность 2ого диф-ла

*>0 – min (<0 – max)*

# Числовые ряды, основные понятия, сходимость ряда.

Рассмотрим числовую последовательность а1, а2, …, аn,… и образуем из нее выражение вида , которая называется числовым рядом, а слагаемые называется n-ым членом ряда.

Конечная сумма называется n-ой частичной суммой ряда

Если существует конечный предел последовательности частичных сумм {} то его называют суммой ряда , записывают, что и говорят что этот ряд сходится к S.

Простейшие свойства рядов

Лемма 1. Если числовой ряд сходится к S, то ряд A+ сходится к A+S

Лемма 2. Если ряд сходится к S то ряд сходится к .

Лемма 3. Если =А, , то ряд .

Свойства сходящихся рядов.

Лемма 1. Отбрасывание конечного числа первых слагаемых ряда не влияет на его сходимость.

Док-во. Пусть ряд сходится к S, отбросим k первых членов ряда и получим бесконечную сумму Рассмотрим его последовательность частичных сумм:

Бесконечная сумма называется n-ым остатком ряда.

Лемма 2. Если ряд сходится, то

Док-во Пусть ряд сходится =>

Лемма 3.(Необходимый признак сходимости числового ряда)

Если ряд сходится то

Док-во Пусть ряд сходится: =>

Лемма 4. Критерий коши

Ряд сходится ⬄

Док-во Применяя критерий коши к последовательности частичных сумм {} и учитывая, что |

# Абсолютная и условная сходимость рядов.

Если для ряда (1) сходится ряд (2) то ряд (1) называется абсолютно сходящимся.

Теорема

Из абсолютной сходимости ряда (1) следует сходимость ряда (1)

Док-во

Пусть (1) сходится абсолютно => сходится знакоположительный ряд (2) ⬄ |Критерий Коши| ⬄

Обратное в общем случае неверно.

Числовой ряд (1), который сходится , но не сходится абсолютно называется не абсолютно сходящимся рядом (условно сходящийся).

# Ряды с неотрицательными членами. Признаки сходимости положительных рядов.

**Опр:** nN Тогда ряд наз знакоположительным

**Теорема:** Знакоположит ряд сх-ся

**Теорема(признак сравнения)**

Ряды и знакоположит и n 0. Тогда

1. из сх-ти ряда сх-ть ряда
2. из расх расх

**Д-во:** т.к. отбрасывание конечного числа слагаемых ряда не влияет на его сх-ть, то будем считать, что . Т.к. , то

1) из сх-ти – сх-ся

**Теорема(предельный признак сравнения)**

Пусть для знакоположит рядов и = L Тогда ряды и сх-ся или расх одновременно.

**Д-во:** = L

- ;

L –< < L+

(L –)< < (L+)

Т.к. начиная с номера N выполнено нер-во < (L+), то из сх-ти ряда , а значит и из ряда следует сх-ть ряда

Аналогично из нер-ва (L –)< и сх-ти ряда сх-ть ряда сх-ть

# Степенные ряды. Промежуток сходимости степенного ряда.

Опр: Функ. Ряд вида наз степенным рядом.

Если a=0, получим ряд вида

Если ряд сх-ся хотя бы в одной точке и расх-ся хотя бы в одной т , то такое число R, что ряд сх-ся абсолютно в (-R,R) и расх-ся в (-,-R)(R,+). В точках требуется дополнит исследование. Число R наз радиусом сх-сти степенного ряда

Замечание: если ряд сх-ся только при x=0, то R=0 если сх-ся x(-,+), то R=

1. Признак Даламбера

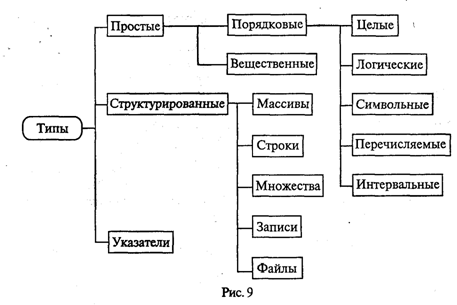
Ряд –знакоположит ряд xR для него по признаку Даламбера сх-сть выполняется при условии = <, т.о. R =

1. Признак Коше

*,* т.о. R = =

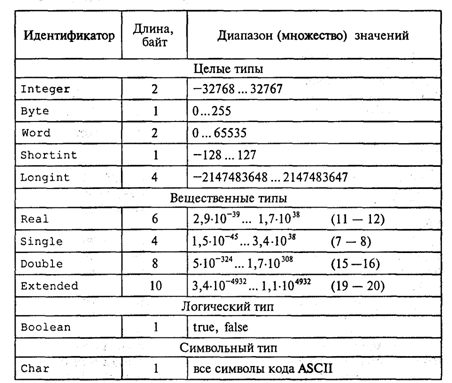
# Понятие типа данных в различных языках программирования. Основные типы (на примере языка Паскаль или др.).

Концепция типов данных является одной из центральных в любом языке программирования. С типом величины связаны три ее свойства: форма внутреннего представления, множество принимаемых значений и множество допустимых операций. Турбо Паскаль характеризуется большим разнообразием типов данных.



В стандартном Паскале отсутствует строковый тип. Кроме того, в Турбо Паскале целые и вещественные — это группы типов. В старших версиях Турбо Паскаля существует процедурный тип и тип объект. Каждый тип имеет свой идентификатор.

В таблице ниже представлена информация о простых типах данных, определенных в Турбо Паскале. Для вещественных типов в скобках указано количество сохраняемых значащих цифр мантиссы в десятичном представлении числа.



В стандарте Паскаля из вещественных типов определен только тип Real; из целых типов — Integer.

 Особенными для этих типов данных являются операции div и mod.Операция div – это операция целочисленного деления, то есть результатом будет целая часть от деления целого числа на целое. Операция mod – это операция остатка отделения, т.е она вернет делимое минус делитель умноженное на целую часть.

Типы Single, Double, Extended употребляются в Паскаль-программах только в том случае, если ПК снабжен сопроцессором «плавающей арифметики» (для процессоров IBM PC, начиная с Intel-80486 и старше, это условие всегда выполняется).

Тип данных называется порядковым, если он состоит из счетного количества значений, которые можно пронумеровать. Отсюда следует, что на этом множестве значений существуют понятия «следующий» и «предыдущий».

Символьные типы данных

Символьный тип данных это тип char. Очевидно, что переменная типа char можно присвоить значение в виде символьной константы, например c:=’A’.Но символьный тип данных может иметь значения, задающиеся кодам в таблице ASCII-кодов, то есть, то, что приведено в первом примере может быть c:=#65. 65 – ASCII код символа А. Таблицу символов ASCII можно найти в Интернете или в учебниках по программированию (рекламой не занимаюсь). Теперь понятно, почему символьный тип данных – это порядковый тип данных, потому что мы можем любой символ заменить числом в диапазоне 0..255, а все числа идут по определенному порядку. Функции и процедуры, применяемые  к порядковым типам данных, будут рассмотрены ниже.

Функции и процедуры, применяемые к целым, символьным и логическим типам данных

Сначала рассмотрим функции:

ord(x) - возвращает порядковый номер значения переменной x (относительно того типа, к которому принадлежит переменная х).

pred(x) - возвращает значение, предшествующее х (к первому элементу типа неприменима).

succ(x) - возвращает значение, следующее за х (к последнему элементу типа неприменима).

 Теперь процедуры:

inc(x) – оператор инкремента, возвращает значение x:=x+1

inc(x,k) - возвращает значение x:=x+k.

dec(x) – оператор декремента, возвращает значение x:=x-1.

dec(x,k) - возвращает значение x:=x-k.

Преобразование типов данных

Преобразование типов данных бывает явным и неявным.

Неявное преобразование типов данных – это когда не используются функции для преобразований типов. Например, если мы переменной типа real присваиваем значения частного двух переменных типа integer, таким образом, понятно, что типinteger неявно преобразовался в тип real.

Явное преобразование типов данных – это когда преобразование выполняется с помощью специальных функций.

Функции округления:

trunc                     real -> integer

round                   real -> integer

Функция преобразования строки в число:

val                                         string -> byte/integer/real

Получение символа по заданному ASCII-коду:

сhr                                         byte -> char

Преобразование порядковых типов:

ord                                        <порядковый\_тип> -> longint

Также можно преобразовать выражение из одного типа в другой, просто указав новый тип данных. Например, a:=byte(b).

# Конструирование новых типов данных.

Имеется 4 стандартных простых типа данных:

целый;

вещественный;

логический;

литерный

и стандартные правила определения новых типов. На основании этих стандартных типов и нестандартного абстрактного перечисляемого типа (в этом типе значения задаются именами) программистом строится огромное многообразие новых типов данных.

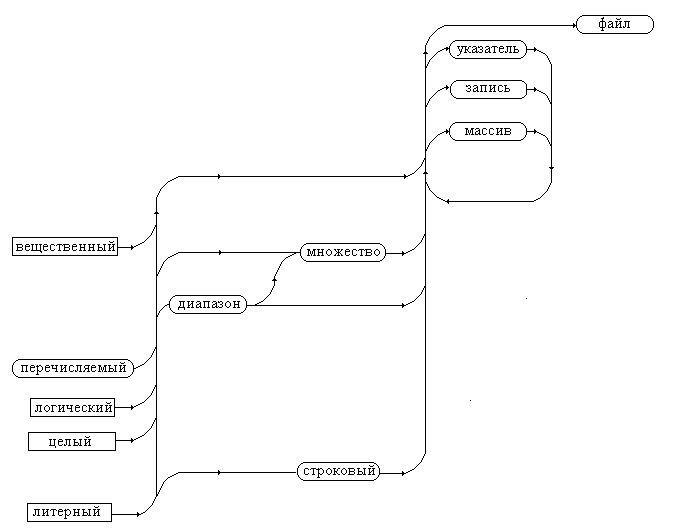


Рис. 2.2. – Дерево типов

Принцип конструирования новых типов из набора стандартных показан на рис.2.2. - Дерево типов языка программирования Паскаль. Дерево построено по принципу "вырастания" одних типов из других. Если взять любой из типов, не являющийся корневым, то легко определить, какие типы можно использовать при его построении (это все типы, расположенные левее и ниже и от которых идут стрелки к данному типу). Типы, из которых строятся новые типы, называются базовыми для этих новых типов.

# Структурные типы данных в языках программирования. Особенности их представления в памяти ЭВМ.

Любой из структурированных типов характеризуется множественностью образующих этот тип элементов, т.е. переменная или сонстанта структурированного типа всегда имеет несколько компонентов. Каждый компонент, в свою очередь, может принадлежать структурированному типу, что позволяет говорить о возможной вложености типов. В Турбо Паскале допускается произвольная глубина вложености типов, однако суммарная длина любого из них во внутреннем представлении не должна превышать 65520 байт.

МАССИВЫ

Массивы в Турбо Паскале во многом схожи с аналогичными типами данных в других языках програмирования. Отличительная особенность массивов заключается в том, что все их компоненты суть данные одного типа (возможно структурированного). Эти компоненты можно легко упорядочить и обеспечить доступ к любому из них простым указанием порядкового номера.

Описание массива задаётся следующим образом:

<имя типа> = array [<сп.инд.типов>] of <тип>

Здесь <имя типа> - правильный индификатор;

Array, of – зарезирвированые слова (массив, из);

<сп.инд.типов> - список из одного или нескольких индексных типов, разделённых запятыми; квадратные скобки, обрамляющие список, - требование синтаксиса;

<тип> - любой тип Турбо Паскаля.

В качестве индексных типов в Турбо Паскале можно использовать любые порядковые типы, кроме LongInt и типов-диапазонов с базовым типом LongInt.

Глубина вложенности структурированных типов вообще, а следовательно, и массивов – произвольная, поэтому количество элементов в списке индексов типов (размерность массива) не ограничено, однако суммарная длина внутреннего представления любого массива не может быть больше 65520 байт.

ЗАПИСИ

Запись – это структура данных, состоящая из фиксированного числа компонентов, называемыхполями записи. В отличие от массива, компоненты (поля) записи могут быть различного типа. Чтобы можно было ссылаться на тот или иной компонент записи, поля именуются.

Структура объявления типа записи такова:

<имя типа> = RECORD <спюполей> END

Здесь <имя типа> - правильный индификатор;

RECORD, END – зарезервированые слова (запись, конец);

<сп.полей> - список полей; представляет собой последовательность разделов записи, между которыми ставится точка с запятой.

МНОЖЕСТВА

Множества – это набор однотипных логическх связанных друг с другом объектов. Характер связей между объектами лишь подразумевается програмистом и никак не контролируется Турбо Паскалем.количество элементов, входящих в множество, может менятся в пределах от 0до 256 (множество, не содержащее элементов, называется пустым).именно непостоянством количества своих элементов множества отличаются от массивов и записей.

Два множества считаются эквивалентными тогда и только тогда, когда все их элементы одинаковы, причём порядок следования элементов множества безразличен. Если все элементы одного множества входят также и в другое, говорят о включении первого множества во второе.

Описание типа множества имеет вид:

<имя типа> = SET OF <баз.тип>

Здесь <имя типа> - правильный индификатор;

SET, OF – зарезирвированные слова (множество, из);

<баз.тип> - базовый тип элементов множества, в качестве которого может использоваться любой порядковый тип, кроме WORD, INTEGER и LONGINT.

Для задания множества используется так называемый конструктор множества: список спецификаций элементов множества, отделяемых друг от друга запятыми; список обрамляется квадратными скобками. Спецификациями элементов могут быть константы или выражения базового типа, а также – тип-диапазон того же базового типа.

СТРОКИ

Тип STRING (строка) в Турбо Паскале широко используется для обработки текстов. Он во многом похож на одномерный массив символов ARRAY [0..N] OF CHAR, однако, в отличие от последнего, количество символов в строке – переменной может менятся от 0 до N, где N – максималльное количество символов в строке. Значение N определяется объявлением типа STRING[N] N и может быть любой константой порядкового типа, но ен больше 255. Турбо Паскаль разрешает не указывать N, в том случае длина строки принимается максимально возможной, а именно N=255.

Строка в Турбо Паскале трактуется как цепочка символов. К любому символу в строке можно обратиться точно так же, как к элементу одномерного массива ARRAY [0..N] OF CHAR.

ФАЙЛЫ

Под файлом понимается либо именованная область внешней памяти ПК, либо логическое устройство – потенциальный источник или приёмник информации.

Любой файл имеет три характерные особенности. Во-первых, у него есть имя, что даёт возможность программе работать одновременно с несколькими файлами. Во-вторых, он содержит компоненты одного типа. Типом компонентов может быть любой тип Турбо Паскаля, кроме файлов. Иными словами, недьзя создать «файл файлов». В-третьих, длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только ёмкостью устройств внешней памяти.

Файловый тип или переменную файлового типа можно задать одним из трёх способов:

<имя>= FILE OF <тип>;

<имя>=TEXT;

<имя> = FILE;

Здесь <имя> - имя файлового типа (правильный индификатор);

FILE, OF – зарезервированные слова (файл, из);

TEXT – имя стандартного типа текстовых файлов;

<тип> - любой тип Турбо Паскаля, кроме файлов.

В зависимости от способа объявления можно выделить три вида файлов:

· типизированные файлы (задаются предложением FILE OF…);

· текстовые файлы (определяются типом TEXT);

· нетипизированные файлы (определяются типом FILE).

# Файловые типы данных в языках программирования. Операции с файлами.

Любой файл имеет три характерные особенности. Во-первых, у него есть имя, что дает возможность программе работать одновременно с несколькими файлами. Во-вторых, он содержит компоненты одного типа. Типом компонентов может быть любой тип Паскаля, кроме файлов. Иными словами, нельзя создать «файл файлов». В-третьих, длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только емкостью устройств внешней памяти.

Файловый тип или переменную файлового типа в Паскале можно задать одним из трех способов:

Type <имя\_ф\_типа>=file of<тип\_элементов>;  
<имя\_ф\_типа>=text;  
<имя\_ф\_типа>=file;

Здесь <имя\_ф\_типа> – имя файлового типа (правильный идентификатор); File, of – зарезервированные слова (файл, из); <тип\_элементов> – любой тип Паскаля, кроме файлов.

Пример описания файлового типа в Паскале

Type  
    Product= record  
        Name: string;  
        Code: word;  
    End;  
    Text80= file of string[80];  
Var  
    F1: file of char;  
    F2: text;  
    F3: file;  
    F4: Text80;  
    F5: file of Product;

В зависимости от способа объявления можно выделить три вида файлов Паскаля:

-типизированные файлы Паскаля(задаются предложением file of..);

-текстовые файлы Паскаля(определяются типом text);

-нетипизированные файлы Паскаля(определяются типом file).

Следует помнить, что физические файлы на магнитных дисках и переменные файлового типа в программе на Паскале – объекты различные. Переменные файлового типа в Паскале могут соответствовать не только физическим файлам, но и логическим устройствам, связанным с вводом/выводом информации. Например, клавиатуре и экрану соответствуют файлы со стандартными именами Input, Output.

Как известно, каждый тип данных в Паскале, вообще говоря, определяет множество значений и множество операций над значениями этого типа. Однако над значениями файлового типа Паскаля не определены какие-либо операции, в том числе операции отношения и присваивания, так что даже такое простое действие, как присваивание значения одной файловой переменной другой файловой переменной, имеющей тот же самый тип, запрещено. Все операции могут производиться лишь с элементами (компонентами) файлов. Естественно, что множество операций над компонентами файла определяется типом компонент.

Переменные файлового типа используются в программе только в качестве параметров собственных и стандартных процедур и функций.

Основные процедуры и функции для работы с файлами

1.До начала работы с файлами в Паскале необходимо установить связь между файловой переменной и именем физического дискового файла:

Assign(<файловая\_переменная>, <имя\_дискового\_файла>)

Следует помнить, что имя дискового файла при необходимости должно содержать путь доступа к этому файлу, включая имя дисковода. При этом имя дискового файла – строковая величина, т.е. должна быть заключена в апострофы. Например:

пример доступа assign в паскале

Assign (chf, 'G:\Home\ Student\ Lang\ Pascal\ primer.dat');

Если путь не указан, то программа будет искать файл в своем рабочем каталоге и по указанным путям в autoexec.bat.

Вместо имени дискового файла можно указать имя логического устройства, каждое из которых имеет стандартное имя:

CON – консоль, т.е. клавиатура-дисплей;

PRN – принтер. Если к компьютеру подключено несколько принтеров, доступ к ним осуществляется по именам LPT1, LPT2, LPT3.

Не разрешается связывать с одним физическим файлом более одной файловой переменной.

2.После окончания работы с файлами на Паскале, они должны быть закрыты.

Close(<список файловых переменных>);

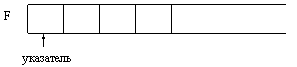
При выполнении этой процедуры закрываются соответствующие физические файлы и фиксируются сделанные изменения. Следует иметь в виду, что при выполнении процедуры close связь файловой переменной с именем дискового файла, установленная ранее процедурой assign, сохраняется, следовательно, файл можно повторно открыть без дополнительного использования процедуры assign.

Работа с файлами заключается, в основном, в записи элементов в файл и считывании их из файла. Для удобства описания этих процедур введем понятие указателя, который определяет позицию доступа, т.е. ту позицию файла, которая доступна для чтения (в режиме чтения), либо для записи (в режиме записи). Позиция файла, следующая за последней компонентой файла (или первая позиция пустого файла) помечается специальным маркером, который отличается от любых компонент файла. Благодаря этому маркеру определяется конец файла.

3.Подготовка к записи в файл Паскаля

Rewrite(<имя\_ф\_переменной>);

Процедура Rewrite(f) (где f – имя файловой переменной) устанавливает файл с именем f в начальное состояние режима записи, в результате чего указатель устанавливается на первую позицию файла. Если ранее в этот файл были записаны какие-либо элементы, то они становятся недоступными. Результат выполнения процедуры rewrite(f); выглядит следующим образом:

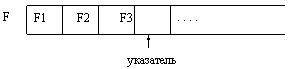


4.Запись в файл Паскаля

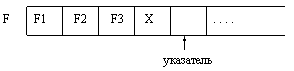
Write(<имя\_ф\_переменной>, <список записи>);

При выполнении процедуры write(f, x) в ту позицию, на которую показывает указатель, записывается очередная компонента, после чего указатель смещается на следующую позицию. Естественно, тип выражения х должен совпадать с типом компонент файла. Результат действия процедуры write(f, x) можно изобразить так:

Состояние файла f до выполнения процедуры



Состояние файла f после выполнения процедуры

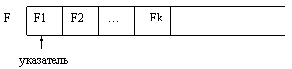


Для типизированных файлов выполняется следующее утверждение: если в списке записи перечислено несколько выражений, то они записываются в файл, начиная с первой доступной позиции, а указатель смещается на число позиций, равное числу записываемых выражений.

5.Подготовка файла к чтению Паскаля

Reset(<имя\_ф\_переменной>);

Эта процедура ищет на диске уже существующий файл и переводит его в режим чтения, устанавливая указатель на первую позицию файла. Результат выполнения этой процедуры можно изобразить следующим образом:



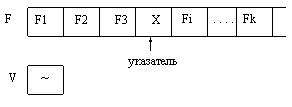
Если происходит попытка открыть для чтения не существующий еще на диске файл, то возникает ошибка ввода/вывода, и выполнение программы будет прервано.

6.Чтение из файла в Паскале

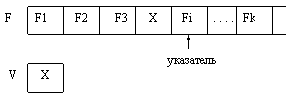
Read(<имя\_ф\_переменной>,<список переменных>);

Рассмотрим результат действия процедуры read(f, v):

Состояние файла f и переменной v до выполнения процедуры:



Состояние файла f и переменной v после выполнения процедуры:



Для типизированных файлов при выполнении процедуры read() последовательно считывается, начиная с текущей позиции указателя, число компонент файла, соответствующее числу переменных в списке, а указатель смещается на это число позиций.

В большинстве задач, в которых используются файлы, необходимо последовательно перебрать компоненты и произвести их обработку. В таком случае необходимо иметь возможность определять, указывает ли указатель на какую-то компоненту файла, или он уже вышел за пределы файла и указывает на маркер конца файла.

7.Функция определения достижения конца файла в Паскале

Eof(<имя\_ф\_переменной>);

Название этой функции является сложносокращенным словом от end of file. Значение этой функции имеет значение true, если конец файла уже достигнут, т.е. указатель стоит на позиции, следующей за последней компонентой файла. В противном случае значение функции – false.

8.Изменение имени файла в Паскале

Rename(<имя\_ф\_переменной>, <новое\_имя\_файла>);

Здесь новое\_ имя\_ файла – строковое выражение, содержащее новое имя файла, возможно с указанием пути доступа к нему.

Перед выполнением этой процедуры необходимо закрыть файл, если он ранее был открыт.

9.Уничтожение файла в Паскале

Erase(<имя\_ф\_переменной>);

Перед выполнением этой процедуры необходимо закрыть файл, если он ранее был открыт.

10.Уничтожение части файла от текущей позиции указателя до конца в Паскале

Truncate(<имя\_ф\_переменной>);

11.Файл Паскаля может быть открыт для добавления записей в конец файла

Append(<имя\_ф\_переменной>);

Типизированные файлы Паскаля.

Длина любого компонента типизированного файла строго постоянна, т.к. тип компонент определяется при описании, а, следовательно, определяется объем памяти, отводимый под каждую компоненту. Это дает возможность организовать прямой доступ к каждой компоненте (т.е. доступ по порядковому номеру).

Перед первым обращением к процедурам ввода/вывода указатель файла стоит в его начале и указывает на его первый компонент с номером 0. После каждого чтения или записи указатель сдвигается к следующему компоненту файла. Переменные и выражения в списках ввода и вывода в процедурах read() и write() должны иметь тот же тип, что и компоненты файла Паскаля. Если этих переменных или выражений в списке несколько, то указатель будет смещаться после каждой операции обмена данными на соответствующее число позиций.

Для облегчения перемещения указателя по файлу и доступа к компонентам типизированного файла существуют специальные процедуры и функции:

fileSize(<имя\_ф\_переменной>) – функция Паскаля, определяющая число компонентов в файле;

filePos(<имя\_ф\_переменной>) – функция Паскаля, значением которой является текущая позиция указателя;

seek(<имя\_ф\_переменной>,n) – процедура Паскаля, смещающая указатель на компоненту файла с номером n. Так, процедура seek(<имя\_ф\_переменной>,0) установит указатель в начало файла, а процедура seek(<имя\_ф\_переменной>, FileSize(<имя\_ф\_переменной>)) установит указатель на признак конца файла.

Текстовые файлы Паскаля.

Текстовые файлы предназначены для хранения текстовой информации. Именно в таких файлах хранятся, например, исходные тексты программ. Компоненты текстовых файлов могут иметь переменную длину, что существенно влияет на характер работы с ними. Доступ к каждой строке текстового файла Паскаля возможен лишь последовательно, начиная с первой. К текстовым файлам применимы процедуры assign, reset, rewrite, read, write и функция eof. Процедуры и функции seek, filepos, filesize к ним не применяются. При создании текстового файла в конце каждой записи (строки) ставится специальный признак EOLN(end of line – конец строки). Для определения достижения конца строки существует одноименная логическая функция EOLN(<имя\_ф\_переменной>), которая принимает значение true, если конец строки достигнут.

Форма обращения к процедурам write и read для текстовых и типизированных файлов одинакова, но их использование принципиально различается.

В списке записываемых в текстовый файл элементов могут чередоваться в произвольном порядке числовые, символьные, строковые выражения. При этом строковые и символьные элементы записываются непосредственно, а числовые из машинной формы автоматически преобразуются в строку символов.

текстовые файлы удобнее для восприятия человеком, а типизированные соответствуют машинному представлению объектов;

текстовые файлы, как правило, длиннее типизированных;

длина текстовых файлов зависит не только от количества записей, но и от величины переменных.

Так, в типизированном файле числа 6, 65 и 165 как целые будут представлены одним и тем же числом байт. А в текстовых файлах, после преобразования в строку, они будут иметь разную длину. Это вызывает проблемы при расшифровке текстовых файлов. Пусть в текстовый файл пишутся подряд целые числа (типа byte): 2, 12, 2, 128. Тогда в файле образуется запись 2122128. При попытке прочитать из такого файла переменную типа byte программа прочитает всю строку и выдаст сообщение об ошибке, связанной с переполнением диапазона.

Но, вообще-то, такой файл не понимает не только машина, а и человек.

Чтобы избежать этой ошибки, достаточно вставить при записи в файл после каждой переменной пробел. Тогда программа при каждом чтении берет символы от пробела до пробела и правильно преобразует текстовое представление в число.

Кроме процедур read и write при работе с текстовыми файлами используются их разновидности readln и writeln. Отличие заключается в том, что процедура writeln после записи заданного списка записывает в файл специальный маркер конца строки. Этот признак воспринимается как переход к новой строке. Процедура readln после считывания заданного списка ищет в файле следующий признак конца строки и подготавливается к чтению с начала следующей строки.

Работа с нетипизированными файлами

Нетипизированные файлы - это последовательность компонент произвольного типа.

Открытие нетипизированного файла:

Reset(f, BufSize)

Rewrite(f, BufSize)

Параметр BufSize задает число байтов, считываемых из файла или записываемых в него за одно обращение. Минимальное значение BufSize - 1 байт, максимальное - 64 К байт. Если BufSize не указан, то по умолчанию он принимается равным 128.

Чтение данных из нетипизированного файла:

BlockRead(f, X, Count, QuantBlock);

Эта процедура осуществляет за одно обращение чтение в переменную X количества блоков, заданное параметром Count, при этом длина блока равна длине буфера. Значение Count не может быть меньше 1. За одно обращение нельзя прочесть больше, чем 64 К байтов.

Необязательный параметр QuantBlock возвращает число блоков, прочитанных текущей операцией BlockRead. В случае успешного завершения операции чтения QuantBlock = Count, в случае аварийной ситуации параметр QuantBlock будет содержать число удачно прочитанных блоков. Отсюда следует, что с помощью параметра QuantBlock можно контролировать правильность выполнения операции чтения.

Запись данных в нетипизированный файл:

BlockWrite(f, X, Count, QuantBlock);

Эта процедура осуществляет за одно обращение запись из переменной X количества блоков, заданное параметром Count, при этом длина блока равна длине буфера.

Необязательный параметр QuantBlock возвращает число блоков, записанных успешно текущей операцией BlockWrite.

Для нетипизированных файлов можно использовать процедуры Seek, FIlePos и FileSize, аналогично соответствующим процедурам типизированных файлов.

# Понятие формата файла. Примеры известных форматов файлов.

**Формат** — спецификация [структуры данных](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), записанных в компьютерном [файле](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB). Формат файла иногда указывается в его имени, как часть, отделённая точкой (обычно эту часть называют расширением имени файла, хотя, строго говоря, это неверно). Например, окончание имени (расширение) «.txt» обычно используют для обозначения файлов, содержащих только текстовую информацию, а «.doc» — содержащих текстовую информацию, структурированную в соответствии со стандартами программы Microsoft Word. Файлы, содержимое которых соответствует одному формату (реже — одному семейству форматов), иногда называют файлами одного *типа*.

Так как общепринятая в вычислительной технике концепция файла — неструктурированная последовательность [байтов](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82), [компьютерные программы](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), сохраняющие в файлах структурированные данные, должны как-то преобразовывать их в последовательность байтов и наоборот (в [ООП](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) эти операции называются, соответственно, «[сериализацией](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F)» и «десериализацией»; для текстовой информации последнее также называется «разбор» или «парсинг»). [Алгоритм](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) этого преобразования, а также соглашения о том, как различные фрагменты информации располагаются внутри файла, и составляют его «формат».

Различные форматы файлов могут различаться степенью детализации, один формат может быть «надстройкой» над другим или использовать элементы других форматов. Например, [текстовый формат](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) накладывает только самые общие ограничения на структуру данных. Формат [HTML](http://ru.wikipedia.org/wiki/HTML)устанавливает дополнительные правила на внутреннее устройство файла, но при этом любой HTML-файл является в то же время текстовым файлом.

Для многих форматов файлов существуют опубликованные [спецификации](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B5%D1%86%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F), в которых подробно описана структура файлов данного формата, то, как программы должны кодировать данные для записи в этот формат и как декодировать их при чтении. Большинство таких спецификаций свободно доступны, некоторые распространяются за плату.

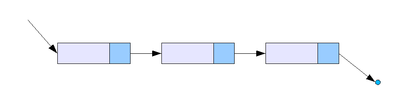
Иногда компании могут считать определённые форматы файлов своей [коммерческой тайной](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BC%D0%B5%D1%80%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%82%D0%B0%D0%B9%D0%BD%D0%B0) и не публиковать их. Хорошо известный пример — форматы файлов пакета [Microsoft Office](http://ru.wikipedia.org/wiki/Microsoft_Office). В некоторых случаях компания, выпустившая приложение, просто не считает нужным тратить время на написание подробной спецификации.

Если спецификация формата недоступна, то для обеспечения совместимости программы с данным форматом приходится заниматься [обратной разработкой](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0). В большинстве или во всех странах форматы файлов не защищены законами об авторских правах. Однако в некоторых странах патентами могут быть защищены алгоритмы, используемые для кодирования данных в какой-либо формат. Например, в широко распространённом формате [GIF](http://ru.wikipedia.org/wiki/GIF) использовался патентованный алгоритм (срок действия патентов в разных странах истек в 2003-2004 гг.), что привело к разработке альтернативного формата [PNG](http://ru.wikipedia.org/wiki/PNG).

# Работа со списками в языках программирования. Представление списков с использованием динамической (управляемой) памяти (на примере языка Паскаль или др.).

Списки в функциональных языках являются фундаментальной структурой. Большинство функциональных языков имеет встроенные средства для работы со списками вроде получения длины списка, головы (первый элемент списка), хвоста (часть списка, идущая за первым элементом), применения функции к каждому элементу списка.

**Линейный однонаправленный список** — это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей. Каждый элемент списка имеет указатель на следующий элемент. Последний элемент списка указывает на NULL. Элемент на который нет указателя является первым (головным) элементом списка.

