1. Понятие архитектуры ВС и общие требования, предъявляемые к ней.

Термин "архитектура", применительно к вычислительным системам, употребляют широком и узком смыслах.

В узком смысле архитектура понимается как архитектура набора команд, которая служит границей между аппаратным и программным обеспечением и представляет собой ту часть системы, с которой взаимодействуют системные программисты.

В широком смысле термин архитектура охватывает понимание организации системы, включающие высокоуровневые аспекты разработки компьютера, такие как система памяти, структура системных шин, организация ввода и вывода, вплоть до физических уровней, т.е. уровня электрических импульсов и включая архитектуру программных средств.

Применительно к вычислительным системам термин архитектура может быть определён как распределение функций между различными уровнями системы или определение границ между этими уровнями, т.е. архитектура определяет многоуровневую систему.

Появление серийно выпускаемых сверхбольших надежных и дешевых интегральных схем, массовое производство микропроцессоров, возобновившийся интерес к разработке языков программирования и программного обеспечения порождают возможность при проектировании компьютеров качественно продвинуться вперед за счет улучшения программно аппаратного интерфейса, т. е. семантической связи между возможностями аппаратных средств современных ЭВМ и их программного обеспечения. Организация вычислительной системы (ВС) на этом уровне лежит в основе понятия "архитектура". Для неспециалистов в области программного обеспечения термин "архитектура" ассоциируется, как правило, со строительными объектами. И здесь, как увидим далее, есть много общего.

Действительно, архитектура компьютера, характеризующая его логическую организацию, может быть представлена как множество взаимосвязанных компонент, включающих, на первый взгляд, элементы различной природы: программное обеспечение (software), аппаратное обеспечение (hardware), алгоритмическое обеспечение (brainware), специальное фирменное обеспечение (firmware) – и поддерживающих его слаженное функционирование в форме единого архитектурного ансамбля, позволяющего вести эффективную обработку различных объектов.

С другой стороны, архитектура может быть задана как абстрактное многоуровневое представление физической системы с точки зрения программиста, с закреплением функций за каждым уровнем и установлением интерфейса между различными уровнями.

Знание особенностей разнообразных архитектурных решений дает возможность пользователям компьютеров эффективно распоряжаться всеми предоставляемыми ресурсами, осуществляя их направленный выбор и тем самым повышая эффективность обработки данных.

Архитектура как набор взаимодействующих компонент

Ранее область применения вычислительных систем определялась ее быстродействием. Однако существует достаточно большое количество ВС, обладающих равным быстродействием, но имеющих совершенно разные способы представления данных, методы организации памяти, режимы работы, системы команд, набор ВнУ и т. д. Таким образом, ВС имеет, кроме быстродействия, ряд других характеристик, необычайно важных в той или иной области применения. Это стало особенно заметно при переходе к ВС четвертого и пятого поколений. Совокупность таких характеристик и легла в основу понятия архитектуры ВС.

Архитектура ВС определяет основные функциональные возможности системы, сферу применения (научно техническая, экономическая, управление и т. д.), режим работы (пакетный, мультипрограммный, разделения времени, диалоговый и т. д.), характеризует параметры ВС (быстродействие, набор и объем памяти, набор периферийных устройств и т. д.), особенности структуры (одно , многопроцессорная) и т. д. Составные части понятия "архитектура" можно определить следующей схемой (рис. 1.1).

Рис. 1.1. Функциональные возможности ВС

Вычислительные и логические возможности ВС. Они обусловливаются системой команд (СК), характеризующей гибкость программирования, форматами данных и скоростью выполнения операций, определяющих класс задач, наиболее эффективно решаемых на ВС. Система команд ВС, базирующихся на архитектуре фон Неймана, сегодня мало чем отличается от СК ЭВМ 50 х годов. Большинство достижений в этой области остались незамеченными проектировщиками и соответственно не нашли адекватного воплощения в архитектуре современных компьютеров.

Анализ показывает, что в различных программах чаще всего встречаются достаточно простые команды: команды пересылки и команды процессора с использованием регистров и простых режимов адресации. Не нашли широкого применения и нетрадиционные способы кодирования данных, несмотря на значительные возможности их в плане разработки быстродействующих алгоритмов арифметических операций. Среди них знакоразрядные системы, системы в коде вычетов и др.

Рассмотрим структуру системы команд в зависимости от класса решаемых задач (рис. 1.2).

К командам управления мы относим команды ввода-вывода данных и команды управления состоянием процессора, памяти и каналов.

Как видно из рис. 1.2, для решения задач любого класса необходимы команды типов 2 и 3. Следовательно, эти типы команд должны присутствовать в любом компьютере.

Рис. 1.2. Классификация СК по назначению

Большое влияние на точность выполнения операций оказывают форматы данных. Современные компьютеры имеют развитую систему форматов. Например, компьютеры фирм ЕС ЭВМ и IВМ имеют форматы в 2, 4, 8 и 16 байт.

Алгоритмы выполнения операций достаточно полно отражают производительность только однопроцессорных ВС.

Аппаратные средства. Простейшая ВС включает модули пяти типов: центральный процессор, основная память, каналы, контроллеры и внешние устройства.

Процессор (УУ + АЛУ + память) управляет работой системы и обеспечивает вычисления непосредственно по программе. Выполнение машинных команд, команд ввода-вывода (I/О), обращение к памяти, управление состоянием устройств инициализируются или выполняются с помощью процессора.

Основная память предназначена для хранения команд и данных и обеспечивает адресный доступ к ним от процессора. Современная память работает со скоростью, близкой к скорости работы процессора.

Каналы – спецустройства, управляющие обменом данных с внешними устройствами. Каналы инициируют свою работу с помощью процессора и затем переходят в автономный режим работы. Это, по сути, спецпроцессор ввода – вывода, обеспечивающий работу внешних устройств, контроль информации и т. д.

Контроллеры ввода-вывода служат для подсоединения внешних устройств (ВнУ) к каналам и обеспечивают обмен управляющей информацией с внешними устройствами, присвоение приоритетов и выдачу информации о состоянии ВнУ для канала, т. е. это устройства управления ВнУ.

ВнУ служат для ввода-вывода информации с различных носителей.

Память может быть организована как многоуровневая с различным объемом и временем доступа к ней – сверхоперативная (СОЗУ), оперативная (ОП), внешняя (ВнП) (рис. 1.3), так и одноуровневая, виртуальная. Почти всегда виртуальная память есть переупорядоченное подмножество реальной памяти.

Рис. 1.3. Типы памяти (V – объем, S – быстродействие)

Уровни иерархии памяти взаимосвязаны между собой: все данные одного уровня могут быть найдены на более низком уровне.

Успешное или неуспешное обращение к уровню памяти называют соответственно попаданием (hit) или промахом (miss), а соответствующее время – временем обращения (hit time или miss penalty).

Существенное влияние на производительность ВС оказывают каналы ввода-вывода. Мультиплексный канал обеспечивает работу группы медленных устройств, блок-мультиплексный – группы быстрых устройств, селекторный – монополизирует информационную магистраль только одним быстродействующим устройством.

Для повышения пропускной способности каналов используют некоторые дополнительные меры, например буферизацию ВнУ путем введения памяти в состав самого устройства или контроллера.

Аппаратные средства защиты памяти служат для управления доступом к различным областям памяти в соответствии с имеющимися у пользователя полномочиями.

Программное обеспечение. Оно является составной частью архитектуры компьютера и существенно влияет на весь вычислительный процесс, в частности позволяет эффективно эксплуатировать аппаратные средства системы.

Операционная система (ОС) управляет ресурсами, разрешает конфликтные ситуации, оптимизирует функционирование системы в целом.

Широкий спектр языков программирования позволяет описывать практически любые задачи, а разнообразие компиляторов – их эффективно реализовывать.

Роль прикладного программного обеспечения (ПО) необычайно велика для решения тематических задач.

Фон-Неймановская архитектура.

Архитектурные принципы фон Нейманом формулировались применительно к созданию автоматического устройства для решения дифференциальных уравнений.

Основные характеристики архитектуры фон Неймановского типа следующие:

• последовательно адресуемая единственная память линейного типа для хранения программ и данных;

• команды и данные различаются через идентификатор неявным способом лишь при выполнении операций. Принимаемые по умолчанию соглашения типа: операнды операции умножения – это данные, а объект, на который указывает команда перехода – это команда, позволяют обращаться с командой как с данными, например, для ее модификации;

• назначение данных определяется лишь логикой программы, так как в памяти машины набор бит может представлять собой как десятичное число с фиксированной точкой, так и строку символов.

Указанные свойства были исключительно важными для своего времени. Однако появление языков высокого уровня (ЯВУ), новых методов решения, логических способов ускорения операций, более совершенной элементной базы требует наряду с имеющимися возможностями архитектуры и принципиально новых. Среди них требования ЯВУ имеют следующие особенности:

• память состоит из набора дискретных именуемых переменных. Вовсе не требуется, например, чтобы память для значений переменных одной программы располагалась рядом с памятью для значений переменных другой программы. Таким образом, принцип единственной последовательной памяти имеет мало общего с организацией памяти в ЯВУ;

• ЯВУ наряду с линейными данными оперируют и с многомерными: массивами, структурами, списками;

• в ЯВУ четко разграничены операции и данные;

• данные определяют и операции над ними.

Например, смысл операции A + C определяется описанием A и С. Cемантика операции "+" совершенно различна, например, для целых чисел и символьных переменных.

Архитектура фон Неймана плохо ориентирована на выполнение программ на ЯВУ. Действительно,

• объем кодов, генерируемых компилятором, из за несоответствия требуемой ЯВУ и предлагаемой архитектурой организации памяти значительно превосходит необходимый объем для решения запрограммированной задачи;

• примитивность выполняемых операций в объектном коде требует сложной работы компилятора.

К сожалению, ЯВУ имитируют в своей структуре архитектуру фон Неймановского типа: переменные – пассивное ЗУ; оператор присваивания – арифметическое устройство (АУ); последовательное выполнение операторов управления (IF, GOTO) – устройство управления (УУ).

Много сложностей из за работы компьютера в двоичной системе. Как отразится на экране дисплея вводимое десятичное вещественное число? Не будет ли потерь в младшем разряде?

Язык АДА, особенно беспокоящийся о точности вычислений, также не ушел от аппроксимации десятичных вещественных чисел двоичными.

Общая взаимосвязь между ЯВУ и ЭВМ в зависимости от уровня языка машины может быть представлена следующей схемой (рис. 1.5).

Введение программно адресуемых регистров (так называемых регистров общего назначения) существенно увеличивает количество используемых команд LOAD (загрузка в регистр) и STORE (запись в память), т. е. команд перемещения данных из регистров в память и обратно. Исследования на компьютере PDP для различных ЯВУ показывают, что 42 % всех выполняемых команд затрачивается на перемещение данных между памятью и регистрами. Имеются исследования, утверждающие, что использование кэш-памяти позволяет достичь увеличения быстродействия за счет использования регистров общего назначения.

В настоящее время имеются конкретные проекты архитектур машин, ориентированные на ЯВУ: ПАСКАЛЬ, PL, ЛИСП, КОБОЛ.

Однако больший интерес представляют архитектуры, ориентированные не на конкретный язык, а на общие семантические возможности некоторой группы языков. Так, архитектура машины SWARD ориентирована на языки PL/1, АДА, ФОРТРАН, КОБОЛ и некоторые другие, создавая тем самым благоприятную среду для программиста.

Компьютеры фирмы Burroughs, начиная с модели B 1700, за счет вызова различных микропрограмм динамически настраиваются на язык написания выполняемой программы.

Рис. 1.5. Соотношение программ на ЯВУ и машинном языке:

1 – это традиционный подход. После компилирования программа переводится на машинный язык, а затем интерпретируется машиной; 2 – компиляция идет на машинный язык более высокого уровня, сокращая тем самым семантический разрыв между ЯВУ и машиной; 3 – здесь ЯВУ можно рассматривать как язык ассемблера, т. е. имеется взаимно однозначное соответствие между типами операторов и знаков операций ЯВУ с командами машинного языка. Здесь идет ассемблирование, а не компилирование, во время которого удаляются комментарии и пробелы в исходной программе, преобразуются разделители, ключевые слова и знаки операций в машинные коды, имена – в адреса полей памяти. Таким образом, многих привычных функций компилятора здесь нет. Остальная привязка программы к ЭВМ происходит перед выполнением программы; 4 – здесь машинный язык является ЯВУ и идет процесс интерпретации программы на компьютере.

Отметим, что разработчику архитектуры ЭВМ не следует строго ориентироваться на языки программирования, компиляторы, операционные системы и другие программы, чтобы не потерять главное в архитектуре – ориентацию на современные требования задач и окружение пользователя.

Необходимо искать разумный компромисс в распределении функций системы между аппаратной и программной реализацией с учетом стоимости, возможности модификации, работы на микроуровне и т. д. Необходимо помнить, что компьютер создается для эффективного выполнения программ.

**1. Понятие «базы данных». Основные компоненты базы данных**

Под базой данных обычно понимается именованная совокупность данных, отображающая состояние объектов и их отношений в рассматриваемой предметной области. Характерной чертой баз данных является постоянство: данные постоянно накапливаются и используются; состав и структура данных, необходимых для решения тех или иных прикладных задач, обычно постоянны и стабильны во времени; отдельные или даже все элементы данных могут меняться – но это и есть проявление постоянства – постоянная актуальность.

Основные компоненты БД:

*Поле* - это минимальный элемент базы данных, содержащий один неделимый квант информации.Каждое поле характеризуется *именем* и *типом* хранящихся в нем данных.

*Запись* - это совокупность нескольких разнородных полей, описывающая некоторую сущность предметной области.

*Таблица базы данных* - это набор однородных записей.Таблица позволяет читать, изменять, добавлять и удалять записи, а также сортировать их по определенному условию и осуществлять поиск по заданным значениям.

Также компонентами БД явл. Триггеры, Индексы, ХП, Генераторы, Схемы.

*Схема базы данных* – это описание базы данных в терминах конкретной модели данных.*Триггеры –* активный объект БД, который имеет возможность срабатывать на некоторое событие, выполняет заданные действия.

*Генераторы –* объекты БД, которые предназначны для получения последовательностей уникальных чисел. *ХП –* объекты БД, которые представляют собой некоторое количество SQL операторов с возможностью использования дополнительных конструкций, объед. в единый логический участок кода.

**3. Базовые и улучшенные сортировки на основе выбора, включения и обмена. Сравнительный анализ алгоритмов сортировки.**

**Сортировка вставками**

Этот метод обычно используют игроки в карты. Элементы (карты) условно разделяются на готовую последовательность а1,…аi-1 и входную последовательность аi,…аn. На каждом шаге, начиная с i=2 и увеличивая i на единицу, берут 1-й элемент входной последовательности и передают его в готовую последовательность, вставляя на подходящее место.

**Алгоритм сортировки вставками**

1. Запоминаем элемент, подлежащий вставке.

2. Перебираем справа налево отсортированные элементы и сдвигаем каждый элемент вправо на одну позицию, пока не освободится место для вставляемого элемента.

3. Вставляем элемент на освободившееся место. Пункты 1-3 выполняем для всех элементов массива, кроме первого.

При поиске подходящего места для элемента x удобно чередовать сравнения и пересылки, т. е. как бы "просеивать"x, сравнивая его с очередным элементом m[j] и либо вставляя x, либо пересылая m[j] вправо. Просеивание может быть закончено при двух различных условиях:

1) найден элемент аi с ключом меньшим, чем ключ x,

2) достигнут левый конец готовой последовательности.

**Анализ сортировки вставками**

Если первоначальный массив отсортирован, то на каждом просмотре делается только одно сравнение, так что эта сортировка имеет порядок O(N). Если массив первоначально отсортирован в обратном порядке, то данная сортировка имеет порядок O(N2), поскольку общее число сравнений равно

1+2+3+...+(N-2)+(N-1)=(N-1)\*N/2, что составляет O(N2).

Чем ближе к отсортированному виду, тем более эффективной становится сортировка простыми вставками. Среднее число сравнений в сортировке простыми вставками также составляет O(N2).

**Сортировка Шелла**

Сортировка Шелла – это улучшенный метод сортировки вставками. Был предложен Д.Л. Шеллом в 1959 г. Рассмотрим этот метод на примере.

Пример:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Первичный массив  ключей | X1  25 | X2  57 | X3  82 | X4  63 | X5  90 | X6  75 | X7  80 |
| Просмотр 1  Шаг 5  Проходов 5 | 25 | 57 | 82 | 63 | 90 | 75 | 80 |
| Просмотр 2  Шаг 3  Проходов 3 | 25 | 57 | 82 | 63 | 90 | 75 | 80 |
| Просмотр 3  Шаг 1  Проходов 1 | 25 | 57 | 75 | 63 | 90 | 82 | 80 |
| Отсортированный массив ключей | 25 | 57 | 63 | 75 | 80 | 82 | 90 |

При первом просмотре группируются и сортируются элементы, отстоящие друг от друга на 5 позиций: (Х1,Х6), (Х2,Х7), (Х3), (Х4), (Х5); при втором проходе — элементы, отстоящие друг от друга на три позиции: (Х1,Х4,Х7), (Х2,Х5), (Х3,Х6); при третьем — на 1 позицию.

Поскольку первый в сортировке Шелла используемый шаг является большим, отдельные подмассивы достаточно малы по отношению к исходному. Таким образом, сортировка включением над этими подмассивами работает достаточно быстро – O(N2) (эффективно при малом N). Сортировка каждого подмассива приводит к тому, что весь массив становиться ближе к отсортированному виду, что также делает эффективным использование сортировки включением. Так как массив становится более упорядоченным, то *O(N) < порядок ФBC <O(N2).*

Если некоторый массив частично отсортирован с использованием шага k, а затем сортируется частично с использованием шага j, то данный массив остается частично отсортированным по шагу k, т.е. последующие частичные сортировки не нарушают результата предыдущих сортировок. Следующий шаг отличается от предыдущего следующим образом: ht=1, hi+1<hi.

**Анализ сортировки Шелла**

Анализ сортировки Шелла математически сложен. В случае правильного выбора шагов порядок ФВС будет выглядеть как O(N1.2). Д. Кнут предложил выбирать шаги из следующего ряда: 1, 4, 13, 40, 121, … , а вычислять их по следующей формуле: *hk–1 = 3 hk +1; ht = 1; t = [log 3 N] – 1* (количество просмотров), где N – размерность исходного массива. Т.е. элементы должны быть взаимно простыми числами. Исходя из этого порядок сортировки может быть аппроксимирован величиной О(N(log 2N)).

**Базовая и улучшенная сортировки выбором, их сравнительный анализ.**

**Сортировка выбором.**

При сортировке выбором из множества неотсортированных элементов выбирают наименьший и присоединяют его к отсортированным ключам.

**Алгоритм сортировки выбором**

1. Находим наименьший ключ в неупорядоченной части массива.

2. Меняем местами найденный элемент с тем, который соседствует с упорядоченной частью.

3. Пункты 1 и 2 выполняем, пока в неупорядоченной части имеется более одного элемента.

Алгоритм сортировки выбором в некотором смысле противоположен алгоритму сортировки вставками. При сортировке вставками на каждом шаге рассматривается только один очередной элемент входной последовательности и все элементы готового массива для нахождения места вставки. При сортировке простым выбором рассматриваются все элементы входного массива, и наименьший из них отправляется в готовую последовательность.

**Анализ сортировки простым выбором**

При сортировке простым выбором число сравнений ключей не зависит от их начального порядка. На первом просмотре выполняется N-1 сравнение, на втором — N-2 и т.д. Следовательно, общее число сравнений равно (N-1) + (N-2) + (N-3) + ... + 1=N\*(N-1)/2, что составляет O(N2). Порядок функции ВС не зависит от упорядоченности сортируемого массива, однако время сортировки упорядоченного массива будет минимальным, т.к. от упорядоченности массива зависит число перестановок элементов.

**Пирамидальная сортировка**

Пирамидальная сортировка является улучшенной сортировкой выбором. Из массива, состоящего из N элементов, выбирается максимальный и меняется местами с последним. Затем рассматривается массив из N=N-1 элементов. Процесс повторяется до тех пор, пока количество рассматриваемых элементов больше одного. Пирамидальная сортировка отличается от сортировки выбором поиском максимального элемента. Чтобы понять пирамидальную сортировку, массив нужно интерпретировать как бинарное дерево, в корне которого находится первый элемент массива, на втором уровне – второй и третий, на третьем – с четвёртого по седьмой и т.д. Для i-го элемента массива можно определить номер P элемента, являющегося родителем, как

P=i div 2, номер L элемента, являющегося левым сыном, как

L=2\*i, номер R элемента, являющегося правым сыном, как

R=2\*i+1. Если в массиве (в дереве) N элементов, то последний элемент, имеющий хотя бы одного левого сына, имеет номер

N div 2. Массив M (см. табл.3.4) можно представить деревом на рис.3.1.