[](http://commons.wikimedia.org/wiki/File:Single_linked_list.png?uselang=ru)

Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

**type** spisok=^element;

element=**record**; *//обозначаем элементы списка как запись, чтобы поместить туда данные и указатель на следующий элемент*

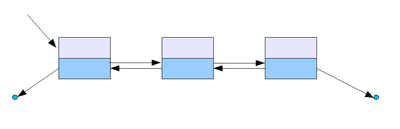
info:**integer**; *//поле info отвечает за данные, которые хранит элемент списка (может быть любого типа)*

next:spisok; *//так как список линейный, у нас будет указатель только на следующий элемент*

**end**;

**var** first,elem,last:spisok; *//создаём переменные начального элемента, вспомогательного элемента и конечного элемента типов список*

#### Двусвязный список (Двунаправленный связный список)

[](http://commons.wikimedia.org/wiki/File:Doubly_linked_list.png?uselang=ru)

Здесь ссылки в каждом узле указывают на предыдущий и на последующий узел в списке. По двусвязному списку можно передвигаться в любом направлении — как к началу, так и к концу. В этом списке проще производить удаление и перестановку элементов, так как всегда известны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

## Достоинства

* лёгкость добавления и удаления элементов
* размер ограничен только объёмом памяти [компьютера](http://ru.wikipedia.org/wiki/%D0%AD%D0%92%D0%9C) и разрядностью указателей
* динамическое добавление и удаление элементов

## Недостатки

* сложность определения адреса элемента по его [индексу](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81) (номеру) в списке
* на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в[массивах](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), например, указатели не нужны)
* работа со списком медленнее, чем с массивами, так как к любому элементу списка можно обратиться, только пройдя все предшествующие ему элементы
* элементы списка могут быть расположены в памяти разреженно, что окажет негативный эффект на кэширование процессора
* над связными списками гораздо труднее (хотя и в принципе возможно) производить параллельные векторные операции, такие как вычисление суммы

**О́чередь** — [структура данных](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) с дисциплиной доступа к элементам «первый пришёл — первый вышел» ([FIFO](http://ru.wikipedia.org/wiki/FIFO_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), First In — First Out). Добавление элемента (принято обозначать словом enqueue — поставить в очередь) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть словом dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.

**Стек** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *stack* — стопка) — [структура данных](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), представляющая собой [список элементов](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), организованных по принципу [*LIFO*](http://ru.wikipedia.org/wiki/LIFO) ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *last in — first out*, «последним пришёл — первым вышел»).

Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

# Операции со списками.

**type** spisok=^element;

element=**record**; *//обозначаем элементы списка как запись, чтобы поместить туда данные и указатель на следующий элемент*

info:**integer**; *//поле info отвечает за данные, которые хранит элемент списка (может быть любого типа)*

next:spisok; *//так как список линейный, у нас будет указатель только на следующий элемент*

**end**;

**var** first,elem,last:spisok; *//создаём переменные начального элемента, вспомогательного элемента и конечного элемента типов список*

**begin**

New(elem); *//создаём новый элемент в динамической памяти*

elem^.next:=**nil**; *//зануляем указатель на следующий элемент (так как его нету)*

elem^.info:=random(40)-20; *//заполняем поле информации генератором рандома*

first:=elem;

last:=elem; *//так как элемент пока первый и последний, присваиваем оба элемента ему*

*// Процедура добавления элемента в конец списка*

New(elem);

elem^.info:=random(40)-20;

elem^.next:=**nil**;

last^.next:=elem; *//следующий от последнего теперь наш элемент*

last:=elem; *// и наш элемент и есть теперь последним*

*//Процедура добавления элемента в начало списка*

New(elem);

elem^.info:=random(40)-20;

elem^.next:=first; *//следующим есть первый элемент*

first:=elem; *//наш элемент теперь первый*

*//Процедура удаления элемента с начала списка*

New(elem);

elem:=first; *//присваиваем нашему элементу значение первого*

first:=first^.next; *//первым стал следующий элемент*

dispose(elem); *//удаляем наш элемент*

*//Процедура удаления элемента с конца списка (тут потяжелее, так как список линейный)*

New(elem);

elem:=first;

**while**(p^.next<>last) **do** *//крутим цикл, пока не дойдем до элемента стоящего перед последним*

p:=p^.next;

p^.next=**nil**; *//делаем следующий от него элемент 0*

dispose(last); *//удаляем последний элемент*

last:=p; *//теперь наш элемент последний*

### Операции со стеком

Возможны три операции со стеком: добавление элемента (иначе проталкивание, *push*), удаление элемента (*pop*) и чтение головного элемента (*peek*)[[5]](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA_(%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80)#cite_note-5).

При проталкивании (*push*) указывается новый элемент, указывающий на элемент бывший до этого головой. Новый элемент теперь становится головным.

При удалении элемента убирается первый, а головным становится тот, на который был указатель у этого объекта (следующий элемент).

void push( STACK \*ps, int x ) // Добавление в стек нового элемента

{

if ( ps->size == STACKSIZE ) // Не переполнен ли стек?

{

fputs( "Error: stack overflow**\n**", stderr );

abort();

}

else

{

ps->items[ps->size++] = x;

}

}

int pop( STACK \*ps ) // Удаление из стека

{

if ( ps->size == 0 ) // Не опустел ли стек?

{

fputs( "Error: stack underflow**\n**", stderr );

abort();

}

else

{

return ps->items[--ps->size];

}

}

Алгоритм достаточно прост: на входе имеются указатели на первые элементы объединяемых списков. Началом результирующего списка из них выбирается элемент с наименьшим ключом. Затем в качестве следующих элементов результирующего списка выбирается последующие элементы из первого или второго исходного списка, с меньшим значением ключа. Когда достигнут последний элемент одного из исходных списков, указатель последнего элемента результирующего списка устанавливается на остаток другого входного списка.

**function** IntersectSorted(**const** pList1, pList2: PList\_Item): PList\_Item;

**var**

pCurItem: PList\_Item;

p1, p2: PList\_Item;

**begin**

p1 := pList1;

p2 := pList2;

**if** p1^.Key <= p2^.Key **then**

**begin**

pCurItem := p1;

p1 := p1^.Next;

**end**

**else** **begin**

pCurItem := p2;

p2 := p2^.Next;

**end**;

Result := pCurItem;

**while** (p1 <> **nil**) and (p2 <> **nil**) **do**

**begin**

**if** p1^.Key <= p2^.Key **then**

**begin**

pCurItem^.Next := p1;

pCurItem := p1;

p1 := p1^.Next;

**end**

**else** **begin**

pCurItem^.Next := p2;

pCurItem := p2;

p2 := p2^.Next;

**end**;

**end**;

**if** p1 <> **nil** **then**

pCurItem^.Next := p1

**else**

pCurItem^.Next := p2;

**end**;

## Сортировка односвязного списка[[править](http://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D0%BE%D0%B3%D0%BE_%D1%81%D0%BF%D0%B8%D1%81%D0%BA%D0%B0&veaction=edit&vesection=2) | [править вики-текст](http://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D0%BE%D0%B3%D0%BE_%D1%81%D0%BF%D0%B8%D1%81%D0%BA%D0%B0&action=edit&section=2)]

Процесс сортировки списка представляет из себя последовательный проход по списку с сортировкой сначала пар элементов, затем каждой двойки пар элементов, с объединением в списки из 4х элементов, затем объединяются получившиеся списки из 8, 16 и т. д. элементов. В предложенной реализации используется [стек](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA) списков. Необходимый размер стека равен [log2*n*] + 1, где n — количество элементов списка. Если количество элементов заранее не известно, то можно заранее заложить достаточную глубину стека. Так, например, стек глубиной в 32 элемента может быть использован для сортировки списков длиной до 4 294 967 295 элементов. В стеке сохраняются указатели на отсортированные части списка и уровень — число фактически равное log2*i* + 1, где i — количество элементов в этой части списка. Суть алгоритма в следующем: идёт последовательный обход списка, при этом каждый элемент преобразуется к вырожденному списку, путём удаления указателя на следующий элемент. Указатель на созданный таким образом список помещается в стек, при этом уровень указывается равным 1, после чего проводится проверка: если два последних элемента стека имеют одно и то же значение уровня, то они извлекаются из стека, производится объединение списков, на которые указывают эти элементы, а результирующий список помещается в стек с уровнем на единицу большем предыдущего. Такое объединение повторяется пока уровни двух последних элементов равны, или пока не будет достигнута вершина стека. После того как исходный список полностью пройден, перечисленные в стеке списки объединяются вне зависимости от их уровня. Получившийся в результате объединения список и есть искомый, с отсортированными элементами

**type**

TSortStackItem = **record**

Level: **Integer**;

Item: PList\_Item;

**end**;

**var**

Stack: **Array**[0..31] **of** TSortStackItem;

StackPos: **Integer**;

p: PList\_Item;

**begin**

StackPos := 0;

p := List;

**while** p <> **nil** **do**

**begin**

Stack[StackPos].Level := 1;

Stack[StackPos].Item := p;

p := p^.Next;

Stack[StackPos].Item^.Next := **nil**;

Inc(StackPos);

**while** (StackPos > 1) and (Stack[StackPos - 1].Level = Stack[StackPos - 2].Level) **do**

**begin**

Stack[StackPos - 2].Item := IntersectSorted(Stack[StackPos - 2].Item, Stack[StackPos - 1].Item);

Inc(Stack[StackPos - 2].Level);

Dec(StackPos);

**end**;

**end**;

**while** StackPos > 1 **do**

**begin**

Stack[StackPos - 2].Item := IntersectSorted(Stack[StackPos - 2].Item, Stack[StackPos - 1].Item);

Inc(Stack[StackPos - 2].Level);

Dec(StackPos);

**end**;

**if** StackPos > 0 **then**

List := Stack[0].Item;

**end**;

# Работа с бинарными деревьями в языках программирования. Представление бинарных деревьев с использованием динамической (управляемой) памяти (на примере языка Паскаль или др.).

**Двои́чное де́рево** — [древовидная](http://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)) [структура данных](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками.

Для практических целей обычно используют два подвида бинарных деревьев — [двоичное дерево поиска](http://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0) и [двоичная куча](http://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0).

Существует следующее рекурсивное определение двоичного дерева (см. [БНФ](http://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0_%D0%91%D1%8D%D0%BA%D1%83%D1%81%D0%B0_%E2%80%94_%D0%9D%D0%B0%D1%83%D1%80%D0%B0)):

<дерево> ::= ( <данные> <дерево> <дерево> ) | nil .

То есть двоичное дерево либо является пустым, либо состоит из данных и двух поддеревьев (каждое из которых может быть пустым). Очевидным, но важным для понимания фактом является то, что каждое поддерево в свою очередь тоже является деревом. Если у некоторого узла оба поддерева пустые, то он называется листовым узлом (листовой вершиной) или терминальным элементом.

Элементы дерева должны иметь тип:

Type TEL=…;

PTree=^TTree;

TTree=Record

Inf:Tel;

Left,Right:Ptree;

End;

**Двоичное дерево поиска** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *binary search tree*, BST) — это [двоичное дерево](http://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):

* Оба поддерева — левое и правое, являются двоичными деревьями поиска.
* У всех узлов левого поддерева произвольного узла X значения ключей данных *меньше*, нежели значение ключа данных самого узла X.
* В то время, как значения ключей данных у всех узлов правого поддерева (того же узла X) *больше*, нежели значение ключа данных узла X.

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения *меньше*.

Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных. Однако, это касается реализации, а не природы двоичного дерева поиска.

Для целей реализации двоичное дерево поиска можно определить так:

* Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла - левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля parent в каждом узле (кроме корневого) - ссылки на родительский элемент.
* Данные (data) обладают ключом (key), на котором определена операция сравнения "меньше". В конкретных реализациях это может быть пара (key, value) - (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
* Для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], т. е. ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.

**Двои́чная ку́ча**, **пирами́да**[[1]](http://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0#cite_note-1), или **сортиру́ющее де́рево** — такое [двоичное дерево](http://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), для которого выполнены три условия:

1. Значение в любой вершине не меньше, чем значения её потомков.
2. Глубина листьев (расстояние до корня) отличается не более чем на 1 слой.
3. Последний слой заполняется слева направо.

Существуют также кучи, где значение в любой вершине, наоборот, не больше, чем значения её потомков. Такие кучи называются min-heap, а кучи, описанные выше — max-heap. В дальнейшем рассматриваются только max-heap. Все действия с min-heap осуществляются аналогично.

# Операции с бинарными деревьями.

С помощью дерева поиска можно организовать эффективный способ поиска, который значительно эффективнее поиска по списку.

Поиск в упорядоченном дереве выполняется по следующему рекурсивному алгоритму:

* Если дерево не пусто, то нужно сравнить искомый ключ с ключом в корне дерева:

- если ключи совпадают, поиск завершен;

- если ключ в корне больше искомого, выполнить поиск в левом поддереве;

- если ключ в корне меньше искомого, выполнить поиск в правом поддереве.

* Если дерево пусто, то искомый элемент не найден.

**Пример функции поиска**

function find(root:tree; key:integer; var p, parent:tree):Boolean;   
begin   
   p:=root;   
   while p<>nil do begin   
      if key=p^.inf then begin{ узел с таким ключом есть }   
         find:=true;   
         exit;   
      end;   
      parent:=p {запомнить указатель на предка}   
      if key<p^.inf then  
         p := p ^. left {спуститься влево}   
      else p := p ^. right ; {спуститься вправо}   
   end;   
find:=false;   
end;

**Пример обхода дерева с помощью рекурсии**

Procedure obhod(p:tree);   
Begin   
   if p<>nil then   
   begin   
      obhod(p^.left);   
      writeln(p^.inf);   
      obhod(p^.right);   
   end;   
end;

Вставка узла в двоичное дерево поиска не представляет сложности. Для того чтобы вставить узел, необходимо найти его место. Для этого мы сравниваем вставляемый ключ с корнем, если ключ больше, чем ключ корня, уходим в правое поддерево, а иначе – в левое. Тем же образом продвигаемся дальше, пока не дойдем до конечного узла (листа). Сравниваем вставляемый ключ с ключом листа. Если ключ меньше ключа листа, то добавляем листу левого сына, а иначе – правого сына.

Если узел является конечным (то есть не имеет потомков), то его удаление не вызывает трудностей, достаточно обнулить соответствующий указатель узла-родителя.

Сложнее всего случай, когда у удаляемого узла есть оба потомка.

Есть простой особый случай: если у правого потомка удаляемого узла нет левого потомка, удаляемый узел заменяется на своего правого потомка, а его левый потомок подключается вместо отсутствующего левого потомка к замещающему узлу.

В общем же случае на место удаляемого узла ставится самый левый лист его правого поддерева (или наоборот – самый правый лист его левого поддерева). Это не нарушает свойств дерева поиска.

Корень дерева удаляется по общему правилу за исключением того, что заменяющий его узел не требуется присоединять к узлу-родителю.

**Пример программы удаления узла из дерева**

procedure del ( var root : tree ; key : integer );   
var   
   p : tree ; {удаляемый узел}   
   parent : tree ; {предок удаляемого узла}   
   y : tree ; {узел, заменяющий удаляемый}   
function spusk(p:tree):tree;   
var   
   y : tree ; {узел, заменяющий удаляемый}   
   pred:tree; { предок узла “y”}   
begin   
   y:=p^.right;   
   if y^.left=nil then y^.left:=p^.left {1}   
   else {2}   
   begin   
      repeat   
         pred:=y; y:=y^.left;   
      until y^.left=nil;   
      y^.left:=p^.left; {3}   
      pred^.left:=y^.right; {4}   
      y^.right:=p^.right; {5}   
   end;   
   spusk:=y;   
end;   
begin   
   if not find(root, key, p, parent) then {6}   
   begin writeln(‘ такого элемента нет ’); exit; end;   
   if p^.left=nil then y:=p^.right {7}   
   else   
      if p^.right=nil then y:=p^.left {8}   
      else y:=spusk(p); {9}   
   if p=root then root:=y {10}   
   else {11}   
      if key<parent^.inf then   
         parent^.left:=y   
      else parent^.right:=y;   
   dispose(p); {12}   
end.

В функцию **del** передаются указатель **root** на корень дерева и ключ **key** удаляемого элемента. С помощью функции **find** определяются указатели на удаляемый элемент **p** и его предка **parent** . Если искомого элемента в дереве нет, то выдается сообщение ( {6}) .

В операторах {7}-{9} определяется указатель на узел **y** , который должен заменить удаляемый. Если у узла **p** нет левого поддерева, на его место будет поставлена вершина (возможно пустая) его правого поддерева ({7}).

Иначе, если у узла **p** нет правого поддерева, на его место будет поставлена вершина его левого поддерева ({8}).

В противном случае, когда оба поддерева существуют, для определения замещающего узла вызывается функция**spusk** , выполняющая спуск по дереву ({9}).

В этой функции первым делом проверяется особый случай, описанный выше ({1}). Если же этот случай (отсутствие левого потомка у правого потомка удаляемого узла) не выполняется, организуется цикл ({2}), на каждой итерации которого указатель на текущий элемент запоминается в переменной **pred** , а указатель **y**смещается вниз и влево до того момента, пока не станет ссылаться на узел, не имеющий левого потомка (он-то нам и нужен).

В операторе {3} к этой пустующей ссылке присоединяется левое поддерево удаляемого узла. Перед тем как присоединять к этому узлу правое поддерево удаляемого узла ({5}), требуется «пристроить» его собственное правое поддерево. Мы присоединяем его к левому поддереву предка узла y , заменяющего удаляемый ({4}), поскольку этот узел перейдет на новое место.

Функция spusk возвращает указатель на узел, заменяющий удаляемый. Если мы удаляем корень дерева, надо обновить указатель на корень ({10}), иначе – присоединить этот указатель к соответствующему поддереву предка удаляемого узла ({11}).

После того как узел удален из дерева, освобождается занимаемая им память ({12}).

# Алгоритмы и задачи работы с множествами в языках программирования.

Классификация: нестандартный, структурированный (сложный) тип.

Имя определяет программист.

Множество - аналог множества в математике.

Структурная организация

По структуре - множество есть набор из фиксированного числа однотипных элементов (рис. 20.1).. Для того, чтобы рассматривать дальнейшую работу с множествами имеет смысл представлять внутреннюю структуру множества. Все элементы поименованы значениями возможных элементов множества. Если в текущий момент времени элемент принадлежит множеству, то значение этого элемента во внутренней структуре 1, и значение 0 - если элемент в этот момент времени не принадлежит множеству.

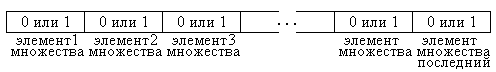


Рис. 20.1. – Структура данного типа множество

Определение типа

Для того, чтобы определить тип множество необходимо перечислить все элементы, которые могут входить в множество. Есть ограничение на мощность множества - не более 256 элементов. Элементами множества могут быть только данные простого порядкового типа.

Правило определения типа множество на рис. 20.2. По правилам структурного программирования тип множество всегда определяется в разделе нестандартных типов.

рисунки\10_02.bmp

Рис. 20.2. – Определение типа множество

Множество значений

Множество значений определяется набором признаков - принадлежат ли возможные элементы множеству или не принадлежат.

Значения множества задаются с помощью конструктора множества, определение которого приведено на рис. 20.3.

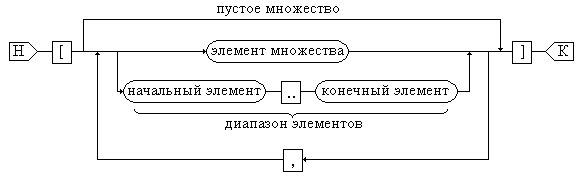


Рис. 20.3. – Конструктор множества

Пример определения значений множества:

type raduga=(kr,orn,gel,zel,gol,sini,fiol);

sveta = set of raduga;

var a : cveta;

begin

a:=[kr,gol..fiol]

end.

После выполнения действий программы содержимое множества а будет таким:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| kr | orn | gel | zel | gol | sini | fiol |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |

Рис. 20.4. – Содержимое множества а

Множество операций

Множественные операции

Это бинарные операции. Операнды и результат одного типа множество. Эти операции являются аналогами теоретико-множественных операций:

объединение ;

пересечение ;

разность .

Геометрическая интерпретация этих операций приведена на рис. 20.5.

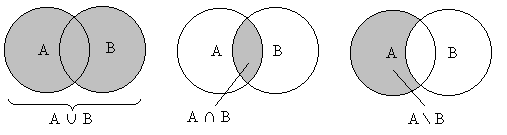


Рис. 20.5. – Операции над множествами

Рассмотрим эти операции на примере:

type raduga=(kr,orn,gel,zel,gol,sini,fiol);

sveta = set of raduga;

var a,b,c: cveta;

a:=[kr,gol..fiol,gel];

b:=[fiol,kr,zel];

После выполнения операторов присваивания имеем:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | kr | orn | gel | zel | gol | sini | fiol |
| a | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | kr | orn | gel | zel | gol | sini | fiol |
| b | 1 | 0 | 0 | 1 | 0 | 0 | 1 |

Рис. 20.6. – Содержимое множеств а и b

Объединение множеств

Бинарная аддитивная операция со знаком +. Пример выполнения показан на рис. 20.7.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | kr | orn | gel | zel | gol | sini | fiol |
| c:=a+b | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

Рис. 20.7. –Операция объединения множеств

Разность множеств

Бинарная аддитивная операция со знаком – (рис. 20.8).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | kr | orn | gel | zel | gol | sini | fiol |
| c:=a-b | 0 | 0 | 1 | 0 | 1 | 1 | 0 |

Рис. 20.8. –Операция разности множеств

Пересечение множеств

Бинарная мультипликативная операция со знаком \* (рис. 20.9).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | kr | orn | gel | zel | gol | sini | fiol |
| c:=a\*b | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

Рис. 20.9. –Операция пересечения множеств

Операции сравнения

Сравнение множеств

Бинарные операции со знаками =|<>|>=|<= , операнды одного и того же множественного типа. Результат логического типа.

a=b ,значение TRUE, если множества полностью совпадают;

a<>b ,значение TRUE, если множества не совпадают;

a>=b ,значение TRUE, если b является подмножеством множества a;

a<=b ,значение TRUE, если a является подмножеством множества b.

2.2.2 Проверка на принадлежность к множеству

Бинарная операция сравнения со знаком IN . Первый операнд – элемент множества, второй операнд множество. Результат типа BOOLEAN - TRUE, если элемент принадлежит множеству, FALSE в противном случае.

Синтаксис операции: *<элемент множества> in <множество>*

Синтаксическое ограничение: тип первого операнда совпадает с типом компонента второго операнда.

Операция определения адреса

Унарная операция со знаком @. Операнд переменная множественного типа. Результат - указатель на участок оперативной памяти, в котором располагается переменная.

. Задача обработки множеств

Постановка задачи

Было куплено 2 коробки одинаковых цветных карандашей. Какое-то время ими пользовались. Определить карандаши какого цвета имеются в наличии, какого отсутствуют, какого имеются в двух экземплярах.

Каждую из коробок можно представить в виде множеств из цветов возможных карандашей. Тогда нам потребуется вычислить три множества:

множество наличных цветов карандашей (определяется как объединение двух исходных множеств);

множество отсутствующих карандашей (определяется как вычитание из множества всех возможных цветов множества наличных цветов);

множество дублирующих цветов карандашей (определяется как пересечение двух исходных множеств).

Структура программы

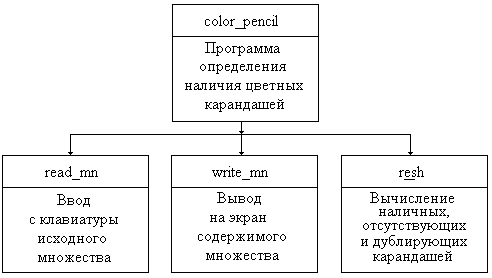


Рис. 20.10. - Структурная диаграмма программы

Выделим в поставленной задаче три подзадачи:

ввод с клавиатуры множества (содержимого коробки карандашей);

вывод на экран содержимого множества из карандашей;

вычисление множеств наличных, отсутствующих и дублирующих карандашей.

Структурная диаграмма программы представлена на рис. 20.10

Разработка подпрограммы ввода множества

Спецификация

Назначение: ввод с клавиатуры значения множества, элементами которого являются названия цветных карандашей

Имя: read\_mn

Вид: процедура

Перечень параметров:

Таблица 20.1. Перечень параметров

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Статус | Назначение | Имя | Тип | Вид |
| Выход | Формируемое множество | n | tkorobka | параметр-переменная |

type

{названия возможных цветов карандашей}

tpalitra = (krasn, gelt, zelen, sini, korichn, chern);

tkorobka = set of tpalitra;{тип информационной модели коробки карандашей}

Заголовок: *procedure read\_mn(var n:tkorobka);*

Метод решения

Суть метода решения заключается в том, что к исходному пустому множеству последовательно добавляются возможные наличные элементы (объединение множеств)

1. n:=
2. i   krasn, gelt, zelen, sini, korichn, chernесли otvet='включить i в множество'n:=n{i}

Информационная модель

Таблица 20.2. Информационная модель

|  |  |  |
| --- | --- | --- |
| Назначение | Имя | Тип |
| Очередной цвет | i | tpalitra |
| Ответ пользователя (да - включать/нет – не включать элемент) | otvet | char |

type tpalitra = (krasn, gelt, zelen, sini, korichn, chern);

В алгоритме используется глобальная константа – массив с названиями цветов карандашей

const colors:array[tpalitra]of string=('красный', 'желтый', 'зеленый', 'синий',   
 'коричневый', 'черный');

Программная модель

procedure read\_mn(var n:tkorobka);

var i:tpalitra;

otvet:char;

begin

n:=[];{коробка пуста - пустое множество}

{перебор возможных цветов}

for i:=krasn to chern do

begin

writeln('Есть ',colors[i],' карандаш? Ответьте: д/н');

readln(otvet);

if (otvet='Д')or(otvet='д')or(upcase(otvet)='L') then

begin

n:=n+[i];{добавление элемента в множество}

writeln(colors[i],' карандаш находится в коробке')

end

else writeln(colors[i],' карандаш отсутствует в коробке')

end

end;

Разработка подпрограммы вывода множества

Спецификация

Назначение: вывод на экран значения множества, элементами которого являются названия цветных карандашей

Имя: write\_mn

Вид: процедура

Перечень параметров:

Таблица 20.3. Перечень параметров

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Статус | Назначение | Имя | Тип | Вид |
| Вход | Выводимое множество | n | tkorobka | параметр-константа |

type

{названия возможных цветов карандашей}

tpalitra = (krasn, gelt, zelen, sini, korichn, chern);

tkorobka = set of tpalitra;{тип информационной модели коробки карандашей}

Заголовок: procedure write\_mn(const n:tkorobka);

Метод решения

1. Обнулить счетчик выведенных элементов *k:=0*

2*. i   krasn, gelt, zelen, sini, korichn, chernесли i  n  вывод (i)*

*k:=k+1*

3*. Если k=0 вывод ('множество пусто')*

Информационная модель

Таблица 20.4. Информационная модель

|  |  |  |
| --- | --- | --- |
| Назначение | Имя | Тип |
| Очередной цвет | i | tpalitra |
| Количество выведенных значений | k | integer |

type tpalitra = (krasn, gelt, zelen, sini, korichn, chern);

В алгоритме используется глобальная константа – массив с названиями цветов карандашей const colors:array[tpalitra]of string=('красный','желтый','зеленый','синий',  
 'коричневый','черный');

Программная модель

procedure write\_mn(const n:tkorobka);

var k:integer;

i:tpalitra;

begin

k:=0;

for i:=krasn to chern do

if i in n then

begin

if k<>0 then write(', ');

k:=k+1;

write(colors[i])

end;

if k=0 then writeln('нет карандашей')

else if k=1 then writeln(' карандаш')

else writeln(' карандаши')

end;

Разработка подпрограммы resh

Спецификация

1. Назначение: вычисление множеств наличных карандашей, отсутствующих карандашей, дублирующих карандашей (все множества типа tkorobka).

2. Имя: resh

3. Вид: процедура

4. Перечень параметров

Таблица 20.5. Перечень параметров

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Статус | Назначение | Имя | Тип | Вид |
| Вход | Первая коробка | n1 | tkorobka | параметр-константа |
| Вход | Вторая коробка | n2 | tkorobka | параметр-константа |
| Выход | Наличные карандаши | nal | tkorobka | параметр-значение |
| Выход | Отсутствующие карандаши | otsut | tkorobka | параметр-значение |
| Выход | Дублирующие карандаши | dubl | tkorobka | параметр-значение |

Метод решения

1) nal:=n1n2;

2) otsut:={ krasn, gelt, zelen, sini, korichn, chern } \ nal;

3) dubl:=n1n2.

Текст процедуры

procedure resh(const n1,n2:tkorobka;var nal, otsut, dubl :tkorobka);

begin

{имеющиеся карандаши}

nal:=n1+n2; {объединение двух множеств - коробок}

{отсутствующие карандаши}

otsut:=[krasn..chern]-nal;

{вычитание из множества возможных цветов множества наличных цветов}

{дублирующие карандаши}

dubl:=n1\*n2

end;

Разработка программы

Метод решения

1) read\_mn(n1);

2) read\_mn(n2);

3) resh(n1,n2,nal,otsut,dubl);

4) write\_mn(nal);

5) write\_mn(otsut);

6) write\_mn(dubl).

Информационная модель

Таблица 20.6. Информационная модель

|  |  |  |  |
| --- | --- | --- | --- |
| Статус | Назначение | Имя | Тип |
| Вход | Первая коробка | n1 | tkorobka |
| Вход | Вторая коробка | n2 | tkorobka |
| Выход | Наличные карандаши | nal | tkorobka |
| Выход | Отсутствующие карандаши | otsut | tkorobka |
| Выход | Дублирующие карандаши | dubl | tkorobka |

Текст программы

program color\_pencil;

type

{названия возможных цветов карандашей}

tpalitra = (krasn, gelt, zelen, sini, korichn, chern);

tkorobka = set of tpalitra;{тип информационной модели коробки карандашей}

const

{массив с символьными названиями цветов карандашей}

colors:array[tpalitra]of string=('красный','желтый','зеленый','синий','коричневый','черный');

procedure read\_mn(var n:tkorobka);

var i:tpalitra;

otvet:char;

begin

n:=[];{коробка пуста - пустое множество}

{перебор возможных цветов}

for i:=krasn to chern do

begin

writeln('Есть ',colors[i],' карандаш? Ответьте: д/н');

readln(otvet);

if (otvet='Д')or(otvet='д')or(upcase(otvet)='L') then

begin

n:=n+[i];{добавление элемента в мнжество}

writeln(colors[i],' карандаш находится в коробке')

end

else writeln(colors[i],' карандаш отсутствует в коробке')

end

end;

procedure write\_mn(const n:tkorobka);

var k:integer;

i:tpalitra;

begin

k:=0;

for i:=krasn to chern do

if i in n then

begin

if k<>0 then write(', ');

k:=k+1;

write(colors[i])

end;

if k=0 then writeln('нет карандашей')

else if k=1 then writeln(' карандаш')

else writeln(' карандаши')

end;

procedure resh(const n1,n2:tkorobka;var nal, otsut, dubl :tkorobka);

begin

{имеющиеся карандаши}

nal:=n1+n2; {объединение двух множеств - коробок}

{отсутствующие карандаши}

otsut:=[krasn..chern]-nal;

{вычитание из множества возможных цветов множества наличных цветов}

{дублирующие карандаши}

dubl:=n1\*n2

end;

var n1,n2 : tkorobka; {коробки с карандашами}

nal,otsut,dubl:tkorobka; {результаты вычислений: наличие, отсутствие, дубль}

begin

writeln('заполнение первой коробки');

read\_mn(n1);

writeln('заполнение второй коробки');

read\_mn(n2);

resh(n1,n2,nal,otsut,dubl);

writeln('Имеющиеся:');

write\_mn(nal);

writeln('Отсутствующие:');

write\_mn(otsut);

writeln('Дублирующие:');

write\_mn(dubl)end.

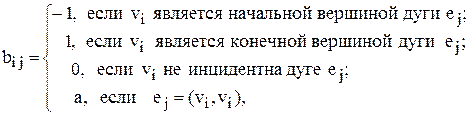
# Программное представление графов.

Машинное представление графа.

Приведем сравнительную характеристику существующих способов представления графа в памяти ЭВМ, их достоинства и недостатки.

Рассмотрим конечный граф G = (V, E), где |V | = n, |E| = m.

**Матрица инциденций**. Ориентированный граф задается прямоугольной матрицей B(n´m), элементы которой определяются по правилу:



где a – любое натуральное число, отличное от 1. У неориентированного графа оба элемента матрицы, соответствующие вершинам, инцидентным ребру ej, равны 1.