Поиск максимального элемента массива в этом алгоритме основан на понятии пирамиды. Дерево (поддерево) является пирамидой, если каждый элемент в нём больше или равен элементам, которые являются его сыновьями. Корень пирамиды – максимальный элемент массива. Пирамида для массива M (см. табл.3.4) представлена на рис.3.2.

Построив пирамиду для массива, можно в соответствии с алгоритмом сортировки вставками, максимальный элемент, который находится в корне пирамиды (корень пирамиды – первый элемент массива), поменять местами с последним элементом массива. В результате получим массив М (см. табл.3.5) и его представление без последнего элемента в виде дерева на рис.3.3.

Дерево на рис.3.3 не является пирамидой, но левое и правое поддерево – пирамиды. Для того, чтобы это дерево перестроить в пирамиду, достаточно значение из корня дерева “опустить вниз” меняя его местами с сыном, имеющим наибольшее значение. Обмен производить до тех пор, пока есть сын, больший чем элемент в корне. При перестроении дерева на рис.3.3 в пирамиду на рис.3.4 корневой элемент опускается на седьмой, а третий и седьмой поднимаются соответственно на первый и третий. Пирамида построена и теперь корень можно поменять с последним элементом дерева и в дальнейшем его не обрабатывать. Для полной сортировки процесс повторяется, пока в дереве количество вершин больше одного.

Сформулируем алгоритм перестроения дерева, у которого левое и правое поддерево – пирамиды, в пирамиду.

**Алгоритм MakeHeap.**

1. Запоминаем корневой элемент.

2. Перебираем элементы в направлении большего сына и сдвигаем каждый элемент “вверх” на одну позицию, пока не освободится место для корневого элемента.

3. Вставляем корневой элемент на освободившееся место.

Для построения пирамиды из произвольного дерева (массива) необходимо построить пирамиду по алгоритму MakeHeap для каждого элемента, имеющего хотя бы одного сына, причём построение должно идти от последнего такого элемента к первому, т.е. снизу в верх.

Теперь можем сформулировать алгоритм пирамидальной сортировки.

**Алгоритм HeapSort.**

1. Построить пирамиду для исходного массива.

2. Пока в массиве более одного элемента, переставить первый и последний элемент, уменьшить размер массива на еди-ницу и перестроить дерево в пирамиду по алгоритму MakeHeap.

**Анализ пирамидальной сортировки**

Пусть дерево массива на нижнем (нулевом) уровне содер-жит максимальное число вершин. Для построения пирамиды из произвольного дерева (первая часть алгоритма) обработать все вершины, начиная с первого уровня. Для обработки вершины на i-том уровне число сравнений пропорционально i. Число вершин на i-том уровне равно 2[log2N]-1 , а всего уревней m=[log2N], следовательно общее число сравнений определяется формулой

1\*2[log2N]-1 + 2\*2[log2N]-2 + … +m\*2[log2N]-m = O(N).

Во второй части алгоритма последовательно обрабатывают-ся с N, N-1, … ,2 вершинами. Число сравнений для перестройки дерева, состоящего из i вершин, в пирамиду пропорционально [log2i], следовательно общее число сравнений

[log2N]+[log2(N-1)]+…+[log22] = O(N\*log2N).

Т.о. порядок функции ВС алгоритма пирамидальной сорти-ровки O(N\*log2N).

**Быстрая сортировка выбором.**Улучшить сортировку выбором можно, если после каждого прохода получать больше информации, чем просто идентификация минимального элемента.

Пример:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| k1  30 | k2  10 | k3  40 | k4  20 | k5  15 | k6  17 | k7  45 | k8  60 |
| 10 | | 20 | | 15 | | 45 | |
| 10 | | | | 15 | | | |
| 10 | | | | | | | |

10 – элемент MaxInt.

k1(N-1) – количество просмотров; log 2 N – высота бинарного дерева.

Мы проходим от корня к листу по тому пути, по которому шел минимальный элемент (шаг 2). Затем осуществляем модификацию бинарного дерева. Общее количество модификаций: k2log 2 N  (N-1).

**Алгоритм быстрой сортировки выбором:**

1. Сравниваем пары соседних ключей и запоминаем значение меньшего ключа из каждой пары.
2. Выполняем п.1 по отношению к значениям, полученным на предыдущем шаге. Так повторяем, пока не определим наименьший ключ и не построим бинарное дерево.
3. Вносим значение корня, найденное в п.2, в массив упорядоченных ключей.
4. Проходим от корня к листу дерева, следуя по пути, отмеченному минимальным ключом, и заменяем значение в листе на наибольшее целое число.
5. Проходим от листа к корню, по пути обновляя значения в узлах дерева, и определяем новый минимальный элемент.
6. Повторяем пп.3-6, пока минимальным элементом не будет MaxInt.

**Анализ быстрой сортировки выбором.**

*k1(N-1)+ k2log 2 N(N-1) = О(N  log 2 N)* (ФВС = порядок)

**Базовая и улучшенная сортировки обменом, их сравнительный анализ.**

**Сортировка обменом**

Идея обменной сортировки заключается в том, что два элемента, нарушающие требуемый порядок, меняются местами.

**Алгоритм сортировки обменом**

1. Перебираем поочередно все пары соседних элементов, начиная с последнего, и меняем местами элементы в парах, нарушающих порядок.

2. Пункт 1 повторяем n-1 раз.

После первого прохода (п.1) минимальный элемент массива занимает первую позицию и в дальнейшем в сортировке не участвует. После второго прохода образуется упорядоченная последовательность из первых двух элементов массива и т.д., пока все элементы массива не будут отсортированы.

**Анализ сортировки обменом**

При использовании сортировки обменом число сравнений элементов не зависит от их начального порядка. На первом просмотре выполняется N-1 сравнение, на втором — N-2 и т.д. Следовательно, общее число сравнений равно

(N-1)+(N-2)+(N-3)+...+1=N\*(N-1)/2, что составляет O(N2).

Перестановки в упорядоченном массиве не выполняются, а в массиве, упорядоченном в обратном порядке, их число равно числу сравнений элементов. Следовательно время выполнения алгоритма зависит от упорядоченности массива.

**Улучшенная сортировка обменом 1**

После каждого прохода в сортировке обменом может быть сделана проверка, были ли совершены перестановки в течение данного прохода. Если перестановок не было, то это означает, что массив упорядочен и дальнейших проходов не требуется.

**Анализ улучшенной сортировки обменом 1**

Число сравнений для этого метода зависит от числа проходов, необходимых для сортировки. В худшем случае, когда массив упорядочен в обратном порядке, выполняется N-1 проходов. На первом проходе выполняется N-1 сравнение, на втором – N-2 и т.д. Следовательно, общее число сравнений равно

(N-1)+(N-2)+(N-3)+...+1=N\*(N-1)/2, что составляет O(N2).

В этом случае выполняется максимальное число перестановок, что увеличивает время выполнения алгоритма.

В лучшем случае, когда массив уже упорядочен, потребуется всего один проход и N-1 сравнение, что составляет O(N). Перестановки в этом случае не выполняются.

**Улучшенная сортировка обменом 2**

В отличие от улучшенной сортировки обменом 1 здесь в течение прохода фиксируется последний элемент, участвующий в обмене. В очередном проходе этот элемент и все предшествующие в сравнении не участвуют, т.к. все элементы до этой позиции уже отсортированы.

**Анализ улучшенной сортировки обменом 2**

Анализ улучшенной сортировки обменом 2 аналогичен анализу улучшенной сортировки обменом 1. Порядок функций ВС этих алгоритмов в лучшем и худшем случаях одинаковый.

**Сортировка Хоара**

Сортировка Хоара–лучший из известных до сего времени метод сортировки массивов. Он обладает столь блестящими характеристиками, что его изобретатель Ч. Хоар окрестил его быстрой сортировкой*.* Быстрая сортировка основана на том факте, что для достижения наибольшей эффективности желательно разбить массив на подмассивы и сортировать подмассивы меньшего размера.

**Алгоритм быстрой обменной сортировки**

1. Выбираем элемент массива в качестве разделителя.

2. Располагаем элементы меньшие разделителя в первой части массива, а большие – во второй.

3. Если число элементов в первой части массива больше 1, то применяем к ней алгоритм в целом.

4. Если число элементов во второй части массива больше 1, то применяем к ней алгоритм в целом. Конец алгоритма.

Данный алгоритм основан на том утверждении, что ни один из элементов, расположенных в первой части массива не обменяется с каким либо элементом, расположенным во второй части.

П.2 выполняется следующим образом: просматриваем массив от краев к центру и меняем местами элементы, нарушающие порядок.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 30 | 10 | 40 | 20 | 15 | 17 | 45 | 60 |
| 30 | ( 10 | 17 | 20 | 15 | 40 | 45 | 60 ) |
| ………………………………………………………………………. | | | | | | | |
| ( 10 | 17 | 20 | 15 ) | 30 | ( 40 | 45 | 60 ) |
| ………………………………………………………………………. | | | | | | | |

**Анализ сортировки Хоара.**

Пусть m=log2N, где N – количество элементов. Будем считать, что количество элементов, меньших разделителя, равно количеству элементов, больших разделителя, т.е. массив разбивается пополам на две равные части. Определим количество сравнений в этом случае:

N+2\*(N/2)+…+m\*(N/m)=O(N\*m)=O(N\*log2N).

Если каждый раз одна из частей массива содержит не более одного элемента, то порядок будет O(N2).

Характер разбиения массива, а, следовательно, и порядок функции ВС, зависит от соотношения разделителя и остальных элементов массива. Для определения “хорошего” разделителя имеются различные алгоритмы.

**3. Нормальные формы БД. Нормализация данных**

В рамках реляционной модели данных Э.Ф. Коддом был разработан аппарат нормализации отношений и предложен механизм, позволяющий любое отношение преобразовать к третьей нормальной форме. Нормализация схемы отношения выполняется путём декомпозиции схемы.

Таблица 3.1. Исходное отношение *КНИГИ*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***ID*** | *Code* | *Theme* | *Author* | *Title* | *Editor* | *Type* | *Year* | *Pg* |
| 200 | 681.3 | ПО ВТ | Бочков С. | Язык СИ | Садчиков П. | учебник | 1990 | 384 |
| Субботин Д. |
| 100 | 681.3 | ПО ВТ | Джехани Н. | Язык АДА |  | учебник | 1960 | 552 |
| 876 | 007 | ИИ | Гик Е.Я. | Шахматы и математика | Кикоин И. | учебное пособие | 1983 | 176 |
| Капица С. |

**Первая нормальная форма (1НФ).**

Отношение приведено к 1НФ, если все его атрибуты простые.

Отношение *КНИГИ* содержит сложные атрибуты *Author* ("Авторы") и *Editor* ("Редакторы"). Для приведения к 1НФ требуется сделать ключ отношения составным – атрибуты *ID*, *Author* и *Editor* (табл. 3.2).

Таблица 3.2. Отношение *КНИГИ*, приведённое к 1НФ

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***ID*** | *Code* | *Theme* | ***Author*** | *Title* | ***Editor*** | *Type* | *Year* | *Pg* |
| 200 | 681.3 | ПО ВТ | Бочков С. | Язык СИ | Садчиков П. | учебник | 1990 | 384 |
| 200 | 681.3 | ПО ВТ | Субботин Д. | Язык СИ | Садчиков П. | учебник | 1990 | 384 |
| 100 | 681.3 | ПО ВТ | Джехани Н. | Язык АДА |  | учебник | 1960 | 552 |
| 876 | 007 | ИИ | Гик Е.Я. | Шахматы и математика | Кикоин И. | учебное пособие | 1983 | 176 |
| 876 | 007 | ИИ | Гик Е.Я. | Шахматы и математика | Капица С. | учебное пособие | 1983 | 176 |

Понятие **функциональной зависимости**. Пусть X и Y – атрибуты (группы атрибутов) некоторого отношения. Говорят, что Y функционально зависит от X, если в любой момент времени каждому значению X=х соответствует единственное значение Y=y (X®Y). (При этом любому значению Y=y может соответствовать несколько значений Х=(х1, х2,…)).

В нормализованном отношении все неключевые атрибуты функционально зависят от ключа отношения. Говорят, что неключевой атрибут функционально полно зависит от составного ключа, если он функционально зависит от ключа, но не находится в функциональной зависимости ни от какой части составного ключа.

**Вторая нормальная форма (2НФ).**

Отношение находится во 2НФ, если оно приведено к 1НФ и каждый неключевой атрибут функционально полно зависит от составного ключа.

Для того чтобы привести отношение ко 2НФ, нужно:

построить его проекцию, исключив атрибуты, которые не находятся в функционально полной зависимости от составного ключа;

построить дополнительные проекции на часть составного ключа и атрибуты, функционально зависящие от этой части ключа.

Ключом отношения *КНИГИ* (табл. 3.2) является комбинация полей (***ID*, *Author*, *Editor***). Все поля, не входящие в состав ключа, зависят только от идентификатора книги. Поэтому отношение должно быть разбито на два: *КНИГИ* (табл. 3.3) и *КНИГИ–АВТОРЫ–РЕДАКТОРЫ* (табл. 3.4). Эти отношения связаны по внешнему ключу, которым является поле ID.

Таблица 3.3. Отношение *КНИГИ*, приведённое к 2НФ Таблица 3.4. Отношение *КНИГИ–АВТОРЫ–РЕДАКТОРЫ* (2НФ)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***ID*** | *Code* | *Theme* | *Title* | *Type* | *Year* | *Pg* |
| 200 | 681.3 | ПО ВТ | Язык СИ для ПК | учебник | 1990 | 384 |
| 100 | 681.3 | ПО ВТ | Язык АДА | учебник | 1960 | 552 |
| 876 | 007 | ИИ | Шахматы и математика | учебное пособие | 1983 | 176 |

|  |  |  |
| --- | --- | --- |
| ***ID*** | ***Author*** | ***Editor*** |
| 200 | Бочков С. | Садчиков П. |
| 200 | Субботин Д. | Садчиков П. |
| 100 | Джехани Н. |  |
| 876 | Гик Е.Я. | Кикоин И. |
| 876 | Гик Е.Я. | Капица С. |

Рассмотрим понятие **транзитивной зависимости**. Пусть X, Y, Z – атрибуты некоторого отношения. При этом X® Y и Y® Z, но обратное соответствие отсутствует, т.е. Z не зависит от Y или Y не зависит от X. Тогда говорят, что Z транзитивно зависит от X (X®® Z).

**Третья нормальная форма (3НФ).**

Отношение находится в 3НФ, если оно находится во 2НФ и в нем отсутствуют транзитивные зависимости.

Для отношения *КНИГИ* (табл. 3.3) атрибут *Theme* зависит от атрибута *Code*, а не от ключа (хотя название рубрики, естественно, соответствует её шифру). Поэтому для приведения отношения к 3НФ (табл. 3.5) нужно выделить из него ещё одно отношение *РУБРИКАТОР* (табл. 3.6).

Таблица 3.5. Отношение *КНИГИ*, приведённое к 3НФ Таблица 3.6. Отношение *РУБРИКАТОР*, приведённое к 3НФ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***ID*** | *Code* | *Title* | *Type* | *Year* | *Pg* |
| 200 | 681.3 | Язык СИ для ПК | учебник | 1990 | 384 |
| 100 | 681.3 | Язык АДА | учебник | 1960 | 552 |
| 876 | 007 | Шахматы и математика | учебное пособие | 1983 | 176 |

|  |  |
| --- | --- |
| ***Code*** | *Theme* |
| 681.3 | ПО ВТ |
| 007 | ИИ |

Введём понятие **многозначной зависимости**. Многозначная зависимость существует, если заданным значениям атрибута X соответствует множество, состоящее из нуля (или более) значений атрибута Y (X–»Y). Если в отношении присутствуют многозначные зависимости, то схема отношения должна находиться в 4НФ.

Различают тривиальные и нетривиальные многозначные зависимости. **Тривиальной** называется такая многозначная зависимость X–»Y, для которой Y Ì  X или X  U Y = R, где R – рассматриваемое отношение. Тривиальная многозначная зависимость не нарушает 4НФ. Если хотя бы одно из двух этих условий не выполняется (т.е. Y не является подмножеством X или X   U  Y состоит не из всех атрибутов R), то такая многозначная зависимость называется нетривиальной.

**Четвертая нормальная форма (4НФ).**

Отношение находится в 4НФ, если оно находится в 3НФ и в нем отсутствуют нетривиальные многозначные зависимости.

Для отношения *КНИГИ–АВТОРЫ–РЕДАКТОРЫ* (табл. 3.4) атрибуты *Author* и *Editor* зависит образуют две многозначные зависимости от первичного ключа, и при этом значения этих атрибутов не зависят друг от друга. Поэтому для приведения отношения к 4НФ нужно разбить его на два отношения *КНИГИ–АВТОРЫ* и *КНИГИ–РЕДАКТОРЫ* (табл. 3.7, 3.8).

Таблица 3.7. Отношение *КНИГИ–АВТОРЫ* (4НФ) Таблица 3.8. Отношение *КНИГИ–РЕДАКТОРЫ* (4НФ)

|  |  |
| --- | --- |
| ***ID*** | ***Author*** |
| 200 | Бочков С. |
| 200 | Субботин Д. |
| 100 | Джехани Н. |
| 876 | Гик Е.Я. |

|  |  |
| --- | --- |
| ***ID*** | ***Editor*** |
| 200 | Садчиков П. |
| 876 | Кикоин И. |
| 876 | Капица С. |

Нормализация сокращает дублирование данных, но появление новых отношений усложняет поддержку логической целостности данных.

**4. Каналы передачи данных. Физический канал. Логический канал. Понятие блока данных. Пример формата блока данных любого протокола.**

*Канал* – средство или путь по которому передаются сигналы либо данные.

В распределённых системах понятие канал используется в нескольких смыслах:

*Физический канал* – средство передачи сигналов между системами и их компонентами. Он состоит из одной физической среды передачи данных либо последовательности физических сред и аппаратуры передачи данных (АПД).

Каналы классифицируют в зависимости от характера носителя сигналов:

* оптический;
* проводной;
* радиоканал.

Важной характеристикой физического канала является *полоса пропускания* – частотный спектр передаваемых по нему сигналов. В зависимости от ширины полосы пропускания (разности между двумя граничными частотами полосы пропускания) и методики передачи сигналов физические каналы делятся на:

узкополосные (основополосные) и широкополосные.

Современные технологии передачи данных обеспечивают по одному физическому каналу одновременное взаимодействие группы пар систем, причем эти пары ведут передачу данных независимо друг от друга. Путь, по которому происходит такая передача, называется *логическим каналом*. Логический канал может использовать частотную полосу либо интервалы времени, выделенные в физическом канале при передаче данных между двумя системами.

Логический канал образуются между объектами любых уровней, например, на рисунке физические каналы a и b соединяют три абонентские системы А, В и С. В каждом физическом канале организовано по три логических канала. Каналы 1 и 2 связывают объекты физического уровня. Между объектами канального уровня образованы логические каналы 3 и 4 (их иногда называют каналом передачи данных). Логический канал 5 соединяет объекты транспортного уровня, проходит через 2 физических канала (a,b) и транзитную систему В. Такой канал, проходящий через всю коммуникационную сеть или ее часть, называется *виртуальный канал*.



Существует два класса каналов:

* в синхронном канале обеспечивается синхронизация выполняемого процесса передачи;
* асинхронный канал характерен тем, что передача данных через него не требует синхронизации работы отправителя и приемника данных.

В соответствии с формой передаваемых сигналов каналы делятся на:

* аналоговые – передают аналоговые сигналы и характеризуются частотной полосой пропускания (на выходе модема телефонной линии);
* дискретные – передают дискретные сигналы и характеризуются, например, тактовой частотой, уровнем напряжения и т.п.

Блок данных – последовательность символов фиксированной длины, используемая для представления данных и самостоятельно передаваемая в сети. Каждый блок данных состоит из частей, называемых полями. Размер блока данных определяется числом содержащихся в нем битов.



Заголовок блока данных является начальной частью, в которой указываются все сведения, необходимые для передачи блока данных, например, его номер, размер, адреса отправителя и получателя и т.д.

Тело блока данных – является основной частью. В нем содержится информация для передачи которой был создан рассматриваемый блок данных.

Концевик – данное поле может отсутствовать в блоке данных. В него перед передачей записывается последовательность битов, позволяющая определить возникновение ошибок в процессе прохождения блока данных по физическим каналам.

Блок данных на разных уровнях может иметь разные размеры, поэтому при передачи блока с одного уровня на другой часто выполняются две операции:

* *сегментирование* блока
* *сборка* блока данных

Пакет IPX состоит из заголовка и блока передаваемых данных. Блок передаваемых данных имеет длину от 0 до 546 байт. Формат заголовка приведен ниже, его особенностью является то, что все поля хранят информацию в "перевернутом" виде (старший байт распологается по меньшему адресу). Формат заголовка пакета IPX длина 30 байт содержит: контрольную сумму, длину пакета, счетчик пройденных мостов, тип пакета, номер сети получателя пакета, адрес станции-получателя, сокет (уникальный номер) программы-получателя, номер сети отправителя пакета, адрес станции-отправителя, сокет программы-отправителя.