Это представление графа является самым неудобным, так как объем занимаемой памяти равен n×m единиц, причем в каждом столбце только две ненулевые [ячейки](http://click01.begun.ru/click.jsp?url=mVvfjdHe394dhFsBAkN7B3b1B1zLEmbWj2g0KaOozhcGKy2owDIlbrAZ5UsX77MZHwue4UaSOFGJhGmjMHHxuDJWwTEHxQpnj6RQKtlgb6BJ1OlKqDUb3oXNk-pVlcboPb2tizQsY6x3sA6abTXTe0akz1rHm51NbO8GjVDG9FrGdhI7Qrk9rhX5X*7m0n2BO3VKq8wt-uly6oNpKEiQ9I4iOoZKkPEOGxbCPyM9JQJ8daguvOykl6tAHSO*C*TgIwp24LzVul33pnilVP693JZfBRn47TWS-3cfyQCwPBNelRLFL*T*MLNFGfv6sf7vVhiaHAUrvLOmltikk-uvqrXMpx5nqbywyK8bTwdsRhP2ZwoSnm-7mPyP87gKiapnV3axIqDYozmJG41Eq4Gc-NFQZMsA2u6g1WgTrxOTI9TYHxB1Lnw3OCTTjjpxWuIqhF*C3o1EIyakCe1SKPfxCvrWiBJDRXx1cW3TOFbixIDaLmyOwguoYcxq6otbZGx*sLMw73jsuu1BoPiY9t0HqcUP4v4tj0lESeh8qy7fBFM*kdKmYHfdyiuUWneOHl8RaSHKOJrYJw8RLbQHdymEWt9M*YHxyeo9SU3IWSvokyXLCCGN9AWmSc9C4gU&eurl%5B%5D=mVvfjZGRkJEfh2CEXXRTnK5MUJijVjV6y1q*vNBPMH3-qfG3EL2HVrg0UncqgeIg5jHONw). Кроме нерационального использования памяти недостатком этого способа представления является неудобный [доступ к информации](http://click01.begun.ru/click.jsp?url=mVvfjWVramtaimoQE1JqFmfkFk3aA3fHnnklOHTbMszQAA4*2q7gmvbhioV67jdP*JBRk87dGq8XJ4JFZOJzlpn*Y*JL3N4qgYXs6B1yGAT4Bs1*tNJbaEMqaXsXIdjK3qkDe5cxF4y6KU-KKGvxqfy-B2PNXQZfd2GsjWXZ8VacZGXyIFaXiYU*yRADb8JTTzCpWxvzSI3kkGzHzno9ww14z0ImJKTX7dmMcKOh*CKQc2P5H7P-umk5c2gg7pVjPX3Q9J-VQTVnZ0hKAy03C9lUcfv6Mp3wYlUDssMQptF*V1GDrSYgSc63SUPAbeSd2dTR1fsJJhPKwPMyXMzPUS0duoDnQeMkBlxIJOyNF5xEPc0kNiADEyRuTSwNIdLgL*pT9K68M4BXt*1cq17r*QGmS5U*Fpf0*MqVAb4Uoib-Jvrp8g-Tk-afjD*UNaiv7wymL*2*Ns6zXGUmhOP0mPWRrAngkb7S*D0F81Esz6gOWKlcV71cQ5*XKfGJZdnWWpVGICZm2eiOr5mIKerHaAkq4K-c7oEXsuq1xXAqxWDq0c7BkqBp09Jm-JhaNldCDChMzV0e80lzkK1etCL0CK*E2UF8w4hWgCcT2jJo71wQTZEH00-oieLs3HjTeFw-tOwvXlJHm5knYH4mZL32gMI4vUkLoEkZe20agl8PuEKuH6-jdY6ZY7vwwik&eurl%5B%5D=mVvfjZWVlJVzjZJPlr*YV2WHm1PtSTdP6yoIZfZ6mk1ajQaMGpwTk5tUV4-qfFsTexyCZA). Например, для ответа на вопросы:

а) существует ли ребро (дуга) (vi, vj);

б) к каким вершинам ведут ребра (дуги) из вершины vi

требуется, в худшем случае, перебор n×m элементов, т.е. порядка n×m шагов алгоритма. От этого недостатка свободен следующий способ представления графа.

**Матрица смежности.** Элементы квадратной матрицы A(n´n) определяются следующим образом:

http://ok-t.ru/life-prog/baza1/100121816771.files/image029.gif

Такая форма эквивалентна матричному представлению бинарного отношения (см. тема “Отношения”). Проверка существования ребра (дуги) (vi, vj) осуществляется за один шаг, в отличие от матрицы инциденций, однако, проверка свойств графа на основе такого представления требует, в худшем случае, порядка n2 шагов алгоритма. При этом способе объем неиспользованной памяти по-прежнему велик.

При работе со взвешенными графами для хранения весов ребер (дуг) требуются дополнительные одномерные массивы размера m (для случая матрицы инциденций) или матрицы размера n´n (для случая матрицы смежности). Это обстоятельство делает неприемлемым использование матрицы смежности для взвешенных графов, так как количество неиспользованных единиц памяти увеличивается в k раз, гдеk *–*число весов ребер (дуг).

**Таблица ребер.**Она представляет собой матрицу размером m´2, каждая строка которой содержит вершины инцидентные i-му ребру (i-ой дуге). Для работы со взвешенными графами нужно добавить к матрице столбцы, соответствующие весам ребер (дуг). Такая форма эквивалентна табличному представлению бинарного отношения

Однако, этому способу представления графа присущ тот же недостаток, что и матрице инциденций, – неудобство доступа к информации, хотя число шагов при поиске ребра здесь значительно меньше (порядка m). Наиболее удобной и экономичной формой представления графа являются

**Списки инцидентности.**Для каждой вершины viÎV создается список записей, характеризующих ребра (дуги), инцидентные этой вершине. Таким образом, это представление использует объем памяти порядка n + m, поиск вершины смежной с данной требует порядка n + m шагов, проверка свойств графа осуществляется за число шагов порядка n. Такая форма эквивалентна представлению бинарного отношения в виде верхних срезов.

# Реализация операций над графами.

Реализация простейших операций над графами, представленными списками смежности

    На этом шаге мы рассмотрим ***несколько операций над графами, представленными списками смежности***.

    Теперь приведем реализацию на языке **C++** простейших операций над графами с использованием представления графов списками смежности.

    Вначале опишем типы данных:

#define N 12; // Количество вершин графа.

typedef struct zveno \*svqz;

typedef struct zveno

{

int Key; //Вершина графа.

svqz Sled; // Указатель на следующую смежную вершину.

} Leader;

svqz beg[N]; // Описание типа списков смежности.

***1. Построение списков смежности, соответствующих данному ориентированному графу.*** Перед первым обращением к функции **MakeGraph** (создание графа) необходима инициализация списков смежности:

for (i=0;i<N;i++) beg[i] = NULL;

void MakeGraph (svqz beg[N])

// Построение списков смежности beg графа.

{

int x,y;

svqz ukzv,uzel; //Рабочие указатели.

cout<<"Вводите начало дуги: ";

cin>>x;

while (x!=0)

{

cout<< "Вводите конец дуги: ";

cin>>y;

AddGraph (x,y,beg);

cout<< "Вводите начало дуги: "; cin>>x;

}

}

***2. Вывод содержимого списков смежности, соответствующих ориентированному графу.***

void PrintGraph (svqz beg[N])

{

int i;

svqz ukzv; //Рабочий указатель.

for (i=1;i<N;i++)

{

cout<<i<<" ...";

ukzv = beg[i];

if (ukzv==NULL) cout<<"Пустой список!\n";

else {

while (ukzv!=NULL)

{ cout<< (\*ukzv).Key; ukzv = (\*ukzv).Sled; }

cout<<endl; }

}

}

    Теперь рассмотрим реализацию унарных операций [1, с.22] на графе.

***3. Добавление дуги* (x,y) *(если ее не было!) к спискам смежности, соответствующим ориентированному графу.***

void AddGraph (int x, int y, svqz beg[N])

{

svqz ukzv,uzel; //Рабочие указатели.

if (beg[x]!=NULL)

{

Poisk (beg[x],y,&ukzv);

if (ukzv==NULL)

{ // Добавление элемента в конец списка,

// заданного указателем beg[x].

uzel = new (Leader);

(\*uzel).Key = y; (\*uzel).Sled = NULL; ukzv = beg[x];

while ((\*ukzv).Sled!=NULL)

ukzv = (\*ukzv).Sled;

(\*ukzv).Sled = uzel;

}

}

else

{

beg[x] = new (zveno);

(\*beg[x]).Key = y; (\*beg[x]).Sled = NULL;

}

}

***4. Удаление дуги между двумя заданными вершинами графа, представленного списками смежности (заметим, что вершины, инцидентные дуге, из графа не удаляются).***

void DeleteGraph (int x, int y, svqz beg[N])

{

svqz ukzv;

if (beg[x]!=NULL)

//Удаление звена из списка без заглавного звена.

{ //Вершины в графе есть.

Poisk (beg[x],y,&ukzv);

if (ukzv!=NULL) Udalenie (&beg[x],&ukzv);

else cout<<"Такой дуги в графе нет!\n";

}

else cout<<"Список пуст!\n";

}

void Poisk (svqz uksp, int ment, svqz \*res)

// Поиск звена с информационным полем ment в

// однонаправленном списке uksp. \*res - указатель

// на найденное звено или NULL.

{

svqz q;

\*res = NULL; q = uksp;

while ((q!=NULL)&&(\*res==NULL))

{ if ((\*q).Key==ment) \*res = q;

q = (\*q).Sled; }

}

void Udalenie (svqz \*ukstr, svqz \*zv)

// Удаление звена, на которое ссылается указатель \*zv,

// из однонаправленного списка, заданного указателем \*ukstr.

{

svqz ukzv,z;

if (((\*\*ukstr).Sled==NULL)&&(\*zv==\*ukstr))

// В списке - один элемент!

{ \*ukstr = NULL; delete zv; }

else

if (\*zv==\*ukstr) // Удаляемый элемент - первый.

{ \*ukstr = (\*\*ukstr).Sled; delete zv; }

else {

z = \*ukstr; ukzv = (\*\*ukstr).Sled;

while (ukzv!=\*zv)

{ z = ukzv; ukzv = (\*ukzv).Sled; }

(\*z).Sled = (\*(\*zv)).Sled; delete zv;

}

}

    Отметим, что удаление вершины **v** из графа **G** приводит к подграфу, содержащему все вершины графа **G** за исключением **v**, и все ребра графа **G**, не инцидентные **v**. Заметим, что при данном представлении графа удаление вершин становится достаточно громоздким.

# Понятие рекурсии в языках программирования. Рекурсивные процедуры и функции.

**Рекурсия** (от латинского recursio - возвращение) - это такой способ организации вычислительного процесса, при котором процедура или функция в ходе выполнения составляющих ее операторов обращается сама к себе.

В программировании рекурсия — вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция A вызывает функцию B, а функция B — функцию A. Количество вложенных вызовов функции или процедуры называется глубиной рекурсии. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

Структурно рекурсивная функция на верхнем уровне всегда представляет собой команду ветвления (выбор одной из двух или более альтернатив в зависимости от условия (условий), которое в данном случае уместно назвать «условием прекращения рекурсии»), имеющей две или более альтернативные ветви, из которых хотя бы одна является рекурсивной и хотя бы одна —терминальной. Рекурсивная ветвь выполняется, когда условие прекращения рекурсии ложно, и содержит хотя бы один рекурсивный вызов — прямой или опосредованный вызов функцией самой себя. Терминальная ветвь выполняется, когда условие прекращения рекурсии истинно; она возвращает некоторое значение, не выполняя рекурсивного вызова. Правильно написанная рекурсивная функция должна гарантировать, что через конечное число рекурсивных вызовов будет достигнуто выполнение условия прекращения рекурсии, в результате чего цепочка последовательных рекурсивных вызовов прервётся и выполнится возврат.

Помимо функций, выполняющих один рекурсивный вызов в каждой рекурсивной ветви, бывают случаи «параллельной рекурсии», когда на одной рекурсивной ветви делается два или более рекурсивных вызова. Параллельная рекурсия типична при обработке сложных структур данных, таких как деревья. Простейший пример параллельно-рекурсивной функции — вычисление ряда Фибоначчи, где для получения значения n-го члена необходимо вычислить (n-1)-й и (n-2)-й.

Реализация рекурсивных вызовов функций в практически применяемых языках и средах программирования, как правило, опирается на механизм стека вызовов — адрес возврата и локальные переменные функции записываются в стек, благодаря чему каждый следующий рекурсивный вызов этой функции пользуется своим набором локальных переменных и за счёт этого работает корректно. Оборотной стороной этого довольно простого по структуре механизма является то, что на каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека вызовов.

Вопрос о желательности использования рекурсивных функций в программировании неоднозначен: с одной стороны, рекурсивная форма может быть структурно проще и нагляднее, в особенности, когда сам реализуемый алгоритм по сути рекурсивен. Кроме того, в некоторых декларативных или чисто функциональных языках (таких как Пролог или Haskell) просто нет синтаксических средств для организации циклов, и рекурсия в них — единственный доступный механизм организации повторяющихся вычислений. С другой стороны, обычно рекомендуется избегать рекурсивных программ, которые приводят (или в некоторых условиях могут приводить) к слишком большой глубине рекурсии. Так, широко распространённый в учебной литературе пример рекурсивного вычисления факториала является, скорее, примером того, как не надо применять рекурсию, так как приводит к достаточно большой глубине рекурсии и имеет очевидную реализацию в виде обычного циклического алгоритма.

Имеется специальный тип рекурсии, называемый «хвостовой рекурсией». Интерпретаторы и компиляторы функциональных языков программирования, поддерживающие оптимизацию кода (исходного или исполняемого), автоматически преобразуют хвостовую рекурсию к итерации, благодаря чему обеспечивается выполнение алгоритмов с хвостовой рекурсией в ограниченном объёме памяти. Такие рекурсивные вычисления, даже если они формально бесконечны (например, когда с помощью рекурсии организуется работа командного интерпретатора, принимающего команды пользователя), никогда не приводят к исчерпанию памяти. Однако далеко не всегда стандарты языков программирования чётко определяют, каким именно условиям должна удовлетворять рекурсивная функция, чтобы транслятор гарантированно преобразовал её в итерацию. Одно из редких исключений — язык Scheme (диалект языка Lisp), описание которого содержит все необходимые сведения.

Теоретически, любую рекурсивную функцию можно заменить циклом и стеком. Однако такая модификация, как правило, бессмысленна, так как приводит лишь к замене автоматического сохранения контекста в стеке вызовов на ручное выполнение тех же операций с тем же расходом памяти. Исключением может быть ситуация, когда рекурсивный алгоритм приходится моделировать на языке, в котором рекурсия запрещена.

# Механизм выполнения рекурсивных процедур.

Хорошей иллюстрацией механизма рекурсии является функция для вычисления факториала натурального числа. Вспомним, что факториалом числа называется произведение всех натуральных чисел от 1 до этого числа включительно:

N! = 1\*2\*3\* . . . \*(N-2)\*(N-1)\*N  
1! = 1  
0! = 1

Сначала покажем обычную не рекурсивную функцию для вычисления факториала, которая реализует итеративный алгоритм вычисления:

|  |
| --- |
| Function NonRecFact(N:integer) : LongInt; Var   i : integer; {переменная цикла }   Res : LongInt; {результат} Begin   Res := 1;   for i := 1 to N do     res := Res\*i;   NonResFact := Res; End; |

Вторая функция использует рекурсивные обращения, что делает ее гораздо компактнее, и основана на очевидном соотношении:

N! = (N-1)!\*N

Иными словами, чтобы получить значение факториала от числа N, достаточно умножить на N значение факториала от предыдущего числа:

|  |
| --- |
| Function RecFact(N:integer) : LongInt; Begin   if N <= 1      then       ResFact := 1     else       ResFact := N\*ResFact(N-1); End; |

Полностью программа, вычисляющая факториал числа, будет выглядеть так:

|  |
| --- |
| Program Rekurs; Var   N : integer;   F : Longint;  Function RecFact(N:integer) : LongInt; Begin   if N <= 1      then       ResFact := 1     else       ResFact := N\*ResFact(N-1); End;  Begin   writeln('Введите число N > ';   read(N);   F := RecFact(N);   writeln('Для числа ',N,' значение факториала равно ',F); End. |

После запуска программы на экран выводится запрос "Введите число N > ", затем с клавиатуры считывается введенное значение и в выражении F:=RecFact(N) вызывается функция RecFact с параметром-значением N. В подпрограмме-функции проверяется условие N<=1. Если оно выполняется, то функции ResFact присваивается значение 1 и на этом выполнение подпрограммы завершается. Если условие N<=1 не соблюдается, то выполняется вычисление произведения N\*ResFact(N-1).

Вычисление произведения носит рекурсивный характер, так как при этом осуществляется вызов функции ResFact(N-1), значение которой вычисляется, в свою очередь, через вызов функции ResFact, параметром которой также будет функция ResFact, и т.д., до тех пор пока значение формального параметра N не будет равно 1. Так как базовая часть описания рекурсивной функции ResFact определяет значение ResFact для N=1, равным единице, то рекурсивные вызовы функции ResFact больше не выполняются, а наоборот выполняется вычисление функции ResFact для чисел, возрастающих от 1 до N , причем функция ResFact всякий раз возвращает значение, равное произведению очередного числа на факториал от предыдущего числа. Последнее возвращение результата вычисления функции ResFact присвоит переменной F значение произведения всех чисел от 1 до N, т.е. факториал числа N.

Итак, при выполнении рекурсивной подпрограммы осуществляется многократный переход от некоторого текущего уровня организации алгоритма к нижнему уровню последовательно до тех пор, пока не будет получено тривиальное решение поставленной задачи. В нашем примере решение при N=1 тривиально, т.е. ResFact=1. Затем осуществляется возврат на верхний уровень с последовательным вычислением значения функции ResFact.

# Методы и алгоритмы сортировки одномерных массивов.

Под сортировкой понимается упорядочение элементов массива по тем или иным признакам. Поле, по которому будет проводиться сортировка, будем называть ключом (key). Остальные поля – дополнительными. Фактически сортировка производится по ключу, но вместе с ключом перемещаются дополнительные поля. Различают внутреннюю и внешнюю сортировки. Внутренняя предполагает, что данные находятся в оперативной памяти. В этом случае время доступа к элементу данных минимально. Внешняя сортировка подразумевает расположение данных на внешнем носителе. При этом время доступа к данным может быть значительным. Очевидно, алгоритмы внешней сортировки работают на несколько порядков медленнее. Часто оптимальным оказывается вариант смешанной сортировки. При этом данные в память загружаются блоками, т.е. сортировка производится в памяти, а затем отсортированный блок переписывается на внешнее устройство, потом загружается новый блок, сортируется и т.д. Такой подход требует выполнение операции слияния отсортированных блоков. Для сортировки необходимо иметь однозначное трактование критерия сравнения данных.

К простым внутренним сортировкам относят методы, сложность которых пропорциональна (C\*), т.е. при сортировке массива из N элементов такие алгоритмы будут выполнять C\* действий, где C – некоторая константа.

Количество действий необходимых для упорядочения последовательности, конечно же зависит не только от длины последовательности, но и от её структуры.

Понятно, что на сортировку уже упорядоченной последовательности будет затрачено значительно меньше действий, чем в случае переменных данных. Как правило, сложность алгоритмов подсчитывают раздельно по количеству сравнений и по количеству перемещений данных в памяти, т.к. время выполнения этих операций различно. Точные значения найти практически невозможно, поэтому для оценки алгоритмов ограничиваются понятием пропорционально, т.е. не учитываются конкретные значения констант, входящих в итоговую формулу. Общую же эффективность алгоритма обычно оценивают в среднем, т.е. как среднее арифметическое от сложности алгоритма в “лучшем” и в “худшем” случаях.

При реализации алгоритмов будем использовать следующее описание:

**Const** N=...{количество элементов};

**Type** TInd=1..N;

Tel=Integer;

TArr1=**Array** [TInd] **of** Tel;

**Var** A:TArr1;

i,j:TInd;

В силу того, что часто приходится менять местами некоторые элементы, будем использовать процедуру обмена Swap.

**Procedure** Swap(**var** A1,A2:Tel);

**var** tmp:Tel;

**begin**

tmp:=A1;

A1:=A2;

A2:=tmp;

**end**;

Самый простой способ сортировки – это упорядочение данных по мере их поступления. В этом случае при вводе каждого нового значения можно опираться на тот факт, что все предыдущие элементы уже отсортированы. Тогда этот алгоритм можно описать так:

1)Включить 1-ый элемент;

2)Пока не закончится последовательность вводимых данных для каждого нового элемента выполнять следующие действия:

а)Начав с конца уже существующей последовательности все её элементы, которые больше чем вновь вводимый элемент сдвинуть на 1 элемент вправо;

б)Записываем новый элемент на освободившееся место. При этом можно полагать, что все элементы массива введены, а потом “представить”, что каждый очередной элемент был введен только что.

Для оценки времени работы посчитаем количество повторений для всех операций: K=(N-1)(3+(N-1)2).

Отсюда следует, что в случае, когда на вход подается уже упорядоченная последовательность, внутренний цикл не выполняется ни разу, а общее число сравнений и операций будет линейно зависеть от N. В худшем случае (упорядоченный в обратном порядке) алгоритм имеет сложность порядка .

Для сокращения количества сравнений введем в массив нулевой элемент (!!! TInd=0..N;), в который будем помещать поочередно каждый вставляемый элемент. В тех случаях, когда вставляемое значение оказывается меньше чем A[i], A[0] будет работать как ” барьер ”, не позволяющий индексу i выйти за нижнюю границу массива. Кроме этого A[0] заменяет дополнительную переменную key.

Эффективность алгоритма: Наилучшим будет случай упорядоченной последовательности. Тогда будет произведено N-1 сравнение и 0 пересылок. В худшем случае сравнений будет , а пересылок (N-1)(N+3).

Таким образом, этот алгоритм имеет сложность порядка по обоим параметрам.

**Сортировка бинарными вставками**

Сортировку простыми вставками можно немного улучшить путем поиска “подходящего места” в упорядоченной последовательности более экономным способом, который называется двоичный поиск. Схема алгоритма основана на методе дихотомии (деление отрезка пополам). Этот метод часто используется при реализации численных методов (поиск корня уравнения, минимума функции и т.д.). В нашем случае позволяет после каждого сравнения обрабатываемую последовательность сокращать в 2 раза.

Эффективность алгоритма Insert\_bin:

На каждом шаге выполняется не проверок, а N, что значительно лучше.

Следовательно в этом алгоритме будет N сравнений. При этом у количества пересылок по-прежнему сложность .

**Сортировка простым выбором**

Этот алгоритм был реализован при иллюстрации обработки одномерных массивов.

Эффективность алгоритма Select:

В лучшем случае – сравнений и 0 пересылок. В остальных случаях количество сравнений остается прежним, а количество пересылок будет 3(N-1). Т.е. по сравнениям алгоритм имеет сложность , а по пересылкам – N.

**Сортировка слиянием**

Идея алгоритма состоит в следующем: последовательность разбивается на 2 половины. Сортируется каждая половина отдельно, после чего они объединяются в одну. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер таких частей не станет равным единице.

**Сортировки обменом**

Существуют алгоритмы, основанные на обмене (перемене мест) 2-х соседних элементов: пузырьковая и шейкерная сортировки. Обе имеют сложность порядка , однако по скорости работы на любых входных данных и по сложности реализации они проигрывают другим простым сортировкам.

**Улучшенные сортировки**

В отличие от простых сортировок, имеющих сложность порядка , к улучшенным сортировкам относятся алгоритмы с общей сложностью N. Однако, необходимо отметить, что на небольших наборах данных (N<100) эффективность быстрых сортировок не столь очевидна. Выигрыш становится значительным при больших N.

**Сортировка Шелла**

Шелл предложил усовершенствование сортировки включением. Смысл ее состоит в раздельной сортировке методом Incert нескольких частей, на которые разбивается исходный массив. Эти разбиения помогают сократить количество пересылок.

На каждом шаге этого алгоритма нужно произвести следующие действия:

1)Вычленить все подпоследовательности, расстояние между элементами которых равно , где t – номер шага;

2)Каждую из этих подпоследовательностей сортируем процедурой Insert.

Главная проблема алгоритма: нахождение не убывающей последовательности расстояний.

Кнут предложил 2 “хороших” последовательности:

: 1,4,13

-1

Алгоритм Шелла имеет порядок и хотя это хуже чем N эта сортировка относится к улучшенным.

# Трансляторы языков программирования. Назначение основных блоков транслятора, схема их взаимодействия и используемые ими структуры данных.

Транслятор – обслуживающая программа, преобразующая исходную программу, предоставленную на входном языке программирования, в рабочую программу, представленную на объектном языке. В настоящее время трансляторы разделяются на три основные группы: ассемблеры, компиляторы и интерпретаторы.

Ассемблер - системная обслуживающая программа, которая преобразует символические конструкции в команды машинного языка. Специфической чертой ассемблеров является то, что они осуществляют дословную трансляцию одной символической команды в одну машинную (т.е. состоят из аналогов машинных команд). Т.о. язык ассемблера предназначен для облегчения восприятия системы команд компьютера и ускорения программирования в этой системе команд. Вместе с тем язык содержит множество дополнительных директив, облегчающих, в частности, управление ресурсами компьютера. Поэтому выразительность языка намного богаче, чем просто языка символического кодирования, что значительно повышает эффективность программирования.

Компилятор - это обслуживающая программа, выполняющая трансляцию на машинный язык программы, записанной на исходном языке программирования. Команды исходного языка значительно отличаются по организации и мощности от команд машинного языка. Существуют языки, в которых одна команда исходного языка транслируется в 7-10 машинных команд. Это языки, ориентированных на решение конкретных задач. Фортран ориентирован на научные расчеты, C предназначен для системного программирования, Пролог эффективно описывает задачи логического вывода, Лисп используется для рекурсивной обработки списков

Интерпретатор - программа, осуществляющая пооператорную трансляцию и выполнение исходной программы. В отличие от компилятора, интерпретатор не порождает на выходе программу на машинном языке. Распознав команду исходного языка, он тут же выполняет ее. Это делает процесс разработки и отладки программ более гибким. Интерпретатор можно достаточно легко адаптировать к любым машинным архитектурам, разработав его только один раз на широко распространенном языке программирования. Поэтому языки, типа Java Script, VB Script, получили широкое распространение. Недостатком интерпретаторов является низкая скорость выполнения программ (в 50-100 раз медленнее программ, написанных в машинных кодах)

Трансляторам присущ ряд общих черт, что упрощает процесс создания транслирующих программ. В состав любого транслятора входят три основных компонента:

      1. лексический анализатор

      2. синтаксический анализатор

      3. генератор кода машинных команд

На фазе лексического анализа исходный текст программы в виде цепочки несвязных друг с другом символов разбивается на единицы, называемые лексемами. Такими текстовыми единицами являются специальные слова, используемые в языке (например, **IF**, **DO**, **BEGIN** и др.), имена переменных, константы и знаки операций (например, \* или +).

      Далее эти слова рассматриваются как неделимые образования, а не как группы отдельных символов.

      После разбиения на лексемы следует фаза синтаксического анализа или грамматического разбора, на которой проверяется правильность следования лексем, т.е. грамматическая правильность исходной цепочки. Например, для оператора **IF**, имеющего вид :

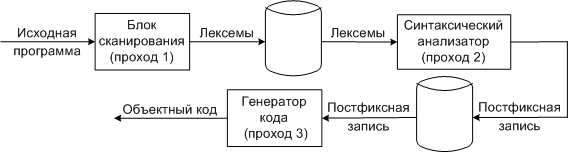
**IF** <выражение> **THEN** <предложение>

грамматический разбор состоит в том, чтобы убедится, что вслед за лексемой **IF** следует правильное выражение, за этим выражением следует лексема **THEN**, за которой в свою очередь следует правильное предложение, оканчивающееся знаком ';' .

      Последним выполняется процесс генерации кода, который использует результаты синтаксического анализа и формирует выход транслятора : программу на машинном коде, готовую к выполнению, либо специальную структуру данных, зависящих от языка и его назначения.

Хотя в состав любого транслятора входят все три описанных выше компонента, их взаимодействие может осуществляться разнообразными способами.

Ниже приведена схема взаимодействия вышеописанных блоков называемая трехпроходным транслятором, т.к. программа считывается трижды (исходный текст программы, файл лексем и файл в постфиксной форме считываются трижды) :



      Сначала блок сканирования считывает исходную программу и представляет ее в форме файла лексем. Синтаксический анализатор читает это файл и выдает новое представление программы в постфиксной форме. Наконец, этот файл считывается генератором кода, который создает объектный код программы.

**Преимущества** такой организации:

      1) относительная независимость каждой фазы трансляции. Связь между блоками осуществляется только через файлы, каждый блок независим, что придает транслятору гибкость – возможность изменения каждой фазы;

      2) минимальный объем памяти, что существенно, например, при использовании мини ЭВМ.

**Недостаток** : При такой организации скорость транслятора не может быть высокой, т.к. обычно операции, связанные с обращением к файлам выполняется медленно.

**Однопроходный транслятор**:

      Ниже приведена схема однопроходного транслятора :



      В данном случае синтаксический анализатор выступает в роли основной управляющей программы, вызывая блок сканирования и генератор кода, организованные в виде подпрограмм. Синтаксический анализатор обращается к блоку сканирования, получая от него лексему за лексемой из просматриваемой программы до тех пор, пока не построит новый элемент постфиксной записи, после чего он обращается к генератору кода, который создает объектный код для этой программы.

**Достоинства** :

      1) эффективность - программа считывается 1 раз;

      2) нет операций обращения к файлам;

**Недостаток**: неоптимальность создаваемой объектной программы. Например, при обработке фраз **GOTO** <метка>; могут встретиться трудности, когда метка ещё не встречалась в программе.

# Компиляторы. Основные функции компилятора.

Компиля́тор — [программа](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), выполняющая [трансляцию программы](http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B), составленной на исходном [языке высокого уровня](http://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в эквивалентную программу на [низкоуровневом языке](http://ru.wikipedia.org/wiki/%D0%9D%D0%B8%D0%B7%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), близком к [машинному коду](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) ([абсолютный код](http://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D0%BE%D0%BB%D1%8E%D1%82%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4), [объектный модуль](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C), иногда на [язык ассемблера](http://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0)). Входной информацией для компилятора ([исходный код](http://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4)) является описание алгоритма или программа на [объектно-ориентированном языке](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), а на выходе компилятора — эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

Процесс компиляции состоит из следующих этапов:

[Лексический анализ](http://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7). На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем.

[Синтаксический (грамматический) анализ](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7). Последовательность лексем преобразуется в дерево разбора.

[Семантический анализ](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7). Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки.

[Оптимизация](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80%D0%B0). Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах — например, над промежуточным кодом или над конечным машинным кодом.

[Генерация кода](http://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F_%D0%BA%D0%BE%D0%B4%D0%B0). Из промежуточного представления порождается код на целевом языке.

В конкретных реализациях компиляторов эти этапы могут быть разделены или, наоборот, совмещены в том или ином виде.

Основные функции компилятора:

1) проверка исходной цепочки символов на принадлежность к входному языку;

2) генерация выходной цепочки символов на языке машинных команд или ассемблере.

# Описание языков программирования с помощью БНФ.

Простая БНФ нотация позволяет описывать все достоверные конструкции языка программирования, используя следующие символы:

::= - имеет значение: по определению есть;

| - имеет значение: или (используется для перечисления альтернативных вариантов).

<..> - ограничивают имя синтаксической единицы.

Для каждого понятия языка должна существовать единственная метаформула , определяющая данное понятие.

Например: <переменная>::= A|B

<выражение>::=<переменная> |<переменная> +< переменная> |<переменная> - < переменная>

Тогда под описание выражения попадают любые из следующих записей: A; B; A+A; A-A; A+B; A-B; B+B; B-B; B-A; B+A.

Пример описания двоичного кода.

Под двоичным кодом будем понимать любую не пустую последовательность нулей и единиц, тогда

<двоичная цифра>::=0 | 1.

<двоичный код>::=<двоичная цифра> |<двоичный код><двоичная цифра>

Легко заметить, что при описании понятия двоичный код используется само это понятие. Подобного рода определения являются рекурсивными, чтобы при этом не получилось порочного круга правил часть формулы должна содержать, по крайней мере, 1 частное определение, в котором определяемый символ не используется.

Для задания синтаксических конструкций произвольной длины используется ещё 2 метасимвола {...}.Заключённые в фигурные скобки конструкции означают повторение 0 или более раз.

Тогда понятие двоичный код можно записать так: <двоичный код>::=<двоичная цифра>{двоичная цифра}.

<пусто>:: - (отсутствие каких-либо знаков и т.п.)

Тогда понятие двоичный код можно определить: <двоичный код>::=<пусто> |<двоичный код><двоичная цифра>

В квадратных скобках […] указывается не обязательный параметр.

# Описание языков программирования с помощью синтаксических диаграмм.

**Синтаксическая диаграмма (СД)** представляет собой графический способ описания языка. Элементами диаграммы являются прямоугольники и овалы с надписями, соединенные стрелками. В прямоугольниках записываются названия метапеременных, в овалах – основные символы. В диаграмме есть одна входная стрелка (не исходящая из какого-либо элемента) и одна выходная (не входящая в какой-либо элемент). Входная и выходная стрелки означают соответственно начало и конец синтаксического определения. Приведем диаграмму для понятия *число 5*. Само определяемое понятие будем, как и в случае БНФ, заключать в угловые скобки и отделять от синтаксического определения знаком ::= .

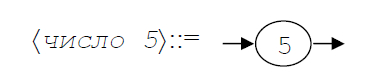
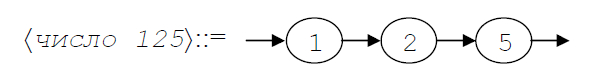
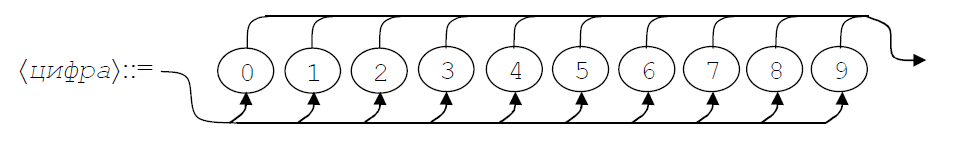


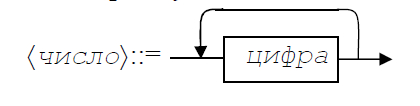
Диаграмма для числа 125 будет содержать три овала, по одному на каждую цифру.



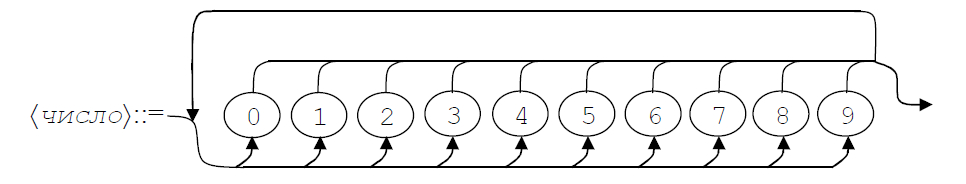
Покажем на примере понятия цифра, как с помощью диаграммы изображаются альтернативы.



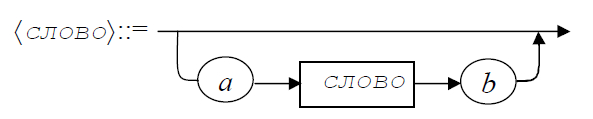
Опишем теперь понятие «число» с помощью СД. В ней будет использовано вспомогательное понятие «цифра», изображенное в виде прямоугольника.



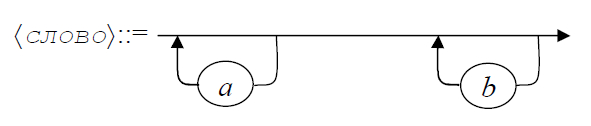
Подставим вместо прямоугольника соответствующую диаграмму и получим СД для понятия «число», не содержащую вспомогательных понятий (метапеременных).



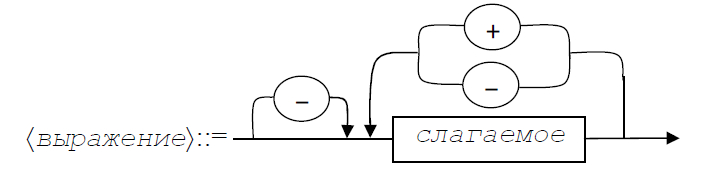
Существуют языки, которые невозможно описать, не используя в диаграмме вспомогательных понятий. Рассмотрим, к примеру, язык, цепочки которого состоят из букв *a* и *b*, причем сначала в цепочке следуют n букв *a* (n≥0), а за ними столько же букв *b*: *ab*, *aabb*, *aaabbb* и т.д. Пустая цепочка тоже принадлежит данному языку (случай n=0). Определим понятие «слово» для данного языка с помощью СД.

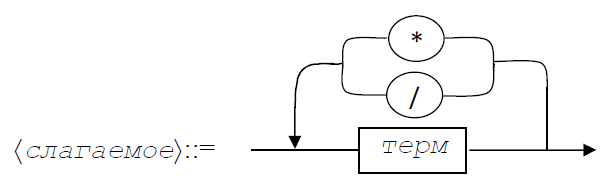


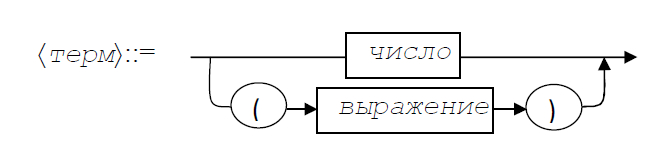
Попытаемся устранить рекурсию – построим СД, в которой нет вспомогательного понятия (метапеременной) «слово»:



Однако полученная диаграмма описывает другой язык – в его словах количество букв *a* может отличаться от количества букв *b*. Опишем с помощью СД понятие «выражение».







Как металингвистические формулы, так и синтаксические диаграммы используются для описания синтаксиса языков программирования (см., например, [2]). Синтаксические диаграммы обеспечивают большую наглядность. Преимущество БНФ в том, что они могут использоваться в системах автоматизированной обработки языков.

# Восходящие методы грамматического разбора.

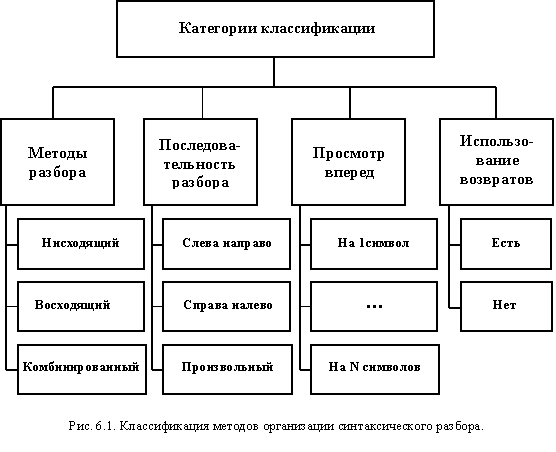
***Термины и введение.***

* **Терминал (терминальный символ)** — объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»). В формальных языках, используемых на компьютере, в качестве терминалов обычно берут все или часть стандартных символов ASCII — латинские буквы, цифры и спец. символы.
* **Нетерминал (нетерминальный символ)** — объект, обозначающий какую-либо *сущность* языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

Если попытаться формализовать задачу на уровне элементарного метаязыка, то она будет ставиться следующим образом. Дан язык **L(G)** с грамматикой **G**, в которой **S** - начальный нетерминал. Построить дерево разбора входной цепочки:

**a = a1a2a3...an.**

Естественно, что существует огромное количество путей решения данной задачи, и целью разработчика распознавателя является выделение приемлемых вариантов его реализации. Для того, чтобы понять, что, и каким образом, влияет на принципы функционирования распознавателя, а следовательно, и на организацию разбора, рассмотрим некоторые возможные варианты. Общая классификация рассматриваемых вариантов построения распознавателя представлена на рис. 6.1.



На самом верхнем уровне выделяются:

* методы разбора;
* последовательность разбора;
* использование просмотра вперед;
* использование возвратов.

### Методы разбора

Выделяются два основных метода синтаксического разбора:

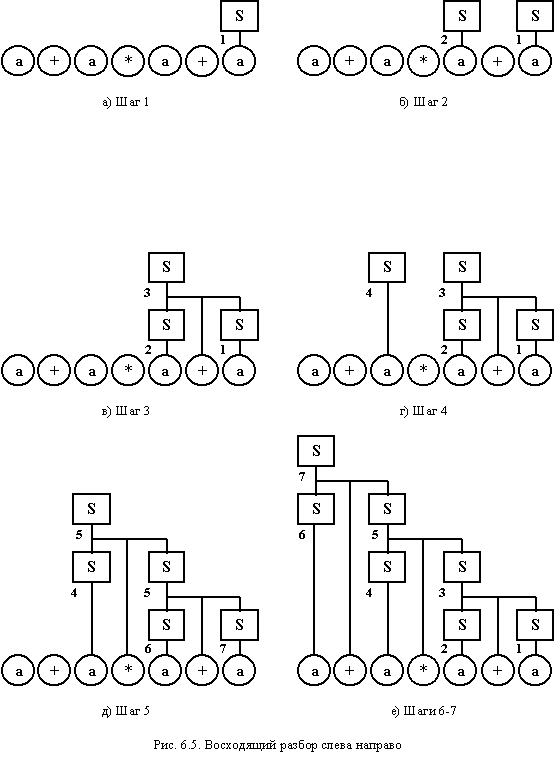
* нисходящий разбор;
* восходящий разбор.

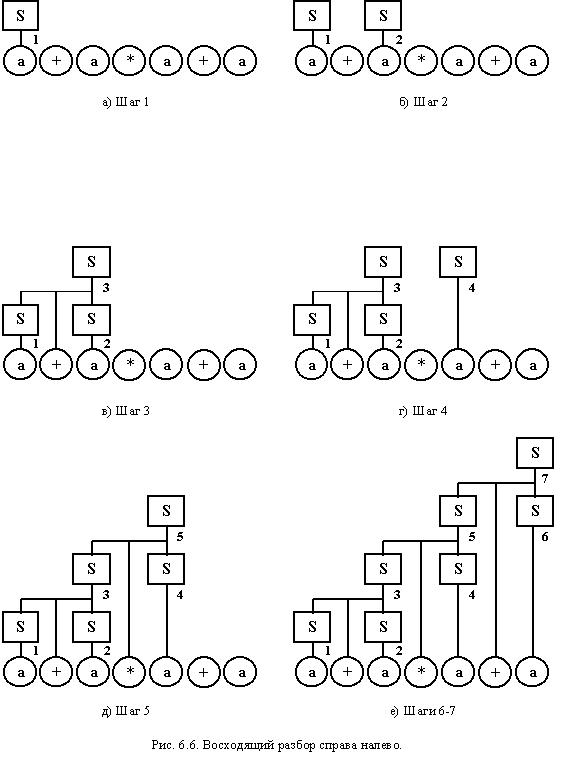
Кроме этого можно использовать комбинированный разбор, сочетающий особенности двух предыдущих.

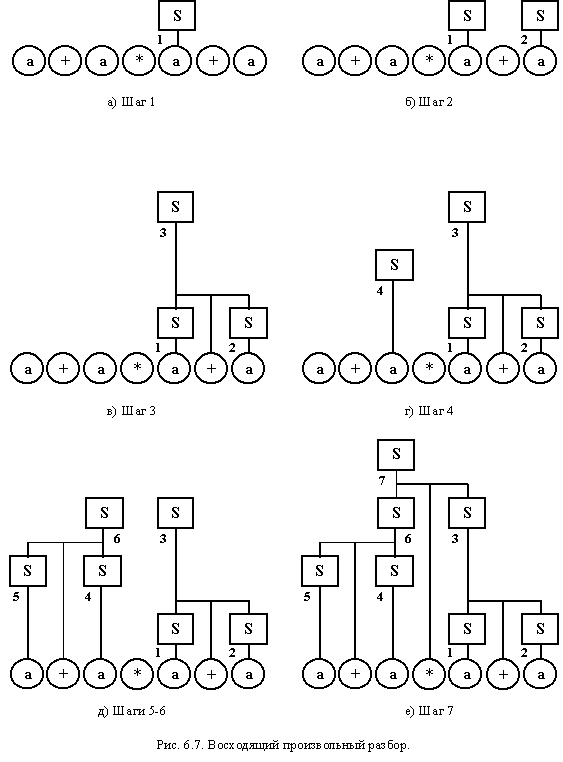
Нисходящие и восходящие подходы широко используются в различных областях человеческой деятельности, особенно в тех из них, которые связаны с анализом и синтезом искусственных систем. В частности, можно отметить методы разработки программного обеспечения сверху вниз (нисходящий) и снизу вверх (восходящий).

При **восходящем разборе** дерево начинает строиться от терминальных листьев путем подстановки правил, применимых к входной цепочке, в общем случае, в произвольном порядке. На следующем шаге новые узлы полученных поддеревьев используются как листья во вновь применяемых правилах. Процесс построения дерева разбора завершается, когда все символы входной цепочки будут являться листьями дерева, корнем которого окажется начальный нетерминал. Если, в результате полного перебора всех возможных правил, мы не сможем построить требуемое дерево разбора, то рассматриваемая входная цепочка не принадлежит данному языку.

Восходящий разбор также непосредственно связан с любым возможным выводом цепочки из начального нетерминала. Однако, эта связь, по сравнению с нисходящим разбором, реализуется с точностью до "наоборот". На рис. 6.5, 6.6, 6.7 приведены примеры построения деревьев разбора для грамматики **G6** и процессов порождения цепочек, представленных выражениями (6.1). Из рисунков видно, что шаги порождения дерева соответствуют движению по представленным цепочкам вывода справа налево.







# Нисходящие методы грамматического разбора.

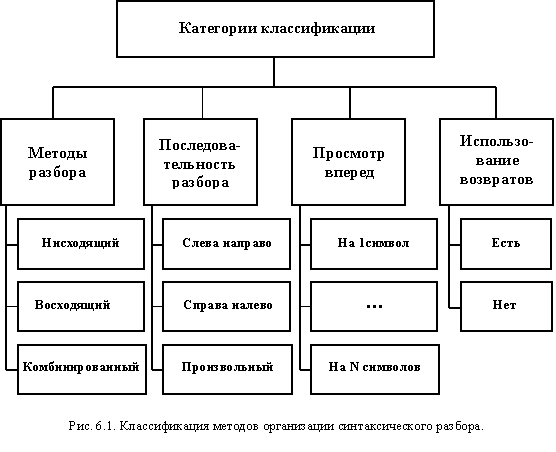
***Термины и введение.***

* **Терминал (терминальный символ)** — объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»). В формальных языках, используемых на компьютере, в качестве терминалов обычно берут все или часть стандартных символов ASCII — латинские буквы, цифры и спец. символы.
* **Нетерминал (нетерминальный символ)** — объект, обозначающий какую-либо *сущность* языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

Если попытаться формализовать задачу на уровне элементарного метаязыка, то она будет ставиться следующим образом. Дан язык **L(G)** с грамматикой **G**, в которой **S** - начальный нетерминал. Построить дерево разбора входной цепочки:

**a = a1a2a3...an.**

Естественно, что существует огромное количество путей решения данной задачи, и целью разработчика распознавателя является выделение приемлемых вариантов его реализации. Для того, чтобы понять, что, и каким образом, влияет на принципы функционирования распознавателя, а следовательно, и на организацию разбора, рассмотрим некоторые возможные варианты. Общая классификация рассматриваемых вариантов построения распознавателя представлена на рис. 6.1.



На самом верхнем уровне выделяются:

* методы разбора;
* последовательность разбора;
* использование просмотра вперед;
* использование возвратов.

### Методы разбора

Выделяются два основных метода синтаксического разбора:

* нисходящий разбор;
* восходящий разбор.

Кроме этого можно использовать комбинированный разбор, сочетающий особенности двух предыдущих.

Нисходящие и восходящие подходы широко используются в различных областях человеческой деятельности, особенно в тех из них, которые связаны с анализом и синтезом искусственных систем. В частности, можно отметить методы разработки программного обеспечения сверху вниз (нисходящий) и снизу вверх (восходящий).

**Нисходящий разбор** заключается в построении дерева разбора, начиная от корневой вершины. Разбор заключается в заполнении промежутка между начальным нетерминалом и символами входной цепочки правилами, выводимыми из начального нетерминала. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой терминалов и нетерминалов одного из альтернативных правил, порождаемых начальным нетерминалом. Подставляемое правило в общем случае выбирается произвольно. Вместо новых нетерминальных вершин осуществляется подстановка выводимых из них правил. Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил. В последнем случае входная цепочка отвергается. Построение дерева разбора подтверждает принадлежность входной цепочки данному языку. При этом, в общем случае, для одной и той же входной цепочки может быть построено несколько деревьев разбора. Это говорит о том, что грамматика данного языка является недетерминированной.

Эти рассуждения иллюстрируются следующим примером. Пусть будет дана грамматика **G**:

**G6** **= ({S}, {a, +, \*}, P, S)**,

Где **P** определяется как:

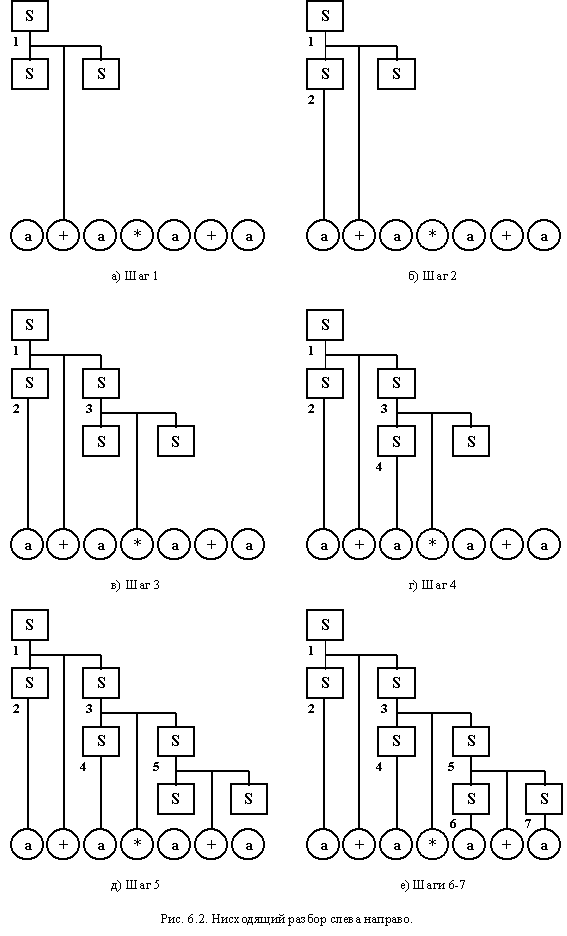
1. S ****a
2. S ****S + S
3. S ****S \* S

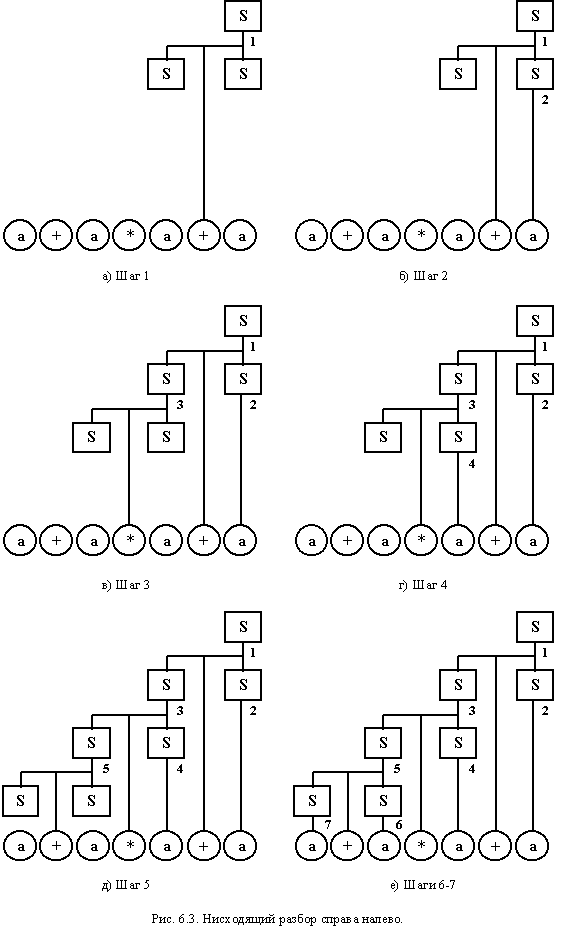
Цепочки, порождаемые данной грамматикой можно интерпретировать как выражения, состоящие из операндов **"a"**, а также операций **"+"** и **"\*"**. Недетерминированность грамматики позволяет порождать одну и ту же терминальную цепочки с использованием различных выводов. Например, выражение**"a+a\*a+a"** можно получить следующими способами:

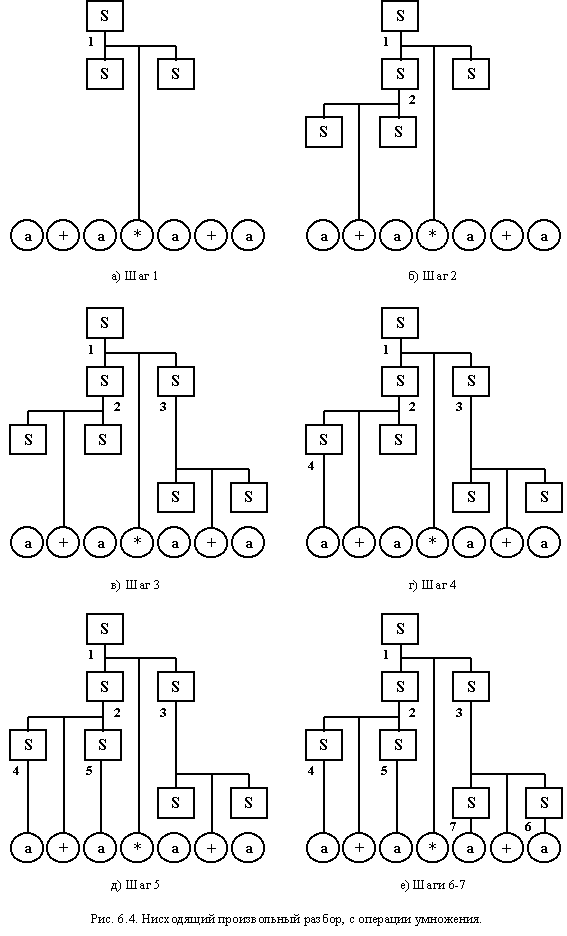
1. S **** S+S **** a+S **** a+S\*S **** a+ a\*S **** a+a\*S+S **** a+a\*a+S **** a+a\*a+a
2. S **** S+S **** S+a **** S\*S+a **** S\*a+a **** S+S\*a+a **** S+a\*a+a **** a+a\*a+a (6.1)
3. S **** S\*S **** S+S\*S **** S+S\*S+S ****

a+S\*S+S **** a+a\*S+S **** a+a\*S+a **** a+a\*a+a

И так далее. В этом пример число вариантов одной и той же произвольной цепочки вывода настолько велико, что не имеет и смысла говорить о практическом применении данной грамматики. Но в данном случае она позволяет показать, каким образом могут порождаться различные деревья при нисходящем разборе. Пошаговое построение различных деревьев показано на рис. 6.2, 6.3, 6.4. Можно отметить, что процесс построения дерева совпадает с последовательностью шагов вывода входной цепочки.





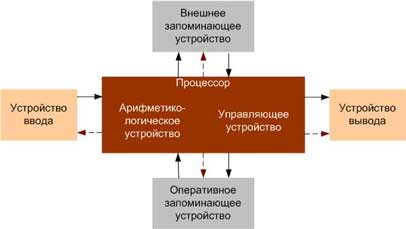


# Основы архитектуры ЭВМ. Номенклатура сегментных регистров.

**Архитектура вычислительной машины** (Архитектура ВМ, [англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Computer architecture) — концептуальная структура вычислительной машины, определяющая проведение обработки информации и включающая методы информации в данные и принципы взаимодействия технических средств и программного обеспечения.

Архитектура ЭВМ включает в себя как структуру, отражающую состав ПК, так и программно – математическое обеспечение. Структура ЭВМ - совокупность элементов и связей между ними. Основным принципом построения всех современных ЭВМ является программное управление.  
  
Основы учения об архитектуре вычислительных машин были заложены Джон фон Нейманом. Совокупность этих принципов породила классическую (фон-неймановскую) архитектуру ЭВМ.

Фон Нейман не только выдвинул основополагающие принципы логического устройства ЭВМ, но и предложил ее структуру, представленную на рисунке. 



Положения фон Неймана:

* Компьютер состоит из нескольких основных устройств (арифметико-логическое устройство, управляющее устройство, память, внешняя память, устройства ввода и вывода)
* Арифметико-логическое устройство – выполняет логические и арифметические действия, необходимые для переработки информации, хранящейся в памяти
* Управляющее устройство – обеспечивает управление и контроль всех устройств компьютера (управляющие сигналы указаны пунктирными стрелками)
* Данные, которые хранятся в запоминающем устройстве, представлены в двоичной форме
* Программа, которая задает работу компьютера, и данные хранятся  в одном и том же запоминающем устройстве
* Для ввода и вывода информации используются устройства ввода и вывода.

Один из важнейших принципов – принцип хранимой программы – требует, чтобы программа закладывалась в память машины так же, как в нее закладывается исходная информация.

Арифметико-логическое устройство и устройство управления в современных компьютерах образуют процессор ЭВМ. Процессор, который состоит из одной или нескольких больших интегральных схем называется микропроцессором или микропроцессорным комплектом.

Процессор – функциональная часть ЭВМ, выполняющая основные операции по обработке данных и управлению работой других блоков. Процессор является преобразователем информации, поступающей из памяти и внешних устройств.

Запоминающие устройства обеспечивают хранение исходных и промежуточных данных, результатов вычислений, а также программ. Они включают: оперативные (ОЗУ), сверхоперативные СОЗУ), постоянные (ПЗУ) и внешние (ВЗУ) запоминающие устройства.

Оперативные ЗУ хранят информацию, с которой компьютер работает непосредственно в данное время (резидентная часть операционной системы, прикладная программа, обрабатываемые данные). В СОЗУ хранится наиболее часто используемые процессором данные. Только та информация, которая хранится в СОЗУ и ОЗУ, непосредственно доступна процессору.

Внешние запоминающие устройства (накопители на магнитных дисках, например, жесткий диск или винчестер) с емкостью намного больше, чем ОЗУ, но с существенно более медленным доступом, используются для длительного хранения больших объемов информации. Например, операционная система (ОС) хранится на жестком диске, но при запуске компьютера резидентная часть ОС загружается в ОЗУ и находится там до завершения сеанса работы ПК.

ПЗУ (постоянные запоминающие устройства) и ППЗУ (перепрограммируемые постоянные запоминающие устройства) предназначены для постоянного хранения информации, которая записывается туда при ее изготовлении, например, ППЗУ для BIOS.

В качестве устройства ввода информации служит, например, клавиатура. В качестве устройства вывода – дисплей, принтер и т.д.  
В построенной по схеме фон Неймана ЭВМ происходит последовательное считывание команд из памяти и их выполнение. Номер (адрес) очередной ячейки памяти, из которой будет извлечена следующая команда программы, указывается специальным устройством – счетчиком команд в устройстве управления.

# Варианты задания исполнительного адреса в архитектурах Intel или др.

Тип архитектуры, как правило, определяется фирмой производителем оборудования. Все крупнейшие фирмы, производящие электронное оборудование для IBM-PC-совместимых компьютеров и выпускающие свои типы центральных процессоров вносят изменения в базовую архитектуру процессоров серии Intel x86 или разрабатывают свою. С типом архитектуры тесно связан набор поддерживаемых команд или инструкций, и их расширений. Эти два параметра, в основном, определяют качественный уровень возможностей персонального компьютера и в большой степени уровень его производительности.

**Адресный код** - это информация об адресе операнда, содержащаяся в команде.

**Исполнительный адрес** - это номер ячейки  памяти,  к  которой производится фактическое обращение.

В современных ЭВМ адресный код,  как правило, не совпадает с исполнительным адресом.

Выбор способов адресации, формирования исполнительного адреса и преобразования адресов является одним из важнейших  вопросов разработки ЭВМ.  Рассмотрим способы  адресации,  используемые  в современных ЭВМ:

**1) Подразумеваемый операнд.**

В команде может не содержаться явных указаний об операнде; в этом случае операнд подразумевается и фактически задается кодом операции команды.

**2) Подразумеваемый адрес.**

В команде может не содержаться явных указаний об адресе участвующего в операции операнда или адресе, по которому должен быть размещен результат операции, но этот адрес подразумевается.

**3) Непосредственная адресация.**

В команде  содержится  не  адрес операнда,  а непосредственно сам операнд.  При непосредственной адресации не требуется обращения к памяти для выборки операнда и ячейки памяти для его хранения. Это способствует уменьшению времени выполнения программы и занимаемого ею объема памяти. Непосредственная адресация удобна для хранения различного рода констант.

**4) Прямая адресация.**

В адресной части команды может быть непосредственно указан исполнительный адрес.

**5) Относительная (базовая) адресация.**

При этом  способе адресации исполнительный адрес определяется как сумма адресного кода команды и базового адреса,  как правило, хранящегося в специальном регистре - регистре базы.

Относительная адресация позволяет при меньшей длине адресного кода команды обеспечить доступ  к  любой  ячейке  памяти.  Для этого число  разрядов  в  базовом регистре выбирают таким,  чтобы можно было адресовать любую ячейку оперативной памяти, а адресный код команды используют для представления лишь сравнительно короткого "смещения".  Смещение определяет положение операнда  относительно начала массива, задаваемого базовым адресом.

**6) Укороченная адресация.**

Для уменьшения длины  кода  команды  часто  применяется  так называемая укороченная адресация. Суть ее сводится к тому, что  в команде задаются только младшие разряды адресов, а старшие разряды при этом подразумеваются нулевыми.  Такая адресация  позволяет использовать только  небольшую часть фиксированных ячеек в начале всей адресуемой области памяти,  и поэтому применяется лишь  совместно с другими способами адресации.

Регистровая адресация  является частным случаем укороченной, когда в качестве фиксированных ячеек с короткими адресами используются регистры (ячейки сверхоперативной или местной памяти) процессора.  Например, если таких регистров 16, то для адреса достаточно  четырех двоичных разрядов.  Регистровая адресация наряду с сокращением длины адресов операндов позволяет увеличить  скорость выполнения операций, так как уменьшается число обрашений к оперативной памяти.

**7) Косвенная адресация.**

Адресный код команды в этом случае указывает адрес ячейки памяти, в которой находится адрес операнда или команды. Косвенная адресация широко используется в малых и микроЭВМ,  имеющих короткое машинное  слово,  для преодоления ограничений короткого формата команды (совместно используются регистровая и косвенная адресация).

**8) Адресация слов переменной длины.**

Эффективность  вычислительных  систем,  предназначенных  для обработки данных,  повышается, если имеется возможность выполнять операции со словами переменной длины.  В  этом  случае  в  машине может быть предусмотрена адресация слов переменной длины, которая обычно  реализуется  путем  указания в команде местоположения в памяти начала слова и его длины.

**9) Стековая адресация.**

Стековая память,  реализующая безадресное задание операндов, особенно широко используется в микропроцессорах и миниЭВМ.

Стек представляет собой группу последовательно пронумерованных регистров  или ячеек памяти,  снабженных указателем стека,  в котором  автоматически  при  записи и считывании  устанавливается номер (адрес) последней занятой ячейки стека (вершины стека). При операции записи заносимое в стек слово помещается в следующую  по порядку свободную ячейку стека,  а при считывании из стека извлекается последнее поступившее в него слово.

**10) Автоинкрементная и автодекрементная адресации.**

Поскольку регистровая  косвенная адресация требует предварительной загрузки регистра косвенным адресом из оперативной памяти, что связано с потерей времени,  такой тип адресации особенно эффективен при обработке массива данных, если  имеется  механизм автоматического приращения  или  уменьшения  содержимого регистра при каждом обращении к  нему.  Такой  механизм  называется  соответственно автоинкрементной и автодекрементной адресацией. В этом случае достаточно один раз загрузить в регистр адрес первого  обрабатываемого элемента  массива,  а  затем при каждом обращении к регистру в нем  будет  формироваться  адрес  следующего  элемента массива.

При автоинкрементной адресации сначала  содержимое  регистра используется как  адрес  операнда,  а  затем получает приращение, равное числу байт в элементе массива. При автодекрементной адресации сначала содержимое указанного в команде регистра уменьшается на число байт в элементе массива,  а  затем  используется  как адрес операнда.

**11) Индексация.**

Для реализуемых на ЭВМ методов решения математических  задач и обработки  данных  характерна  цикличность  вычислительных процессов, когда одни и те же процедуры выполняются  над  различными операндами, упорядоченно расположенными в памяти.  Поскольку операнды, обрабатываемые при повторениях цикла, имеют разные адреса, без использования  индексации требовалось бы для каждого повторения составлять свою последовательность команд,  отличающихся  адресными частями.

Программирование циклов существенно упрощается,  если  после каждого выполнения  цикла  обеспечено  автоматическое изменение в соответствующих командах их адресных частей согласно расположению в памяти обрабатываемых операндов. Такой процесс называется модификацией команд,  и основан на возможности выполнения над кодами команд арифметических и логических операций.

Управление вычислительным циклом должно обеспечивать  повторение цикла нужное число раз, а затем выход из него.