*Стек* – это последовательность, в которой включение и исключение элемента осуществляется с одной стороны последовательности (вершины стека). Так же осуществляется и операция доступа. Структура функционирует по принципу LIFO (последний пришедший обслуживается первым). Условные обозначения стека:

Вкл.

Искл.

Вкл.

Искл.

При реализации стека рассматриваются стек как отображение на массив и стек как отображение на список.

Совокупность ***операций***, определяющих структуру типа стек:

1. Операция инициализации.
2. Операция включения элемента в стек.
3. Операция исключения элемента из стека.
4. Операция проверки: стек пуст / стек не пуст.
5. Операция проверки: стек переполнен / стек не переполнен (данная операция характерна для стека как отображения на массив).
6. Операция чтения элемента (доступ к элементу).

Все операции не зависят от размерности стека, т.е. порядок временной сложности – О(1).

Имеется две модификации стека:

* Указатель находится на вершине стека, показывая на первый пустой элемент.
* Указатель указывает на первый заполненный элемент.

слот

Стек в последовательной памяти (схема на физическом уровне):

*Дескриптор*

S

|  |  |
| --- | --- |
| 1. **Условные обозначения** | Название стека |
| Нижний адрес стека | |
| Верхний адрес стека | |
| Адрес указателя | |
| Описание элемента | |

***Применение стека:***

1. Стек используется при преобразовании рекурсивных алгоритмов в нерекурсивные. В частности, с помощью стека можно модифицировать алгоритм сортировки Хоара.

**Алгоритм сортировки Хоара (стек используется для хранения границ сортируемой области в таблице):**

Включить в стек начало и конец сортируемой таблицы.

Исключить из стека адрес конца сортируемой области (В). Если стек пуст, то сортировка окончена.

Исключить из стека адрес начала сортируемой области (А).

Выбираем ключ-разделитель в сортируемой области (при этом используем тот или иной алгоритм).

Размещаем меньшие ключи до разделителя, а большие – после него. Адрес разделителя – С.

Если А < (С-1), то записываем в стек адрес А и (С-1).

Если (С+1) < В, то записываем в стек начало (С+1) и конец сортируемой области В и переходим к п.2.

1. Стек используется при разработке компиляторов.

Например, вычисление арифметических выражений. Имеем некоторое выражение: (А+В)С. Это выражение в инфиксной записи. Необходимо перевести его в постфикс (польскую запись): А В + С . С помощью стека выражение вычисляется за один проход.

*Пример:* Имеем выражение (5 + 3)  7 + 2  4. Преобразуем его к виду 5 3 + 7  2 4  +

< > - стек. Заносим в стек цифры и когда появляется операция, то выполняем ее над занесенными в стек элементами: < >  < 5 >  < 5 3 >  < 8 >  < 8 7 >  …

1. Cтеки оказали влияние и на архитектуру ЭВМ, послужили основой для стековых машин. В такой ЭВМ аккумулятор выполнен в виде стека, позволяющего расширить спектр безадресных команд.

д, т.е. команд, не требующих явных заданий адресов операндов. Следствием использования стека является увеличение скорости обработки.

**5. Структуризация сетей. Понятие и характеристики основных сетевых топологий. Структурообразующие аппаратные средства и программное обеспечение.**

Топология сети – способ соединения узлов сети в среде связи.

Физическими структурными элементами сети являются ЭВМ, каналы связи, устройства расширения сети (репитеры), концентраторы, коммутаторы, мосты, маршрутизаторы, шлюзы.

Примерами логических структур элементов сети являются: протокол, драйвер, направляющий граф соединений в виде маршрутной таблицы и т.д.

**1**

**4**

**2**

**3**

**6**

**5**

1. Концентрация управляющей информации в одном узле обеспечивает более простое управление сетью. Система является полностью централизованной.
2. Обеспечивается частично децентрализованное управление. В такой топологии всегда должна решаться задача маршрутизации.
3. Сигналы передаются по кольцу в одном направлении и проходят через каждый компьютер, и тем самым отпадает необходимость в маршрутизации. Если выйдет из строя один компьютер, прекращает функционировать вся сеть.
4. Шина (горизонтальная топология) — простая с точки зрения управления трафиком, поскольку шина допускает, чтобы каждое сообщение принималось всеми станциями, т.е. одна единственная станция работает в широковещательном режиме на несколько станций. Поэтому, если один из компьютеров выйдет из строя, это не скажется на работе остальных. Один разделяемый ресурс (канал связи).
5. Многосвязная топология – используется как правило в отказоустойчивых системах. Предполагает наличие многочисленных каналов связи с множественным резервированием.
6. Комбинированная топология – объединяет все рассмотренные топологии и реализует смешанное управление (децентрализованное и централизованное). Асинхронна. Является сложной с точки зрения проектирования и последующего управления, обеспечивается наибольшая производительность с точки зрения передачи.

Структурообразующее оборудование: повторители (repeater), концентраторы (hub), мосты (bridge), коммутаторы, маршрутизаторы.

*Повторитель* (repeater) – простейшее из коммуникационных устройств, используется для физического соединения различных сегментов кабеля локальной сети с целью увеличения общей длинны среды передачи.

Мост (bridge) – коммуникационное устройство, разделяющее среду передачи на части (логические сегменты) и передающее информацию из одного сегмента в другой только в том случае, если такая передача действительно необходима, т.е. если адрес компьютера-получателя принадлежит другой подсети.

Маршрутизатор – коммуникационное устройство, используемое для локализации трафика в отдельных частях сети.

*Очередь* – последовательность, в которую включают элементы с одной стороны, а исключают – с другой. Структура функционирует по принципу FIFO (поступивший первым обслуживается первым). Условные обозначения очереди:

Искл. (“голова”)

Вкл.

Искл.

Вкл. (“хвост”)

При реализации очереди рассматриваются очередь как отображение на массив (полустатическая реализация) и очередь как отображение на список.

Совокупность операций, определяющих структуру типа очередь:

Операция инициализации.

Операция включения элемента в очередь.

Операция исключения элемента из очереди.

Операция проверки: очередь пуста / очередь не пуста.

Операция проверки: очередь переполнена / очередь не переполнена.

В последовательной памяти очередь имеет следующий вид:

1

N

слот

Uk 2

Здесь: Uk 1 – указатель на “голову” (начало) очереди, Uk 2 – указатель на “хвост” (конец) очереди.

Чтобы избежать переполнения, рациональней использовать кольцевую очередь. При этом Uk 1 > Uk 2 или Uk 2 > Uk 1. В случае же если очередь не кольцевая, то всегда Uk 2 > Uk 1. Если очередь пуста или переполнена, то Uk 1 = Uk 2.

Пусть n – текущее состояние очереди, а N – количество элементов в очереди, тогда имеем условие 0  n  N и значение n может являться условием проверки состояния очереди. Операция включения возможна, если n < N, а операция исключения, если n > 0. В случае кольцевой очереди при операции модуля указатели будут меняться следующим образом: Uk 1 = Uk 1 mod N + 1, Uk 2 = Uk 2 mod N + 1.

Схема на физическом уровне СД типа очередь выглядит следующим образом:

*Дескриптор:*

|  |  |
| --- | --- |
| 1. **Условные обозначения** | Название очереди |
| Нижний адрес очереди | |
| Верхний адрес очереди | |
| Указатель начала очереди | |
| Указатель конца очереди | |
| Свойства элементов очереди | |

const fifoksize=100; { размер очереди }

type index=0..fifosize;

fifo=record st:array[index] of char;

uk:index;

end;

***Применение очереди:***

Очередь используется при передаче данных из оперативной во вторичную память (при этом происходит процедура буферизации: накапливается блок и передается во вторичную память). Наличие буфера обеспечивает независимость взаимодействия процессов между производителем и потребителем (ранжирование задач пользователя). Задачи разделяются по приоритетам: 1) задачи, решаемые в режиме реального времени (высший приоритет) (очередь 1); 2) задачи, решаемые в режиме разделения времени (очередь 2); задачи, решаемые в пакетном режиме (фоновые задачи) (очередь 3). Очереди выполняются последовательно.

**5. Язык SQL для работы с реляционными базами данных**

SQL - Структурированный Язык Запросов. Это - язык который дает вам возможность создавать и работать в реляционных базах данных, которые являются наборами связанной информации сохраняемой в таблицах.

SQL это язык ориентированный специально на реляционные базы данных. Он устраняет много работы которую вы должны были бы сделать если бы вы использовали универсальный язык программирования, напрмер C.

Интерактивный SQL используется для функционирования непосредственно в базе данных чтобы производить вывод для использования его заказчиком. В этой форме SQL, когда вы введете команду, она сейчас же выполнится и вы сможете увидеть вывод - немедленно.

Вложенный SQL состоит из команд SQL помещенных внутри программ, которые обычно написаны на некотором другом языке.

Запрос - команда которую вы даете вашей программе базы данных, и которая сообщает ей чтобы она вывела определенную информацию из таблиц в память. Эта информация обычно посылается непосредственно на экран компьютера или терминала которым вы пользуетесь, хотя, в большинстве случаев, ее можно также послать принтеру, сохранить в файле ( как объект в памяти компьютера ), или представить как вводную информацию для другой команды или процесса.

Операторы SQL делятся на:

- операторы определения данных (*Data Definition Language,* [*DDL*](http://ru.wikipedia.org/wiki/DDL))

**CREATE** создает объект БД (саму базу, таблицу, [представление](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)), пользователя и т. д.)

**ALTER** изменяет объект

**DROP** удаляет объект

- операторы манипуляции данными (*Data Manipulation Language,* [*DML*](http://ru.wikipedia.org/wiki/DML))

**SELECT** считывает данные, удовлетворяющие заданным условиям

**INSERT** добавляет новые данные

**UPDATE** изменяет существующие данные

**DELETE** удаляет данные

- операторы определения доступа к данным (*Data Control Language,* [*DCL*](http://ru.wikipedia.org/wiki/DCL))

**GRANT** предоставляет пользователю (группе) разрешения на определенные операции с объектом

**REVOKE** отзывает ранее выданные разрешения

- операторы управления [транзакциями](http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D1%8F) (*Transaction Control Language,* [*TCL*](http://ru.wikipedia.org/wiki/TCL_(SQL)))

**COMMIT** применяет транзакцию.

**ROLLBACK** откатывает все изменения, сделанные в контексте текущей транзакции.

Общий вид оператора выборки **select** выглядит следующим образом. С помощью этого оператора можно выбирать данные.

**select** выражение [**from** table\_references] [**where** условие]

[**group by** { имя столбца | номер столбца } [**asc** | **desc**], ...]

[**having** условие]

[**order by** { имя столбца | номер столбца } [**asc** | **desc**], ...]

Операторы модификации данных.

Добавление данных. Оператор **insert**.

**insert** **into** **< таблица >** [(список столбцов)] **values** (вставляемые значения);

Список столбцов таблицы необязателен и обязан присутствовать лишь тогда, когда вставляемых значений меньше чем столбцов таблицы.

Изменение данных. Оператор **update**.

Для изменения данных следует применять оператор **update**.

**update** **< имя таблицы > set** < атрибут > = значение,... [**where** <условное выражение>];

Удаление данных. Оператор **delete**.

Для удаления кортежей из таблиц следует применять оператор **delete**.

**delete** **from** **< имя таблицы >** [**where** <условное выражение>]

**Храни́мая процеду́ра** — объект [базы данных](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), представляющий собой набор [SQL](http://ru.wikipedia.org/wiki/SQL)-инструкций, который компилируется один раз и хранится на сервере. Хранимые процедуры очень похожи на обыкновенные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные, в них могут производиться числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам. В хранимых процедурах могут выполняться стандартные операции с базами данных (как [DDL](http://ru.wikipedia.org/wiki/DDL), так и [DML](http://ru.wikipedia.org/wiki/DML)). Кроме того, в хранимых процедурах возможны циклы и ветвления, то есть в них могут использоваться инструкции управления процессом исполнения.

Хранимые процедуры позволяют повысить производительность, расширяют возможности программирования и поддерживают функции безопасности данных. Вместо хранения часто используемого запроса, клиенты могут ссылаться на соответствующую хранимую процедуру. При вызове хранимой процедуры её содержимое сразу же обрабатывается сервером.

Кроме собственно выполнения запроса, хранимые процедуры позволяют также производить вычисления и манипуляцию данными — изменение, удаление, выполнять DDL-операторы (не во всех СУБД!) и вызывать другие хранимые процедуры, выполнять сложную транзакционную логику. Один-единственный оператор позволяет вызвать сложный сценарий, который содержится в хранимой процедуре, что позволяет избежать пересылки через сеть сотен команд и, в особенности, необходимости передачи больших объёмов данных с клиента на сервер.

В большинстве СУБД при первом запуске хранимой процедуры она компилируется (выполняется синтаксический анализ и генерируется план доступа к данным). В дальнейшем её обработка осуществляется быстрее. В СУБД Oracle выполняется интерпретация хранимого процедурного кода, сохраняемого в [словаре данных](http://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85). Начиная с версии [Oracle 10g](http://ru.wikipedia.org/w/index.php?title=Oracle_10g&action=edit&redlink=1)поддерживается так называемая [естественная компиляция](http://ru.wikipedia.org/w/index.php?title=%D0%95%D1%81%D1%82%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F&action=edit&redlink=1) (native compilation) хранимого процедурного кода в [Си](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%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)) и затем в [машинный код](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) целевой машины, после чего при вызове хранимой процедуры происходит прямое выполнение её скомпилированного объектного кода.

Созданную хранимую процедуру можно вызвать в любой момент, что обеспечивает [модульность](http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%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 стимулирует [повторное использование кода](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0). Последнее облегчает сопровождение базы данных, так как она становится изолированной от меняющихся бизнес-правил. Модифицировать хранимую процедуру в соответствии с новыми правилами можно в любой момент. После этого все приложения, использующие её, автоматически придут в соответствие с новыми бизнес-правилами без непосредственной модификации.

Использование хранимых процедур позволяет ограничить или вообще исключить непосредственный доступ пользователей к таблицам базы данных, оставив пользователям только разрешения на выполнение хранимых процедур, обеспечивающих косвенный и строго регламентированный доступ к данным. Кроме того, некоторые СУБД поддерживают шифрование текста (wrapping) хранимой процедуры. Эти функции безопасности позволяют изолировать от пользователя структуру базы данных, что обеспечивает целостность и надежность базы. Снижается вероятность таких действий как «[внедрение SQL-кода](http://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_SQL-%D0%BA%D0%BE%D0%B4%D0%B0)», поскольку хорошо написанные хранимые процедуры дополнительно проверяют входные параметры перед тем, как передать запрос СУБД.

**Три́ггер** — это [хранимая процедура](http://ru.wikipedia.org/wiki/%D0%A5%D1%80%D0%B0%D0%BD%D0%B8%D0%BC%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0) особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено наступлением определенного события (действием) — по сути добавлением INSERT или удалением DELETE строки в заданной таблице, или модификации UPDATE данных в определенном столбце заданной таблицы[реляционной базы данных](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85). Триггеры применяются для обеспечения целостности данных и реализации сложной [бизнес-логики](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B7%D0%BD%D0%B5%D1%81-%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0). Триггер запускается сервером автоматически при попытке изменения данных в таблице, с которой он связан. Все производимые им модификации данных рассматриваются как выполняемые в [транзакции](http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), в которой выполнено действие, вызвавшее срабатывание триггера. Соответственно, в случае обнаружения ошибки или нарушения целостности данных может произойти откат этой транзакции.

Момент запуска триггера определяется с помощью ключевых слов BEFORE (триггер запускается до выполнения связанного с ним события; например, до добавления записи) или AFTER(после события). В случае, если триггер вызывается до события, он может внести изменения в модифицируемую событием запись (конечно, при условии, что событие — не удаление записи). Некоторые СУБД накладывают ограничения на операторы, которые могут быть использованы в триггере (например, может быть запрещено вносить изменения в таблицу, на которой «висит» триггер, и т. п.)

Кроме того, триггеры могут быть привязаны не к таблице, а к [представлению](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)) (VIEW). В этом случае с их помощью реализуется механизм «обновляемого представления». В этом случае ключевые слова BEFORE и AFTER влияют лишь на последовательность вызова триггеров, так как собственно событие (удаление, вставка или обновление) не происходит.

В некоторых серверах триггеры могут вызываться не для каждой модифицируемой записи, а один раз на изменение таблицы. Такие триггеры называются табличными.

**Транза́кция**— группа последовательных операций, которая представляет собой логическую единицу работы с данными. Транзакция может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще и тогда она не должна произвести никакого эффекта. Транзакции обрабатываются [транзакционными системами](http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), в процессе работы которых создаётся [история транзакций](http://ru.wikipedia.org/wiki/%D0%96%D1%83%D1%80%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%B8%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B9). Различают последовательные (обычные), [параллельные](http://ru.wikipedia.org/w/index.php?title=%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%82%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D1%8F&action=edit&redlink=1) и [распределённые транзакции](http://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D0%B0%D1%8F_%D1%82%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D1%8F&action=edit&redlink=1). Распределённые транзакции подразумевают использование больше чем одной транзакционной системы и требуют намного более сложной логики (например, two-phase commit — [двухфазный протокол фиксации транзакции](http://ru.wikipedia.org/w/index.php?title=%D0%94%D0%B2%D1%83%D1%85%D1%84%D0%B0%D0%B7%D0%BD%D1%8B%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB_%D1%84%D0%B8%D0%BA%D1%81%D0%B0%D1%86%D0%B8%D0%B8_%D1%82%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D0%B8&action=edit&redlink=1)). Также, в некоторых системах реализованы [автономные транзакции](http://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BD%D0%BE%D0%BC%D0%BD%D1%8B%D0%B5_%D1%82%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D0%B8), или под-транзакции, которые являются автономной частью родительской транзакции. Одним из наиболее распространённых наборов требований к транзакциям и транзакционным системам является набор ACID (Atomicity, Consistency, Isolation, Durability). Вместе с тем, существуют специализированные системы с ослабленными транзакционными свойствам. В идеале транзакции разных пользователей должны выполняться так, чтобы создавалась иллюзия, что пользователь текущей транзакции — единственный. Однако в реальности, по соображениям производительности и для выполнения некоторых специальных задач, СУБД предоставляют различные уровни изоляции транзакций. Уровни описаны в порядке увеличения изоляции транзакций и надёжности работы с данными

0 — Неподтверждённое чтение (Read Uncommitted, Dirty Read, грязное чтение) — чтение незафиксированных изменений своей транзакции и параллельных транзакций, возможны нечистые, неповторяемые чтения и фантомы

1 — Подтверждённое чтение (Read Committed) — чтение всех изменений своей транзакции и зафиксированных изменений параллельных транзакций, нечистые чтения невозможны, возможны неповторяемые чтения и фантомы;

2 — Повторяемое чтение (Repeatable Read, Snapshot) — чтение всех изменений своей транзакции, любые изменения, внесённые параллельными транзакциями после начала своей, недоступны, нечистые и неповторяемые чтения невозможны, возможны фантомы;

3 — Упорядоченный — (Serializable, сериализуемый) — [упорядоченные (сериализуемые)](http://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D1%83%D0%B5%D0%BC%D0%BE%D1%81%D1%82%D1%8C&action=edit&redlink=1) транзакции. Идентичен ситуации, при которой транзакции выполняются строго последовательно, одна после другой, то есть результат действия которых не зависит от порядка выполнения шагов транзакции (запрещено чтение всех данных, изменённых с начала транзакции, в том числе и своей транзакцией). Фантомы невозможны.

Чем выше уровень изоляции, тем больше требуется ресурсов, чтобы их поддерживать.

В СУБД уровень изоляции транзакций можно выбрать как для всех транзакций сразу, так и для одной конкретной транзакции. По умолчанию в большинстве баз данных используется уровень 1 (Read Committed). Уровень 0 используется в основном для отслеживания изменений длительных транзакций или для чтения редко изменяемых данных. Уровни 2 и 3 используются при повышенных требованиях к изолированности транзакций. Полноценная реализация уровней изоляции и свойств ACID представляет собой нетривиальную задачу. Обработка поступающих данных приводит к большому количеству маленьких изменений, включая обновление как самих таблиц, так и индексов. Эти изменения потенциально могут потерпеть неудачу: закончилось место на диске, операция занимает слишком много времени (timeout) и т. д. Система должна в случае неудачи корректно вернуть базу данных в состояние до транзакции.

Первые коммерческие СУБД (к примеру, IBM [DB2](http://ru.wikipedia.org/wiki/DB2)), пользовались исключительно блокировкой доступа к данным для обеспечения свойств ACID. Но большое количество блокировок приводит к существенному уменьшению производительности. Есть два популярных семейства решений этой проблемы, которые снижают количество блокировок:

* [Журнализация изменений](http://ru.wikipedia.org/wiki/%D0%96%D1%83%D1%80%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%B8%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B9) (write ahead logging, WAL)
* [механизм теневых страниц](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%85%D0%B0%D0%BD%D0%B8%D0%B7%D0%BC_%D1%82%D0%B5%D0%BD%D0%B5%D0%B2%D1%8B%D1%85_%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86) (shadow paging).

В обоих случаях, блокировки должны быть расставлены на всю информацию, которая обновляется. В зависимости от уровня изоляции и [имплементации](http://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D0%B8%D1%8F), блокировки записи также расставляются на информацию, которая была прочитана транзакцией.

При *упреждающей журнализации*, используемой в [Sybase](http://ru.wikipedia.org/wiki/Sybase) и [MS SQL Server](http://ru.wikipedia.org/wiki/MS_SQL_Server) до версии 2005, все изменения записываются в журнал, и только после успешного завершения — в базу данных. Это позволяет СУБД вернуться в рабочее состояние после неожиданного падения системы. *Теневые страницы* содержат копии тех страниц базы данных на начало транзакции, в которых происходят изменения. Эти копии активизируются после успешного завершения. Хотя теневые страницы легче реализуются, упреждающая журнализация более эффективна[[3]](http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)#cite_note-2)

Дальнейшее развитие технологий управления базами данных привело к появлению безблокировочных технологий. Идея контроля за параллельным доступом с помощью временных меток (timestamp-based concurrency control) была развита и привела к появлению многоверсионной архитектуры [MVCC](http://ru.wikipedia.org/wiki/MVCC). Эти технологии не нуждаются ни в журнализации изменений, ни в теневых страницах. Архитектура, реализованная в [Oracle](http://ru.wikipedia.org/wiki/Oracle_(%D0%A1%D0%A3%D0%91%D0%94)) 7.х и выше, записывает *старые* версии страниц в специальный сегмент отката, но они все ещё доступны для чтения. Если транзакция при чтении попадает на страницу, временная метка которой новее начала чтения, данные берутся из сегмента отката (то есть используется «старая» версия). Для поддержки такой работы ведётся журнал транзакций, но в отличие от «упреждающей журнализации», он не содержит данных. Работа с ним состоит из трёх логических шагов:

Записать намерение произвести некоторые операции

Выполнить задание, копируя оригиналы изменяемых страниц в сегмент отката

Записать, что всё сделано без ошибок

Журнал транзакций в сочетании с *сегментом отката* (область, в которой хранится копия всех изменяемых в ходе транзакции данных) гарантирует целостность данных. В случае сбоя запускается процедура восстановления, которая просматривает отдельные его записи следующим образом:

* Если повреждена запись, то сбой произошёл во время проставления отметки в журнале. Значит, ничего важного не потерялось, игнорируем эту ошибку.
* Если все записи помечены как успешно выполненные, то сбой произошёл между транзакциями, здесь также нет потерь.
* Если в журнале есть незавершённая транзакция, то сбой произошёл во время записи на диск. В этом случае мы восстанавливаем старую версию данных из сегмента отката.

[Firebird](http://ru.wikipedia.org/wiki/Firebird) вообще не имеет ни журнала изменений, ни сегмента отката, а реализует [MVCC](http://ru.wikipedia.org/wiki/MVCC), записывая новые версии строк таблиц прямо в активное пространство данных. Так же поступает MS SQL 2005. Теоретически это даёт максимальную эффективность при параллельной работе с данными, но ценой является необходимость «сборки мусора», то есть удаления старых и уже не нужных версий данных.

**6. Понятие адресации в сетях. Типы адресов. Адресация в IP-сетях.**

Адресация – способ указания объектов в сети. Адресация определяет пункты отправления и назначения посылаемых сообщений, блоков данных, сигналов. В процессе адресации указывается, где расположен объект, или как их достичь, но не то, что они собой представляют.

Наиболее широкое распространение получили 3 схемы адресации узлов:

1. Аппаратные адреса. Эти адреса не имеют иерархической структуры. Типичным представителем является адрес сетевого адаптера локальной сети. Такой адрес как правило не изменяется и используется только аппаратными средствами сети. Адреса встраиваются в аппаратуру производителем, либо генерируются автоматически при каждом новом запуске оборудования. Помимо отсутствия иерархии у такого типа адресации есть еще один недостаток: при замене аппаратуры (сетев. адаптер) изменяется и адрес компьютера
2. Символьные адреса или имена. Эти адреса несут смысловую нагрузку. В больших сетях символьные имя может иметь сложную иерархическую структуру. Например: int.bel.ru
3. Числовые составные адреса. Типичными представителями являются IP и IPX адреса. В них поддерживается двух уровневая иерархия, адрес делится на старшую часть (номер сети) и младшую часть (номер узла). Такое разделение позволяет передавать сообщения между сетями только на основании номера сети, а номер узла используется только после доставки сообщения в нужную сеть.

Для адресации узлов применяется обычно все три схемы адресации. Пользователи адресуют компьютеры символьными именами, которые автоматически заменяются в сообщениях, передаваемых по сети на числовые, с помощью этих числовых адресов сообщения передаются из одной сети в другую, а после доставки сообщения в сеть назначения вместо числового номера сети используется аппаратный адрес компьютера.

Проблемой установления соответствия между адресами различных типов занимается служба разрешения имён.

В IP-сетях используются три типа адресов:

1. физический (Mac); 2. сетевой (IP-адрес); 3. символьное имя (DNS-имя).

Каждый компонент в сети TCP/IP имеет адреса трёх уровней:

1. Локальный адрес узла, определенный технологией, с помощью которой построена данная отдаленная сеть. Для узлов, входящих в локальные сети – это MAC адрес сетевого адаптера или порта маршрутизатора. Эти адреса назначаются изготовителем оборудования и являются уникальными, т.к. управляются централизовано. Для всех существующих технологий локальных сетей MAC адрес имеет формат 6 байтов. Старшие три – идентификатор фирмы производителя, а младшие три – назначаются уникальным образом самим производителем.
2. IP-адрес, состоящий из 4-х байт. Например, 130.26.117.100. Этот адрес используется на сетевом уровне. Он назначается администратором во время конфигурирования компьютеров и маршрутизаторов. IP-адрес состоит из двух частей – номер сети и номер узла. Номер узла в протоколе IP называется независимо от локального адреса узла. Деление IP адреса на поле номера сети и на поле номера узла гибкое и граница между этими полями может устанавливаться весьма произвольно. Узел может входить в несколько IP сетей. В этом случае узел должен иметь несколько IP адресов. Т.о. адрес характеризует не отдельный компьютер или маршрутизатор, а одно сетевое соединение.
3. Символьный идентификатор имя (EDU.BSTU.ru). Этот адрес состоит из нескольких частей: имени машины, имени организации, имени домена. Такой адрес, называется DNS–имя и используется на прикладом уровне, например в протоколах FTR, HTTP.

*6. Структура данных типа таблица.*

*Таблица* – последовательность записей, которые имеют одну и ту же организацию. Такая отдельная запись называется *элементом таблицы*. Чаще всего используется простая запись. Таким образом, таблица – это агрегация элементов. Если последовательность записей упорядочена относительно какого-либо признака, то такая таблица называется *упорядоченной*, иначе – таблица *неупорядоченная*.

Классификация СД типа таблица:

|  |  |  |
| --- | --- | --- |
| СД типа таблица | | |
|  | | |
| неупорядоченная таблица | упорядоченная таблица | хеш-таблица |
|  | | |
| как отображение на массив | как отображение на список |  |

кортеж

атрибут

Если один элемент d i , то *кортеж* – это <d 1, d 2, …, d n>, причем D Ti  d i. Множество значений элементов типа Т (множество допустимых значений СД типа таблица) будет определяться с помощью прямого декартова произведения:

*DT = DT1  DT2  …  DТn*, причем D Ti  <d 1, d 2, …, d n>.

Сам же элемент таблицы можно представить в виде двойки <K, V>, где К – ключ, а V – тело элемента. В качестве ключа может быть разное число полей, которые определяют этот элемент. Ключ используется для операции доступа к элементу, так как каждый из ключей уникален для данного элемента. Таким образом, таблица является совокупностью двоек <K, V>.

На логическом уровне элемент СД типа таблица описывается следующим образом:

*Type Element = record*

*Key: integer;*

*{описание остальных полей}*

*end;*

При реализации таблицы как отображения на массив ее описание выглядит так:

*Tabl = array [0 .. N] of Element*

Во время выполнения программы количество элементов может меняться. Структура, в которой изменяется количество элементов во время выполнения программы, называется *динамической*. Если рассматривать динамическую структуру как отображение на массив, то такая структура называется *полустатической*.

Перед тем как определить операции, которые можно выполнять над таблицей рассмотрим **классификацию операций**:

1. *Конструкторы* – операции, которые создают объекты рассматриваемой структуры.
2. *Деструкторы* – операции, которые разрушают объекты рассматриваемой структуры. Признаком этой операции является освобождение памяти.
3. *Модификаторы* – операции, которые модифицируют соответствующие структуры объектов. К ним относятся динамические и полустатические модификаторы.
4. *Наблюдатели* – операции, у которых в качестве элемента (входного параметра) используются объекты соответствующей структуры, а возвращают эти операции результаты другого типа. Таким образом, операции-наблюдатели не изменяют структуру, а лишь дают информацию о ней.
5. *Итераторы* – операторы доступа к содержимому объекта по частям в определенном порядке.

**Операции над таблицей:**

1. Операция инициализации (конструктор).
2. Операция включения элемента в таблицу (модификатор).
3. Операция исключения элемента из таблицы (модификатор).
4. Операции-предикаты:

* таблица пуста / таблица не пуста (наблюдатель)
* таблица переполнена / таблица не переполнена (наблюдатель).

1. Чтение элемента по ключу (наблюдатель).

**СД типа хеш-таблица.**

СД типа хеш-таблица использует алгоритм поиска, основанный на идее *хеширования*. До сих пор, для поиска требуемой записи мы использовали организацию просмотра некоторого количества ключей. Очевидно, что эффективными алгоритмами поиска являются те, которые проходят это количество ключей при минимальном числе сравнений. В идеале хотелось бы иметь такую организацию таблицы, у которой не было бы ненужных сравнений, т.е. чтобы каждый ключ находился за одно сравнение, но в этом случае положение записи в таблице должно зависеть от значения ключа.

**Операция включения элемента в таблицу.**

Перед включением элемента в таблицу определяют его адрес с помощью *хеш-функции*: *A = h(K)*, где K – ключ, а А – адрес элемента таблицы, причем 0  А  N-1, т.е. хеш-функция – отображение множества ключей на множество адресов.

*A = K mod N = (C1  k + C2) mod N*, где N – количество адресов в таблице.

“Хорошая” хеш-функция – это функция, которая быстро вычисляется и минимизирует количество коллизий. *Коллизия –* это ситуация, когда для разных ключей адрес один и тот же, т.е. *h(ki) = h(kj)*, где i  j. Минимизации можно добиться, если равномерно распределить элементы по адресному полю таблицы.

После вычисления адреса элемента таблицы есть две возможности размещения элемента:

1. поместить этот элемент по адресу, если адрес свободен;
2. если вычисленный адрес занят, то повторить хеширование, т.е. совершить *рехеширование* по следующей формуле: *A1 = (A+C) mod N*.

Таким образом, **алгоритм включения элемента в таблицу** будет иметь следующий вид:

1. Вычисление ключа с помощью хеш-функции. Если вычисленный адрес таблицы свободен, то помещаем в него элемент. Конец алгоритма.
2. Если вычисленный адрес таблицы занят, то перевычисляем адрес (проводим рехеширование) и переходим к п.1.

Чтобы процесс размещения элементов был равномерным, адресное поле таблицы делается больше множества ключей на 10-15%.

**Операция исключения элемента из таблицы:**

1. С помощью хеш-функции вычисляется предполагаемый адрес элемента таблицы.
2. Если по вычисленному адресу располагается искомый ключ, то запись найдена и ее удаляют. Конец алгоритма.
3. Если вычисленный адрес пуст, то элемента с заданным ключом в таблице нет. Конец алгоритма.

Если элемент таблицы содержит другой ключ, то перевычисляем (рехешируем) адрес и переходим к п.2.

**7. Характеристика протоколов IP, TCP, ARP, ICMP, POP3, SMTP.**

Internet Protocol или IP (англ. *internet protocol* — межсетевой протокол) — маршрутизируемый сетевой протокол, протокол сетевого уровня семейства TCP/IP.

Протокол IP используется для негарантированной доставки данных, разделяемых на так называемые пакеты от одного узла сети к другому. Это означает, что на уровне этого протокола (третий уровень сетевой модели OSI) не даётся гарантий надёжной доставки пакета до адресата. В частности, пакеты могут прийти не в том порядке, в котором были отправлены, продублироваться (когда приходят две копии одного пакета; в реальности это бывает крайне редко), оказаться повреждёнными (обычно повреждённые пакеты уничтожаются) или не прибыть вовсе.

Transmission Control Protocol (TCP) (протокол управления передачей) — предназначенный для управления передачей данных в сетях и подсетях TCP/IP. Выполняет функции протокола транспортного уровня модели OSI.

TCP — это транспортный механизм, предоставляющий поток данных, с предварительной установкой соединения, за счёт этого дающий уверенность в достоверности получаемых данных, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного.

ARP (англ. *Address Resolution Protocol* — протокол определения адреса) — использующийся в компьютерных сетях протокол низкого уровня, предназначенный для определения адреса канального уровня по известному адресу сетевого уровня.

Существуют следующие типы сообщений ARP: запрос ARP (ARP request) и ответ ARP (ARP reply). Система-отправитель при помощи запроса ARP запрашивает физический адрес системы-получателя. Ответ (физический адрес узла-получателя) приходит в виде ответа ARP.

Записи в таблице ARP, созданные динамически, остаются в кэше в течение 2-х минут. Если в течение этих двух минут произошла повторная передача данных по этому адресу, то время хранения записи в кэше продлевается ещё на 2 минуты. Эта процедура может повторяться до тех пор, пока запись в кэше просуществует до 10 минут. После этого запись будет удалена из кэша, и будет отправлен повторный запрос ARP.

ICMP (англ. *Internet Control Message Protocol* — межсетевой протокол управляющих сообщений) — сетевой протокол, входящий в стек протоколов TCP/IP. В основном ICMP используется для передачи сообщений об ошибках и других исключительных ситуациях, возникших при передаче данных, например, запрашиваемая услуга недоступна, или хост, или маршрутизатор  не отвечают.

POP3 (англ. *Post Office Protocol Version 3* — протокол почтового отделения, версия 3) используется почтовым клиентом для получения сообщений электронной почты с сервера. Обычно используется в паре с протоколом SMTP. Перед работой через протокол POP3 сервер прослушивает порт 110. Когда клиент хочет использовать этот протокол, он должен создать TCP соединение с сервером. Когда соединение установлено, сервер отправляет приглашение. Затем клиент и POP3 сервер обмениваются информацией пока соединение не будет закрыто или прервано.

SMTP (англ. *Simple Mail Transfer Protocol* — простой протокол передачи почты) — Для работы через протокол SMTP клиент создаёт TCP соединение с сервером через порт 25. Затем клиент и SMTP сервер обмениваются информацией пока соединение не будет закрыто или прервано. Основной процедурой в SMTP является передача почты (Mail Procedure). Далее идут процедуры форвардинга почты (Mail Forwarding), проверка имён почтового ящика и вывод списков почтовых групп. Самой первой процедурой является открытие канала передачи, а последней - его закрытие.

Между деревьями общего вида (узел дерева может иметь более двух сыновей) и бинарными деревьями существует взаимно однозначное соответствие, поэтому бинарные деревья часто используют для представления деревьев общего вида. Для такого представления используют следующий алгоритм:

1. изображаем корень дерева;
2. по вертикали изображаем старшего сына этого корня;
3. по горизонтали вправо от этого узла представляем всех его братьев;
4. пп. 1, 2, 3 повторяем для всех его узлов.

Таким образом, получаем, что вертикальные ребра изображают левые поддеревья, а горизонтальные – правые.

Данный алгоритм можно распространить и для леса.

a

c d e

b

j

f g h i

k

**Теорема.**  Число бинарных деревьев с n вершинами можно выразить формулой: . Например, для n=3 имеем 

Проиллюстрируем этот пример графически:

**Представление бинарных деревьев в связной памяти.**

**Прошитые деревья.**

При представлении бинарных деревьев в связной памяти, различают также три основных способа представления: стандартный, инверсный, смешанный. Узлы дерева при каждом таком представлении выглядят следующим образом:

Data

L\_Son R\_Son

F

Data

F Data

L\_Son R\_Son

стандартный инверсный смешанный

Если мы воспользуемся прошитым бинарным деревом, то вид узла для него будет таким:

LP RP Data

L\_Son R\_Son

Один из возможных способов прошивки бинарного дерева заключается в том, что для каждого узла, не имеющего левого поддерева, запоминается ссылка на непосредственного предшественника, а для каждого узла, не имеющего правого поддерева ,запоминается ссылка на его непосредственного преемника. Предшественников и преемников для узла определяют исходя из списка, который получается при прохождении рассматриваемого бинарного дерева в симметричном порядке. Для того, чтобы отличать ссылку на левого или правого сына (такие ссылки называют *структурными связями*) от ссылки на предшественника или преемника узла (такие ссылки называют “нитями”), используются индикаторные поля LP и RP.

*Симметричным прохождением* бинарного дерева является такое прохождение, когда мы проходим сначала левое поддерево, затем посещаем корень и последним посещаем правое поддерево: A R B.

A

B

Пример. Рассмотрим бинарное дерево:

В памяти это дерево будет выглядеть так:

a

b e

c d f

g

фиктивный элемент

1 1 a

1 1

1 0 b

1 1 e

При его симметричном прохождении получаем следующий список вершин: c, b, a, d, g, e, f.

0 0 c

0 0 f

0 1 d

0 0 g

Использование “прошивок” существенно убыстряет прохождение дерева, позволяет не использовать стек, но при этом усложняются алгоритмы включения / исключения узла, т.к. в прошитом дереве необходимо управлять не только структурными связями, но и “нитями”.

**Формирование бинарного дерева.**

При формировании бинарного дерева нужно задать некоторые ограничения. Рассмотрим пример, когда необходимо построить дерево из n вершин, имеющее минимальную высоту. Для достижения минимальной высоты необходимо, чтобы все уровни дерева, кроме, может быть, последнего, были заполнены. Для того чтобы сделать такое распределение, все поступающие нужно распределять поровну: слева и справа.

n=1 n=2 n=3 n=4 n=5 n=6 n=7

Таким образом, правила для равномерного распределения при известном числе узлов n можно сформулировать с помощью следующего рекурсивного алгоритма:

1. взять один узел в качестве корня;
2. построить левое поддерево с количеством узлов n l = [n / 2] тем же способом;
3. построить правое поддерево с количеством узлов n r = n – n l – 1 тем же способом.

Описание узла дерева на языке Turbo Pascal выглядит таким образом:

*Type ElPtr = ^Element*

*Element = record*

*Data: integer;*

*L\_Son, R\_Son: ElPtr;*

*end;*

*Идеально сбалансированное дерево* обладает определенным свойством таким, что относительно каждого узла дерева количество узлов в левом и правом поддеревьях может отличаться максимум на 1.

*Function Tree (n: integer): ElPtr;*

*var nl, nr, x: integer;*

*NewElement: ElPtr;*

*begin*

*if n=0 then Tree:=nil*

*else begin*

*nl:=n div 2;*

*nr:=n – nl – 1;*

*read (x); New (NewElement);*

*with NewElement do begin*

*Data:=x;*

*L\_Son:=Tree (nl);*

*R\_Son:=Tree )nr);*

*end;*

*Tree:=NewElement;*

*end;*

*end;*

Эффективность рекурсивного определения заключается в том, что оно позволяет с помощью конечного высказывания определить бесконечное множество объектов.

**Применение бинарных деревьев в алгоритмах поиска.**

В односвязном списке невозможно использовать бинарные методы, они могут использоваться только в последовательной памяти. Однако, если использовать бинарные деревья, то в такой связной структуре можно получить алгоритм поиска со сложностью O(log 2 N). Такое дерево реализуется следующим образом: для любого узла дерева с ключом Ti все ключи в левом поддереве должны быть меньше Ti, а в правом – больше Ti. В дереве поиска можно найти место каждого ключа, двигаясь, начиная от корня и переходя на левое или правое поддерево, в зависимости от значения его ключа. Из n элементов можно организовать бинарное дерево (идеально сбалансированное) с высотой не более чем log 2 N, которое определяет количество операций сравнения при поиске.