Автоматическая модификация команд и управление  вычислительными циклами в современных ЭВМ обеспечиваются механизмом индексации. Это понятие включает в себя специальный  способ  кодирования команд, командные  и  аппаратурные  средства задания и выполнения модификации команд и управления вычислительными циклами. Упомянутые средства часто называют индексной арифметикой.

Для выполнения индексации в машину вводятся  так  называемые индексные регистры.  Исполнительный адрес при индексации формируется  путем сложения адресного кода команды (смещения) с содержимым индексного регистра (индексом), а при наличии базирования – и с базовым адресом.

Для управления индексацией  используются  команды,  задающие операции над  содержимым  индексных регистров - команды индексной арифметики. Можно отметить основные виды индексных операций:

- засылка в  соответствующий  индексный  регистр  начального значения индекса;

- изменение индекса;

- проверка окончания циклических вычислений.

# Особенности языков ассемблерного типа.

Команды языка ассемблера один к одному соответствуют командам [процессора](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80). Фактически, они и представляют собой более удобную для человека символьную форму записи — *мнемокоды* — команд и их [аргументов](http://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). При этом одной команде языка ассемблера может соответствовать несколько вариантов команд процессора.

### Достоинства

* Язык ассемблера позволяет писать самый быстрый и компактный код, какой вообще возможен для данного процессора.
* Если код программы достаточно большой, — данные, которыми он оперирует, не помещаются целиком в [регистрах процессора](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0), то есть частично или полностью находятся в [оперативной памяти](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), — то искусный программист, как правило, способен значительно оптимизировать программу по сравнению с транслятором с языка высокого уровня по одному или нескольким параметрам и создать код близкий к [оптимальному по Парето](http://ru.wikipedia.org/wiki/%D0%AD%D1%84%D1%84%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%BF%D0%BE_%D0%9F%D0%B0%D1%80%D0%B5%D1%82%D0%BE) (как правило, быстродействие программы достигается за счет удлинения кода и наоборот):
  + скорость работы — за счёт оптимизации вычислительного алгоритма и/или более рационального обращения к[оперативной памяти (ОП)](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) (например, если все исходные данные хранятся в регистрах процессора, то можно исключить излишние обращения к ОП), перераспределения данных, табличного вычисления функций;
  + объём кода (в том числе за счёт эффективного использования промежуточных результатов). Сокращение объёма кода также нередко повышает скорость выполнения программы.
* Обеспечение максимального использования специфических возможностей конкретной платформы, что также позволяет создавать более эффективные программы, в том числе менее ресурсоёмкие.
* При программировании на языке ассемблера возможен непосредственный доступ к аппаратуре, и, в частности,[портам ввода-вывода](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D1%82_%D0%B2%D0%B2%D0%BE%D0%B4%D0%B0-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4%D0%B0), регистрам процессора и др. Во многих операционных системах прямое обращение из прикладных программ для записи в регистры периферийного оборудования блокировано для надёжности работы системы и исключения «зависаний».
* Язык ассемблера часто применяется для создания [драйверов](http://ru.wikipedia.org/wiki/%D0%94%D1%80%D0%B0%D0%B9%D0%B2%D0%B5%D1%80) оборудования и [ядра операционной системы](http://ru.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) (или машиннозависимых подсистем ядра ОС), тогда, когда важно временно́е согласование работы периферийных устройств с центральным процессором.
* Язык ассемблера используется для создания «[прошивок](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%88%D0%B8%D0%B2%D0%BA%D0%B0)» [BIOS](http://ru.wikipedia.org/wiki/BIOS).
* С помощью языка ассемблера часто создаются машиннозависимые подпрограммы [компиляторов](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) и [интерпретаторы](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80)[языков высокого уровня](http://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%B2%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D0%B3%D0%BE_%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D1%8F), а также реализуется совместимость [платформ](http://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BB%D0%B0%D1%82%D1%84%D0%BE%D1%80%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%B0).
* С помощью программы [дизассемблера](http://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B7%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80) можно понять алгоритмы работы исследуемой программы при отсутствии листинга на высокоуровневом языке, изучая только машинные коды, но в сложных нетривиальных программах это очень и очень трудоёмко.

### Недостатки

* В силу машинной ориентации («низкого» уровня) языка ассемблера человеку сложнее читать и понимать программу на нём по сравнению с языками программирования высокого уровня; программа состоит из слишком «мелких» элементов — машинных команд, соответственно, усложняются программирование и отладка, растут трудоёмкость и вероятность внесения ошибок.
* Требуется повышенная квалификация программиста для получения качественного кода: код, написанный средним программистом на языке ассемблера, обыкновенно оказывается не лучше или даже хуже кода, порождаемого[оптимизирующим компилятором](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B8%D0%B9_%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) для сравнимых программ, написанных на языке высокого уровня[[5]](http://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0#cite_note-5).
* Программа на языке высокого уровня может быть перекомпилирована с автоматической оптимизацией под особенности новой целевой платформы, программа же на языке ассемблера на новой платформе может потерять своё преимущество в скорости без ручного переписывания кода.
* Как правило, меньшее количество доступных [библиотек](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) по сравнению с современными индустриальными языками программирования.
* Отсутствует переносимость программ на компьютеры с другой архитектурой и системой команд.

# Макросредства языка Ассемблер (на примере архитектуры Intel или др.)

Общая идея макросредств заключается в том, что включением в исходный текст программы предложений специального языка макросредств (макроязыка) мы в какой-то степени управляем процессом трансляции программы. Макроязык позволяет выполнять или не выполнять трансляцию отдельных участков программы в зависимости от некоторого нами же определяемого условия (условная трансляция); осуществлять размножение участка исходного текста программы, в том числе, с модификацией каждого повторения (блоки повторения); включать в программу написанные отдельно фрагменты с настройкой их текста в соответствии с заданными параметрами (макрокоманды). Объекты, создаваемые с помощью директив макроязыка, обычно называют макросами.

К простейшим макросредствам языка ассемблера можно отнести псевдооператоры equ и "=" (равно).

Синтаксис псевдооператора equ:

имя\_идентификатора equ строка или числовое\_выражение

Синтаксис псевдооператора “=”:

имя\_идентификатора = числовое\_выражение

Несмотря на внешнее и функциональное сходство псевдооператоры equ и “=” отличаются следующим:

из синтаксического описания видно, что с помощью equ идентификатору можно ставить в соответствие как числовые выражения, так и текстовые строки, а псевдооператор “=” может использоваться только с числовыми выражениями;

идентификаторы, определенные с помощью “=”, можно переопределять в исходном тексте программы, а определенные с использованием equ— нельзя.

Синтаксис *макроопределения* следующий:

имя\_макрокоманды macro список\_формальных\_аргументов

тело макроопределения

endm

С помощью макросредств ассемблера можно не только частично изменять входящие в макроопределение строки, но и модифицировать сам набор этих строк и даже порядок их следования. Сделать это можно с помощью набора макродиректив (далее — просто директив). Их можно разделить на две группы:

* директивы повторения **[WHILE](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm" \l "while)**, [**REPT**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#while), [**IRP**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#irp) и [**IRPC**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#irpc).   
  Директивы этой группы предназначены для создания макросов, содержащих несколько идущих подряд одинаковых последовательностей строк. При этом возможна частичная модификация этих строк.
* директивы управления процессом генерации макрорасширения [**EXITM**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#exitm) и [**GOTO**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#goto).   
  Они предназначены для управления процессом формирования макрорасширения из набора строк соответствующего макроопределения. С помощью этих директив можно как исключать отдельные строки из макрорасширения, так и вовсе прекращать процесс генерации. Директивы**EXITM** и **GOTO** обычно используются вместе с условными директивами компиляции, поэтому они будут рассмотрены вместе с ними.

#### Директивы WHILE и REPT

Директивы **WHILE** и **REPT** применяют для повторения определенное количество раз некоторой последовательности строк.   
Эти директивы имеют следующий синтаксис:

WHILE константное\_выражение

последовательность\_строк

ENDM

REPT константное\_выражение

последовательность строк

ENDM

Обратите внимание, что последовательность повторяемых строк в обеих директивах ограничена директивой **ENDM**.

При использовании директивы **WHILE** макрогенератор транслятора будет повторять последовательность\_строк до тех пор, пока значение**константное\_выражение** не станет равно нулю. Это значение вычисляется каждый раз перед очередной итерацией цикла повторения (то есть значение **константное\_выражение** должно подвергаться изменению внутри **последовательность\_строк** в процессе макрогенерации).

Директива **REPT**, подобно директиве **WHILE**, повторяет **последовательность\_строк** столько раз, сколько это определено значением**константное\_выражение**. Отличие этой директивы от **WHILE** состоит в том, что она автоматически уменьшает на единицу значение**константное\_выражение** после каждой итерации.

Следующие две директивы, [**IRP**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#irp) и [**IRPC**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#irpc), делают этот процесс более гибким, позволяя модифицировать на каждой итерации некоторые элементы в последовательность\_строк.

#### Директива IRP

Директива **IRP** имеет следующий синтаксис:

IRP формальный\_аргумент,<строка\_символов\_1,...,строка\_символов\_N>

последовательность\_строк

ENDM

Действие данной директивы заключается в том, что она повторяет последовательность\_строк N раз, то есть столько раз, сколько **строк\_символов**заключено в угловые скобки во втором операнде директивы **IRP**. Но это еще не все.   
Повторение **последовательности\_строк** сопровождается заменой в ней **формального\_аргумента** строкой символов из второго операнда.   
Так, при первой генерации **последовательности\_строк** **формальный\_аргумент** в них заменяется на **строка\_символов\_1**.   
Если есть **строка\_символов\_2**, то это приводит к генерации второй копии **последовательности\_строк**, в которой **формальный\_аргумент**заменяется на **строка\_символов\_2**. Эти действия продолжаются до **строка\_символов\_N** включительно.

иректива **IRPC** имеет следующий синтаксис:

IRPC формальный\_аргумент,строка\_символов

последовательность строк

ENDM

Действие данной директивы подобно **IRP**, но отличается тем, что она на каждой очередной итерации заменяет **формальный\_аргумент** очередным символом из **строка\_символов**.   
Понятно, что количество повторений **последовательность\_строк** будет определяться количеством символов в **строка\_символов**.

Последний тип макросредств — директивы условной компиляции.   
Существует два типа этих директив:

* [директивы компиляции по условию](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm" \l "ДирективыКомпиляцииПоУсловию) позволяют проанализировать определенные условия в ходе генерации макрорасширения и, при необходимости, изменить этот процесс;
* [директивы генерации ошибок по условию](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#ДирективыГенерацииОшибокПоУсловию) также контролируют ход генерации макрорасширения с целью генерации или обнаружения определенных ситуаций, которые могут интерпретироваться как ошибочные.

С этими директивами применяются директивы управления процессом генерации макрорасширений **EXITM** и **GOTO**.

Директива **EXITM** не имеет операндов, и ее действие заключается в том, что она немедленно прекращает процесс генерации макрорасширения, начиная с того места, где она встретилась в макроопределении.

Директива **GOTO** **имя\_метки** переводит процесс генерации макроопределения в другое место, прекращая тем самым последовательное разворачивание строк макроопределения. Метка, на которую передается управление, имеет специальный формат:

:имя\_метки

### Директивы компиляции по условию

Данные директивы предназначены для организации выборочной трансляции фрагментов программного кода. Такая выборочная компиляция означает, что в макрорасширение включаются не все строки макроопределения, а только те, которые удовлетворяют определенным условиям. То, какие конкретно условия должны быть проверены, определяется типом условной директивы.

Введение в язык ассемблера этих директив значительно повышает его мощь.   
Всего имеется 10 типов условных директив компиляции. Их логично попарно объединить в четыре группы:

1. Директивы **[IF](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm" \l "if)** и [**IFE**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#if) — условная трансляция по результату вычисления логического выражения.
2. Директивы [**IFDEF**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#ifdef) и [**IFNDEF**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#ifdef) — условная трансляция по факту определения символического имени.
3. Директивы [**IFB**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#ifb) и [**IFNB**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#ifb) — условная трансляция по факту определения фактического аргумента при вызове макрокоманды.
4. Директивы [**IFIDN**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#ifidn), [**IFIDNI**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#ifidn), [**IFDIF**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#ifidn) и [**IFDIFI**](http://cs.mipt.ru/docs/comp/rus/programming/languages/asm/yurov/guide/text/macro.htm#ifidn) — условная трансляция по результату сравнения строк символов.

# Операционные системы, их назначение и функции. Виды операционных систем, их характеристики.

1)ОС как виртуальная машина.

При разработке ОС широко применяется абстрагирование, которое позволяет сконцентрироваться на взаимодействии высоко уровневых компонентах системы, игнорируя детали их реализации. В этом смысле ОС представляет собой интерфейс между пользователем и компьютером. Архитектура большинства компьютеров на уровне машинных команд очень не удобна для программирования.

2)ОС как менеджер ресурсов.

ОС предназначена для управления всеми частями архитектуры компьютера. ОС осуществляет упорядоченное и контролируемое распределение процессоров, памяти и других ресурсов между различными программами.

3)ОС как защитник пользователей и программ.

При совместной работе нескольких пользователей возникает проблема организации их безопасной деятельности, т.е. необходимо обеспечить сохранность информации на диске, нельзя разрешать программам одних пользователей произвольно вмешиваться в работу программ других пользователей, и т.д.

4)ОС как постоянно функционирующее ядро.

ОС – программа, постоянно работающая на компьютере и взаимодействующая со всеми прикладными программами. Во многих современных ОС’ях постоянно работает лишь часть ОС, которую принято называть ее ядром.

Функции:

Исполнение запросов программ (ввод и вывод данных, запуск и остановка других программ, выделение и освобождение дополнительной памяти и др.).

[Загрузка программ](http://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D1%87%D0%B8%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC) в оперативную память и их выполнение.

Стандартизованный доступ к периферийным устройствам ([устройства ввода-вывода](http://ru.wikipedia.org/wiki/%D0%A3%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE_%D0%B2%D0%B2%D0%BE%D0%B4%D0%B0-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4%D0%B0)).

Управление оперативной памятью (распределение между процессами, организация [виртуальной памяти](http://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C)).

Управление доступом к данным на энергонезависимых носителях (таких как [жёсткий диск](http://ru.wikipedia.org/wiki/%D0%96%D1%91%D1%81%D1%82%D0%BA%D0%B8%D0%B9_%D0%B4%D0%B8%D1%81%D0%BA), [оптические диски](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B4%D0%B8%D1%81%D0%BA) и др.), организованным в той или иной [файловой системе](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0).

Обеспечение [пользовательского интерфейса](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8F).

Сохранение информации об ошибках системы.

Виды ОС:

1)Монолитное ядро

ОС по сути обычная программа, поэтому логично организовать ее так как устроено большинство программ т.е. составить из процедур и функций.

В этом случае компоненты ОС являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура ОС называется монолитным ядром, т.е. это набор процедур, каждая из которых может вызывать каждую. Все процедуры работают в привилегированном режиме. Т.о. монолитное ядро – такая схема ОС, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путём непосредственного вызова процедур. Для монолитной ОС ядро совпадает со всей системой. Монолитное ядро – старейший способ организации ОС. Большинство Unix систем является системами с монолитным ядром.

2)Многоуровневые системы.

Всю вычислительную систему можно разделить на несколько уровней с чётко определёнными связями между ними так, чтобы объекты уровня n могли вызывать только объекты уровня (n-1). Нижним уровнем в таких системах обычно является железо, а верхним – интерфейс пользователя.

Чем ниже уровень, тем более привилегированные команды и действия может выполнить модуль, находящийся на этом уровне. Слоёные системы хорошо реализуются и тестируются. При использовании операций нижнего слоя необязательно знать, как они реализованы, важно лишь понимать, что они делают. Слоёные системы хорошо модернизируются. При необходимости можно заменить лишь 1 слой, не трогая остальные.

Но слоёные системы весьма сложны для разработки: очень трудно правильно определить порядок слоёв и что к какому слою относится. Слоёные системы менее эффективны чем монолитные, так, например, для выполнения операций ввода-вывода программе пользователя придётся пройти все слои от верхнего до нижнего.

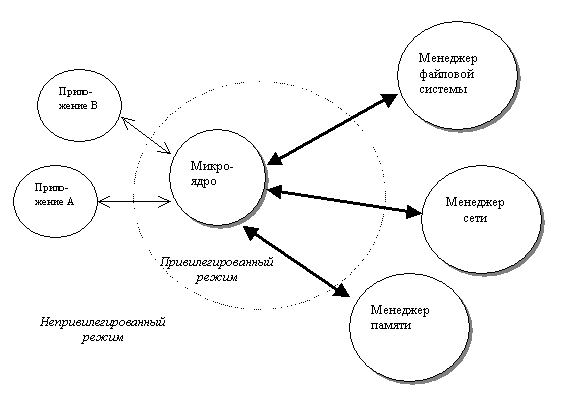
3)Виртуальные машины.

Рассмотрим подход, когда ОС реализуют виртуальные машины для каждого пользователя, но не упрощая ему жизнь, а наоборот осложняя. Каждая такая виртуальная машина предоставляется пользователю как копия всего hardware в вычислительной системе, включая процессор, привилегированные и не привилегированные команды, устройство ввода-вывода, прерывания и т.д.

Такой подход позволяет каждому пользователю загрузить свою ОС на персональную виртуальную машину и обращаться с ней по своему усмотрению. Недостатком таких ОС является снижение эффективности виртуальных машин по сравнению с реальным компьютером. Преимущества – использование в одной вычислительной системе программ написанных для разных ОС.

4)Микроядерная архитектура.

Современная тенденция в разработке ОС состоит в переносе значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Это называют микроядерной архитектурой ОС, когда большинство ее составляющих являются самостоятельными программами. При этом взаимодействие между ними обеспечивает специальный модуль – микроядро, который работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью.



Все компоненты системы взаимодействуют друг с другом путём передачи сообщений через микроядро. Основное достоинство микроядерной архитектуры – высокая степень модуляции ядра ОС. Это существенно упрощает добавление в него новых компонентов. Компоненты ядра принципиально ничем не отличаются от пользовательских программ. Поэтому для их отладки можно применять обычные средства. Микроядерная архитектура повышает надёжность системы, т.к. ошибка на уровне не привилегированной программы менее опасна, чем отказ на уровне режима ядра. В то же время микроядерная архитектура ОС вносит дополнительные накладные расходы, связанные с передачей сообщений, что существенно влияет на производительность.

5)Смешанные системы.

Все рассмотренные подходы к построению ОС имеют свои достоинства и недостатки. В большинстве случаев современные ОС используют различные комбинации этих подходов. Так, например, ядро OC Linux представляет собой монолитную систему с элементами микроядерной архитектуры. При компиляции ядра можно разрешить динамическую загрузку и выгрузку очень многих компонентов ядра, так называемых модулей. В момент загрузки модуля его код загружается на уровне системы и связывается с остальной частью ядра. Внутри модуля могут использоваться любые экспортируемые ядром функции.

Другим примером смешанного подхода может служить возможность запуска ОС с монолитным ядром под управлением микроядра, которое обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов. Все остальные функции, в том числе и взаимодействие с прикладными программами, осуществляется монолитным ядром. Данный подход использует преимущества микроядерной архитектуры, сохраняя по возможности хорошо отлаженные код монолитного ядра. Наиболее тесно элементы микроядерной архитектуры и элементы монолитного ядра переплетены в ядре Windows NT, которую часто называют микроядерная ОС. При размере микроядра NT более 1 мб приставка микро не совсем уместна. Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путём передачи сообщений, как и положено в микроядерной ОС. В то же время все компоненты ядра работают в одном пространстве и активно используют общие структуры данных, что свойственно ОС с монолитным ядром.

**Классификация ОС**

ОС могут отличаться особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера, особенностями использования, методами проектирования, типами аппаратных платформ и т.п.

Приведём основные признаки:

1)Поддержка многозадачности.

По числу одновременно выполняемых задач ОС могут быть разделены на 2 класса:

а)однозадачные: MS DOS, MSX.

б)многозадачные: ОС ЕС, OS/2, Unix, Windows \*, Windows 3.\*

Однозадачные ОС в основном выполняют функцию предоставления пользователю виртуальной машины, делая простым и удобным процесс взаимодействия пользователя с компьютером.

Многозадачные ОС кроме этого управляют разделением совместно используемых ресурсов таких как процессор, ОП, файлы и внешние устройства.

2)Поддержка многопользовательского режима.

По числу одновременно работающих пользователей ОС делятся на:

а)Однопользовательские: MS DOS, MSX, Windows 3.\*

б)Многопользовательские: Unix, Windows \*

Главным отличием является наличие средств защиты информации каждого пользователя от несанкционированного доступа.

Заметим, что не всякая многозадачная система является многопользовательской и не всякая однопользовательская является однозадачной.

3)Вытесняющая и не вытесняющая многозадачность.

Важнейшим разделяемым ресурсом является процессорное время. Способ распределения времени центрального процессора между несколькими одновременно существующими в системе процессами (нитями) во многом определяют специфику ОС. Среди множества вариантов обычно выделяют 2 группы алгоритмов:

а)Вытесняющие: NT, Unix

б)Не вытесняющие: Windows 3.\*

Основным различием между вытесняющими и не вытесняющими вариантами многозадачности является степень централизации механизма планирования процессов. В 1-м случае он целиком сосредоточен в ОС, а во 2-м распределён между системой и прикладными программами.

При не вытесняющей многозадачности активный процесс выполняет до тех пор пока он сам не отдаст управление ОС для того, чтобы она выбрала из очереди готовый к выполнению процесс.

При вытесняющей многозадачности решение о переключении процессора с одного процесса на другой принимается ОС, а не самим активным процессом.

4)Поддержка многонитиевости.

Поддержка многонитиевости означает возможность распараллеливания вычислений в рамках одной задачи. Многонитиевая ОС разделяет процессорное время не между задачами, а между их отдельными ветвями (нитями).

5)Многопроцессорная обработка

Многопроцессорные ОС могут классифицироваться по способу организации вычислительного процесса на ассиметричные и симметричные ОС.

Ассиметричные ОС целиком выполняются только на одном из процессоров, распределяя прикладные задачи по остальным процессорам. Симметричная ОС полностью децентрализована, использует весь пул (набор) процессоров, разделяя их между системными и прикладными задачами. Специфика ОС проявляется и в том каким образом она реализует сетевые функции: распознавание и перенаправление в сеть запросов к удалённым ресурсам, передача сообщений по сети, выполнение удалённых запросов. При реализации сетевых функций возникает комплекс задач, связанных с распределённым характером хранения и обработки данных в сети.

6)Системы реального времени.

Используются для управления различными техническими объектами или технологическими процессами.

Такие системы характеризуются предельно допустимым временем реакции на внешнее событие, в течение которого должна быть выполнена программа, управляющая объектом. Система должна обрабатывать поступающие данные быстрее чем они могут поступать, при чем от нескольких источников одновременно. Такие жёсткие ограничения сказываются на архитектуре систем реального времени. Например, в них может отсутствовать виртуальная память, поддержка которой даёт непредсказуемые задержки в выполнении программ.

# Управление процессами. Понятия процесса, состояния процесса.

Понятие процесса характеризует некоторую совокупность исполняющихся команд, ассоциированных с этим ресурсом (выделенная для исполнения ОП или адресное пространство, стеки, используемые файлы, устройства ввода-вывода и т.п.), и текущего момента его выполнения (значения регистров, программного счётчика, значения переменных).

Не существует взаимно однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. В некоторых ОС для работы определённых программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько программ даже в случае обработки одной программы в рамках одного процесса. Нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением ОС, поэтому в нем может выполняться часть кода ее ядра, как в случаях запланированных программистом (системные вызовы), так и не в предусмотренных ситуациях (внешние прерывания).

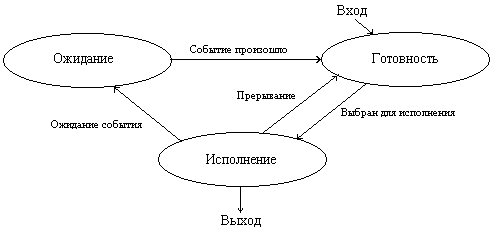
Все, что выполняется в вычислительных системах (не только программы пользователей, но и возможно определённые части ОС) организовано как набор процессов. Реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипроцессорных вычислительных систем псевдо параллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди. Каждый процесс может находиться как минимум в двух состояниях (исполняется и не исполняется).



Процесс, находящийся в состоянии исполнения через некоторое время может быть завершён ОС или приостановлен и снова переведён в состояние “не исполняется”. Приостановка процесса происходит по двум причинам: для его дальнейшей работы требуется какое-нибудь событие (например: завершение ввода) или истёк временной интервал, отведённый ОС для работы данного процесса. После этого ОС по определённому алгоритму выбирает для исполнения один из следующих процессов, находящихся в состоянии “не исполняется” и переводит его в состояние “исполняется”. Новый процесс, появляющийся в системе первоначально помещается в состояние “не исполняется”.

Это очень грубая модель. Она не учитывает в частности то, что процесс выбранный для исполнения все ещё может ожидать события из-за которого был приостановлен и к выполнению не готов.

Чтобы избежать такой ситуации, состояние “процесс не исполняется” разобьём на готовность и ожидание.



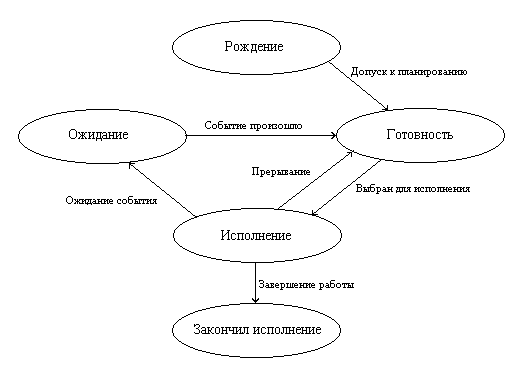
Выйти из состояния “исполняется” процесс может выйти по следующим причинам:

1)ОС прекращает его деятельность.

2)Переход в состояние ожидания некоторого события

3)в результате прерывания.

Из состояния ожидания процесс попадает в состояние готовности после того как событие произошло. Для полноты картины добавим ещё 2 состояния процессов: рождения, завершение исполнения.



При рождении процесс получает в своё распоряжение адресное пространство, в которое загружается программный код процесса, ему выделяется стек и системные ресурсы, устанавливается начальное значение программного счётчика и т.д. Родившийся процесс переводится в состояние готовность. В конкретных ОС состояния процесса могут быть ещё более детализированы. Например возможны варианты переходов из одного состояния в другое. Так например модель состояний процессов для Windows NT содержит 7 различных состояний, а Unix – 9. Тем не менее все ОС подчиняются указанной выше модели.

# Операции над процессами. Асинхронные параллельные процессы.

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояний процессов занимается ОС. Операции удобно объединить в 3 пары:

1)создание – завершение процесса;

2)приостановка – запуск процесса (перевод из состояния исполнения в состояние готовность и наоборот);

3)блокирование – разблокирование процесса (перевод из состояния исполнение в состояние ожидание и наоборот)

В дальнейшем мы введём ещё одну операцию, которая не имеет парной – изменения приоритета процесса. Операции создания и завершения процесса являются одноразовыми, а все остальные, как правило, выполняются неоднократно.

Для того чтобы ОС могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию специфическую для данного процесса:

-состояние, в котором находится процесс;

-программный счётчик процесса, т.е. адрес команды, которая должна быть выполнена следующей;

-содержимое регистров процессора;

-данные, необходимые для планирования использования процессора и управление памятью (приоритет процесса, размер и расположение адресного пространства и т.д.);

-учётные данные (идентификационный номер процесса, информация о пользователе, который его инициировал, общее время использования процессора данным процессом и т.д.);

-сведения об устройствах ввода-вывода, связанных с процессом.

Состав и строение такой структуры зависят от конкретной ОС. Во многих ОС информация, характеризующая процесс хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или часть описанной информации.

Важно то, что для любого процесса, находящегося в вычислительной системе, вся информация, необходимая для выполнения операций над ним, доступна ОС. Для простоты будем считать, что она хранится в одной структуре данных, которую будем называть Process Control Block (PCB) – блок управления процессом (БУП).

БУП является моделью процесса для ОС. Любая операция, выполняемая ОС над процессом, вызывает изменения в БУП.

В рамках принятой модели состояний процессов, содержимое БУП между операциями не изменяется. Информацию, хранящуюся в БУП, обычно разделяют на 2 части.

Содержимое всех регистров процессора, включая программный счётчик, называют регистровым контекстом процесса. Все остальное – системным контекстом. Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его работой, совершая над ним операции. Однако недостаточно для того чтобы полностью охарактеризовать процесс. Ос не интересуют какими именно вычислениями занимается процесс, т.е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя наибольший интерес представляет содержимое адресного пространства процесса, которое вместе с регистровым контекстом определяет последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве процесса будем называть его пользовательским контекстом.

Совокупность регистрового, системного и пользовательского контекстов принято называть контекстом процесса. В любой момент времени процесс полностью характеризуется своим контекстом.

Любая ОС, поддерживающая концепцию процессов, должна обладать средствами для их создания. Достаточно сложные ОС создают процессы динамически по мере необходимости. Инициатором рождения нового процесса после старта ОС может выступить либо процесс пользователя по специальному системному вызову, либо сама ОС, т.е. фактически тоже некоторый процесс. Процесс инициализировавший создание нового процесса принято называть процессом-родителем, а вновь созданный процесс – процессом-ребёнком. Процессы-дети могут в свою очередь порождать новых детей. Т.о. внутри системы образуется набор генеалогических деревьев процесса. При рождении процесса система формирует новый БУП с состоянием процесса рождения и начинает его заполнять. Новый процесс получает уникальный идентификационный номер. Т.к. для хранения этого номера в ОС отводится определённое число битов, то для соблюдения уникальности номеров количество одновременно присутствующих в ней процессов должно быть ограничено. После завершения какого-либо процесса, освободившийся идентификационный номер может быть использован для другого процесса. Обычно для выполнения своих функций процесс-ребёнок требует определённых ресурсов: памяти, файлов, устройств ввода-вывода и т.п. Существует 2 подхода к их выделению. Новый процесс может получить в своё распоряжение некоторую часть родительских ресурсов, возможно разделяя с процессом-родителем и другими процессами-детьми права на них. Другой вариант: получить свои ресурсы непосредственно от ОС.

Информация о выделенных ресурсах заносится в БУП. После наделения процесса-ребёнка ресурсами необходимо занести в его адресное пространство программный код, значения данных, установить программный счётчик. Здесь тоже возможны 2 решения.

В первом случае процесс-ребёнок становится дубликатом процесса-родителя по регистровому и пользовательским контекстам. При этом должен существовать способ определения кто для кого является родителем.

Во втором случае процесс-ребёнок загружается новой программой из какого-либо файла.

ОС Unix разрешает порождение процесса только первым способом, т.е. для запуска новой программы необходимо сначала создать копию процесса-родителя, а затем процесс-ребёнок должен заменить свой пользовательский контекст с помощью специального системного вызова. ОС VAX/VMS допускает только 2-е решение. В Windows NT возможны оба варианта.

Порождение нового процесса как дубликата процесса-родителя приводит к возможности существования программ, т.е. исполняемых файлов, для работы которых организуется более одного процесса. Возможность замены пользовательского контекста по ходу его работы приводит к тому, что в рамках одного процесса может последовательно выполняться несколько различных программ.

Процесс-родитель может продолжать своё выполнение одновременно с выполнением процесса-ребёнка, а может и ожидать завершения работы некоторых или всех своих детей. После завершения работы процесса, ОС переводит его в состояние “завершил исполнение” и освобождает все ассоциированные с ним ресурсы, делая соответствующие записи в БУП. При этом сам БУП не уничтожается, а остаётся в системе ещё некоторое время. Это связано с тем, что процесс-родитель после завершения процесса-ребёнка может запросить ОС о причине смерти порождённого им процесса или статистическую информацию о его работе.

В ОС Unix процессы, находящиеся в состоянии “завершил исполнение” принято называть процессами зомби. В некоторых ОС, например VAX/VMS, гибель процесса-родителя приводит к завершению работы всех его детей. В других ОС, например Unix, процессы-дети продолжают своё существование и после окончания работы процесса-родителя. При этом возникает необходимость изменения информации в БУП процессов-детей о породившем их процессе для того, чтобы генеалогический лес процессов оставался целостным.

Одноразовые операции приводят к изменению количества процессов и всегда связаны с выделением или освобождением определённых ресурсов. Многоразовые операции не приводят к изменению количества процессов в ОС и не обязаны быть связанными с выделением или освобождением процессов.