*Function Search (x: integer; t: ElPtr): ElPtr;*

*{поле Data заменим на поле Key}*

Ti

L\_Son R\_Son

*var f: boolean;*

*begin*

*f:=false;*

< Ti > Ti

*while (t<>nil) and not f do*

*if x=t^. key then f:=true*

*else if x>t^. key then t:=t^. R\_Son else t:=t^. L\_Son;*

*Search:=t;*

*end;*

Если получим, что значение функции = nil, то ключа со значением *x* в дереве не найдено.

Функцию можно упростить, если организовать идею барьера, когда все листья указывают на введенный фиктивный элемент.

S

*Function Search (x: integer; t: ElPtr): ElPtr;*

*begin*

*S^.key:=x;*

*while t^.key<>x do*

*if x > t^.key then then t:=t^. R\_Son else t:=t^. L\_Son;*

*Search:=t;*

*end;*

**Операция включения в СД типа бинарное дерево.**

Рассмотрим случай постоянно растущего, но никогда не убывающего дерева. Хорошим примером этого является построение частотного словаря. В этой задаче задана последовательность слов, и нужно установить число появления каждого слова. Это означает, что, начиная с пустого дерева, каждое слово, встречающееся в тексте, ищется в нем. Если это слово найдено, то счетчик появления данного слова увеличивается; если же слово не найдено, то в дерево включается новое слово со значением счетчика = 1.

*Type ElPtr = ^Element*

*Element = record*

*Key: integer;*

*n: integer;*

*L\_Son, R\_Son: ElPtr;*

*end;*

*Procedure Put (x: integer; var t: ElPtr);*

*begin*

*if t=nil then begin*

*New(t);*

*with t^ do begin*

*key:=x; n:=1; L\_Son:=nil; R\_Son:=nil;*

*end;*

*end*

*else if x> t^.key then Put(x, t^.R\_Son)*

*else if x<t^.key then Put(x, t^.L\_Son)*

*else t^.n:=t^.n+1;*

*end;*

Хотя задача этого алгоритма – *поиск с включением*, но его можно использовать и для сортировки, т.к. он напоминает сортировку с включением. А так как вместо массива используется дерево, то пропадает необходимость перемещения элементов выше места включения. Для того, чтобы сортировка этого алгоритма была устойчивой, алгоритм должен выполняться одинаково как при t^.key=x, так и при t^.key>x.

Пусть задана последовательность: 30, 1, 15, 17, 18, 50, 2, 60.

После построения дерева осуществляется прохождение дерева в симметричном порядке: 1, 2, 15, 17, 18, 30, 50, 60.

***Анализ алгоритма поиска.*** Если дерево вырождается в последовательность то сложность в таком случае O(N) (в худшем случае). В лучшем же случае, если дерево сбалансировано, то сложность будет O(log 2 N). Таким образом, имеем:

*O(N)  ПФВС  O(log 2 N).*

Если считать, что вероятность появления любой структуры дерева одинакова, и ключи появляются с одинаковой вероятностью, то .

Алгоритм сортировки является устойчивым, если элементы с одинаковыми ключами появляются в той же последовательности при симметричном обходе дерева, что и в процессе их включения в дерево.

**Операция исключения из бинарного дерева.**

При удалении узла дерево должно оставаться упорядоченным относительно ключа:

а) удаляется лист;

б) удаляется узел с одним потомком (потомок слева или справа);

в) удаляется узел с двумя потомками, но в левом потомке нет правого поддерева;

г) общий случай.

Рассмотрим случай б):

t

*if g^.L\_Son=nil then begin*

t

*g:=t;*

K

K

*t:=t^.R\_Son;*

*dispose(g);*

g

g

*end;*

*if g^.R\_Son=nil then begin*

*g:=t;*

*t:=t^.L\_Son;*

*dispose(g);*

*end;*

Случай а) является частным случаем этого алгоритма.

Случай в): r указывает на элемент, не имеющий правого поддерева.

*g:=t;*

*g^.key:=r^.key;*

t

*g:=r;*

К

g

*r:=r^.L\_Son;*

r

*dispose(g);*

Случай г):

Мы ищем такой элемент, у которого отсутствует правое поддерево (на такой элемент указывает r). Последовательность операторов та же, т.е. удаляемый элемент нужно заменить самым правым элементом его левого поддерева. Такие элементы не могут иметь более одного потомка.

К

t

r

*Procedure Get (x: integer; var t: ElPtr);*

*var g: ElPtr;*

*begin*

*if t=nil then {обработка исключительной ситуации, когда элемента с заданным ключом в дереве нет}*

*else if x>t^.key then Get(x, t^.R\_Son)*

*else if x<t^.key then Get(x, t^.L\_Son)*

*else begin*

*g:=t;*

*if g^.L\_Son=nil then t:=g^.R\_Son*

*else if g^.R\_Son=nil then t:=g^.R\_Son*

*else D(g^.L\_Son); {найти r)*

*dispose(g);*

*end;*

*end;*

**Представление бинарного дерева в прямоугольной памяти.**

При представлении дерева в прямоугольной памяти одно из полей R\_Son или L\_Son удаляется, т.к. необходимость в нем отпадает. Потомок может размещаться рядом, т.е. в непосредственной близости. Чтобы узнать, есть ли потомок, вводится булево поле L или R. Пусть, например, отсутствует левый потомок. Введем следующие переменные:

*index = 0..max;*

a

b d

c e

f g h

*type Element = record*

*R\_Son: index;*

*Data: BaseType;*

*L: boolean;*

*end;*

*var Tree = array [index] of Element;*

Порядок называется *прямым*, если отсутствует левое поддерево. Если же отсутствует правое поддерево, то такой порядок называется *фамильным*.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R\_Son  Data  L | |  | |  | |  | |  | |  |  | |  | |
| A | | b | | c | | f | | g | | D | e | | h | |
| T | | F | | T | | F | | F | |  |  | |  | |
| 1 | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | 8 | |  |

**Применение бинарных деревьев.**

* Любое алгебраическое выражение, которое содержит переменные, числа, знаки операций и скобки можно представить в виде бинарного дерева (исходя из бинарности этих операций). При этом знак операции помещается в корне, первый операнд – в левом поддереве, а второй операнд – в правом. Скобки при этом опускаются. В результате все числа и переменные окажутся в листьях, а знаки операций – во внутренних узлах.

рассмотрим пример: 3  (x – 2) + 4.

+

4 \*

3 \_

X 2

Если осуществлять просмотр такого дерева в прямом порядке (сначала проходим корень, затем – левое поддерево, а последним проходим правое поддерево), то получим прямую польскую запись выражения.

Если же осуществить просмотр дерева в обратном порядке (левое поддерево, правое поддерево, корень), то получим обратную польскую запись.

* Классическая программа из класса интеллектуальных: **построение деревьев решений**. В этом случае не листовые (внутренние) узлы содержат предикаты – вопросы, ответы на которые могут принимать значения “да” или “нет”. На уровне листьев находятся объекты (альтернативы, с помощью которых распознаются программы). Пользователь получает вопросы, начиная с корня, и, в зависимости от ответа, спускается либо на левое поддерево, либо на правое. Таким образом, он выходит на тот объект, который соответствует совокупности ответов на вопрос.
* Применение практических задач.

**Алгоритмы прохождения деревьев в глубину и в ширину.**

При прохождении в глубину представленного дерева, список его вершин, записанных в порядке их посещения, будет выглядеть следующим образом:

A, B, C, F, G, H, D, E, I, J, K.

***Алгоритм прохождения дерева в глубину:***

<пустой стек S>;

<пройти корень и включить его в стек S>;

while <стек S не пуст> do

begin {пусть P – узел, находящийся в вершине стека S}

if <не все сыновья узла Р пройдены>

then <пройти старшего сына и включить его в стек S>

else begin

<исключить из вершины стека узел Р>;

if <не все братья вершины Р пройдены>

then <пройти старшего брата и включить его в стек S>

end;

end;

При прохождении представленного дерева в ширину (по уровням), список его вершин, записанных в порядке их посещения, будет выглядеть следующим образом:

A, B, C, D, E, F, G, H, I, J, K.

***Алгоритм прохождения дерева в ширину:***

<взять две очереди О1 и О2>;

<поместить корень в очередь О1>;

while <О1 или О2 не пуста> do

if <О1 не пуста> then {Р – узел, находящийся в голове очереди О1}

begin

<исключить узел из очереди О1 и пройти его>;

<поместить все узлы, относящиеся к братьям этого узла Р в очередь О2>;

end

else <O1=O2; O2=> и повторяем цикл.

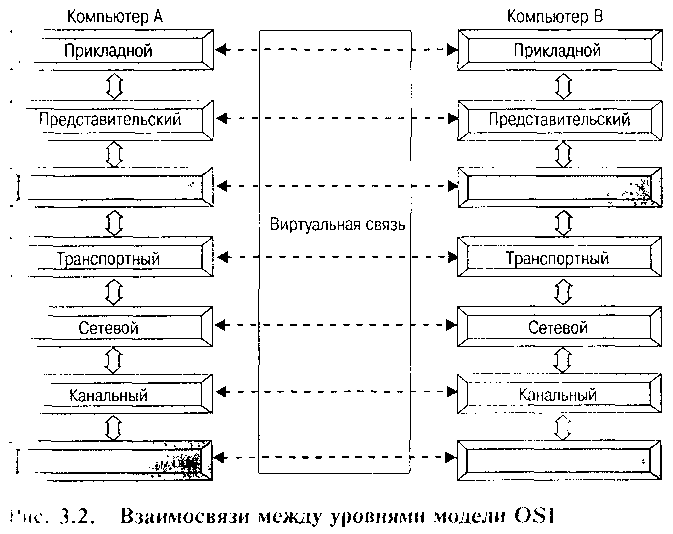
**8. Эталонная модель взаимодействия открытых систем. Понятие уровня и уровневых сетевых сервисов. Характеристика уровней. Преимущества и недостатки уравнения, распределённого по уровням**.

Работа сети заключается в передаче данных от одного компьютера к другому и представляет распознавание данных, разбиение данных на управляемые блоки, добавление информации к каждому блоку, чтобы указать местонахождение данных, получателя, добавление информации синхронизации и информации для проверки ошибок, помещение данных в сеть и отправление их по заданному адресу.

В 1984 году ISO выпустила новую версию своей модели, названную эталонной моделью взаимодействия открытых систем. Являясь многоуровневой системой, она отражает взаимодействие программного и аппаратного обеспечения при осуществлении сеанса связи, а также помогает решить разнообразные проблемы. В модели OSI сетевые функции распределены между семью уровнями. *Уровень* – компонент, слой либо граница иерархической структуры. Каждому уровню соответствуют различные сетевые операции, оборудование и протоколы.

В основу иерархической организации протоколов положены следующие принципы:

1. закрепление функции одного целевого назначения за одним и тем же уровнем;
2. внесение изменений при модификации имеющихся и добавлении новых функций не должно оказывать никакого влияния на протоколы других уровней и обслуживающие интерфейсы;
3. простота обслуживающих интерфейсов;
4. минимальное количество уровней иерархии.

Задача каждого уровня — предоставление услуг вышележащему уровню, «маскируя» детали реализации этих услуг. При этом каждый уровень на одном компьютере pa6oтает так, будто он напрямую связан с таким же уровнем на другом компьютере. Эта логическая, или виртуальная связь между одинаковыми уровнями показана на рисунке. Однако в действительности связь осуществляется между смежными уровнями одного компьютера.

**Физический уровень**Уровень I, Физический, —осуществляет передачу неструктурированного, «сырого» потока битов по физической среде (например, по сетевому кабелю, он также формирует сигналы, которые переносят данные, поступившие от всех вышележащих уровней.

**Канальный уровень**Уровень 2, Канальный, осуществляет передачу кадров данных от Сетевого уровня к Физическому. Кадры — это логически организованная структура, в которую можно помещать данные. Канальный уровень компьютера-получателя упаковывает «сырой» поток битов, поступающих от Физического уровня, в кадры данных.

**Сетевой уровень**Уровень З, Сетевой (Network), отвечает за адресацию сообщений и перевод логических адресов и имен в физические адреса. Одним словом здесь определяется маршрут от компьютера-oтправителя к компьютеру-получателю. На этом уровне решаются также такие задачи и проблемы, связанные с сетевым графиком, как коммутация пакетов, маршрутизация и перегрузки.

**Транспортный уровень**Уровень 4, Транспортный (Transport), обеспечивает дополнительный уровень соединения — ниже Сеансового уровня. Транспортный уровень гарантирует доставку пакетов без ошибок, в той же последовательности, без потерь и дублирования. Транспортный уровень управляет потоком, проверяет ошибки и участвует в решении проблем, связанных с отправкой и получением пакетов.

**Сеансовый уровень**Уровень 5, Сеансовый (Session), позволяет двум приложениям на разных компьютерах устанавливать, использовать и завершить соединение, называемое сеансом. На этом уровне выполняются такие функции, как распознавание имен и защит, необходимые для связи двух приложений в сети.

**Представительский уровень**Уровень 6, Представительский (Presentation), определяет формат, используемый для обмена данными между сетевыми компьютерами. Этот уровень можно назвать переводчиком. На компьютере-отправителе данные, поступившие от Прикладного уровня на этом уровне переводятся в общепонятный промежуточный формат. На компьютере-получателе, на этом уровне происходит перевод из промежуточного формата в тот, который используется Прикладным уровнем данного компьютера. Представительский уровень отвечает за преобразование протоколов, трансляцию данных, их шифрование, смену или преобразование применяемого набора символов (кодовой таблицы) и рас­ширение графических команд. Представительский уровень, кроме того, управляет сжатием данных для уменьшения передаваемых битов.

**Прикладной уровень.**Уровень 7, Прикладной (Application) — обеспечивает услуги, напрямую поддерживающие приложения пользователя, такие, как программное обеспечение для передачи файлов, доступа к базам данных, электронная почта. Нижележащие уровни поддерживают задачи, выполняемые на Прикладном уровне. Прикладной уровень управляет общим доступом к сети, потоком данных и обработкой ошибок.

**Преимущества и недостатки управления распределенного по уровням.**

**«+»**

* Любой уровень может изменяться и развиваться, не затрагивая другие уровни.
* Возникающая при расслоении модульность упрощает создание всей системы в целом.
* Можно осуществить подстановку принципиально других механизмов, затронув не более одного уровня.

**«-»**

* Растут суммарные накладные расходы (заголовки и т.д.).
* Может оказаться, что две связанные между собой вычислительные системы должны выполнять какие-то функции, без которых они могли бы обойтись.

**8. Стили программирования. Программные продукты как сложные системы. Признаки сложных систем. Декомпозиция. Преимущества и недостатки объектно-ориентированного подхода в программировании.**

Стили программирования:

1. структурное программирование (в основе понятия алгоритма, Pascal);
2. логическое программирование (в основе исчисление предикатов, Prolog);
3. функциональное программирование (в основе понятие функции, Lisp);
4. ООП (C++, Object Pascal).

ООП - методология программирования, основанная на представление программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Опыт разработки программного обеспечения, показывает, что большинство программных продуктов представляют собой сложные системы. Быть сложным, значит включать множество частей, аспектов, деталей, понятий, требующих для понимания серьёзного исследования или рассмотрения. Примеры пк, структура растений.

Разработка программ требует анализа сложного неорганизованного реального мира, точного и полного определения зависимостей и исключений, проектирования абсолютно верных решений.

Все остальные технические цели разработки ПО вторичны по отношению к управлению сложностью.

Особенности или признаки сложных систем:

1. Сложные системы являются иерархическими, т. е. состоят из взаимосвязанных подсистем, которые в свою очередь могут быть разделены на подсистемы и т. д. в плоть до самого низкого уровня.
2. Выбор какие компоненты в данной системе являются элементарными относительно произволен и в большей степени оставляется на усмотрение исследователя.
3. Внутрикомпонентные связи обычно сильнее, чем связи между компонентами системы. Это различие внутрикомпонентных и межкомпонентных взаимодействий обуславливает разделение функций между частями системы и даёт возможность относительно независимо изучать каждую часть.
4. Иерархические системы обычно состоят из небольшого количества одинаковых структурных частей, т. е. система обычно состоит из различных подсистем по – разному скомбинированных и организованных.
5. Любая работающая сложная система является результатом развития работающей более простой системы. В процессе развития системы объекты первоначально рассматривающиеся как сложные становятся элементарными из них строят более сложные системы.

Сложность программных продуктов постоянно растёт, а человеческие возможности при рассмотрении сложных систем ограничены. Тогда на помощь пришел принцип «разделяй и властвуй», т. е. рассмотрение любой сложной системы требует применения декомпозиции, т. е. разбиение на составляющие элементы.

При проектирование сложной программной системы необходимо разделять на всё меньшие и меньшие подсистемы, каждую из которых можно совершенствовать не зависимо. В этом случае мы не превысим человеческие возможности. В программирование известны две схемы декомпозиции:

* алгоритмическая;
* объектно-ориентированная.

*Алгоритмическая декомпозиция.* В основе лежит разбиение программы по действиям, т. е. по алгоритму. Данному подходу соответствует структурное программирование сверху вниз.

*Объектно–ориентированная декомпозиция.* Обеспечивает разбиение по объектам, т. е. реальным или виртуальным элементам, каждый из которых несёт в себе описание данных и действия над этими данными. Обеспечивает ряд преимуществ перед алгоритмической декомпозицией.1. уменьшается размер программных систем за счёт повторного исполнения общих механизмов. 2. ОО системы являются более гибкими и проще эволюционируют во времени, поскольку базируются на устойчивых промежуточных формулах. 3. объектная декомпозиция снижает риск при создании сложных программных систем, т. к. она развивается из меньших систем, в которых мы уже уверены. 4 помогает разобраться в сложной программной системе.

Преимущества ООП:

Сокращение количества межмодульных вызовов

Уменьшение объема информации, передаваемой между модулями

Возможность вести независимую разработку объектов программы

Недостатки

Некоторое снижение быстродействия за счет более сложной организации программной системы

**9. Основные понятия ООП. Класс, объект, виды отношений между классами**

**Объектно-ориентированное программирование** — это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса (типа особого вида), а классы образуют иерархию, основанную на принципах наследуемости. При этом объект характеризуется как совокупностью всех своих свойств и их текущих значений, так и совокупностью допустимых для данного объекта действий.

**Класс** – это абстрактный тип данных, который включает в себя не только данные, но и функции и процедуры.

**Класс** - это шаблон, на основе которого может быть создан конкретный программный объект, он описывает свойства и методы, определяющие поведение объектов этого класса. Каждый конкретный объект, имеющий структуру этого класса, называется экземпляром класса.

**Поля** класса являются переменными, объявленными внутри класса. Они предназначены для хранения данных во время работы экземпляра класса (объекта). Ограничений на тип полей в классе не предусмотрено. В описании класса поля должны предшествовать методам и свойствам.

**Методы** — это процедуры и функции, описанные внутри класса и предназначенные для операций над его полями.

Функции и процедуры класса называются методами и содержат исходный код, предназначенный для обработки внутренних данных объекта данного класса.

После того, как Вы объявили состав класса, необходимо определить (описать), что делает каждая функция-член (метод). С помощью класса описывается некоторая сущность (ее характеристики и возможные действия). Например, класс может описывать студента, автомобиль и т.д. Описав класс, мы можем создать его экземпляр – объект.

**Объект**- это осязаемая сущность, которая четко проявляет свое поведение.

Объект состоит из следующих трех частей:

- имя объекта;

- состояние (переменные состояния);

- методы (операции).

**Объект** – это уже конкретный представитель класса.

Теория ООП выделяет три основных отношения между классами:

* Ассоциация
* Агрегация и композиция
* Обобщение/Расширение (наследование)

**Ассоциация**

Ассоциация означает, что объекты двух классов могут ссылаться один на другой, иметь некоторую связь между друг другом. Например, Менеджер может выписать Счет. Соответственно возникает ассоциация между Менеджером и Счетом. Еще пример – Преподаватель и Студент – т.е. какой-то Студент учится у какого-то Преподавателя. Ассоциация и есть описание связи между двумя объектами. Студент учится у Преподавателя. Идея достаточно простая – два объекта могут быть связаны между собой и это надо как-то описать.

**Агрегация и композиция**

Агрегация и композиция на самом деле являются частными случаями ассоциации. Это более конкретизированные отношения между объектами.

**Агрегация** – отношение, когда один объект является частью другого. Например, Студент входит в Группу любителей физики.

**Композиция** – еще более “жесткое отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежат еще кому-то. Например, Машина и Двигатель. Хотя двигатель может быть и без машины, но он вряд ли сможет быть в двух или трех машинах одновременно. В отличии от студента, который может входить и в другие группы тоже. Такие описания всегда несколько условны, но тем не менее.

**Наследование (Генерализация)** — объекты дочернего класса наследуют все свойства родительского класса.

**Класс-Метакласс** — отношение, при котором экземплярами одного класса являются другие классы.

**Операционная система** — это программное обеспечение, управляющее выполнением программ и исполняющее роль интерфейса между приложениями и аппаратным обеспечением компьютера.

Ее предназначение можно разделить на **три основные составляющие**:

• Удобство. Операционная система делает использование компьютера простым и удобным.

• Эффективность. Операционная система позволяет эффективно использовать ресурсы компьютерной системы.

• Возможность развития. Операционная система должна быть организована так, чтобы она допускала эффективную разработку, тестирование и внедрение новых приложений и системных функций, причем это не должно ме¬шать нормальному функционированию вычислительной системы.

**Классификация ОС.**

1. **По режиму обработки задачи:**

* однопрограммные(MSDOS, MSX)
* Мультипрограммные(UNIX, OS/2, Windows 95)

Мультипрограммная ОС-сама распараллеливает задачи.

Мультизадачная ос-распараллеливанием занимается программист.

Используются механизм вытесняющих приоритетов для создания иллюзии выполнения нескольких программ одновременно.

1. **По способу взаимодействия систем:**

* ОС с пакетной обработкой (ОС ЕС)
* диалоговые системы:
* однопользовательские (однотерминальные)ОС (MS-DOS, Windows до NT)
* мультитерминальные ОС (Linux) (UNIX, Windows NT).

1. **По способу построения:**

-Микроядерные -многоуровневая

-Макроядерные(монолитные) -объектно-ориентированная

-распределённая -С множественными прикладными средами

1. **По назначению**

* Общего назначения ;
* Специального назначения:
* ОС для КПК,
* ОС для встроенных систем,
* ОС для организации и ведения баз данных
* ОС для решения задач реального времени

**Функциональные компоненты ОС.**

Программы ОС группируются согласно выполняемым функциям и называются подсистемами ОС. Все подсистемы разделяются на два больших класса по следующим признакам:

по типам локальных ресурсов, которыми управляет ОС;

соответствующие подсистемы –**подсистемы управления ресурсами;**

по специфические задачи, применимым ко всем ресурсам;

соответствующие подсистемы –**подсистемы, общие для всех ресурсов.**

**Основные подсистемы управления ресурсами** – это подсистемы:

-управления процессами;

-управления памятью;

-управления файлами и внешними устройствами.

**Общие для всех ресурсов** – это подсистемы:

-прикладного программного и пользовательского интерфейсов;

-защиты данных и администрирования.

**9. HTML** (от англ. HyperText Markup Language — «язык гипертекстовой разметки») — стандартный язык разметки документов во Всемирной паутине. Большинство веб-страниц содержат описание разметки на языке HTML (или XHTML). Язык HTML интерпретируется браузерами и отображается в виде документа в удобной для человека форме.

HTML позволяет формировать на странице сайта текстовые блоки, включать в них изображения, организовывать таблицы, управлять отображением цвета документа и текста, добавлять в дизайн сайта звуковое сопровождение, организовывать гиперссылки с контекстным переходом в другие разделы сервера или обращаться к иным ресурсам Сети и компоновать все эти элементы между собой. Файлы, содержащие гипертекстовый код, имеют расширение .htm или .html.

Базовая технология Интернета, в своё время (на заре создания единых Сетей) разработанная специально для «общения» двух основных функционалов виртуального пространства: сервера и клиента (говоря проще, «от сохи», сетевого и пользовательского компьютеров). Является основным и единым (но не единственным) стандартом для обмена данными в связке «сервер — клиент».

HTML — теговый язык разметки документов. Любой документ на языке HTML представляет собой набор элементов, причём начало и конец каждого элемента обозначается специальными пометками — тегами. 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. Вот примеры фрагментов HTML-документа:

<strong>Текст между двумя тегами — открывающим и закрывающим.</strong>

<a href="http://www.example.com">Здесь элемент содержит атрибут href, то есть гиперссылку.</a>

А вот пример пустого элемента: <br>

**Версии**

* HTML 0.9
* RFC 1866 — HTML 2.0, одобренный как стандарт 22 сентября 1995 года;
* HTML 3.2[1] — 14 января 1997 года;
* HTML 4.0[2] — 18 декабря 1997 года;
* HTML 4.01[3] (изменения, причём более значительные, чем кажется на первый взгляд) — 24 декабря 1999 года;
* ISO/IEC 15445:2000[4] (так называемый ISO HTML, основан на HTML 4.01 Strict) — 15 мая 2000 года.
* HTML 5[5] — 28 октября 2014 года
* HTML 5.1 начал разрабатываться примерно 19 декабря 2012 года.

**CSS –** это язык, содержащий набор свойств для определения внешнего вида документа. Cпецификация CSS определяет свойства и описательный язык для установления связи с HTML-элементами.

**CSS –** абстракция, в которой внешний вид Web-документа определяется отдельно от его содержания.

Преимущественно используется как средство описания, оформления внешнего вида веб-страниц, написанных с помощью языков разметки HTML и XHTML, но может также применяться к любым XML-документам, например, к SVG или XUL.

CSS обеспечивает более эффективную, быструю и удобную работу электронных документов и web приложений. CSS стили помогают обрабатывать такие элементы оформления страниц, как шрифт, цветовая гамма, позиционирование элементов, обеспечивает более эффективную работу с изображениями.

В отличие от HTML, где для каждого элемента все его свойства задавались отдельно, в CSS создается блочное описание свойств. Впоследствии, полученные блоки могут быть применены для любого элемента документа.

Kаждое правило CSS из таблицы стилей имеет две основные части — селектор и блок объявлений. Селектор, расположенный в левой части правила, определяет, на какие части документа распространяется правило. Блок объявлений располагается в правой части правила. Он помещается в фигурные скобки, и, в свою очередь, состоит из одного или более объявлений, разделённых знаком «;». Каждое объявление представляет собой сочетание свойства CSS и значения, разделённых знаком ": ". Селекторы могут группироваться в одной строке через запятую. В таком случае свойство применяется к каждому из них.

селектор, селектор {

свойство: значение;

свойство: значение;

свойство: значение;

}

**Виды селекторов**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Универсальный селектор** | **Селектор элементов** | **Селектор классов** | **Селектор идентификаторов** | **Селектор атрибутов** |
| **\* {**  **margin: 0;**  **padding: 0;**  **}** | **p {**  **font-family: arial, helvetica, sans-serif;**  **}** | **.note {**  **color: red;**  **background-color: yellow;**  **font-weight: bold;**  **}** | **#paragraph1 {**  **margin: 0px;**  **}** | **a[href="http://www.somesite.com"] {**  **font-weight: bold;**  **}** |

Пример таблицы стилей (в таком виде она может быть либо размещена в отдельном файле .css, либо же обрамлена тегами <style> и размещена в «шапке» той самой веб-страницы, на которую она действует):

p {

**font-family**: arial, helvetica, sans-serif;

}

h2 {

**font-size**: 20pt;

**color**: red;

**background**: white;

}

**JavaScript** - предназначен для написания сценариев для активных HTML-страниц. Язык JavaScript не имеет никакого отношения к языку Java. Первоначальное название - LiveScript. После завоевания языком Java всемирной известности LiveScript из коммерческих соображений переименовали в JavaScript.

JavaScript не предназначен для создания автономных приложений. Программа на JavaScript встраивается непосредственно в исходный текст HTML-документа и интерпретируется брaузером по мере загрузки этого документа. С помощью JavaScript можно динамически изменять текст загружаемого HTML-документа и реагировать на события, связанные с действиями посетителя или изменениями состоятия документа или окна.

Важная особенность JavaScript - объектная ориентированность. Программисту доступны многочисленные объекты, такие, как документы, гиперссылки, формы, фреймы и т.д. Объекты характеризуются описательной информацией (свойствами) и возможными действиями (методами).

Сценарий JavaScript встраивается в HTML-документ с помощью тега <script>.

**Переменные в JavaScript**

Имя переменной не должно совпадать с зарезервированными ключевыми словами JavaScript.

Все переменные в JavaScript объявляются с помощью ключевого слова var. При объявлении тип переменной не указывается. Этот тип присваивается переменной только тогда, когда ей присваивается какое-либо значение.

Числа в строки интерпретатор JavaScript преобразует автоматически. Для преобразования строк в числа используют специальные функции parseInt и parseFloat.

**Объекты JavaScript**

Язык JavaScript является объектно-ориентированным. Объекты JavaScript представляют собой наборы свойств и методов. Можно сказать, что свойства объектов - это данные, связанные с объектом, а методы - функции для обработки данных объекта.

В языке JavaScript имеется три вида объектов: встроенные объекты, объекты брaузера и объекты, создаваемые программистом.

JavaScript поддерживает следующий набор встроенных объектов: Array, Boolean, Date, Global, Function, Math, Number, String.

**Операторы языка JavaScript**

**Унарные операторы**

|  |  |  |  |
| --- | --- | --- | --- |
| **Изменение знака на противоположный** | **Дополнение. Используется для реверсирования значения логических переменных** | **Увеличение значения переменной.** | **Уменьшение значения переменной.** |
| **-** | **!** | **++** | **--** |

**Примечания: ++ и -- могут применяться и как префикс, и как суффикс.**

**Бинарные операторы**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Вычитание** | **Сложение** | **Умножение** | **Деление** | **Остаток от деления** |
| **-** | **+** | \* | / | **%** |

**Операторы для работы с отдельными битами**

|  |  |  |  |
| --- | --- | --- | --- |
| **И** | **ИЛИ** | **ИСКЛЮЧАЮЩЕЕ ИЛИ** | **НЕ** |
| **&** | | | **^** | **~** |

**Операторы сдвига**

|  |  |  |
| --- | --- | --- |
| **Сдвиг вправо** | **Сдвиг влево** | **Сдвиг вправо с заполнением освобождаемых разрядов нулями** |
| **>>** | **<<** | **>>>** |

**Операторы отношения**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Больше** | **Больше или равно** | **Меньше** | **Меньше или равно** | **Равно** | **Не равно** |
| **>** | >= | < | **<=** | **==** | **!=** |

**Условные операторы**

В языке JavaScript два условных оператора: if-else и ?:.

**Операторы цикла**

В языке JavaScript три оператора цикла: for, for-in, while.

10 Архитектура Web-приложений

*Web-приложения* представляют собой особый тип программ, построенных по архитектуре "клиент-*сервер*". Особенность их заключается в том, что само Web-*приложение* находится и выполняется на сервере - клиент при этом получает только результаты работы. Работа приложения основывается на получении запросов от пользователя (клиента), их обработке и выдачи результата. Передача запросов и результатов их обработки происходит через *Интернет* ([рис. 1.1](http://www.intuit.ru/studies/courses/1139/250/lecture/6422?page=1#image.1.1)).

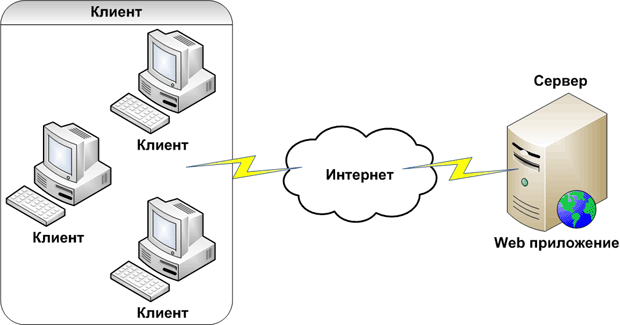


Рис. 1.1. Архитектура Web-приложения

Отображением результатов запросов, а также приемом данных от клиента и их передачей на *сервер* обычно занимается специальное*приложение* — *браузер* (*Internet* Explorer, Mozilla, Opera и т. д.). Как известно, одной из функций браузера является *отображение*данных, полученных из Интернета, в виде страницы, описанной на языке *HTML*, следовательно, результат, передаваемый сервером клиенту, должен быть представлен на этом языке.

На стороне сервера Web-*приложение* выполняется специальным программным обеспечением (Web-сервером), который и принимает запросы клиентов, обрабатывает их, формирует ответ в виде страницы, описанной на языке *HTML*, и передает его клиенту. Одним из таких Web-серверов является *Internet* *Information* Services (*IIS*) компании Microsoft. Это единственный Web-*сервер*, который способен выполнять Web-приложения, созданные с использованием технологии *ASP*.*NET*.

В процессе обработки запроса пользователя Web-*приложение* компонует ответ на основе исполнения программного кода, работающего на стороне сервера, Web-формы, страницы *HTML*, другого содержимого, включая графические файлы. В результате, как уже было сказано, формируется *HTML*-страница, которая и отправляется клиенту. Получается, что результат работы Web-приложения идентичен результату запроса к традиционному Web-сайту, однако, в отличие от него, Web-*приложение* генерирует*HTML*-код в зависимости от запроса пользователя, а не просто передает его клиенту в том виде, в котором этот код хранится в файле на стороне сервера. То есть Web-*приложение* динамически формирует ответ с помощью исполняемого кода — так называемой исполняемой части.

За счет наличия исполняемой части, Web-приложения способны выполнять практически те же *операции*, что и обычные *Windows*-приложения, с тем лишь ограничением, что код исполняется на сервере, в качестве интерфейса системы выступает *браузер*, а в качестве среды, посредством которой происходит *обмен данными*, — *Интернет*. К наиболее типичным операциям, выполняемым Web-приложениями, относятся:

прием данных от пользователя и сохранение их на сервере;

выполнение различных действий по запросу пользователя: извлечение данных из базы данных (БД), добавление, удаление, изменение данных в БД, проведение сложных вычислений;

аутентифицирование пользователя и отображение интерфейса системы, соответствующего данному пользователю;

отображение постоянно изменяющейся оперативной информации и т. д.

Каждое Web-приложение, разрабатываемое на основе ASP.NET состоит из информационной части, программного кода и сведений о конфигурации.

Информационная часть содержит статические и динамические элементы страницы и реализуется в виде Web-форм. Статические элементы представляют собой типичные элементы языка HTML, динамические же компонуются программным кодом приложения во время его выполнения (например, запросы к базе данных).

Программный код реализует логику, определенную в процедурах обработки данных, которые определяют реакцию приложения на запросы пользователя. Программный код исполняется сервером и взаимодействует с динамическими элементами информационной части для формирования отклика приложения.

Сведения о конфигурации представляют собой файлы, содержащие параметры, определяющие способ исполнения приложения на сервере, параметры безопасности, реакцию приложения на возникающие ошибки и т. д.

Основным элементом Web-приложения является Web-форма (или Web-страница), которая, с одной стороны, похожа на Windows-форму, т. к. позволяет размещать внутри себя различные элементы управления, способные отображать данные и реагировать на действия пользователя, а с другой — представляет собой HTML-страницу, т. к. содержит все ее атрибуты. Описания элементов управления, упомянутых ранее, представляются в коде HTML-страницы в виде специальных тегов.

11

Ajax — это сокращение от "Asynchronous JavaScript and XML" (асинхронный JavaScript и XML),

Ajax позволяет отправлять на сервер асинхронные запросы. это означает, что обмен данными между браузером и сервером выполняется в фоновом режиме, и поэтому никоим образом не мешает взаимодействию пользователя с содержимым HTML-документа. Чаще всего Ajax используется для отправки данных формы на сервер. Преимуществом такого подхода является то, что отображение ответной информации, получаемой от сервера, не требует полной перезагрузки веб-страницы, и данные документа могут беспрепятственно отображаться с помощью стандартных функций jQuery.

В библиотеке jQuery определен ряд так называемых прямых методов, которые в действительности представляют собой оболочки для вызовов функций ядра Ajax и позволяют быстро и просто решать типичные задачи Ajax.

Кратко об Ajax

Как программист большую часть времени вы тратите на написание синхронного кода. Вы определяете блок кода, решающий некоторую задачу, и остальная часть программы ждет, пока этот блок не выполнится. Задача завершается выполнением последнего оператора блока. На то время, пока выполняется блок, браузер лишает пользователя возможности взаимодействовать с содержимым веб-страницы.

Если же задача выполняется асинхронно, то вы сообщаете браузеру, что намерены выполнить определенную работу в фоновом режиме. Выражение "в фоновом режиме" звучит довольно туманно, но для браузера оно означает следующее: "Выполни это так, чтобы пользователь в течение всего времени мог взаимодействовать с документом, и сообщи мне, когда все будет сделано".

В случае Ajax вы приказываете браузеру связаться с веб-сервером и сообщить вам, когда запрос будет выполнен. Управление этой связью осуществляется с помощью функций обратного вызова (callback functions). Вы предоставляете браузеру одну или несколько функций, которые должны быть вызваны сразу же по завершении выполнения задачи. Должна быть предусмотрена функция, которая обработает успешный запрос, а кроме того, могут существовать функции, выполняющиеся в случае других исходов, например при возникновении ошибок.

Преимущество асинхронных запросов состоит в том, что они позволяют создавать функционально насыщенные HTML-документы, которые могут непрерывно обновляться на основании полученных от сервера ответов, не прерывая взаимодействия пользователя с приложением и не заставляя его дожидаться окончания полной загрузки документа.

Недостатком такого подхода является то, что он требует тщательного продумывания кода. Нельзя заранее сказать, когда именно будет выполнен асинхронный запрос, и вы не имеете права делать какие-либо предположения относительно возможного исхода запроса. Кроме того, использование функций обратного вызова приводит к созданию еще более сложного кода, который только и ждет, чтобы "наказать" программиста, неосторожно сделавшего какие-либо предположения относительно возможного исхода запроса или времени его выполнения.

Список функций

$.get() производит запрос к серверу методом GET

$.post() производит запрос к серверу методом POST

.load() производит запрос HTML-данных у сервера и помещает их в выбранные элементы страницы

$.getJSON() производит запрос JSON-данных у сервера методом GET

$.getScript() производит запрос файла javascript методом GET, а затем выполняет код из полученного файла.

Обработка ajax-событий

.ajaxSend() Устанавливает пользовательскую функцию, которая будет вызвана при отправке ajax-запроса.

.ajaxComplete() Устанавливает пользовательскую функцию, которая будет вызвана при завершении ajax-запроса.

.ajaxSuccess() Устанавливает пользовательскую функцию, которая будет вызвана при удачном завершении ajax-запроса.

.ajaxError() Устанавливает пользовательскую функцию, которая будет вызвана при неудачном завершении ajax-запроса.

.ajaxStart() Устанавливает пользовательскую функцию, которая будет вызвана перед выполнением первого ajax-запроса.

.ajaxStop() Устанавливает пользовательскую функцию, которая будет вызвана после выполнения всех запущенных ajax-запросов.

12

Веб-сервер — сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, как правило, вместе с HTML-страницей, изображением, файлом, медиа-потоком или другими данными.

Веб-сервером называют как программное обеспечение, выполняющее функции веб-сервера, так и непосредственно компьютер, на котором это программное обеспечение работает.

Клиент, которым обычно является веб-браузер, передаёт веб-серверу запросы на получение ресурсов, обозначенных URL-адресами. Ресурсы — это HTML-страницы, изображения, файлы, медиа-потоки или другие данные, которые необходимы клиенту. В ответ веб-сервер передаёт клиенту запрошенные данные. Этот обмен происходит по протоколу HTTP.

Функции, выполняемые Web-серверами, в сущности очень просты:

прием запроса от Web-браузера по протоколу HTTP с использованием TCP/IP;

поиск и отсылка файла гипертекста или документа в браузер по HTTP;

обслуживание запросов mailto, ftp, telnet и др.;

запуск прикладных программ на Web-сервере с передачей и возвратом параметров обработки через интерфейс CGI;

обслуживание навигационных карт изображения imagemap;

служит источником загружаемых программ на языке Java;

контроль доступа на основе имен и паролей доступа;

ведение регистрационного журнала;

административное и оперативное управление сервером.

Web-браузеры общаются с Web-серверами через протокол передачи гипертекстовых сообщений (HypertextTransferProtocol, HTTP), простой протокол запросов и ответов для пересылки информации с использованием TCP/IP. Web-сервер получает запрос, находит файл, посылает его браузеру и потом разрывает соединение. Имеющаяся на странице графика обрабатывается точно так же. Затем настает очередь браузера вывести на экран загруженный из сети HTML-документ.

Хотя обычно Web-серверы содержат HTML-страницы и графику, на них могут храниться любые файлы, в том числе текстовые, документы текстовых процессоров, видео- и аудиоинформация. Сегодня, если не считать анкет, заполняемых пользователем, основная часть трафика Web передается в одном направлении - браузеры читают файлы с Web-сервера, - но положение изменится после повсеместного принятия описанного в проекте HTTP 1.1 метода put, позволяющего записывать файлы на Web-сервер. Сегодня метод put используется в основном создателями страниц Web, но в перспективе он может пригодиться и конечным пользователям для обратной связи с информационными узлами.