Действия ОС при выполнении многоразовых операций над процессами:

1)Запуск процесса

Из числа процессов, находящихся в состоянии готовность, ОС выбирает один процесс для исполнения. Для него ОС обеспечивает наличие в ОП информации для его выполнения. Состояние процесса меняется на исполнение. Восстанавливаются значения регистров для данного процесса и управление передаётся команде, на которую указывает счётчик команд процесса. Все данные, необходимые для восстановления контекста, извлекаются из БУП процесса, над которым совершается операция.

2)Приостановка процесса

Приостановка процесса происходит в результате какого-либо прерывания. Процессор автоматически сохраняет счётчик команд и некоторые регистры в стеке исполняемого процесса, а затем обрабатывается данное прерывание. ОС сохраняет динамическую часть системного и регистрового контекстов в процессе в его БУП. Переводит процесс в состояние готовность и приступает к обработке прерываний.

3)Блокирование процесса

Оно происходит, когда он не может продолжать работу, пока не произойдёт какое-либо событие. Для этого он обращается к ОС с помощью определённого системного вызова. ОС обрабатывает этот вызов и, сохранив нужную часть контекста в его БУП, переводит процесс из состояния исполнение в состояние ожидание.

4)Разблокирование процесса

ОС проверяет, находится ли какой-нибудь процесс в ожидании некоторого события, и если находится, то переводит процесс в состояние готовность, выполняя необходимые действия, связанные с наступлением события

Понятие процесса характеризует некоторую совокупность исполняющихся команд, ассоциированных с ним ресурсов и текущего момента его выполнения, находящуюся под управлением ОС.

В любой момент процесс полностью описывается своим контекстом, состоящим из регистровой, системной и пользовательской частей.

В ОС процессы представляются определённой структурой данных БУП, отражающей содержимое регистрового и системного контекстов. Процессы могут находиться в 5 основных состояниях: рождение, готовность, исполнение, ожидание, гибель.

Из одного состояния в другое процесс переводится ОС в результате выполнения над ним операций. ОС может выполнить над процессом такие операции: создание и завершение, приостановка и запуск, блокирование и разблокирование, изменение приоритета. Между операциями содержимое БУП не изменяется. Деятельность мультипрограммных ОС состоит из цепочек перечисленных операций, выполняемых над различными процессами и сопровождается переключением контекста.

Процессы называются параллельными, если они существуют одновременно. Параллельные процессы могут работать совершенно независимо друг от друга или они могут быть асинхронными — это значит, что им необходимо периодически синхронизироваться и взаимодействовать.

Параллельная работа процессов в ОС резко повышает производительность компьютера.

# Взаимоисключение.

При выполнении параллельных процессов может возникать проблема, когда каждый процесс, обращающийся к разделяемым данным, исключает для всех других процессов возможность одновременного с ним обращения к этим данным — это называется взаимоисключением (mutual exclusion).  
Ресурс, который допускает обслуживание только одного пользователя за один раз, называется критическим ресурсом. Если несколько процессов хотят пользоваться критическим ресурсом в режиме разделения времени, им следует синхронизировать свои действия таким образом, чтобы этот ресурс всегда находился в распоряжении не более чем одного их них.  
Для каждого процесса могут быть выделены участки, в которых происходит обращение к критическим ресурсам, они называются критическими участками.

Одним из первых алгоритмов синхронизации процессов был Алгоритм Деккера. Его суть: Если два процесса пытаются перейти в [критическую секцию](http://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F) одновременно, алгоритм позволит это только одному из них, основываясь на том, чья в этот момент очередь. Если один процесс уже вошёл в критическую секцию, другой будет ждать, пока первый покинет её. Это реализуется при помощи использования двух [флагов](http://ru.wikipedia.org/w/index.php?title=%D0%A4%D0%BB%D0%B0%D0%B3_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)&action=edit&redlink=1) (индикаторов "намерения" войти в критическую секцию) и переменной turn (показывающей, очередь какого из процессов наступила).

Еще одним механизмом взаимоисключения являются семафоры. Семафоры применяются, когда требуется ограничить доступ к определенному ресурсу или ограничить выполнение раздела кода некоторым количеством.

Для получения доступа к семафору цепочка вызывает функцию CreateSemaphore. При вызове этой функции необходимо указать сколько цепочек могут одновременно использовать данный ресурс или код. Если семафор будет использоваться в рамках только одного процесса, другие цепочки смогут получить дескриптор семафора с помощью глобальной переменной. Если некоторые цепочки находятся в другом процессе, чтобы получить дескриптор следует вызвать функцию OpenSemaphore. Когда цепочке потребуется доступ к совместно используемому ресурсу, она передает имя этого ресурса функции ожидания (WaitForSingleObject). Если семафор еще не затребован максимальным числом цепочка, функция ожидания увеличивает число цепочек - пользователей семафора и цепочка продолжит выполнение. Если семафор работает с максимальным числом цепочек, то цепочка, вызвавшая функцию ожидания, окажется заблокированной до тех пор, пока какая-либо другая цепочка не освободит семафор.

В 1974 году Хоаром (Хоар, Чарльз Энтони Ричард) был предложен механизм более высокого уровня, чем семафоры.

Мониторы представляют собой тип данных, который может быть внедрен в объектно-ориентированные языки программирования.

Монитор обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функции методов, принадлежащих монитору.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активным, т.е. находиться в состоянии “готовность” или “исполнение” внутри данного монитора.

Для организации очередности процессов в мониторах были введены условные переменные, над которыми можно выполнять 2 операции: Wait, Signal, отчасти похожие на операции P и V над семафорами.

Если функция монитора не может быть выполняться дальше, пока не наступит некоторое событие, то она выполняет операцию Wait над некоторой условной переменной. При этом процесс блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс выполняет операцию Signal над этой же условной переменной, что приводит к пробуждению заблокированного процесса.

# Объектно-ориентированное программирование. Классы и объекты. Наследование.

Объе́ктно-ориенти́рованное программи́рование (ООП) — [парадигма программирования](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в которой основными [концепциями](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%86%D0%B5%D0%BF%D1%86%D0%B8%D1%8F) являются понятия [объектов](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [классов](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Класс — это шаблон, или проект, по которому будет сделан объект. Конструирование объекта на основе некоторого класса называется созданием экземпляра (instance) этого класса.

Данные инкапсулируются в класс путем объявления переменных между открывающей и закрывающей фигурными скобками, выделяющими в определении класса его тело.

В ООП определены следующие ключевые свойства объектов.

• Поведение (behavior) объекта— что с ним можно делать и какие методы к нему можно применять.

• Состояние объекта — как этот объект реагирует на применение методов.

• Сущность (identity) объекта — чем данный объект отличается от других, характеризующихся таким же поведением и состоянием.

Все объекты, являющиеся экземплярами одного и того же класса, ведут себя одинаково. Поведение объекта определяется методами, которые можно вызвать. Далее, каждый объект сохраняет информацию о своем состоянии. Со временем состояние объекта может измениться, однако спонтанно это произойти не может.

Состояние объекта может изменяться только в результате вызовов методов. (Если состояние объекта изменилось вследствие иных причин, значит, принцип инкапсуляции не соблюден.)

Состояние объекта не полностью описывает его, поскольку каждый объект имеет свою собственную сущность. Например, в системе обработки заказов два заказа могут отличаться друг от друга, даже если они относятся к одним и тем же товарам. Заметим, что индивидуальные объекты, представляющие собой экземпляры класса, всегда отличаются своей сущностью и обычно отличаются своим состоянием.

Между классами существуют три общих отношения:

• зависимость ("использует");

• агрегирование ("содержит");

• наследование ("является").

Отношение зависимости (dependence— "uses-a") наиболее очевидное и распространенное. Следовательно, класс зависит от другого класса, если его методы выполняют какие-либо действия с экземплярами этого класса. Если класс А не знает о существовании класса В, он тем более ничего не знает о любых его изменениях! (Это значит, что любые изменения класса В не повлияют на поведение объектов класса А.)

Отношение агрегирования (aggregation — "has-a") понять довольно просто, поскольку оно конкретно. Агрегирование означает, что объект класса А содержит объекты класса В.

Наследование (inheritance) выражает отношение между более конкретным и более общим классом.

В ООП языках для создания новых экземпляров используются конструкторы.

Конструктор — это специальный метод, предназначенный для создания и инициализации экземпляра класса. Имя конструктора всегда совпадает с именем класса. У конструкторов нет типа возвращаемого результата.

Методы - это подпрограммы, присоединенные к конкретным определениям классов. Они описываются внутри определения класса на том же уровне, что и переменные объектов. При объявлении метода задаются тип возвращаемого им результата и список параметров. Тип результата, который должен возвращать метод может быть любым, в том числе и типом void - в тех случаях, когда возвращать результат не требуется.

Наследование — механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами. Позволяет строить иерархии классов. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом.

# Объектно-ориентированное программирование. Полиморфизм.

Объе́ктно-ориенти́рованное программи́рование (ООП) — [парадигма программирования](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в которой основными [концепциями](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%86%D0%B5%D0%BF%D1%86%D0%B8%D1%8F) являются понятия [объектов](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [классов](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

В основе ООП заложены 3 парадигмы: Наследование, инкапсуляция и полиморфизм.

Полиморфи́зм — возможность объектов с одинаковой спецификацией иметь различную реализацию. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».  Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс. Общность имеет внешнее и внутреннее выражение: внешняя общность проявляется как одинаковый набор методов с одинаковыми именами и сигнатурами (именем методов и типами аргументов и их количеством); внутренняя общность — одинаковая функциональность методов. Её можно описать интуитивно или выразить в виде строгих законов, правил, которым должны подчиняться методы. Возможность приписывать разную функциональность одному методу (функции, операции) называется перегрузкой метода (перегрузкой функций, перегрузкой операций).

Формы полиморфизма

Полиморфизм включения

Этот полиморфизм называют чистым полиморфизмом. Применяя такую форму полиморфизма, родственные объекты можно использовать обобщенно. С помощью замещения и полиморфизма включения можно написать один метод для работы со всеми типами объектов TPerson. Используя полиморфизм включения и замещения можно работать с любым объектом, который проходит тест «is-A». Полиморфизм включения упрощает работу по добавлению к программе новых подтипов, так как не нужно добавлять конкретный метод для каждого нового типа, можно использовать уже существующий, только изменив в нем поведение системы. С помощью полиморфизма можно повторно использовать базовый класс; использовать любого потомка или методы, которые использует базовый класс.

Параметрический полиморфизм

Используя Параметрический полиморфизм можно создавать универсальные базовые типы. В случае параметрического полиморфизма, функция реализуется для всех типов одинаково и таким образом функция реализована для произвольного типа. В параметрическом полиморфизме рассматриваются параметрические методы и типы.

Параметрические методы.  Если полиморфизм включения влияет на наше восприятие объекта, то параметрические полиморфизм влияет на используемые методы, так как можно создавать методы родственных классов, откладывая объявление типов до времени выполнения. Для избежания написания отдельного метода каждого типа применяется параметрический полиморфизм, при этом тип параметров будет являться таким же параметром, как и операнды.

Параметрические типы. Вместо того, чтобы писать класс для каждого конкретного типа следует создать типы, которые будут реализованы во время выполнения программы то есть мы создаем параметрический тип. Полиморфизм переопределения.  Абстрактные методы часто относятся к отложенным методам. Класс, в котором определен этот метод может вызвать метод и полиморфизм обеспечивает вызов подходящей версии отложенного метода в дочерних классах. Специальный полиморфизм допускает специальную реализацию для данных каждого типа. Полиморфизм-перегрузка - это частный случай полиморфизма. С помощью перегрузки одно и то же имя может обозначать различные методы, причем методы могут различаться количеством и типом параметров, то есть не зависят от своих аргументов. Метод может не ограничиваться специфическими типами параметров многих различных типов.

# Технологии программирования. Основные технологические подходы. Жизненный цикл программного продукта. Модели ЖЦ ПП.

*Технологией программирования* называют совокупность методов и средств, используемых в процессе разработки программного обеспечения. Как любая другая технология, технология программирования представляет собой набор технологических инструкций, включающих:

указание последовательности выполнения технологических операций;

перечисление условий, при которых выполняется та или иная операция;

описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т. п.

Кроме набора операций и их последовательности, технология также определяет способ описания проектируемой системы, точнее модели, используемой на конкретном этапе разработки.

Различают технологии, используемые на конкретных этапах разработки или для решения отдельных задач этих этапов, и технологии, охватывающие несколько этапов или весь процесс разработки. В основе первых, как правило, лежит ограниченно применимый метод, позволяющий решить конкретную задачу. В основе вторых обычно лежит базовый метод или подход (парадигма), определяющий совокупность методов, используемых на разных этапах разработки, или методологию.

Исторически в развитии программирования можно выделить несколько принципиально отличающихся методологий.

Изначально понятие технологии как таковой — это 60-е годы прошлого столетия — это период "стихийного" программирования. В этот период отсутствовало понятие структуры программы, типов данных и т.д. Вследствие этого код получался запутанным, противоречивым. Программирование тех лет считалось искусством. Конец 60-х — кризис в программирование.

Выход из этого кризиса — переход к структурной парадигме программирования. Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т.д.) данный способ получил название процедурной декомпозиции.

Другим базовым принципом структурного программирования является использование при составлении программ только базовых алгоритмических структур (см. билет 4), запрет на использование оператора GOTO.

Структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование осуществлялось "сверху-вниз" и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их описания, а также специальный метод проектирования алгоритмов — метод пошаговой детализации.

Поддержка принципов структурного программирования была заложена в основу так называемых процедурных языков программирования.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития структурирования данных. Как следствие этого в языках появляется возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы, чтобы уменьшить количество ошибок, возникающих при работе с глобальными данными. В результате появилась и стала развиваться технология модульного программирования.

*Модульное программирование* предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные, в отдельно компилируемые модули (библиотеки подпрограмм), например, модуль графических ресурсов. Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым "внутренним" переменным) запрещен.

*Объектно-ориентированное программирование (ООП)* определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений.

Жизненный цикл программного обеспечения – это весь период его разработки и эксплуатации, начиная с момента возникновения замысла и заканчивая прекращением ее использования.

Существует простейшее представление жизненного цикла, которое включает следующие стадии:

анализ;

проектирование;

программирование;

тестирование и отладка;

эксплуатация.

Жизненный цикл программного обеспечения тесно связан с технологиями программирования

Основные технологические подходы

[1. Ранние технологические подходы](http://bourabai.kz/alg/technology05.htm#351)

[1.1. Подход "кодирование и исправление"](http://bourabai.kz/alg/technology05.htm#3511)

[2. Каскадные технологические подходы](http://bourabai.kz/alg/technology05.htm#352)

[2.1. Каскадный подход](http://bourabai.kz/alg/technology05.htm#3521)

[2.2. Каскадно-возвратный подход](http://bourabai.kz/alg/technology05.htm#3522)

[2.3. Каскадный подход с подпроцессами](http://bourabai.kz/alg/technology05.htm#3525)

[2.4. Спиральная модель](http://bourabai.kz/alg/technology05.htm#3526)

[3. Группа ранних подходов быстрой разработки](http://bourabai.kz/alg/technology05.htm#356)

[3.1. Эволюционное прототипирование](http://bourabai.kz/alg/technology05.htm#3561)

[4. Адаптивные технологические подходы](http://bourabai.kz/alg/technology05.htm#357)

[4.1. Экстремальное программирование](http://bourabai.kz/alg/technology05.htm#3571)

1.Ранние технологические подходы

Ранние технологические подходы не используют явных технологий. Обычно их применяют только для очень маленьких проектов, как правило, завершающихся созданием демонстрационного прототипа. В качестве примера подхода, не использующего формализации, в данном разделе будет рассмотрен подход "кодирование и исправление".

1.1Подход "кодирование и исправление"

Подход "кодирование-исправление" (code and fix) упрощенно может быть описан следующим образом. Разработчик начинает кодирование системы с самого первого дня, не занимаясь сколь-либо серьезным проектированием.

Все ошибки обнаруживаются, как правило, к концу кодирования и требуют исправления через повторное кодирование.

Фактически каждый из программистов, так или иначе, применял этот подход. Практически все учебные программы пишутся в таком стиле. Следует заметить, что при использовании данного подхода затрачивается время лишь на кодирование и заказчику легко демонстрировать прогресс в разработке в строках кода.

Этот подход может быть рекомендован к использованию в двух случаях.

Для очень маленьких проектов, которые должны завершиться разработкой демонстрационного прототипа.

Для доказательства некоторой программной концепции.

2.Каскадные технологические подходы

Каскадные технологические подходы задают некоторую последовательность выполнения процессов, обычно изображаемую в виде каскада. Эти подходы также иногда называют подходами на основе модели водопада.

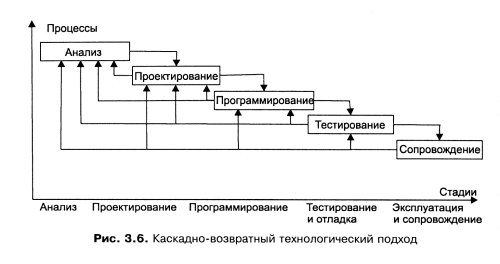
2.1Каскадный подход

Каскадный подход (pure waterfall) считается "дедушкой" технологических подходов к ведению жизненного цикла. Фактически, его можно рассматривать как отправную точку для огромного количества других подходов. Сформировался каскадный подход в период с 1970 по 1985 годы. Специфика "чистого" каскадного подхода такова, что переход к следующему процессу осуществляется только после того, как завершена работа с текущим процессом (см. рис. 3.1). Возвраты к уже пройденным процессам не предусмотрены.

Данный подход может быть рекомендован к применению в тех проектах, где в самом начале все требования могут быть сформулированы точно и полно. Например, в задачах вычислительного характера. Достаточно легко при таком технологическом подходе вести планирование работ и формирование бюджета.

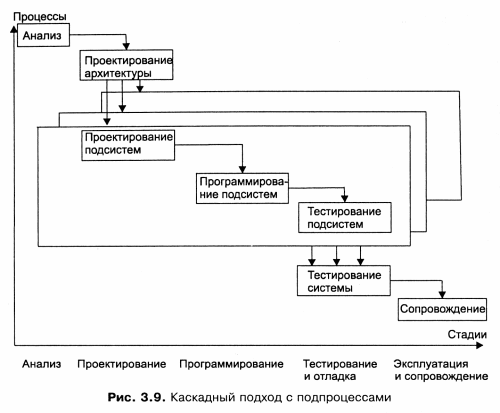
2.2Каскадно-возвратный подход

Основной недостаток каскадного подхода - отсутствие гибкости. Именно этот недостаток преодолевается каскадно-возвратным подходом, в котором разрешены возвраты к предыдущим стадиям и пересмотр или уточнение ранее принятых решений. Каскадно-возвратный подход отражает итерационный характер разработки программного обеспечения. Этот подход в значительной степени отражает реальный процесс создания программного обеспечения, в том числе и существенное запаздывание с достижением результата. На задержку оказывают существенное влияние корректировки при возвратах.



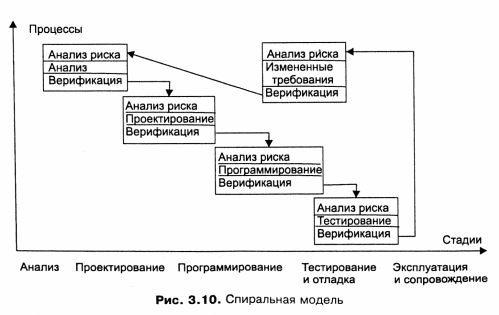
2.3. Каскадный подход с подпроцессами

Особенность его в том, что с архитектурной точки зрения проект достаточно часто может быть разделен на подпроекты, которые могут разрабатываться индивидуально. В данном подходе требуется дополнительная фаза тестирования подсистем до объединения их в единую систему. Следует особое внимание обращать на грамотное деление проекта на подпроекты, которое должно учесть все возможные зависимости между подсистемами.



2.4. Спиральная модель

Спиральная модель (spiral model) была предложена Барри Боэмом (Barry Воет) в середине 80-х годов XX века с целью сократить возможный риск разработки. Фактически, это была первая реакция на устаревание каскадной модели. Спиральная модель использует понятие прототипа - программы, реализующей частичную функциональность создаваемого программного продукта. Создание прототипов осуществляется за несколько витков спирали, каждый из которых состоит из "анализа риска", "некоторого процесса" и "верификации". Обращение к каждому процессу предваряет "анализ риска", причем, если риск превышения сроков и стоимости проекта оказывается существенным, то разработка заканчивается. Это позволяет предотвратить более крупные денежные потери в будущем.



Особенность спиральной модели - в разработке итерациями. Причем каждый следующий итерационный прототип будет обладать большей функциональностью.

3. Группа ранних подходов быстрой разработки

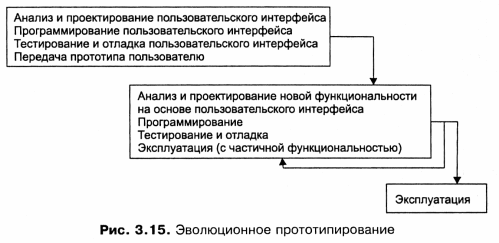
Развитием и одновременно альтернативой каскадных подходов является группа подходов быстрой разработки. Все эти подходы объединяют следующие основные черты:

итерационную разработку прототипа;

тесное взаимодействие с заказчиком.

3.1. Эволюционное прототипирование

Первый прототип при эволюционном прототипировании (evolutionary prototyping) обычно включает создание развитого пользовательского интерфейса. Он может быть сразу же продемонстрирован заказчику для получения от него отзывов и возможных корректив. Основное начальное внимание уделяется стороне системы, обращенной к пользователю. Далее, до тех пор, пока пользователь не сочтет программный продукт законченным, в него вносится необходимая функциональность.



Эволюционное прототипирование разумно применять в тех случаях, когда заказчик не может четко сформулировать свои требования к программному продукту на начальных этапах разработки или заказчик знает, что требования могут кардинально измениться.

Существенным недостатком данного подхода является невозможность определить продолжительность и стоимость проекта. Неочевидным является количество итераций, по истечении которых пользователь сочтет программный продукт законченным.

4. Адаптивные технологические подходы

Адаптивные технологические подходы были задуманы как подходы, поддерживающие изменения. Они только выигрывают от изменений, даже когда изменения происходят в них самих. Данные подходы ориентированы на человека, а не на процесс. В них необходимо учитывать в работе природные качества человеческой натуры, а не действовать им наперекор.

4.1. Экстремальное программирование

Наиболее концентрированно идеи быстрой разработки программ оказались выражены в подходе экстремального программирования (extreme programming). Две основные черты, присущие быстрым разработкам, являются базовыми и в этом подходе. Методы, объединенные в данном подходе, не являются принципиально новыми. Однако именно их рациональное объединение и совокупное использование дает существенные результаты и успешно выполненные проекты [Бек 2000]. Наибольшую пользу подход экстремального программирования может принести в разработке небольших систем, требования к которым четко не определены и вполне могут измениться.

Подход начинается с того, что проводится анализ назначения системы и определения первоочередной функциональности. В результате составляется список историй - возможных применений системы. Каждая история должна быть ориентирована на определенные задачи бизнеса, которые можно оценить с помощью количественных показателей. Наконец, ценность истории определяется материальными и временными затратами на ее разработку командой разработчиков.

Заказчик выбирает истории для очередной итерации, основываясь на их значимости для проекта и ценности. Для первой версии системы заказчик определяет небольшое количество логически связанных наиболее важных историй. Для каждой следующей версии выбираются наиболее важные истории из числа оставшихся (рис. 3.17).



Одним из существенных методов данного подхода является функциональное тестирование. Существуют две особенности процесса тестирования.

Программисты сами пишут тесты для тестирования программы.

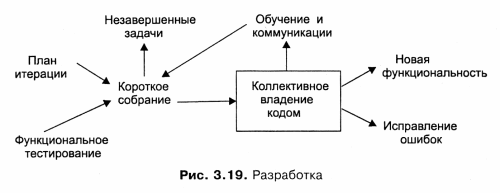
Эти тесты пишутся до начала кодирования.

Уверенность в нормальной работе каждого отдельного теста постепенно формирует у разработчиков уверенность в нормальной работе системы в целом.

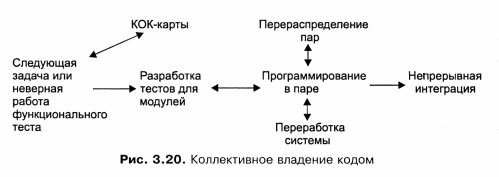
Цель каждой итерации (рис. 3.18) - включить в версию несколько новых историй. На собрании по планированию итерации определяется, какие именно истории будут реализованы и каким образом это будет сделано командой разработчиков.



Коллективное владение кодом в процессе разработки (рис. 3.19) означает возможность для каждого программиста в любое время усовершенствовать любую часть кода в системе, если он сочтет это необходимым. Программист берет на себя ответственность за реализацию определенных задач. В случае возникновения вопросов о разрабатываемой задаче, может быть проведено короткое (15-минутное) собрание с присутствием заказчика.



Для того чтобы выполнить задачу, ответственный за нее программист должен найти себе партнера (рис. 3.20). Окончательный код всегда пишется двумя программистами на одной рабочей станции.



Добавим еще, что экстремальное программирование уделяет значительное внимание организации офисного пространства, отмечая существенное влияние окружающих условий на работу программистов.

# Компьютерные сети. Типы каналов связи. Эталонная модель OSI. Понятия: протокол и интерфейс.

Под компьютерной сетью мы будем понимать любое множество ЭВМ, связанных между собой средствами передачи данных (средствами телекоммуникаций).

По способу организации сети подразделяются на реальные и искусственные.

Искусственные сети (псевдосети) позволяют связывать компьютеры вместе через последовательные или параллельные порты и не нуждаются в дополнительных устройствах. Иногда связь в такой сети называют связью по нуль-модему (не используется модем). Само соединение называют нуль-модемным. Искусственные сети используются когда необходимо перекачать информацию с одного компьютера на другой. MS-DOS и Windows снабжены специальными программами для реализации нуль-модемного соединения.

Основной недостаток - низкая скорость передачи данных и возможность соединения только двух компьютеров.

Реальные сети позволяют связывать компьютеры с помощью специальных устройств коммутации и физической среды передачи данных.

Основной недостаток - необходимость в дополнительных устройствах.

В классификации сетей существует два основных термина: LAN и wAN.

LAN (Local Area Network) - локальные сети, имеющие замкнутую инфраструктуру до выхода на поставщиков услуг. Термин "LAN" может описывать и маленькую офисную сеть, и сеть уровня большого завода, занимающего несколько сотен гектаров. Зарубежные источники дают даже близкую оценку - около шести миль (10 км) в радиусе; использование высокоскоростных каналов.

wAN (wide Area Network) - глобальная сеть, покрывающая большие географические регионы, включающие в себя как локальные сети, так и прочие телекоммуникационные сети и устройства. Пример wAN - сети с коммутацией пакетов (Frame relay), через которую могут "разговаривать" между собой различные компьютерные сети.

Термин "корпоративная сеть" также используется в литературе для обозначения объединения нескольких сетей, каждая из которых может быть построена на различных технических, программных и информационных принципах.

Локальные сети являются сетями закрытого типа, доступ к ним разрешен только ограниченному кругу пользователей, для которых работа в такой сети непосредственно связана с их профессиональной деятельностью. Глобальные сети являются открытыми и ориентированы на обслуживание любых пользователей.

С точки зрения организации взаимодействия компьютеров, сети делят на одноранговые (Peer-to-Peer Network) и с выделенным сервером (Dedicated Server Network).

Канал связи  — система технических средств и среда распространения сигналов для односторонней передачи данных от источника к получателю. В случае использования проводной линии связи, средой распространения сигнала может являться оптическое волокно или витая пара. Канал связи является составной частью [канала передачи данных](http://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BD%D0%B0%D0%BB_%D0%BF%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85).

Существует множество видов каналов связи, среди которых наиболее часто выделяют каналы проводной связи (воздушные, кабельные, световодные и др.) и каналы радиосвязи (тропосферные, спутниковые и др.). Такие каналы в свою очередь принято квалифицировать на основе характеристик входного и выходного сигналов, а также по изменению характеристик сигналов в зависимости от таких явлений, происходящих в канале, как замирания и затухание сигналов.

По типу среды распространения каналы связи делятся на [проводные](http://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%B1%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%BB%D0%B8%D0%BD%D0%B8%D0%B8_%D1%81%D0%B2%D1%8F%D0%B7%D0%B8), [акустические](http://ru.wikipedia.org/wiki/%D0%90%D0%BA%D1%83%D1%81%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B2%D0%BE%D0%BB%D0%BD%D0%BE%D0%B2%D0%BE%D0%B4), [оптические](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B2%D0%BE%D0%BB%D0%BE%D0%BA%D0%BD%D0%BE), [инфракрасные](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D1%80%D0%B0%D0%BA%D1%80%D0%B0%D1%81%D0%BD%D1%8B%D0%B9_%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB) и[радиоканалы](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B4%D0%B8%D0%BE%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB).

Каналы связи также классифицируют на

непрерывные (на входе и выходе канала - непрерывные сигналы),

дискретные или цифровые (на входе и выходе канала - дискретные сигналы),

непрерывно-дискретные (на входе канала - непрерывные сигналы, а на выходе - дискретные сигналы),

дискретно-непрерывные (на входе канала - дискретные сигналы, а на выходе - непрерывные сигналы).

Международной организацией стандартов утверждены определённые требования к организации взаимодействия между системами сети.

Эти требования получили название OSI (Open System Interconnection) - "эталонная модель взаимодействия открытых систем".

Согласно требованиям эталонной модели, каждая система сети должна осуществлять взаимодействие посредствам передачи кадра данных. Согласно модели OSI образование и передача кадра осуществляется с помощью 7-ми последовательных действий, получивших название "уровень обработки".

Основная идея этой модели заключается в том, что каждому уровню отводится конкретная роль, в том числе и транспортной среде. Благодаря этому общая задача передачи данных расчленяется на отдельные легко обозримые задачи.

Так как пользователи нуждаются в эффективном управлении, система вычислительной сети представляется как комплексное строение, которое координирует взаимодействие задач пользователей.

Отдельные уровни базовой модели проходят в направлении вниз от источника данных (от уровня 7 к уровню 1) и в направлении вверх от приемника данных (от уровня 1 к уровню 7). Пользовательские данные передаются в нижерасположенный уровень вместе со специфическим для уровня заголовком до тех пор, пока не будет достигнут последний уровень.

На приемной стороне поступающие данные анализируются и, по мере надобности, передаются далее в вышерасположенный уровень, пока информация не будет передана в пользовательский прикладной уровень.

Уровень 1. Физический

На физическом уровне определяются электрические, механические, функциональные и процедурные параметры для физической связи в системах. Физическая связь и неразрывная с ней эксплуатационная готовность являются основной функцией 1-го уровня. Стандарты физического уровня включают рекомендации V.24 МККТТ (CCITT), EIA rS232 и Х.21. Стандарт ISDN ( Integrated Services Digital Network) в будущем сыграет определяющую роль для функций передачи данных. В качестве среды передачи данных используют трехжильный медный провод (экранированная витая пара), коаксиальный кабель, оптоволоконный проводник и радиорелейную линию.

Уровень 2. Канальный

Канальный уровень формирует из данных, передаваемых 1-м уровнем, так называемые "кадры", последовательности кадров. На этом уровне осуществляются управление доступом к передающей среде, используемой несколькими ЭВМ, синхронизация, обнаружение и исправление ошибок.

Уровень 3. Сетевой

Сетевой уровень устанавливает связь в вычислительной сети между двумя абонентами. Соединение происходит благодаря функциям маршрутизации, которые требуют наличия сетевого адреса в пакете. Сетевой уровень должен также обеспечивать обработку ошибок, мультиплексирование, управление потоками данных. Самый известный стандарт, относящийся к этому уровню, - рекомендация Х.25 МККТТ (для сетей общего пользования с коммутацией пакетов).

Уровень 4. Транспортный

Транспортный уровень поддерживает непрерывную передачу данных между двумя взаимодействующими друг с другом пользовательскими процессами. Качество транспортировки, безошибочность передачи, независимость вычислительных сетей, сервис транспортировки из конца в конец, минимизация затрат и адресация связи гарантируют непрерывную и безошибочную передачу данных.

Уровень 5. Сеансовый

Сеансовый уровень координирует прием, передачу и выдачу одного сеанса связи. Для координации необходимы контроль рабочих параметров, управление потоками данных промежуточных накопителей и диалоговый контроль, гарантирующий передачу, имеющихся в распоряжении данных. Кроме того, сеансовый уровень содержит дополнительно функции управления паролями, подсчета платы за пользование ресурсами сети, управления диалогом, синхронизации и отмены связи в сеансе передачи после сбоя вследствие ошибок в нижерасположенных уровнях.

Уровень 6. Представления данных

Уровень представления данных предназначен для интерпретации данных; а также подготовки данных для пользовательского прикладного уровня. На этом уровне происходит преобразование данных из кадров, используемых для передачи данных в экранный формат или формат для печатающих устройств оконечной системы.

Уровень 7. Прикладной

В прикладном уровне необходимо предоставить в распоряжение пользователей уже переработанную информацию. С этим может справиться системное и пользовательское прикладное программное обеспечение.

|  |  |  |
| --- | --- | --- |
| Модель OSI | | |
| Тип данных | Уровень (layer) | Функции |
| Данные | 7. Прикладной (application) | Доступ к сетевым службам |
| Поток | 6. Уровень представления (presentation) | Представление и шифрование данных |
| Сеансы | 5. Сеансовый (session) | Управление сеансом связи |
| Сегменты | 4. Транспортный (transport) | Прямая связь между конечными пунктами и надежность |
| Пакеты / Датаграммы | 3. Сетевой (network) | Определение маршрута и логическая адресация |
| Кадры | 2. Канальный (data link) | Физическая адресация |
| Биты | 1. Физический (physical) | Работа со средой передачи, сигналами и двоичными данными |

Протоколом будем называть набор процедур, регламентирующих взаимодействие двух абонентских систем на определенных уровнях модели OSI.

Интерфейсом будем называть набор процедур, регламентирующих взаимодействие двух соседних уровней в пределах одной абонентской системы.

Набор протоколов, достаточный для организации связи абонентской системы в сети, будем называть стеком протоколов.

# Маршрутизация в компьютерных сетях: способы и методы.

Основная идея введения сетевого уровня состоит в следующем. Сеть в общем случае рассматривается как совокупность нескольких сетей и называется ***составной сетью*** или ***интерсетью*** *(internetworking* или *internet).* Сети, входящие в составную сеть, называются ***подсетями*** *(subnet),* ***составляющими сетями*** или просто сетями.

Подсети соединяются между собой ***маршрутизаторами.*** Компонентами составной сети могут являться как *локальные*, так и *глобальные* сети. Внутренняя структура каждой сети не имеет значения при рассмотрении сетевого протокола. Все узлы в пределах одной подсети взаимодействуют, используя единую для них технологию. Для организации взаимодействия между любой произвольной парой узлов составной сети требуются дополнительные средства. Такие средства и предоставляет *сетевой уровень*.

Важнейшей задачей сетевого уровня является ***маршрутизация*** — *передача пакетов между двумя конечными узлами в составной сети*.

Маршрутизатор можно рассматривать как совокупность нескольких узлов, каждый из которых входит в свою сеть. Как единое устройство маршрутизатор не имеет ни отдельного сетевого адреса, ни какого-либо локального адреса.

В сложных составных сетях почти всегда существует несколько альтернативных маршрутов для передачи пакетов между двумя конечными узлами. ***Маршрут*** — это последовательность маршрутизаторов, которые должен пройти пакет от отправителя до пункта назначения.

Задачу выбора маршрута из нескольких возможных решают маршрутизаторы, а также конечные узлы. Маршрут выбирается на основании имеющейся у этих устройств информации о текущей конфигурации сети, а также на основании указанного критерия выбора маршрута. Обычно в качестве критерия выступает *задержка прохождения маршрута отдельным пакетом* или *средняя пропускная способность маршрута для последовательности пакетов*. Часто также используется весьма простой критерий, учитывающий только количество пройденных в маршруте промежуточных маршрутизаторов (хопов).

Чтобы по адресу сети назначения можно было бы выбрать рациональный маршрут дальнейшего следования пакета, каждый конечный узел и маршрутизатор анализируют специальную информационную структуру, которая называется ***таблицей маршрутизации***. В первом столбце таблицы перечисляются номера сетей, входящих в интерсеть. В каждой строке таблицы следом за номером сети указывается сетевой адрес следующего маршрутизатора (более точно, сетевой адрес соответствующего порта следующего маршрутизатора), на который надо направить пакет, чтобы тот передвигался по направлению к сети с данным номером по рациональному маршруту.

Когда на маршрутизатор поступает новый пакет, номер сети назначения, извлеченный из поступившего кадра, последовательно сравнивается с номерами сетей из каждой строки таблицы. Строка с совпавшим номером сети указывает, на какой ближайший маршрутизатор следует направить пакет.

Поскольку пакет может быть адресован в любую сеть составной сети, может показаться, что каждая таблица маршрутизации должна иметь записи обо всех сетях, входящих в составную сеть. Но при таком подходе в случае крупной сети объем таблиц маршрутизации может оказаться очень большим, что повлияет на время ее просмотра, потребует много места для хранения и т. п. Поэтому на практике число записей в таблице маршрутизации стараются уменьшить за счет использования специальной записи — ***«маршрутизатор по умолчанию»-*** *(default).* Действительно, если принять во внимание топологию составной сети, то в таблицах маршрутизаторов, находящихся на периферии составной сети, достаточно записать номера сетей, непосредственно подсоединенных к данному маршрутизатору или расположенных поблизости, на тупиковых маршрутах. Обо всех же остальных сетях можно сделать в таблице единственную запись, указывающую на маршрутизатор, через который пролегает путь ко всем этим сетям. Такой маршрутизатор называется *маршрутизатором по умолчанию*, а вместо номера сети в соответствующей строке помещается особая запись, например default

Перед тем как передать пакет следующему маршрутизатору, текущий маршрутизатор должен определить, на какой из нескольких собственных портов он должен поместить данный пакет. Для этого служит третий столбец таблицы маршрутизации. Еще раз подчеркнем, что каждый порт идентифицируется собственным сетевым адресом.

***Задачу маршрутизации решают*** не только промежуточные узлы-маршрутизаторы, но и конечные узлы — ***компьютеры***. Средства сетевого уровня, установленные на конечном узле, при обработке пакета должны, прежде всего, определить, направляется ли он в другую сеть или адресован какому-нибудь узлу данной сети. Если номер сети назначения совпадает с номером данной сети, то для данного пакета не требуется решать задачу маршрутизации. Если же номера сетей отправления и назначения не совпадают, то маршрутизация нужна. Таблицы маршрутизации конечных узлов полностью аналогичны таблицам маршрутизации, хранящимся на маршрутизаторах.

Конечные узлы в еще большей степени, чем маршрутизаторы, пользуются приемом маршрутизации по умолчанию. Хотя они также в общем случае имеют в своем распоряжении таблицу маршрутизации, ее объем обычно незначителен, что объясняется периферийным расположением всех конечных узлов. Конечный узел часто вообще работает без таблицы маршрутизации, имея только сведения об адресе маршрутизатора по умолчанию.

Еще одним отличием работы маршрутизатора и конечного узла при выборе маршрута является способ построения таблицы маршрутизации. Если маршрутизаторы обычно автоматически создают таблицы маршрутизации, обмениваясь служебной информацией, то для конечных узлов таблицы маршрутизации часто создаются вручную администраторами и хранятся в виде постоянных файлов на дисках.

Задача маршрутизации решается на основе анализа таблиц маршрутизации, размещенных во всех маршрутизаторах и конечных узлах сети. Основная работа по созданию таблиц маршрутизации выполняется автоматически, но и возможность вручную скорректировать или дополнить таблицу тоже, как правило, предусматривается.

Для автоматического построения таблиц маршрутизации маршрутизаторы обмениваются информацией о топологии составной сети в соответствии со специальным служебным протоколом. Протоколы этого типа называются ***протоколами маршрутизации*** (или маршрутизирующими протоколами). Протоколы маршрутизации (например, RIP, OSPF, NLSP) следует отличать от собственно сетевых протоколов (например, IP, IPX). И те и другие выполняют функции сетевого уровня модели OSI — участвуют в доставке пакетов адресату через разнородную составную сеть. Но в то время как первые собирают и передают по сети чисто служебную информацию, вторые предназначены для передачи пользовательских данных, как это делают протоколы канального уровня. Протоколы маршрутизации используют сетевые протоколы как транспортное средство. При обмене маршрутной информацией пакеты протокола маршрутизации помещаются в поле данных пакетов сетевого уровня или даже транспортного уровня, поэтому с точки зрения вложенности пакетов протоколы маршрутизации формально следовало бы отнести к более высокому уровню, чем сетевой.

С помощью протоколов маршрутизации маршрутизаторы составляют карту связей сети той или иной степени подробности. На основании этой информации для каждого номера сети принимается решение о том, какому следующему маршрутизатору надо передавать пакеты, направляемые в эту сеть, чтобы маршрут оказался рациональным. Результаты этих решений заносятся в таблицу маршрутизации. *При изменении конфигурации сети* некоторые записи в таблице становятся недействительными. В таких случаях пакеты, отправленные по ложным маршрутам, могут зацикливаться и теряться. От того, насколько быстро протокол маршрутизации приводит в соответствие содержимое таблицы реальному состоянию сети, зависит качество работы всей сети.

Протоколы маршрутизации могут быть построены на основе разных ***алгоритмов***, отличающихся способами построения таблиц маршрутизации, способами выбора наилучшего маршрута и другими особенностями своей работы.

*Одношаговый подход.* Маршрутизация выполняется по распределенной схеме — каждый маршрутизатор ответственен за выбор только одного шага маршрута, а окончательный маршрут складывается в результате работы всех маршрутизаторов, через которые проходит данный пакет. Такие алгоритмы маршрутизации называются *одношаговыми.*

*Многошаговый подход* — *маршрутизация от источника (Source Routing).* В соответствии с ним узел-источник задает в отправляемом в сеть пакете полный маршрут его следования через все промежуточные маршрутизаторы. При использовании многошаговой маршрутизации нет необходимости строить и анализировать таблицы маршрутизации. Это ускоряет прохождение пакета по сети, разгружает маршрутизаторы, но при этом большая нагрузка ложится на конечные узлы.

Одношаговые алгоритмы в зависимости от способа формирования таблиц маршрутизации делятся на три класса:

*• алгоритмы фиксированной (или статической) маршрутизации;*

*• алгоритмы простой маршрутизации;*

*• алгоритмы адаптивной (или динамической) маршрутизации.*

В *алгоритмах фиксированной маршрутизации* все записи в таблице маршрутизации являются статическими. Администратор сети сам решает, на какие маршрутизаторы надо передавать пакеты с теми или иными адресами, и вручную заносит соответствующие записи в таблицу маршрутизации.

В *алгоритмах простой маршрутизации* таблица маршрутизации либо вовсе не используется, либо строится без участия протоколов маршрутизации. Выделяют три типа простой маршрутизации:

• *случайная маршрутизация,* когда прибывший пакет посылается в первом попавшем случайном направлении, кроме исходного;

• *лавинная маршрутизация,* когда пакет широковещательно посылается по всем возможным направлениям, кроме исходного (аналогично обработке мостами кадров с неизвестным адресом);

• *маршрутизация по предыдущему опыту,* когда выбор маршрута осуществляется по таблице, но таблица строится по принципу моста путем анализа адресных полей пакетов, появляющихся на входных портах.

Самыми распространенными являются *алгоритмы адаптивной (или динамической) маршрутизации.* Эти алгоритмы обеспечивают автоматическое обновление таблиц маршрутизации после изменения конфигурации сети. Адаптивные алгоритмы обычно имеют распределенный характер, который выражается в том, что в сети отсутствуют какие-либо выделенные маршрутизаторы, которые собирали бы и обобщали топологическую информацию: эта работа распределена между всеми маршрутизаторами.

Адаптивные протоколы обмена маршрутной информацией, применяемые в настоящее время в вычислительных сетях, в свою очередь делятся на две группы, каждая из которых связана с одним из следующих типов алгоритмов:

*• дистанционно-векторные алгоритмы (DVA);*

*• алгоритмы состояния связей (LSA).*

В алгоритмах *дистанционно-векторного типа* каждый маршрутизатор периодически и широковещательно рассылает по сети вектор, компонентами которого являются расстояния от данного маршрутизатора до всех известных ему сетей. Каждый маршрутизатор узнает информацию обо всех имеющихся в интерсети сетях и о расстоянии до них через соседние маршрутизаторы. Дистанционно-векторные алгоритмы хорошо работают только в небольших сетях.

*Алгоритмы состояния связей* обеспечивают каждый маршрутизатор информацией, достаточной для построения точного графа связей сети. Все маршрутизаторы работают на основании одинаковых графов, что делает процесс маршрутизации более устойчивым к изменениям конфигурации. «Широковещательная» рассылка (то есть передача пакета всем непосредственным соседям маршрутизатора) используется здесь только при изменениях состояния связей, что происходит в надежных сетях не так часто.

# Глобальная компьютерная сеть Интернет. Основные принципы построения и управления сетью.

Интернет – глобальная компьютерная сеть, объединяющая сети, шлюзы, серверы и компьютеры, использующие для связи единый набор протоколов TCP/IP. TCP/IP — это технология межсетевого взаимодействия.  Интернет не является коммерческой организацией и никому не принадлежит; оплачиваются только услуги провайдера - организации, предоставляющей возможность подключения к Интернет.

Как и во всякой другой сети, в Интернете существует 7 уровней взаимодействия между компьютерами: физический, логический, сетевой, транспортный, уровень сеансов связи, представительский и прикладной. Каждому уровню взаимодействия соответствует набор протоколов (т. е. правил взаимодействия).

Для того чтобы в процессе обмена информа­цией компьютеры могли найти друг друга, в Интернете су­ществует единая система адресации, основанная на исполь­зовании Интернет-адреса (IP-адреса).

Компьютеры легко могут най­ти друг друга по числовому IP-адресу, однако человеку за­помнить числовой адрес нелегко, и для удобства была введе­на доменная система имен (DNS —Domain Name System).

Идеологическая основа проекта: отсутствие общего центра управления, полная самостоятельность каждого сегмента. Обмен информацией между узлами сети с гарантированной доставкой сообщений обеспечивался специальным набором протоколов TCP/IP. Сообщения для передачи разбиваются на небольшие специфицированные элементы, названные пакетами. Каждый пакет имеет информацию об адресе назначения, доставка сообщения обеспечивается тем, что каждый узел имеет возможность посылать (или переадресовывать) пакеты по сети к месту назначения. При этом узел сам выбирает наиболее быстрый путь для каждого пакета в соответствии с состоянием соседних участков сети. Таким образом, возможна ситуация, когда разные части файла (пакеты) добираются до адресата различными путями, при этом его окончательная сборка происходит на компьютере получателя.

Кроме того, был разработан ряд стандартных утилит для передачи файлов (FTP), для дистанционного запуска программ (Telnet) и для обмена электронной почтой (SMTP).

Способ соединения компьютеров в сеть называется её топологией. Под топологией сети понимается конфигурация графа, вершинам которого соответствуют конечные узлы сети (например, компьютеры) и коммуникационное оборудование (например, маршрутизаторы), а ребрам — электрические и информационные связи между ними.

Наиболее распространенные виды топологий сетей:

|  |  |
| --- | --- |
| Линейная сеть | Содержит только два оконечных узла, любое число промежуточных узлов и имеет только один путь между любыми двумя узлами. |
| top01 |
| Кольцевая сеть | Сеть, в которой к каждому узлу присоединены две и только две ветви. |
| top02 |
| Звездообразная сеть | Сеть, в которой имеется только один промежуточный узел. |
| top03 |
| Общая шина | В этом случае подключение и обмен данными производится через общий канал связи, называемый общей шиной. |
| top04 |
| Древовидная сеть | Сеть, которая содержит более двух оконечных узлов и по крайней мере два промежуточных узла, и в которой между двумя узлами имеется только один путь. |
| top05 |
| Ячеистая сеть | Сеть, которая содержит по крайней мере два узла, имеющих два или более пути между ними. |
| top06 |

Более подробно о топологиях:

* Ячеистая топология (mesh) получается из полносвязной путем удаления некоторых возможных связей. Ячеистая топология допускает соединение большого количества компьютеров и характерна для крупных сетей.
* В сетях с кольцевой конфигурацией данные передаются по кольцу от одного компьютера к другому. Главное достоинство "кольца" в том, что оно по своей природе обладает свойством резервирования связей. Действительно, любая пара узлов соединена здесь двумя путями — по часовой стрелке и против. "Кольцо" представляет собой очень удобную конфигурацию и для организации обратной связи — данные, сделав полный оборот, возвращаются к узлу-источнику. Поэтому отправитель в данном случае может контролировать процесс доставки данных адресату. Часто это свойство "кольца" используется для тестирования связности сети и поиска узла, работающего некорректно. В то же время в сетях с кольцевой топологией необходимо принимать специальные меры, чтобы в случае выхода из строя или отключения какой-либо станции не прерывался канал связи между остальными станциями "кольца".
* Топология "звезда" образуется в том случае, когда каждый компьютер с помощью отдельного кабеля подключается к общему центральному устройству, называемому концентратором. В функции концентратора входит направление передаваемой компьютером информации одному или всем остальным компьютерам сети. В роли концентратора может выступать как компьютер, так и специализированное устройство, такое как многовходовый повторитель, коммутатор или маршрутизатор. К недостаткам топологии типа "звезда" относится более высокая стоимость сетевого оборудования, связанная с необходимостью приобретения специализированного центрального устройства. Кроме того, возможности наращивания количества узлов в сети ограничиваются количеством портов концентратора.
* Иногда имеет смысл строить сеть с использованием нескольких концентраторов, иерархически соединенных между собой связями типа "звезда". Получаемую в результате структуру называют также деревом. В настоящее время дерево является самым распространенным типом топологии связей, как в локальных, так и в глобальных сетях.
* Особым частным случаем конфигурации звезда является конфигурация "общая шина". Здесь в роли центрального элемента выступает пассивный кабель, к которому по схеме "монтажного ИЛИ" подключается несколько компьютеров (такую же топологию имеют многие сети, использующие беспроводную связь — роль общей шины здесь играет общая радиосреда). Передаваемая информация распространяется по кабелю и доступна одновременно всем присоединенным к нему компьютерам. Основными преимуществами такой схемы являются низкая стоимость и простота наращивания, т.е. присоединения новых узлов к сети. Самым серьезным недостатком "общей шины" является ее недостаточная надежность: любой дефект кабеля или какого-нибудь из многочисленных разъемов полностью парализует всю сеть. Другой недостаток "общей шины" — невысокая производительность, так как при таком способе подключения в каждый момент времени только один компьютер может передавать данные по сети, поэтому пропускная способность канала связи всегда делится между всеми узлами сети. До недавнего времени "общая шина" являлась одной из самых популярных топологий для локальных сетей.
* В то время как небольшие сети, как правило, имеют типовую топологию — "звезда", "кольцо" или "общая шина", для крупных сетей характерно наличие произвольных связей между компьютерами. В таких сетях можно выделить отдельные произвольно связанные фрагменты (подсети), имеющие типовую топологию, поэтому их называют сетями со смешанной топологией.

Существует два основных принципа управления в локальных сетях: централизация и децентрализация.

    В сетях с централизованным управлением функции управления обменом данными возложены на файл-серверы. Файлы, хранящиеся на сервере, доступны рабочим станциям сети. Одна PC к файлам другой PC доступа не имеет. Правда, обмен файлами между PC может происходить и в обход основных путей с помощью специальных программ.

    Существует множество сетевых ОС, реализующих централизованное управление. Среди них Microsoft Windows NT/2000 Server, Novell NetWare (версии З.Х и 4.Х), Microsoft Lan Manager, OS/2 Warp Server Advanced, VINES 6.0 и др.

**Преимуществом централизованных сетей** является высокая защищенность сетевых ресурсов от несанкционированного доступа, удобство администрирования сети, возможность создания сетей с большим числом узлов. Основной недостаток состоит в уязвимости системы при нарушении работоспособности файл-сервера (это преодолевается при наличии нескольких серверов или в результате принятия некоторых других мер), а также в предъявлении довольно высоких требований к ресурсам серверов.

    В централизованной схеме управления все вычислительные ресурсы, данные и программы их обработки были сконцентрированы в одной ЭВМ. Пользователи имели доступ к ресурсам машины с помощью терминалов (дисплеев). Терминалы подключались к ЭВМ через интерфейсные соединения или удаленные телефонные линии связи (так называемые удаленные терминалы). Основной функцией терминала было отображение информации, представляемой пользователю. К достоинствам этой схемы можно отнести удобство администрирования, модификации программного обеспечения и защиты информации. Недостатком схемы является ее низкая надежность (выход из строя ЭВМ влечет за собой разрушение вычислительного процесса), сложность масштабирования (наращивания мощности) модификации аппаратного и программного обеспечения, как правило, резкое снижение оперативности при увеличении числа пользователей системы и др.

    Децентрализованные (одноранговые) сети не содержат в своем составе выделенных серверов. Функции управления сетью в них поочередно передаются от одной PC к другой. Ресурсы одной PC (диски, принтеры и другие устройства) оказываются доступными другим PC.

    Наиболее распространенными программными продуктами, позволяющими строить одноранговые сети, являются следующие программы и пакеты: Novell NetWare Lite, Windows for Workgroups, Artisoft LANtastic, LANsmart, Invisible Software NET-30 и др. Все они могут работать под управлением DOS. Для одноранговой сети могут быть использованы также ОС Windows 95/98 и Windows NT/2000 Prof, Windows XP.

    Развертывание одноранговой сети для небольшого числа PC часто позволяет построить более эффективную и живучую распределенную вычислительную среду. Сетевое программное обеспечение в них является более простым по сравнению с централизованными сетями. Здесь не требуется установка файл-сервера (как компьютера, так и соответствующих программ), что существенно удешевляет систему. Однако такие сети слабее с точки зрения защиты информации и администрирования.

# Адресация в Интернет.

Для того чтобы в процессе обмена информацией компьютеры могли найти друг друга, в Интернете существует единая система адресации, основанная на использовании IP-адреса.

**IP-адрес**  — это уникальный [сетевой адрес](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%B0%D0%B4%D1%80%D0%B5%D1%81) [узла](http://ru.wikipedia.org/wiki/%D0%A3%D0%B7%D0%B5%D0%BB_%D1%81%D0%B5%D1%82%D0%B8) в [компьютерной сети](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C), построенной по [протоколу](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB_%D0%BF%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) [IP](http://ru.wikipedia.org/wiki/IP). В сети [Интернет](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82) требуется глобальная уникальность адреса; в случае работы в [локальной сети](http://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C) требуется уникальность адреса в пределах сети.

В 4-й версии IP-адрес представляет собой 32-[битовое](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82) число. Удобной формой записи IP-адреса ([IPv4](http://ru.wikipedia.org/wiki/IPv4)) является запись в виде четырёх десятичных чисел значением от 0 до 255, разделённых точками

В 6-й версии IP-адрес ([IPv6](http://ru.wikipedia.org/wiki/IPv6)) является 128-[битовым](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82). Внутри адреса разделителем является двоеточие (напр. 2001:0db8:85a3:0000:0000:8a2e:0370:7334). Ведущие нули допускается в записи опускать. Нулевые группы, идущие подряд, могут быть опущены, вместо них ставится двойное двоеточие (fe80:0:0:0:0:0:0:1 можно записать как fe80::1). Более одного такого пропуска в адресе не допускается.

IP-адрес состоит из двух частей: номера сети и номера узла. В случае изолированной сети её адрес может быть выбран администратором из специально зарезервированных для таких сетей блоков адресов (10.0.0.0/8, 172.16.0.0/12 или 192.168.0.0/16).

табличка соответствия для масок подсетей:

|  |  |  |
| --- | --- | --- |
| **Размер сети** | **Короткая маска** | **Длинная маска** |
| 4 | /30 | 255.255.255.252 |
| 8 | /29 | 255.255.255.248 |
| 6 | /28 | 255.255.255.240 |
| 32 | /27 | 255.255.255.224 |
| 64 | /26 | 255.255.255.192 |
| 128 | /25 | 255.255.255.128 |
| 256 | /24 | 255.255.255.0 |

В [протоколе IP](http://ru.wikipedia.org/wiki/IP) существует несколько соглашений об особой интерпретации IP-адресов: если все двоичные разряды IP-адреса равны 1, то [пакет](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BA%D0%B5%D1%82_(%D1%81%D0%B5%D1%82%D0%B5%D0%B2%D1%8B%D0%B5_%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D0%B8)) с таким адресом назначения должен рассылаться всем узлам, находящимся в той же сети, что и источник этого пакета. Такая рассылка называется ограниченным [широковещательным](http://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D1%80%D0%BE%D0%BA%D0%BE%D0%B2%D0%B5%D1%89%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B4%D1%80%D0%B5%D1%81) сообщением (*limited broadcast*). Если в поле номера узла назначения стоят только единицы, то пакет, имеющий такой адрес, рассылается всем узлам сети с заданным номером сети. Например, в сети *192.168.5.0* с [маской](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D0%BA%D0%B0_%D0%BF%D0%BE%D0%B4%D1%81%D0%B5%D1%82%D0%B8) *255.255.255.0* пакет с адресом*192.168.5.255* доставляется всем узлам этой сети. Такая рассылка называется широковещательным сообщением (*direct broadcast*).

IP-адрес называют *статическим* (*постоянным*, *неизменяемым*), если он назначается пользователем в настройках устройства, либо если назначается автоматически при подключении устройства к сети и не может быть присвоен другому устройству.

IP-адрес называют *динамическим* (*непостоянным*, *изменяемым*), если он назначается автоматически при подключении устройства к сети и используется в течение ограниченного промежутка времени, указанного в сервисе назначавшего IP-адрес ([DHCP](http://ru.wikipedia.org/wiki/DHCP)).

Для получения IP-адреса клиент может использовать один из следующих [протоколов](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB_%D0%BF%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85):

* [DHCP](http://ru.wikipedia.org/wiki/DHCP) ([RFC 2131](http://tools.ietf.org/html/rfc2131)) — наиболее распространённый протокол настройки сетевых параметров.
* [BOOTP](http://ru.wikipedia.org/wiki/BOOTP) ([RFC 951](http://tools.ietf.org/html/rfc951)) — простой протокол настройки сетевого адреса, обычно используется для [бездисковых станций](http://ru.wikipedia.org/wiki/%D0%91%D0%B5%D0%B7%D0%B4%D0%B8%D1%81%D0%BA%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D1%82%D0%B0%D0%BD%D1%86%D0%B8%D1%8F).
* [IPCP](http://ru.wikipedia.org/wiki/IPCP) ([RFC 1332](http://tools.ietf.org/html/rfc1332)) в рамках протокола [PPP](http://ru.wikipedia.org/wiki/PPP_(%D1%81%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB)) ([RFC 1661](http://tools.ietf.org/html/rfc1661)).
* [Zeroconf](http://ru.wikipedia.org/wiki/Zeroconf) ([RFC 3927](http://tools.ietf.org/html/rfc3927)) — протокол настройки сетевого адреса, определения имени, поиск служб.
* [RARP](http://ru.wikipedia.org/wiki/RARP) ([RFC 903](http://tools.ietf.org/html/rfc903)) Устаревший протокол, использующий обратную логику (из аппаратного адреса — в логический) популярного и поныне в широковещательных сетях протокола [ARP](http://ru.wikipedia.org/wiki/ARP). Не поддерживает распространения информации о длине маски (не поддерживает [VLSM](http://ru.wikipedia.org/wiki/VLSM)).

Адреса, используемые в локальных сетях, относят к частным. К частным относятся IP-адреса из следующих сетей:

* 10.0.0.0/8
* 172.16.0.0/12
* 192.168.0.0/16

Также для внутреннего использования:

* 127.0.0.0/8
* 169.254.0.0/16 — используется для автоматической настройки сетевого интерфейса в случае отсутствия DHCP.

Человеку запомнить числовой адрес нелегко, поэтому для удобства пользователей Интернета была введена доменная система имен, которая ставит в соответствие числовому Интернет-адресу компьютера уникальное доменное имя.

Доменная система имен имеет иерархическую структуру: домены верхнего уровня - домены второго уровня - домены третьего уровня.

Домены верхнего уровня существуют двух типов: географические и административные. Каждой стране мира выделен свой географический домен, обозначаемый двухбуквенным кодом. Например, России принадлежит географический домен ru, в котором российские организации и граждане имеют право зарегистрировать домен второго уровня.

Административные домены обозначаются тремя или более буквами и предназначены для регистрации доменов второго уровня организациями различных типов (табл. 6.2).

|  |
| --- |
| Таблица 6.2. Некоторые имена доменов верхнего уровня |
| |  |  |  |  | | --- | --- | --- | --- | | Административные | Тип организации | Географические | Страна | | com, biz | Коммерческая | са | Канада | | edu | Образовательная | de | Германия | | net | Коммуникационная | JP | Япония | | org, pro | Некоммерческая | ru | Россия | | name | Персональная | it | Италия | | museum | Музей | uk | Великобритания | |

# Основные службы Интернет.

Благодаря использованию различных сетевых протоколов Интернет может обеспечить выполнение двух основных функций:

быть средством общения между удаленными пользователями;

быть средством доступа к общим информационным ресурсам, размещенным в Интернете.

### Службы (сервисы) – это виды услуг, которые оказываются серверами сети Internet.

 Перечислим те из сервисов, которые не потеряли своей актуальности на данный момент:

* World Wide Web – всемирная паутина – служба поиска и просмотра гипертекстовых документов, включающих в себя графику, звук и видео.
* E-mail – электронная почта – служба передачи электронных сообщений.
* FTP – служба передачи файлов.
* ICQ – служба для общения в реальном времени с помощью клавиатуры.
* Telnet – служба удаленного доступа к компьютерам.

#### WWW – это распределенная информационная система с гиперсвязями, существующая на технической база всемирной компьютерной сети Internet. Данная информационная система представляет собой сеть документов, связанных между собой гиперссылками. Такие документы называются гипертекстовыми. Так как ссылки могут указывать на любой документ, находящийся в Интернете в любом месте земного шара, данная система и названа Всемирной паутиной.

Наименьшей информационной единицей WWW является Web- страница, представляющая собой совокупность текстовых, графических и мультимедийных файлов, связанных гиперссылками. Группа **Web- страниц**, принадлежащих одному владельцу и связанных между собой по содержанию, составляют **Web – сайт**. Хост - компьютер, предназначенный для хранения **Web- страниц** и **Web** – сайтов называется **Web – сервером**. Клиент-программа, предназначенная для просмотра **Web – сайтов**, называется браузером.

E-mail – это наиболее старая и одна из самых массовых служб Сети. Ее назначение — поддержка обмена электронными письмами между пользователями. По своей сущности электронная почта - это система обмена электронными сообщениями в компьютерных сетях (в режиме отложенного общения -offline). Почтовый сервер — это своеобразное почтовое отделение, куда поступает входящая и исходящая корреспонденция зарегистрированных на нем пользователей. Эта корреспонденция помещается в «почтовые ящики» пользователей — специально отведенные разделы на жестком диске. Каждый пользователь получает персональный почтовый адрес, по которому к нему будут поступать письма. Следует отметить, что адреса электронной почты несколько отличаются от других адресов Интернета, но очень похожи. Они состоят из двух частей, разделенных символом @. Справа от символа располагается Интернет-адрес компьютера, на котором располагается почтовое отделение абонента. Этот адрес формируется так же, как и любое другое доменное имя в Интернете. Слева от символа @ расположено имя абонента. Примером почтового адреса может служить: myname@mail.ru.

Для того, чтобы эта служба стала доступной пользователю, необходимо выполнить ряд процедур. Сначала пользователь должен зарегистрироваться на почтовом сервере. При этом фиксируется адрес клиента электронной почты и пароль – набор символов, позволяющий идентифицировать пользователя. Наличие пароля обеспечивает конфиденциальность данной услуги. Зарегистрированный пользователь имеет возможность написать текст письма, указав адрес получателя. Для этого в состав клиент-программы электронной почты включен редактор подготовки писем. Подготовленные письма помещаются в папку «Исходящие». Сервер, принимает все письма из папки «Исходящие» и, кроме того, передает поступившие письма, которые помещаются в папку «Входящие». Соединение пользователя с почтовым сервером необходимо только для передачи письма и доставки на компьютер пользователя писем из папки входящие. Почтовый сервер работает постоянно. Он периодически просматривает «почтовые ящики» и организует передачу по сети исходящих писем. Входящую корреспонденцию почтовый сервер раскладывает по «ящикам».

Клиент-программа электронной почты кроме функции приема-передачи писем во время сеанса связи, может выполнять еще множество сервисных функций:

· подготовка и редактирование писем,

· организация адресной книги,

· просмотр почтового архива,

· сортировка и удаление писем из почтового архива и пр.

Популярным клиентом E-mail является программа Outlook Express, входящая в стандартную поставку операционной системы MS Windows.

Cервер и клиент электронной почты работают по разным протоколам. Сервер программа POP3 (Post Office Protocol — протокол почтового отделения) кроме всего прочего выполняет функцию защиты информации. Во время сеанса связи она устанавливает личность пользователя, обеспечивает связь с его персональным ящиком. При работе клиент-программы никакого установления личности не требуется. Ее задача — передать на сервер исходящие письма и принять поступившие. Здесь используется более простой протокол SMTP (Simple Mail Transfer Protocol — простейший протокол передачи почты).