На Web-сервере выполняются также прикладные программы, наибольшее распространение среди которых получили процессоры поиска и средства связи с базами данных. Для их разработки применяются такие стандарты, как общий шлюзовой интерфейс (CommonGatewayInterface, CGI), языки сценариев, подобные JavaScript, а также полноценные языки программирования, как, например, Java и VisualBasic. Помимо CGI несколько поставщиков Web-серверов разработали интерфейсы прикладного программирования (API) - как, например, NetscapeServerAPI и InternetServerAPI, созданные фирмами Microsoft и ProcessSoftware, - которые позволяют разработчикам непосредственно обращаться к конкретным функциям Web-сервера. Некоторые серверы располагают связующими средствами (middleware) для подключения к базам данных, использование которых может потребовать хороших знаний в программировании.

Базовые процессоры поиска помогают пользователям отсортировывать нужную им информацию, а программы связи с базами данных обеспечивают пользователям Web-браузеров доступ к информации.

Программное обеспечение

Наиболее распространённым веб-сервером, занимающим более 65 % рынка, является Apache — свободный веб-сервер, наиболее часто используемый в UNIX-подобных операционных системах;

Другие известные веб-серверы:

IIS от компании Microsoft, распространяемый с ОС семейства Windows

nginx — свободный веб-сервер, разрабатываемый Игорем Сысоевым с 2002 года и пользующийся большой популярностью на крупных сайтах

lighttpd — свободный веб-сервер.

Google Web Server — веб-сервер, основанный на Apache и доработанный компанией Google.

Resin — свободный веб-сервер приложений.

Cherokee — свободный веб-сервер, управляемый только через web-интерфейс.

Rootage — веб-сервер, написанный на java.

Клиенты

В качестве клиентов для обращения к веб-серверам могут использоваться различные программы и устройства:

веб-браузер, работающий на настольном компьютере или переносном устройстве (например, карманном ПК);

разнообразные программы, самостоятельно обращающиеся к веб-серверам для получения обновлений или другой информации (например, антивирус может периодически запрашивать у определённого веб-сервера обновления своих баз данных);

мобильный телефон, получающий доступ к ресурсам веб-сервера при помощи протокола WAP;

другие цифровые устройства или бытовая техника.

**10. Режим ядра**: Аппаратное обеспечение->HAL уровень аппаратной абстракции->Драйвера->ФС >Системные службы

**Режим пользователя:** Системный интерфейс->Подсистемы Win32, POSIX, OS/2 -> Приложение пользователя.

**Сервисы**

* Сервис АРI – функция или подпрограмма API, которая реализует некоторое действие (сервис) ос, такое как создание файла или работа с графикой
* Системный сервис - недокументированная функция, которая может вызывается из польз режима. Эти функции часто используются функциями Win32 API для предоставления низкоуровневых сервисов
* Внутренний сервис – функция или подпрограмма, которая может вызывается только из кода, выполняемого в режиме ядра. Эти ф-и относ к низкоуровневой части кода Windows: к исполнительной системе Windows NT, к ядру или к слою абстрагирование от аппаратуры (HAL).

**Системные процессы.**

- это особые процессы, обслуживающие ос. В системе Windows постоянно задействованы следующие (все, кроме system, выполняются в пользовательском режиме):

* процесс idle, α состоит из одного потока, управляющего временем простоя процессора;
* процесс system - спец процесс, выполняются только в режиме ядра. Его потоки -системными потоками; состоит из системных потоков, являющих¬ся потоками режима ядра. Windows и многие драйверы устройств создают потоки прoцecca system для различных целей. Например, диспетчер памяти формирует системные потоки для решения задач управления виртуальной памятью, диспетчер кэша использует системные потоки для управления кэш-памятью.
* процесс Session Manager (диспетчер сеансов) - SMSS.EXE; создание переменных окружения системы; задание имен устройств MS DOS; загрузка части подсистемы Win32, α относится к режиму ядра; запуск процесса регистрации в системе WinLogon.
* подсистема Win32 - CSRSS.EXE;
* процесс регистрации в системе — WinLogon (winlogon.exe). управляет входом пользователей в систему и выходом из нее. Вызываетсяся Ctrl+Alt+Delete.отвечает за загрузку оболочки Windows (Explorer).

**Подсистема Win32**

Подсистема Win32 является разновидностью подсистемы среды. Другие подсистемы среды Windows включают POSIX и OS/2.. POSIX является сокращением термина «переносимая ОС па базе UNIX» и реализует ограниченную поддержку ОС UNIX.

Назначение - служить интерфейсом между пользователем приложениями и соотв частью исполнительной системы Windows. Каждая подсистема имеет свои функциональные возможности на базе единой исполнительной системы Windows. Любой выполняемый файл неразрывно связан с одной из этих подсистем. Подсистема Win32 содержит Win32 API в виде набора DLL, таких как kernel32.dll, GDI32. dll user32.dll. В Windows NT перенесена часть подсистемы Win32 из польз режима в режим ядра. В частности, драйвер устройства режима ядра WIN32K.SYS, α управляет отображением окон, выводом на экран, вводом данных с клавиатуры или при помощи мыши и передачей сообщений. Он включает также библиотеку интерфейсов графических устройств (GDL.dll), использующую для создания графических объектов и текста.

**Вызов Win32 API-функций**

Когда приложение вызывает API-ф-ю из подсистемы Win32, может произойти одно из событий:

* если DLL подсистемы, экспортирующая данную API-функцию, содержит весь код, необходимый для выполнения функции то функция выполняется и возвращает результат;
* API-функции может потребоваться вызвать для выполнения вспомогательных действий дополнительный модуль, принадлежащий подсистеме Win32 (но не DLL, α экспортирует данную функцию);
* API-функции могут понадобиться услуги недокументированного системного сервиса.

**Исполнительная система Windows**

Сервисы исполнительной системы Windows составляют низкоуровневую часть Windows NT режима ядра, включенную в файл NTOSKRNL.EXE. Их делят на две группы: исполнительную систему, относятся к верхнему уровню, и ядро. Ядро - это самый нижний уровень ОС, реализующий наиболее фундаментальные сервисы:

* планирование потоков;
* обработку исключений;
* обработку прерываний;
* синхронизацию процессоров в многопроцессорной системе;
* создание объектов ядра.

**Составляющие исполнительной системы:**

* диспетчер процессов и потоков создает и завершает процессы и потоки, использует сервисы низкоуровнего ядра;
* диспетчер виртуальной памяти реализует механизм виртуальной памяти;
* диспетчер ввода/вывода реализует аппаратно-независимый ввод/вывод и взаимодействует с драйверами устройств;
* диспетчер КЭШа управляет кэшированием диска;
* диспетчер объектов создает объекты исполнительной системы Windows и управляет ими. Windows использует объекты для представления разнообразных ресурсов, таких как процессы и потоки;
* библиотеки времени выполнения содержат такие функции, как обработки строк и арифметические операции .

**Уровень абстрагирования от аппаратуры (HAL)**

- это библиотека режима ядра (HAL.DLL), α реализует низкоуровневый интерфейс с аппаратурой. Компоненты Windows и драйверы устройств от других компаний взаимодействуют с аппаратурой посредством HAL. Существует много версий HAL под различные аппаратные платформы. Подходящий уровень выбирается в процессе установки Windows.

**10. Основные принципы ООП. Инкапсуляция, наследование,полиморфизм**

**Инкапсуля́ция** — свойство языка программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента (что у него внутри?), а взаимодействовать с ним посредством предоставляемого интерфейса (публичных методов и членов), а также объединить и защитить жизненно важные для компонента данные. При этом пользователю предоставляется только спецификация (интерфейс) объекта. Пользователь может взаимодействовать с объектом только через этот интерфейс. Реализуется с помощью ключевого слова: public. Пользователь не может использовать закрытые данные и методы. Реализуется с помощью ключевых слов: private, protected, internal. Сокрытие реализации целесообразно применять в следующих случаях: предельная локализация изменений при необходимости таких изменений, прогнозируемость изменений (какие изменения в коде надо сделать для заданного изменения функциональности) и прогнозируемость последствий изменений.

**Насле́дование** —позволяет описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом. Другими словами, класс-наследник реализует спецификацию уже существующего класса (базовый класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса. Простое наследование: Класс, от которого произошло наследование, называется базовым или родительским (англ. base class). Классы, которые произошли от базового, называются потомками, наследниками или производными классами (англ. derived class). В некоторых языках используются абстрактные классы. Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. То есть от абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник вуза», от которого наследуются классы «аспирант», «профессор» и т. д. Так как производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза».

**Множественное наследование**

При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости. Множественное наследование реализовано в C++. Из других языков, предоставляющих эту возможность, можно отметить Python и Эйфель. Множественное наследование поддерживается в языке UML. Множественное наследование — потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках. В языках, которые позиционируются как наследники C++ (Java, C# и др.), от множественного наследования было решено отказаться в пользу интерфейсов. Практически всегда можно обойтись без использования данного механизма. Однако, если такая необходимость все-таки возникла, то, для разрешения конфликтов использования наследованных методов с одинаковыми именами, возможно, например, применить операцию расширения видимости — «::» — для вызова конкретного метода конкретного родителя. Попытка решения проблемы наличия одинаковых имен методов в предках была предпринята в языке Эйфель, в котором при описании нового класса необходимо явно указывать импортируемые члены каждого из наследуемых классов и их именование в дочернем классе. Большинство современных объектно-ориентированных языков программирования (C#, Java, Delphi и др.) поддерживают возможность одновременно наследоваться от класса-предка и реализовать методы нескольких интерфейсов одним и тем же классом. Этот механизм позволяет во многом заменить множественное наследование — методы интерфейсов необходимо переопределять явно, что исключает ошибки при наследовании функциональности одинаковых методов различных классов-предков.

**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).

**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.

**11. Общая характеристика объектов в объектно-ориентированном программировании. Объектная декомпозиция. Виды отношений между объектами.**

Объект- это совокупность способная сохранять своё состояние и обеспечивает набор операций для проверки и изменения этого состояния.

Объект является конкретным представлением абстракции, структура и поведение определяется в их общем классе. Пример объекта:

|  |
| --- |
| Стол-название |
| Вес-св-во  Стоимость  Размер |
| Операции |
| Взвесить  Переместить  Купить  Продать |

Каждый объект обладает индивидуальным состоянием и поведением.

Индивидуальность-совокупность характеристик объекта, которые отличают его от всех других.

Состояние характеризуется перечнем его свойств и их текущим значением.

Поведение характеризует, то, как объект воздействует на другие объекты или подвергался воздействию в терминах изменения его состояний и передачи сообщений.

Операция означает обслуживание, которое объект предполагает своим клиентам.

Возможны 5 видов операций над объектом:

-модификатор, изменение состояний объекта.

-селектор, даёт доступ к состоянию не изменяя его.

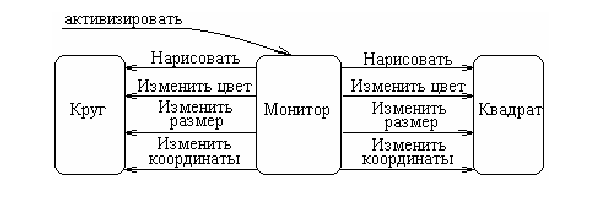
-Итератор, предоставляет доступ к содержимому объекта по частям в строго определённом порядке.

-конструктор создаёт объект и инициализирует его состояние.

-Дескриптор разгружает объект и освобождает занимаемую им память.

Все методы связанные с объектом образуют его протокол. Протокол определяет оболочку допустимого поведения объекта.

Объектная декомпозиция процесс представления предметной области задачи в виде совокупности объектов, обменивающихся сообщениями.



Все отношения между объектами могут быть сведены к двум типам:

* Ассоциация (связь) – отношение, позволяющее реализовать взаимодействие клиент-сервер;
* Агрегация (композиция) – отношение, служащее для определения понятия целое-часть и организации иерархий объектов.

Ассоциация: чтобы объекты могли взаимодействовать, между ними должна быть прямая или косвенная связь, с помощью которой объект-клиент может вызвать метод (послать сообщение) у сервера – объекта-адресата взаимодействия.  
Прямая связь означает, что сервер достижим со стороны клиента напрямую с использованием переменной, ссылки или указателя, а также «глобальной» видимости. Косвенная связь может быть реализована с помощью ассоциативного контейнера, ключа или специального механизма, который может обеспечить доступ от клиента к серверу.

При рассмотрении взаимодействия, можно выделить три основные категории объектов:

* Объекты-актеры – объекты, которые действуют на другие, но сами не подвергаются воздействию;
* объекты-серверы – могут подвергаться воздействию со стороны, но не выступающие в роли взаимодействующих объектов;
* объекты-агенты – выступают как в активной, так и пассивной роли.

Агрегация означает отношение объектов в иерархии «Целое часть»

Два вида:

1)Включение по величине, физическое включение или композиция.

2)Включение по ссылке.

***15 Упрощённая модель компилятора. Блоки и проходы компилятора***

ЛА

СА

ГК

таблица

СА- синтаксический анализатор; ЛА – лексический анализатор; ГК – генератор кода; СемА – семантический анализатор.

Исходный текст программы обрабатывает ЛА. Основное назначение ЛА – разбить исходный текст, представляющий собой последовательность символов, на слова или лексемы. После этого исходная программа представляет собой последовательность слов, которые будет обрабатывать СА. Кроме разбиения исходного текста на слова, ЛА проверяет правильность написания каждого слова, то есть обнаруживает лексические ошибки в программе. Для ЛА порядок слов не имеет значения.

СА определяет, является ли синтаксически правильной последовательность слов, полученная ЛА. Если последовательность слов не верная, то в программе есть синтаксическая ошибка. Если ошибок нет, то СА переводит последовательность слов в последовательность лексем в таком представлении прогшраммы, в котором удобно использовать при генерации кода. Например, если рассматривать арифметические выражения, то ЛА разбивает его на операции, операнды и скобки, а СА может преобразовать исходное арифметическое выражение в обратную польскую запись или 3х адресный код. Эти формы удобны для ГК, так как они записаны в той последовательности, в которой они должны выполняться.

ЛА и СА машинно-независимы. ГК переводит полученные данные в программу для конкретной машины. ГК может содержать оптимизатор кода, который преобразует последовательность команд в такую, которая получает тот же результат, но работает более эффективно. СА и ГК содержат семантический анализатор. СемА может генерировать различный код для вычисления значений выражений в зависимости от типов операндов. СемА проверяет семантические соглашения исходного языка. Если они нарушены, то объектный код не может быть получен. Пример семантического соглашения:

* 1. каждый идентификатор должен быть описани и только один раз;
  2. каждый идентификатор должен быть описан до его использования;
  3. соблюдение согласования типов в выражениях, в операторах присваивания;
  4. соблюдение согласованности параметров подпрограмм.

СемА может проверять смысловые нормы языка – правила, нарушение которых говорит о не качественности программы, но объектный код может быть получен. Пример смысловых норм:  
1) каждый идентификатор должен быть использован;

2) использованию переменной должна предшествовать ее инициализация;

3) результат функции должен быть определен при ее выполнении;

4) каждый оператор должен иметь возможность выполниться.

Основной блок – СА. Он может выполнять функции и ЛА и ГК. Наличие ЛА позволяет:

* Сокращать размер информации, обрабатываемой СА
* Упростить функции СА
* Использовать различные методы проектирования СА и ЛА
* Позволяет упростить переход на новые версии языков
* Отделить машинно-зависимое от машинно-независимого

Взаимодействие блоков компилятора: параллельное и последовательное. Рассмотрим ЛА и СА. При последовательном взаимодействии блоков ЛА обрабатывает весь текст исходной программы и создает таблицу лексем, каждая строка таблицы – лексема и порядок следования лексем таблицы совпадает с порядком следования лексем в исходном тексте, затем СА обрабатывает эту таблицу.

При параллельном взаимодействии работает СА, и если ему нужна очередная лексема, он обращается к ЛА, который ее выделяет и передает СА. После ее обработки СА опять обращается к ЛА и тд.

Аналогично могут взаимодействовать ГК и СА. В зависимости от взаимодействия блоков компиляторы могут быть однопроходные и многопроходные. Проход – процесс последовательного чтения данных из внешней памяти, их обработка и помещение результата во внешнюю память. Если блоки взаимодействуют последовательно, то требуется 3 прохода ЛА-> СА->ГК. Если параллельно, то один проход ГК-СА-ЛА. Возможны двухпроходные схемы ЛА<->СА->ГК или ЛА-СА<->ГК.

ЛА-первая фаза программной обработки языков. Читает символы программы на исходном я зыке и строит лексемы (слова) исходного языка. Лексема – структурная единица языка, которая состоит из символов алфавита и не содержит в себе других структурных единиц языка. Лексемами языка программирования являются цепочки символов, представляющие собой идентификаторы, константы, ключевые слова, знаки операции и тд. Каждая лексема имеет тип и, возможно, значение, позволяющее отличить ее от другой лексемы того же типа. Последовательность типов лексем обрабатывается СА. Значением лексемы типа идентификатор может быть ссылка на элемент таблицы идентификаторов, которая хранит основные характеристики идентификаторов. Для переменной это может быть: имя, ее тип, область памяти ну и тп. Не все характеристики идентификаторов могут быть определены ЛА. Например, для переменной, ее имя может быть определено ЛА, а тип- нет. Тип может быть определен СА, а область памяти – ГК.

ЛА может представлять собой подпрограмму – функцию, которая возвращает тип очередной выделенной лексемы, а значение заносит в некоторую глобальную переменную. Подпрограмма ЛА может вызывать СА при параллельном взаимодействии, и тип, и значение сразу обрабатываются СА. При последовательном взаимодействии эта подпрограмма вызывается многократно, пока не будет обработан текст программы или не найдена ошибка. На каждом шаге тип лексем и ее значения заносятся в таблицу лексем. Порядок следования лексем в таблице такой же, как в исходной программе. Например, если ЛА обрабатывает цепочку for i:=15 to n\*(m-k) do таблица лексем будет иметь следующий вид:

|  |  |  |
| --- | --- | --- |
| имя | тип | значение |
| for | key for | - |
| i | id |  |
| := | op := | - |
| 15 | const | 15 |
| to | key to | - |
| n | id |  |
| \* | mul | 1 (по номеру операции) |
| ( | ( | - |
| m | id |  |
| - | add | 2 |
| k | id |  |
| ) | ) | - |

|  |  |
| --- | --- |
| имя | параметры |
| i |  |
| n |  |
| m |  |
| k |  |

Ла может при его вызове из очередной последовательности символов создать лексемму. Такой анализатор называется прямым. Непрямому ЛА задается тип лексемы, которая определяет действие «распознать» на очередном шаге. Такой анализатор возвращает истину, если очередная последовательность обрабатываемых символов – есть лексемы заданного типа, и ложь иначе. Непрямой ЛА (НЛА) вызывается из СА. Если ЛА возвращает ложь, возможно, СА опять обратиться к ЛА с другим типом лексемы, тогда ЛА будет заново обрабатывать ту же последовательность символов с целью формирования лексемы другого типа. НЛА может быть использован при параллельном взаимодействии с СА, когда одна последовательность символов может представлять собой различные лексемы, в зависимости от ее окружения. Кроме основной задачи выделения лексем, ЛА выполняет также и другие задачи, например: удаление из исходной программы комментариев, удаление пустых символов и тп, также обнаружение ошибок (лексических и синтаксических), например:

* 1. ЛА проверяет парность лексем (if…then, (), и тп). Для того, чтобы обнаружить такого типа ошибки, используется массив. Индекс массива соответствует паре лексем. Сначала все элементы массива равны 0, далее, если встречается первый элемент пары соответствующий элемент массива увеличивается на 1, если второй элемент пары - уменьшается на 1. Если после обработки всего текста исходной программы все элементы такого массива равны 0, то этого типа синтаксических ошибок в программе нет, если хотя бы один элемент не равен нулю, то может быть выдано сообщение об ошибке, чего и сколько не хватает.
  2. Сочетаемость/порядок ключевых слов. (за for – to, за while – do и тп). Для обнаружения таких ошибок используется матрица, в которой строки и столбцы соответствуют ключевым словам. Если за i-тым может следовать j – ое ключевое слово, то табл[i,j] =1 и 0 иначе. Для того,чтобы обнаружить ошибку такого типа, нужно знать предыдущее ключевое слово. После прочтения очередного ключевого слова – обращение к элементу матрицы, и если он равен нулю – есть ошибка.
  3. Сочетаемость рядом стоящих лексем. Например, после for может быть идентификатор, но не наоборот. Для обнаружения можно составить матрицу, как в 2.
  4. Согласование местоположения ошибки, лексической или синтаксической и текста исходной программы.
  5. Можно выполнить простейший синтаксический анализ. Например, распознавание унарного минуса, а не бинарного, так как с точки зрения ЛА – это одна и та же лексема.

***31 Преобразования КС-грамматик. Применение преобразований при проектировании трансляторов.***

*Формальная грамматика* G =(N,A,P,S), где

N – конечное множество нетерминальных символов (нетерминалов),

A – конечное множество терминальных символов (терминалов),

P – конечное множество правил (продукций) вида α→β, где α и β - последовательности терминальных и нетерминальных символов,

S – начальный нетерминал.