#### Служба передачи файлов

Часто эту службу называют по имени используемого протокола: FTP (File Transfer Protocol — протокол передачи файлов). Со стороны Сети работу службы обеспечивают так называемые FTP-серверы, а со стороны пользователей — FTP-клиенты. Назначение FTP-сервера — хранение набора файлов самого разнообразного назначения (обычно в архивированном виде). Чаще всего это программные файлы: средства системного и прикладного программного обеспечения. Но в наборах могут храниться файлы и любых других форматов: графические, звуковые, документы MS Word, MS Excel и пр. Вся эта информация образует иерархическую структуру папок (каталогов и подкаталогов).

После соединения FTP-клиента с сервером на экране пользователя открывается файловый интерфейс хранилища папок и файлов на сервере (наподобие Проводника Windows). Далее работа происходит так же, как с файловой системой на ПК: папки и файлы можно просматривать, сорти­ровать, копировать на свои диски. Клиент FTP входит в состав программы Internet Explorer и поэтому всегда имеется на ПК, работающем под управлением MS Windows.

# Одна из популярнейших служб Интернет - это ICQ (I seek you - я ищу тебя).

### ICQ - это способ общения в сети, который позволяет вести беседу с любым зарегистрированным в системе ICQ и подключенным в данный момент к Интернету пользователем.

     В настоящее время в системе ICQ зарегистрировано более 150 миллионов пользователей, причем каждый пользователь имеет уникальный идентификационный номер. Программа уведомляет о присутствии в данное время в Интернете абонентов из предварительно составленного списка и дает возможность инициализировать контакт с ними.

     Для того, чтобы стать абонентом системы ICQ, достатаочно скачать программу ICQ-клиент с файлового сервера (например, www.freeware.ru) и в процессе ее установки на компьютер зарегистрироваться.

     Возможности ICQ:

* отправка электронных писем и SMS-сообщений;
* обмен текстовыми сообщениями (chat);
* голосовая связь;
* поиск по интересам, по номеру, по e-mail;
* интернет-телефония (звонки с компьютера на компьютер, с компьютера на телефон, с телефона на компьютер)

и многое другое.

Данный сервис обеспечивает взаимодействие с удаленным компьютером. Он позволяет превратить компьютер пользователя в удаленный терминал другого компьютера. Поэтому данный сервис еще называют э**муляцией удаленного терминала**. Терминал от обычного компьютера отличается тем, что не выполняет собственные вычисления. Все, что вводится на клавиатуре рабочей станции, передается удаленному компьютеру, а получаемые результаты передаются обратно и выводятся на монитор рабочей станции. В качестве удаленных компьютеров, в основном, используются машины, работающие под управлением операционной системы UNIX [Юникс]. Поэтому для работы в режиме удаленного терминала требуется знание основных команд данной операционной системы. С развитием графических операционных систем, таких, как Windows, командный режим работы стал менее популярен, и **сервис Telnet** в последнее время большинство пользователей не применяют. Многие информационные системы, ранее доступные исключительно с помощью **Telnet**, сегодня доступны из Всемирной паутины, о которой речь пойдет ниже.

Установив связь с помощью **Telnet**, пользователь получает возможность работать с удаленным компьютером, как со "своим", т.е. теоретически получить в свое распоряжение все ресурсы, если к ним разрешен доступ. Реально **Telnet** предоставляет открытый доступ, но организация взаимодействия полностью определяется удаленным компьютером. Два вида услуг Internet требуют подключения к серверам через **Telnet**: библиотечные каталоги и электронные доски объявлений (BBS).

 Работа с удаленной системой может вестись в "прозрачном" режиме, когда программы на сервере и у клиента только обеспечивают протокол соединения, и в командном, когда клиент получает в свое распоряжение набор команд сервера. Следует заметить, что из соображений безопасности намечается тенденция сокращения числа узлов Internet, позволяющих использовать **Telnet** для подключения к ним.

# Инструментальные средства разработки Web-сайтов. Языки HTML, XML.

***Web-сайт***- это набор документов, хранящийся на сервере, управляемый Web-сервером и имеющий имя - адрес URL.

Веб-сайты можно создавать с помощью обычного редактора. Но это не очень удобно. Поэтому для создания сайтов используют различные инструментальные средства разработки.

Notepad++ - Отличный текстовый редактор, который можно использовать как более функциональную замену Блокноту Windows, так и в качестве редактора с языками [верстки и веб-программирования](http://www.imagecms.net/blog/obzory/veb-dizain-dlia-nachinaiushchih-sovety-po-izucheniiu): HTML, CSS, Java Script, PHP.

Работа в режиме вкладок, подсветка синтаксиса, кодировки, макросы, возможность установки дополнительных плагинов, проверка кода и сравнение файлов – эти и другие возможности делают Notepad++ действительно полезным инструментом для веб-разработчика.

Стоит отметить легкость дистрибутива и скорость работы программы.

Adobe Dreamweaver – мощное программное обеспечение для web-дизайна и создания визуальных проектов. Программа Adobe Dreamweaver помогает разрабатывать реалистичную среду для интерактивного просмотра, управлять проектами на уровне пикселов или с помощью специально созданного кода. Встроенные подсказки в Adobe Dreamweaver позволяют более точно создавать код в HTML, JavaScript, Ajax, Spry, jQuery и Prototype.

Новая версия Adobe Dreamweaver CS6 предлагает «резиновые» макеты и панель «Многоэкранный просмотр» в целях одновременного создания проектов для телефонов, планшетных устройств и компьютеров. Расширенная поддержка jQuery Mobile и Adobe PhoneGap позволяет сокращать время создания мобильных приложений.

* Поддержка CSS3/HTML5. Работа со стилями через панель CSS, в которой реализована поддержка CSS3. Создание кода с помощью технологии HTML5 с удобными подсказками по коду и визуализацией в представлении «Дизайн».

## Adobe Muse. Программное обеспечение ориентировано на работу по визуальному созданию сайтов (без необходимости написания кода).

**HTML** (от [англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *HyperText Markup Language* — «язык [гипертекстовой](http://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%BF%D0%B5%D1%80%D1%82%D0%B5%D0%BA%D1%81%D1%82) разметки») — стандартный [язык разметки](http://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%80%D0%B0%D0%B7%D0%BC%D0%B5%D1%82%D0%BA%D0%B8) документов во [Всемирной паутине](http://ru.wikipedia.org/wiki/%D0%92%D1%81%D0%B5%D0%BC%D0%B8%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D1%83%D1%82%D0%B8%D0%BD%D0%B0). Большинство [веб-страниц](http://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86%D0%B0) содержат описание разметки на языке HTML.

HTML — теговый язык разметки [документов](http://ru.wikipedia.org/wiki/HTML-%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82). Любой документ на языке HTML представляет собой набор [элементов](http://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D1%8B_HTML), причём начало и конец каждого элемента обозначается специальными пометками — [*тегами*](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%B3_(%D1%8F%D0%B7%D1%8B%D0%BA%D0%B8_%D1%80%D0%B0%D0%B7%D0%BC%D0%B5%D1%82%D0%BA%D0%B8)). Элементы могут быть *пустыми*, то есть не содержащими никакого текста и других данных (например, тег перевода строки <br>). В этом случае обычно не указывается закрывающий тег. Кроме того, элементы могут иметь *атрибуты*, определяющие какие-либо их свойства (например, размер шрифта для элемента font). Атрибуты указываются в открывающем теге.

Каждый HTML-документ, отвечающий спецификации HTML какой-либо версии, должен начинаться со строки объявления версии HTML <!DOCTYPE…>

В настоящее время [Консорциум Всемирной паутины](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%81%D0%BE%D1%80%D1%86%D0%B8%D1%83%D0%BC_%D0%92%D1%81%D0%B5%D0%BC%D0%B8%D1%80%D0%BD%D0%BE%D0%B9_%D0%BF%D0%B0%D1%83%D1%82%D0%B8%D0%BD%D1%8B) разработал [HTML версии 5](http://ru.wikipedia.org/wiki/HTML5).

В HTML5 реализовано множество новых синтаксических особенностей. Например, элементы <video>, <audio> и <canvas>, а также возможность использования [SVG](http://ru.wikipedia.org/wiki/SVG) и [математических формул](http://ru.wikipedia.org/wiki/MathML). Эти новшества разработаны для упрощения создания и управления графическими и мультимедийными объектами в сети, без необходимости использования сторонних API. Другие новые элементы, такие как <section>, <article>, <header> и <nav>, разработаны для того, чтобы обогащать[семантическое](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D0%B0%D1%83%D1%82%D0%B8%D0%BD%D0%B0) содержимое документа (страницы). Новые атрибуты были введены с той же целью, хотя ряд элементов и атрибутов был удален. Некоторые элементы, например <a>, <menu> и <cite>, были изменены, переопределены или стандартизированы. API и [DOM](http://ru.wikipedia.org/wiki/Document_Object_Model) являются фундаментальными частями спецификации HTML5[[3]](http://ru.wikipedia.org/wiki/HTML5#cite_note-HTML5diffHTML4-3). HTML5 также определяет некоторые особенности обработки ошибок вёрстки, поэтому синтаксические ошибки должны рассматриваться одинаково всеми совместимыми браузерами

**XML** (*e****X****tensible****M****arkup****L****anguage* — расширяемый язык разметки) Спецификация XML описывает XML-документы и частично описывает поведение XML-процессоров (программ, читающих XML-документы и обеспечивающих доступ к их содержимому). XML разрабатывался как язык с простым формальным [синтаксисом](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), удобный для создания и обработки документов программами и одновременно удобный для чтения и создания документов человеком, с подчёркиванием нацеленности на использование в Интернете. Язык называется расширяемым, поскольку он не фиксирует разметку, используемую в документах: разработчик волен создать разметку в соответствии с потребностями к конкретной области, будучи ограниченным лишь синтаксическими правилами языка. Сочетание простого формального синтаксиса, удобства для человека, расширяемости, а также базирование на кодировках [Юникод](http://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D0%BA%D0%BE%D0%B4) для представления содержания документов привело к широкому использованию как собственно XML, так и множества производных специализированных языков на базе XML в самых разнообразных программных средствах.

XML является подмножеством [SGML](http://ru.wikipedia.org/wiki/SGML).

**Логическая и физическая структура документа**

С логической точки зрения, документ состоит из *пролога* и ***корневого элемента***. Корневой элемент — обязательная часть документа, пролог, вообще говоря, может отсутствовать.

Пролог может включать ***объявления***, ***инструкции обработки***, ***комментарии***.

Пролог следует начинать с *объявления XML*, хотя в определённой ситуации допускается отсутствие этого объявления.

***Корневой элемент*** может включать (а может не включать) вложенные в него ***элементы*** и *символьные данные*, а также комментарии. Вложенные в корневой элемент элементы, в свою очередь, могут включать вложенные в них элементы, символьные данные и комментарии, и так далее.

Элементы документа должны быть *правильно вложены*: любой элемент, начинающийся внутри другого элемента (то есть любой элемент документа, кроме корневого), должен заканчиваться внутри элемента, в котором он начался.

Символьные данные могут встречаться внутри элементов как непосредственно так и в специальных *секциях CDATA*.

Объявления, инструкции обработки и элементы могут иметь связанные с ними *атрибуты*. Атрибуты ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *attribute*) используются для связывания с логической единицей текста пар имя-значение.

С физической точки зрения, символы, составляющие документ, делятся на ***разметку*** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *markup*) и ***символьные данные*** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *character data*).

Объявления и инструкции обработки, включая их атрибуты, а также комментарии полностью принадлежат разметке. К разметке также принадлежат теги, обозначающие границы элементов, включая их атрибуты, а также последовательности символов, обрамляющие секции CDATA.

Часть документа, не принадлежащая разметке, составляет символьные данные документа.

Разметка всегда начинается символом < и заканчивается символом >. Наряду с символами < и >, специальную роль для разметки играет также символ &. Употребление разметочных символов в символьных данных затрудняет распознавание конструкций разметки и может создать проблему неоднозначности структуры. В XML эта проблема решается следующим образом: три упомянутые символа не могут присутствовать в символьных данных и в значениях атрибутов в их непосредственном виде, для их представления в этих случаях зарезервированы специальные последовательности символов, начинающиеся с & и заканчивающиеся ;

|  |  |
| --- | --- |
| **Символ** | **Замена** |
| **<** | &lt; |
| **>** | &gt; |
| **&** | &amp; |

Кроме того, для употребления апострофов и кавычек внутри значений атрибутов используются следующие замены:

|  |  |
| --- | --- |
| **'** | &apos; |
| **"** | &quot; |

Правило замены символов, используемых в разметке, на зарезервированные последовательности символов не распространяется на символьные данные в секциях CDATA.

Объявление XML объявляет версию языка, на которой написан документ. Поскольку интерпретация содержимого документа, вообще говоря, зависит от версии языка, то Спецификация предписывает начинать документ с объявления XML. В первой (1.0) версии языка использование объявления не было обязательным, в последующих версиях оно обязательно. Таким образом, версия языка определяется из объявления, и если объявление отсутствует, то принимается версия 1.0.

Кроме версии XML, объявление может также содержать информацию о *кодировке* документа.

Пример:

**<?xml** version="1.1" encoding='UTF-8' **?>**

или

**<?xml** version="1.0" encoding="windows-1251"**?>**

**Комментарий**

Комментарии не относятся к символьным данным документа. Комментарий начинается последовательностью «<!--» и заканчивается последовательностью «-->», внутри не может встречаться комбинация символов «--». Символ & не используется внутри комментария в качестве разметки.

Пример:*<!-- это комментарий -->*

**Элемент и его разметка**

Элемент ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *element*) является понятием логической структуры документа. Каждый документ содержит один или несколько элементов. Границы элементов представлены *начальным*и *конечным* *тегами*. Имя элемента в начальном и конечном тегах элемента должно совпадать. Элемент может быть также представлен тегом *пустого*, то есть не включающего в себя другие элементы и символьные данные, *элемента*.

Тег ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *tag*) — конструкция разметки, которая содержит имя элемента.

Начальный тег: <element1>

Конечный тег: </element1>

Тег пустого элемента: <empty\_element1 />

В элементе атрибуты могут использоваться только в начальном теге и теге пустого элемента.

# Основные понятия и проблемы защиты информации (угрозы, требования, критерии, способы, средства).

Современные методы обработки, передачи и накопления информации способствовали появлению угроз, связанных с возможностью потери, искажения и раскрытия данных, адресованных или принадлежащих конечным пользователям. Поэтому обеспечение информационной безопасности компьютерных систем и сетей является одним из ведущих направлений развития ИТ.

Защита информации — это деятельность по предотвращению утечки защищаемой информации, несанкционированных и непреднамеренных воздействий на защищаемую информацию.  
Объект защиты — информация, носитель информации или информационный процесс, в отношении которых необходимо обеспечивать защиту в соответствии с поставленной целью защиты информации.

типы угроз информации:

преднамеренные воздействия

перехват (в случае реализации данного вида угроз становится возможным копирование, чтение, разглашение или использование сведений закрытого характера);

хищение (в этом случае злоумышленнику представляются широчайшие возможности осуществления своих намерений);

модификация (угрозы данного типа осуществляются путем несанкционированного доступа и чреваты уничтожением информации или ея фальсификацией);

разрушение (угрозы данного типа имеют, как правило, одноразовый характер проявления);

непреднамеренные воздействия отказы и сбои устройств или программ; угрозы, проявляемые в результате неаккуратности или непрофессионализма сотрудников при обработке, хранении или передаче информации; стихийные бедствия.

На практике защита информации представляет собой комплекс регулярно используемых средств и методов, принимаемых мер и осуществляемых мероприятий с целью систематического обеспечения требуемой надежности информации, генерируемой, хранящейся и обрабатываемой на каком-либо объекте какой-либо информационной системы, а также передаваемой по каким-либо каналам. Защита должна носить системный характер, то есть для получения наилучших результатов все разрозненные виды защиты информации должны быть объединены в одно целое и функционировать в составе единой системы, представляющей собой слаженный механизм взаимодействующих элементов, предназначенных для выполнения определенных задач по обеспечению информационной безопасности. Иначе говоря, при организации обеспечения информационной безопасности закономерно возникает необходимость в создании комплексной системы защиты информации, включающей в себя:

организационную защиту информации;

правовую защиту информацию;

программно-аппаратную защиту информации;

инженерно-техническую защиту информации.

Качество (надежность и эффективность) защиты зависит не только от видов составляющих системы, но и от их полноты, которая может быть обеспечена только при учете всех факторов и обстоятельств, сопутствующих функционированию информационной системы. Поэтому комплексная система защиты информации призвана объединить логические и технологические составляющие защиты, учитывая все факторы, которые оказывают или могут оказывать влияние на качество защиты.

Более того, комплексная система защиты информации предназначена обеспечивать, с одной стороны, организацию и обеспечение надежных механизмов защиты, а с другой - управление механизмами защиты информации. В связи с этим, кроме принятия организационных мер и внедрения программно-аппаратных и инженерно-технических средств, руководством должна предусматриваться организация четкой и отлаженной системы управления защитой информации на предприятии.

В настоящее время в США разработан стандарт оценок безопасности компьютерных систем – критерии оценок пригодности. В нем учитываются четыре типа требований к компьютерным системам:

·     требования к проведению политики безопасности – security policy;

·     ведение учета использования компьютерных систем – accounts;

·     доверие к компьютерным системам;

·     требования к документации.

Требования к проведению последовательной политики безопасности и ведение учета использования компьютерных систем зависят друг от друга и обеспечиваются средствами, заложенными в систему, т.е. решение вопросов безопасности включается в программные и аппаратные средства на стадии проектирования.

Нарушение доверия к компьютерным системам, как правило, бывает вызвано нарушением культуры разработки программ: отказом от структурного программирования, неисключением заглушек, неопределенным вводом и т.д. Для тестирования на доверие нужно знать архитектуру приложения, правила устойчивости его поддержания, тестовый пример.

Требования к документации означают, что пользователь должен иметь исчерпывающую информацию по всем вопросам. При этом документация должна быть лаконичной и понятной.

Методы обеспечения безопасности информации в ИС:

·   препятствие;

·   управление доступом;

·   механизмы шифрования;

·   противодействие атакам вредоносных программ;

·   регламентация;

·   принуждение;

·   побуждение.

Препятствие – метод физического преграждения пути злоумышленнику к защищаемой информации (к аппаратуре, носителям информации и т.д.).

Управление доступом – методы защиты информации регулированием использования всех ресурсов ИС и ИТ. Эти методы должны противостоять всем возможным путям несанкционированного доступа к информации.

Управление доступом включает следующие функции зашиты:

·    идентификацию пользователей, персонала и ресурсов системы (присвоение каждому объекту персонального идентификатора);

·    опознание (установление подлинности) объекта или субъекта по предъявленному им идентификатору;

·    проверку полномочий (проверка соответствия дня недели, времени суток, запрашиваемых ресурсов и процедур установленному регламенту);

·    разрешение и создание условий работы в пределах установленного регламента;

·    регистрацию (протоколирование) обращений к защищаемым ресурсам;

·    реагирование (сигнализация, отключение, задержка работ, отказ в запросе и т.п.) при попытках несанкционированных действий.

Механизмы шифрования – криптографическое закрытие информации. Эти методы защиты все шире применяются как при обработке, так и при хранении информации на магнитных носителях. При передаче информации по каналам связи большой протяженности этот метод является единственно надежным.

Противодействие атакам вредоносных программ предполагает комплекс разнообразных мер организационного характера и исполь­зование антивирусных программ. Цели принимаемых мер – это уменьшение вероятности инфицирования АИС, выявление фактов заражения системы; уменьшение последствий информационных инфекций, локализация или уничтожение вирусов; восстановление информации в ИС. Овладение этим комплексом мер и средств требует знакомства со специальной литературой.

Регламентация – создание таких условий автоматизированной обработки, хранения и передачи защищаемой информации, при кото­рых нормы и стандарты по защите выполняются в наибольшей степени.

Принуждение – метод защиты, при котором пользователи и персонал ИС вынуждены соблюдать правила обработки, передачи и использования защищаемой информации под угрозой материальной, административной или уголовной ответственности.

Побуждение – метод защиты, побуждающий пользователей и персонал ИС не нарушать установленные порядки за счет соблюдения сложившихся моральных и этических норм.

Вся совокупность технических средств подразделяется на аппаратные и физические.

Аппаратные средства – устройства, встраиваемые непосредственно в вычислительную технику, или устройства, которые сопрягаются с ней по стандартному интерфейсу.

Физические средства включают различные инженерные устройства и сооружения, препятствующие физическому проникновению злоумышленников на объекты защиты и осуществляющие защиту персонала (личные средства безопасности), материальных средств и финансов, информации от противоправных действий. Примеры физических средств: замки на дверях, решетки на окнах, средства электронной охранной сигнализации и т.п.

Программные средства – это специальные программы и программные комплексы, предназначенные для защиты информации в ИС. Как отмечалось, многие из них слиты с ПО самой ИС.

Из средств ПО системы защиты выделим еще программные средства, реализующие механизмы шифрования (криптографии). Криптография – это наука об обеспечении секретности и/или аутентичности (подлинности) передаваемых сообщений.

Организационные средства осуществляют своим комплексом регламентацию производственной деятельности в ИС и взаимоотношений исполнителей на нормативно-правовой основе таким образом, что разглашение, утечка и несанкционированный доступ к конфиденциальной информации становится невозможным или существенно затрудняется за счет проведения организационных мероприятий. Комплекс этих мер реализуется группой информационной безопасности, но должен находиться под контролем первого руководителя.

Законодательные средства защиты определяются законодательными актами страны, которыми регламентируются правила пользования, обработки и передачи информации ограниченного доступа и устанавливаются меры ответственности за нарушение этих правил.

Морально-этические средства защиты включают всевозможные нормы поведения (которые традиционно сложились ранее), складываются по мере распространения ИС и ИТ в стране и в мире или специально разрабатываются. Морально-этические нормы могут быть неписаные (например честность) либо оформленные в некий свод (устав) правил или предписаний. Эти нормы, как правило, не являются законодательно утвержденными, но поскольку их несоблюдение приводит к падению престижа организации, они считаются обязательными для исполнения. Характерным примером таких предписаний является Кодекс профессионального поведения членов Ассоциации пользователей ЭВМ США.

# Методы построения систем защиты информации (аппаратные, программные, организационные и др. аспекты).

Защита информации должна основываться на следующих основных принципах:

системности;

комплексности;

непрерывности защиты;

разумной достаточности;

гибкости управления и применения;

открытости алгоритмов и механизмов защиты;

простоты применения защитных мер и средств.

Принцип системности

Системный подход к защите компьютерных систем предполагает необходимость учета всех взаимосвязанных, взаимодействующих и изменяющихся во времени элементов, условий и факторов, существенно значимых для понимания и решения проблемы обеспечения безопасности АС.

При создании системы защиты необходимо учитывать все слабые, наиболее уязвимые места системы обработки информации, а также характер, возможные объекты и направления атак на систему со стороны нарушителей (особенно высококвалифицированных злоумышленников), пути проникновения в распределенные системы и НСД к информации. Система защиты должна строиться с учетом не только всех известных каналов проникновения и НСД к информации, но и с учетом возможности появления принципиально новых путей реализации угроз безопасности.

Принцип комплексности

В распоряжении специалистов по компьютерной безопасности имеется широкий спектр мер, методов и средств защиты компьютерных систем. Комплексное их использование предполагает согласованное применение разнородных средств при построении целостной системы защиты, перекрывающей все существенные каналы реализации угроз и не содержащей слабых мест на стыках отдельных ее компонентов. Защита должна строиться эшелонированно. Внешняя защита должна обеспечиваться физическими средствами, организационными и правовыми мерами. Одной из наиболее укрепленных линий обороны призваны быть средства защиты, реализованные на уровне операционных систем (ОС) в силу того, что ОС - это как раз та часть компьютерной системы, которая управляет использованием всех ее ресурсов. Прикладной уровень защиты, учитывающий особенности предметной области, представляет внутренний рубеж обороны.

Принцип непрерывности защиты

Защита информации - это не разовое мероприятие и даже не определенная совокупность проведенных мероприятий и установленных средств защиты, а непрерывный целенаправленный процесс, предполагающий принятие соответствующих мер на всех этапах жизненного цикла АС, начиная с самых ранних стадий проектирования, а не только на этапе ее эксплуатации.

Разработка системы защиты должна вестись параллельно с разработкой самой защищаемой системы. Это позволит учесть требования безопасности при проектировании архитектуры и, в конечном счете, позволит создать более эффективные (как по затратам ресурсов, так и по стойкости) защищенные системы.

Большинству физических и технических средств защиты для эффективного выполнения своих функций необходима постоянная организационная (административная) поддержка (своевременная смена и обеспечение правильного хранения и применения имен, паролей, ключей шифрования, переопределение полномочий и т.п.). Перерывы в работе средств защиты могут быть использованы злоумышленниками для анализа применяемых методов и средств защиты, для внедрения специальных программных и аппаратных "закладок" и других средств преодоления системы защиты после восстановления ее функционирования.

Разумная достаточность

Создать абсолютно непреодолимую систему защиты принципиально невозможно. При достаточном количестве времени и средств можно преодолеть любую защиту. Поэтому имеет смысл вести речь только о некотором приемлемом уровне безопасности. Высокоэффективная система защиты стоит дорого, использует при работе существенную часть мощности и ресурсов компьютерной системы и может создавать ощутимые дополнительные неудобства пользователям. Важно правильно выбрать тот достаточный уровень защиты, при котором затраты, риск и размер возможного ущерба были бы приемлемыми (задача анализа риска).

Гибкость системы защиты

Часто приходится создавать систему защиты в условиях большой неопределенности. Поэтому принятые меры и установленные средства защиты, особенно в начальный период их эксплуатации, могут обеспечивать как чрезмерный, так и недостаточный уровень защиты. Естественно, что для обеспечения возможности варьирования уровнем защищенности, средства защиты должны обладать определенной гибкостью. Особенно важным это свойство является в тех случаях, когда установку средств защиты необходимо осуществлять на работающую систему, не нарушая процесса ее нормального функционирования. Кроме того, внешние условия и требования с течением времени меняются. В таких ситуациях свойство гибкости спасает владельцев АС от необходимости принятия кардинальных мер по полной замене средств защиты на новые.

Открытость алгоритмов и механизмов защиты

Суть принципа открытости алгоритмов и механизмов защиты состоит в том, что защита не должна обеспечиваться только за счет секретности структурной организации и алгоритмов функционирования ее подсистем. Знание алгоритмов работы системы защиты не должно давать возможности ее преодоления (даже автору). Однако, это вовсе не означает, что информация о конкретной системе защиты должна быть общедоступна.

Принцип простоты применения средств защиты

Механизмы защиты должны быть интуитивно понятны и просты в использовании. Применение средств защиты не должно быть связано со знанием специальных языков или с выполнением действий, требующих значительных дополнительных трудозатрат при обычной работе законных пользователей, а также не должно требовать от пользователя выполнения рутинных малопонятных ему операций (ввод нескольких паролей и имен и т.д.).

Комплексный (системный) подход к построению любой системы включает в себя:

изучение объекта внедряемой системы

оценку угроз безопасности объекта

анализ средств, которыми будем оперировать при построении системы

оценку экономической целесообразности

изучение самой системы, ее свойств, принципов работы и возможность увеличения ее эффективности

соотношение всех внутренних и внешних факторов

возможность дополнительных изменений в процессе построения системы и полную организацию всего процесса от начала до конца

Комплексный (системный) подход — это принцип рассмотрения проекта, при котором анализируется система в целом, а не ее отдельные части. Его задачей является оптимизация всей системы в совокупности, а не улучшение эффективности отдельных частей. Это объясняется тем, что, как показывает практика, улучшение одних параметров часто приводит к ухудшению других, поэтому необходимо стараться обеспечить баланс противоречий требований и характеристик. Комплексный (системный) подход не рекомендует приступать к созданию системы до тех пор, пока не определены следующие ее компоненты:

Входные элементы. Это те элементы, для обработки которых создается система. В качестве входных элементов выступают виды угроз безопасности, возможные на данном объекте

Ресурсы. Это средства, которые обеспечивают создание и функционирование системы (например, материальные затраты, энергопотребление, допустимые размеры и т. д.)

Окружающая среда. Следует помнить, что любая реальная система всегда взаимодействует с другими системами, каждый объект связан с другими объектами. Очень важно установить границы области других систем, не подчиняющихся руководителю данного предприятия и не входящих в сферу его ответственности. Характерным примером важности решения этой задачи является распределение функций по защите информации, передаваемой сигналами в кабельной линии, проходящей по территориям различных объектов. Как бы ни устанавливались границы системы, нельзя игнорировать ее взаимодействие с окружающей средой, ибо в этом случае принятые решения могут оказаться бессмысленными. Это справедливо как для границ защищаемого объекта, так и для границ системы защиты.

Назначение и функции. Для каждой системы должна быть сформулирована цель, к которой она (система) стремится. Эта цель может быть описана как назначение системы, как ее функция. Чем точнее и конкретнее указано назначение или перечислены функции системы, тем быстрее и правильнее можно выбрать лучший вариант ее построения. Так, например, цель, сформулированная в самом общем виде как обеспечение безопасности объекта, заставит рассматривать варианты создания глобальной системы защиты. Если уточнить ее, определив, например, как обеспечение безопасности информации, передаваемой по каналам связи внутри здания, то круг возможных решений существенно сузится. Следует иметь в виду, что, как правило, глобальная цель достигается через достижение множества менее общих локальных целей (подцелей). Построение такого «дерева целей» значительно облегчает, ускоряет и удешевляет процесс создания системы;

Критерий эффективности. Необходимо всегда рассматривать несколько путей, ведущих к цели, в частности нескольких вариантов построения системы, обеспечивающей заданные цели функционирования. Для того чтобы оценить, какой из путей лучше, необходимо иметь инструмент сравнения — критерий эффективности. Он должен: характеризовать качество реализации заданных функций; учитывать затраты ресурсов, необходимых для выполнения функционального назначения системы; иметь ясный и однозначный физический смысл; быть связанным с основными характеристиками системы и допускать количественную оценку на всех этапах создания системы.

При построении любой системы необходимо определить принципы, в соответствии с которыми она будет построена. Комплексной системы защиты информации (КСЗИ) — сложная система, функционирующая, как правило, в условиях неопределенности, требующая значительных материальных затрат. Поэтому определение основных принципов КСЗИ позволит определить основные подходы к ее построению:

принцип законности заключается в соответствии принимаемых мер законодательству РФ о защите информации, а в случае отсутствия соответствующих законов — другим государственным нормативным документам по защите. В соответствии с принципом полноты защищаемой информации защите подлежит не только информация, составляющая государственную, коммерческую или служебную тайну, но и та часть несекретной информации, утрата которой может нанести ущерб ее собственнику либо владельцу. Реализация этого принципа позволяет обеспечить и охрану интеллектуальной собственности.

принцип обоснованности защиты информации заключается в установлении путем экспертной оценки целесообразности засекречивания и защиты той или другой информации, вероятных экономических и других последствий такой защиты исходя из баланса жизненно важных интересов государства, общества и граждан. Это, в свою очередь, позволяет расходовать средства на защиту только той информации, утрата или утечка которой может нанести действительный ущерб ее владельцу.

принцип создания специализированных подразделений по защите информации заключается в том, что такие подразделения являются непременным условием организации комплексной защиты, поскольку только специализированные службы способны должным образом разрабатывать и внедрять защитные мероприятия и осуществлять контроль за их выполнением.

принцип участия в защите информации всех соприкасающихся с нею лиц исходит из того, что защита информации является служебной обязанностью каждого лица, имеющего по роду выполняемой работы отношение к защищаемой информации, и такое участие дает возможность повысить качество защиты.

принцип персональной ответственности за защиту информации требует, чтобы каждое лицо персонально отвечало за сохранность и неразглашение вверенной ему защищаемой информации, а за утрату или распространение такой информации оно несет уголовную, административную или иную ответственность.

принцип наличия и использования всех необходимых сил и средств заключается в том, что КСЗИ требует, во-первых, участия в ней руководства предприятия и специальной службы защиты информации; во-вторых, использования различных организационных форм и методов защиты; в-третьих, наличия необходимых материально-технических ресурсов, включая технические средства защиты.

принцип превентивности предполагает заблаговременное принятие мер по защите информации. Из этого принципа вытекает, в частности, необходимость разработки защищенных информационных технологий.

Среди рассмотренных принципов едва ли можно выделить более или менее важные. А при построении комплексной системы защиты информации (КСЗИ) важно использовать их в совокупности.