Формальная грамматика G порождает цепочки языка L(G), поэтому она называется порождающей. Множество A терминальных символов грамматики представляет собой алфавит языка L(G). Если A состоит только из строчных букв, то в качестве элементов N будем использовать заглавные буквы, иначе элементы N (нетерминалы) будем заключать в угловые скобки.

*Цепочка, выводимая в грамматике* G = (N,A,P,S) , удовлетворяет следующим условиям:

1) S – выводимая цепочка;

2) если цепочка αβγ - выводимая, и β→δ∈P, то αδγ - выводимая (обозначается αβγ ⇒ αδγ , читается “αδγ непосредственно выводится из αβγ”)

Если α1 ⇒ α2 ⇒ … ⇒ αn , то αn - цепочка, выводимая из α1 , обозначается α1 ⇒ \*αn .

*Терминальная цепочка –* это цепочка, выводимая в грамматике G и несодержащая нетерминалов – принадлежит языку L(G).

*Промежуточная цепочка (сентенциальная форма вывода) –* это цепочка, выводимая в грамматике G и содержащая нетерминалы.

*КС-грамматика* – это класс формальных грамматик, правила которых имеет вид B→α , где B∈N, т.е. левая часть каждого правила представляет собой только один нетерминал, а на правую часть правил ограничений нет. Правила грамматики будем нумеровать. Пример КС-грамматики G1:

1. S → aABc

2. S → ε

3. A → cSB

4. A → Ab

5. B → bB

6. B → a

Вывод в КС-грамматике начинается с начального нетерминала. В процессе вывода один из нетерминалов промежуточной цепочки заменяется правой частью правила, левая часть которого представляет собой заменяемый нетерминал. Будем говорить, что выполняется замена нетерминала по правилу. При записи вывода под заменяемыми нетерминалами будем записывать номер правила, по которому выполняется замена.

**Пример**. Вывод терминальной цепочки в грамматике G1:

S ⇒ aABc ⇒ aAbBc ⇒ acSBbBc ⇒ acSabBc ⇒ acabBc ⇒ acabac

1 4 3 6 2 6

Вывод можно представить *деревом вывода* . Дерево вывода удовлетворяет следующим условиям:

- каждая вершина дерева обозначается терминалом или нетерминалом;

- корнь дерева обозначается начальным нетерминалом;

- каждый лист дерева обозначается терминалом или символом пустой цепочки ε;

- если некоторая вершина дерева обозначена нетерминалом А, то её сыновья обозначены символами правой части правила A → α, применяемого при выводе.

Если прочитать листья дерева вывода слева направо, то получим терминальную цепочку.

**Пример**. Дерево вывода цепочки acabac в грамматике G1:

S

a A B c

A b a

c S B

ε b

Дерево вывода может быть представлено в *линейной скобочной форме* (ЛСФ). Записываем начальный нетерминал (корень дерева), за которым в скобках перечисляем сыновей. Если сын представляет собой нетерминал, то за ним в скобках перечисляем его сыновей и т.д. ЛСФ дерева вывода цепочки acabac в грамматике G1 (см. выше) имеет вид: S(aA(A(cS()B(b))b)B(a)c). ЛСФ представляет собой язык описания деревьев вывода. Грамматика этого языка следующая:

1. <Дерево вывода> → нетерминал (<Деревья>)

2. <Деревья> → <Дерево><Деревья>

3. <Деревья> →ε

4. <Дерево> → терминал

5. <Дерево> → нетерминал (<Деревья>)

ЛСФ можно получить, выполнив обход дерева вывода в глубину, расставляя скобки. Если в ЛСФ оставить только терминалы, то получим выводимую цепочку.

В дереве вывода отражено, какие правила применялись и к каким нетерминалам, но порядок применения правил скрыт, поэтому может быть много выводов, соответствующих одному и тому же дереву вывода.

**Пример**. Приведённому выше дереву соответствует другой вывод:

S ⇒ aABc ⇒ aAbBc ⇒ acSBbBc ⇒ acBbBc ⇒ acabBc ⇒ acabac

1 4 3 2 6 6

Вывод, на каждом шаге которого правило вывода применяется к самому левому нетерминалу промежуточной цепочки, называется *левым* (или *левосторонним*).

Вывод, на каждом шаге которого паравило вывода применяется к самому правому нетерминалу промежуточной цепочки, называется *правым* (или *правосторонним*).

Для каждого дерева существует единственный левый и единственный правый вывод (могут совпадать).

Выводы, соответствующие одному и тому же дереву, называются *эквивалентными.* Эквивалентные выводы отличаются только последовательностью применения правил.

Если одной терминальной цепочке соответствуют несколько деревьев вывода, то грамматика называется *неоднозначной*. Грамматика G1 – неоднозначная, т.к. цепочка acabac имеет другое дерево вывода:

S

a A B c

c S B b B

ε a a

Вывод, соответствующий этому дереву:

S ⇒ aABc ⇒ aAbBc ⇒ acSBbBc ⇒ acSabBc ⇒ acabBc ⇒ acabac

1 5 3 6 2 6

Проблема определения неодназначности грамматики алгоритмически неразрешима [ ], однако существуют определённого вида правила грамматики, по наличию которых в множестве правил можно утверждать, что грамматика является неоднозначной. Эти правила имеют следующий вид:

1. A → AA A → α

2. A → АαА A → β

3. A → αА A → Аβ A → χ

4. A → αА A → αАβА A → χ

Грамматики Gi и GJ называются *эквивалентными*, если равны порождаемые ими языки. Получение грамматики GJ, эквивалентной исходной грамматике Gi , назовём *эквивалентным преобразованием грамматики Gi в GJ*. Преобразование грамматики Gi в GJ выполняется применением определённых правил преобразования. Множество правил образуют *систему преобразований*. Система преобразований называется *полной*, если для любых двух эквивалентных грамматик Gi и GJ существует последовательность правил из заданной системы преобразований, результатом применения которой к грамматике Gi  является грамматика GJ. Для КС-грамматик не существует полной системы преобразований, следовательно рассматри-ваемые ниже правила преобразований не образуют полную систему преобра-зований, но обычно позволяют получить грамматику, эквивалентную исходной, обладающую заданными свойствами.

**Устранение лишних символов**

Среди символов КС-грамматики можно выделить две группы лишних символов:

1) *бесплодные нетерминалы* – это нетерминалы, которые не порожда-ют ни одной терминальной цепочки. Такие нетерминалы не могут участво-вать в выводе терминальных цепочек.

2) *недостижимые символы* – это символы (терминалы и нетерми-налы), которые не встречаются ни в одной цепочки (терминальной или промежуточной), выводимой из начального нетерминала.

Правила, в которые входят лишние символы (в левую или правую часть правила), можно исключить из множества правил грамматики.

Поиск бесплодных нетерминалов сводится к нахождению продук-тивных нетерминалов и исключению их из множества всех нетерминалов грамматики. Терминальный или нетерминальный символ называется *продуктивным (живым)*, если из него выводится терминальная цепочка. Любой терминальный символ продуктивный. Нетерминальный символ А будет продуктивным, если существует правило А→α, в котором все символы в правой части продуктивны.

*Алгоритм нахождения всех продуктивных нетерминалов.*

1. Принять множество продуктивных нетерминалов Р=∅.

2. Если существует правило А→α, в котором все символы в правой части продуктивны, то нетерминал А включить в множество Р.

3. Повторять п.2, пока множество Р растёт.

Поиск недостижимых символов сводится к нахождению всех достижимых символов и исключению их из множества всех нетерминалов грамматики. Терминальный или нетерминальный символ называется *достижимым*, если он может появиться в какой-нибудь цепочке, выводимой из начального нетерминала. Начальный нетерминал - достижимый. Терминальный или нетерминальный символ будет достижимым, если он находится в правой части правила А→α и А – достижимый нетерминал.

*Алгоритм нахождения всех достижимых символов.*

1. Принять множество достижимых символов Р={S}, где S – начальный нетерминал.

2. Если существует правило А→α и нетерминал А принадлежит множеству Р, то все символы правой части включить в множество Р.

3. Повторять п.2, пока множество Р растёт.

*Алгоритм устранения всех лишних символов.*

1. В исходной грамматике G найти все бесплодные нетерминалы и исключить правила, связанные с ними. В результате получим грамматику G’ без бесплодных нетерминалов.

2. В грамматике G’ найти все недостижимые символы и исключить правила, связанные с ними. В результате получим грамматику G’’ без лишних символов.

После исключения бесплодных нетерминалов в грамматике могут появиться недостижимые символы, которые в исходной грамматике были достижимыми. Если же в грамматике нет бесплодных нетерминалов, то они не могут появиться в результате исключения недостижимых символов.

**Пример** устранения лишних символов.

Грамматика:

1. S→ac 4. B→aSA

2. S→bA 5. C→bC

3. A→cBC 6. C→d

1. Поиск продуктивных нетерминалов.

В множество продуктивных нетерминалов Р включаем нетерминал S (правило 1) и нетерминал С (правило 6). Получаем Р={S,C} и увеличить множество Р не можем.

2. Поиск бесплодных нетерминалов.

Из множества всех нетерминалов исключаем все продуктивные и получаем множество {A,B} бесплодных нетерминалов.

3. Исключение бесплодных нетерминалов.

Исключаем правила 2, 3 и 4, т.к. они содержат бесплодные нетерминалы. Получаем грамматику:

1. S→ac

5. C→bC

6. C→d

4. Поиск достижимых символов.

Достижимыми символами являются S, a и c.

5. Поиск недостижимых символов.

Из множества всех символов исключаем все достижимые и получаем множество {C,b,d} недостижимых символов.

6. Исключение недостижимых символов.

Исключаем правила 5 и 6, т.к. они содержат недостижимые символы. Получаем грамматику:

1. S→ac

Заметим, что в исходной грамматике все символы достижимы.

**Исключение ε-правил**

Правило вида А→ε называется *ε-правилом*. Грамматику, порождаю-щую язык, несодержащий пустую цепочку, можно преобразовать в эквивалентную ей грамматику без ε-правил, а грамматику, порождающую язык, содержащий пустую цепочку, можно преобразовать в эквивалентную ей грамматику с единственным ε-правилом S→ε, где S – начальный нетерминал. Рассмотрим два алгоритма ислючения ε-правил. В первом алгоритме ис-пользуется понятие *аннулирующий нетерминал* – нетерминал, который может породить пустую цепочку. Для нахождения всех аннулирующих нетермина-лов грамматики можно использовать следующий алгоритм:

1. Из множества правил грамматики исключить все правила, содер-жащие хотя бы один терминал в правой части (ε-правила не исключаются), т.к. они не могут участвовать в выводе пустой цепочки.

2. В полученной грамматике найти множество всех продуктивных не-терминалов. Оно так же является множеством всех аннулирующих нетер-миналов в исходной грамматике.

*Алгоритм 1 исключения ε-правил*.

1. Найти множество всех аннулирующих нетерминалов.

2. Заменить каждое из правил, правые части которых содержат хотя бы по одному аннулирующему нетерминалу, множеством новых правил. Если правая часть правила содержит k вхождений аннулирующих нетерминалов, то множество, заменяющее это правило, состоит из 2k правил, соответствующим всем возможным способам удаления некоторых (или всех) из этих вхождений.

3. Исключить из множества правил грамматики все ε-правила и правила вида А→А. Если в результате выполнения п.2 получены множества одинаковых правил, то из каждого такого множества оставить только одно.

4. Если исходная грамматика порождает пустую цепочку, то добавить правило S→ε, где S – начальный нетерминал.

**Пример**.

Грамматика:

1. S→AaB 6. A→b

2. S→aB 7. B→Ba

3. S→cC 8. B→ε

4. A→AB 9. C→AB

5. A→B 10. C→c

1. Находим множество аннулирующих нетерминалов. Исключая правила, содержащие хотя бы один терминал в правой части, получим грамматику:

4. A→AB

5. A→B

8. B→ε

9. C→AB

В этой грамматике все нетерминалы продуктивные, следовательно {A,B,C} – множество аннулирующих нетерминалов.

2. Исключаем из каждого правила исходной грамматики всеми возможными способами аннулирующие нетерминалы, полученные правила добавляем в множество правил грамматики.

1\_1. S→AaB 4\_1. A→AB 7\_1. B→Ba

1\_2. S→Aa 4\_2. A→A 7\_2. B→a

1\_3. S→aB 4\_3. A→B 8\_1. B→ε

1\_4. S→a 4\_4. A→ε 9\_1. C→AB

2\_1. S→aB 5\_1. A→B 9\_2. C→A

2\_2. S→a 5\_2. A→ε 9\_3. C→B

3\_1. S→cC 6\_1. A→b 9\_4. C→ε

3\_2. S→c 10\_1. C→c

3. Исключаем из множества правил грамматики все ε-правила, правила вида А→А и из повторяющихся оставляем только одно.

1\_1. S→AaB 4\_1. A→AB 9\_1. C→AB

1\_2. S→Aa 4\_3. A→B 9\_2. C→A

1\_3. S→aB 6\_1. A→b 9\_3. C→B

1\_4. S→a 7\_1. B→Ba 10\_1. C→c

3\_1. S→cC 7\_2. B→a

3\_2. S→c

*Алгоритм 2 исключения ε-правил*.

Пока в грамматике есть ε-правила, выполнять п.1, 2 и 3.

1. Выбрать ε-правило А→ε.

2. Заменить каждое из правил, правые части которых содержат хотя бы один нетерминал А, множеством новых правил. Если правая часть правила содержит k вхождений нетерминала А, то множество, заменяющее это правило, состоит из 2k правил, соответствующим всем возможным способам удаления некоторых (или всех) из этих вхождений.

3. Исключить из множества правил грамматики правило А→ε и правила вида В→В. Если в результате выполнения п.2 получены множества одинаковых правил, то из каждого такого множества оставить только одно.

**Пример**.

Грамматика:

1. S→AaB 6. A→b

2. S→aB 7. B→Ba

3. S→cC 8. B→ε

4. A→AB 9. C→AB

5. A→B 10. C→c

1. Выбираем ε-правило 8.В→ε .

2. Исключаем из правой части каждого правила исходной грамматики всеми возможными способами вхождение нетерминала В, полученные правила добавляем в множество правил грамматики.

1\_1. S→AaB 4\_1. A→AB 7\_1. B→Ba

1\_2. S→Aa 4\_2. A→A 7\_2. B→a

2\_1. S→aB 5\_1. A→B 8\_1. B→ε

2\_2. S→a 5\_2. A→ε 9\_1. C→AB

3\_1. S→cC 6\_1. A→b 9\_2. C→A

10\_1. C→c

3. Исключаем из множества правил грамматики правило 8\_1.В→ε и правило 4\_2.А→А. Получаем грамматику:

1\_1. S→AaB 4\_1. A→AB 7\_1. B→Ba

1\_2. S→Aa 5\_1. A→B 7\_2. B→a

2\_1. S→aB 5\_2. A→ε 9\_1. C→AB

2\_2. S→a 6\_1. A→b 9\_2. C→A

3\_1. S→cC 10\_1. C→c

4. Выбираем ε-правило 5\_2.А→ε .

5. Исключаем из правой части каждого правила исходной грамматики всеми возможными способами вхождение нетерминала А, полученные правила добавляем в множество правил грамматики.

1\_1\_1. S→AaB 4\_1\_1. A→AB 9\_1\_1. C→AB

1\_1\_2. S→aB 4\_1\_2. A→B 9\_1\_2. C→B

1\_2\_1. S→Aa 5\_1\_1. A→B 9\_2\_1. C→A

1\_2\_2. S→a 5\_2\_1. A→ε 9\_2\_2. C→ε

2\_1\_1. S→aB 6\_1\_1. A→b 10\_1\_1. C→c

2\_2\_1. S→a 7\_1\_1. B→Ba

3\_1\_1. S→cC 7\_2\_1. B→a

6. Исключаем из множества правил грамматики правило 5\_2\_1.А→ε. Из каждого множества одинаковых правил {1\_1\_2, 2\_1\_1}, {1\_2\_2, 2\_2\_1}, {4\_1\_2, 5\_1\_1} оставляем по одному.

1\_1\_1. S→AaB 4\_1\_1. A→AB 9\_1\_1. C→AB

1\_1\_2. S→aB 4\_1\_2. A→B 9\_1\_2. C→B

1\_2\_1. S→Aa 6\_1\_1. A→b 9\_2\_1. C→A

1\_2\_2. S→a 7\_1\_1. B→Ba 9\_2\_2. C→ε

3\_1\_1. S→cC 7\_2\_1. B→a 10\_1\_1. C→c

7. Выбираем ε-правило 9\_2\_2.С→ε .

8. Исключаем из правой части каждого правила исходной грамматики всеми возможными способами вхождение нетерминала С, полученные правила добавляем в множество правил грамматики.

1\_1\_1\_1. S→AaB 4\_1\_1\_1. A→AB 9\_1\_1\_1. C→AB

1\_1\_2\_1. S→aB 4\_1\_2\_1. A→B 9\_1\_2\_1. C→B

1\_2\_1\_1. S→Aa 6\_1\_1\_1. A→b 9\_2\_1\_1. C→A

1\_2\_2\_1. S→a 7\_1\_1\_1. B→Ba 9\_2\_2\_1. C→ε

3\_1\_1\_1. S→cC 7\_2\_1\_1. B→a 10\_1\_1\_1. C→c

3\_1\_1\_2. S→c

9. Исключаем правило 9\_2\_2\_1. С→ε.

1\_1\_1\_1. S→AaB 4\_1\_1\_1. A→AB 9\_1\_1\_1. C→AB

1\_1\_2\_1. S→aB 4\_1\_2\_1. A→B 9\_1\_2\_1. C→B

1\_2\_1\_1. S→Aa 6\_1\_1\_1. A→b 9\_2\_1\_1. C→A

1\_2\_2\_1. S→a 7\_1\_1\_1. B→Ba 10\_1\_1\_1. C→c

3\_1\_1\_1. S→cC 7\_2\_1\_1. B→a

3\_1\_1\_2. S→c

В полученной грамматике нет ε-правил, правил вида В→В и оди-наковых правил.

**Устранение цепных правил**

*Циклом (циклическим выводом)* называется вывод вида А⇒\*А, где А – нетерминал грамматики. Циклический вывод бесполезен. Циклы возможны только в том случае, если в грамматике есть *цепные правила* вида А→В, где А и В – нетерминалы грамматики. В алгоритме устранения цепных правил будем использовать множество МА – множество нетерминалов, достижимых из нетерминала А только применением цепных правил.

*Алгоритм нахождения множества МА*.

1. Исключить из правил грамматики все нецепные правила.

2. Принять множество МА ={А}, где А – нетерминал.

3. Если существует правило В→С и нетерминал В принадлежит множеству МА, то нетерминал С включить в множество МА .

4. Повторять п.3, пока множество МА растёт.

5. Исключить нетерминал А из множества МА.

*Алгоритм устранения цепных правил.*

1. Для каждого нетерминала А из множества нетерминалов грамма-тики найти множество МА.

2. Исключить из множества правил грамматики все цепные правила.

3. Для правил А→α из множества правил грамматики добавить прави-ло В→α, если А принадлежит множеству МВ .

**Пример.**

Грамматика с цепными правилами:

1. S→S+T 4. T→E

2. S→T 5. E→(S)

3. T→T\*E 6. E→a

1. Для каждого нетерминала находим множество нетерминалов, дости-жимых применением только цепных правил. Очевидно МS={T, E}, MT={E} и ME=∅.

2. Исключить из множества правил грамматики все цепные правила.

1. S→S+T

3. T→T\*E

5. E→(S)

6. E→a

3. Для правила 3. T→T\*E добавляем правило 3\_1. S→T\*E, поскольку T принадлежит МS={T, E}.

Для правила 5. Е→(S) добавляем правило 5\_1. S→(S) и 5\_2. T→(S), поскольку E принадлежит МS={T, E} и MT={E}.

Для правила 6. E→a добавляем правило 6\_1. S→а и 6\_2. Т→а, поскольку Е принадлежит МS={T, E} и MT={E}.

В результате получим:

1. S→S+T

3. T→T\*E 3\_1. S→T\*E

5. E→(S) 5\_1. S→(S) 5\_2. T→(S)

6. E→a 6\_1. S→а 6\_2. Т→а

**Замена**

Если грамматика содержит n правил А→αi, где 1≤i≤n и других правил с левой частью А нет, и в грамматике есть правило В→βАχ, то его можно заменить на n правил вида В→βαiχ, где 1≤i≤n. Такое преобразование грамматики называется *заменой*, а нетерминал А в правиле В→βАχ - *заменяемым.*

Если в грамматике есть хотя бы одно правило А→α, такое, что цепочка α содержит в себе нетерминал А, то нетерминал А назовём *саморекурсивным*, в противном случае нетерминал А назовём *несаморекурсивным.*

Если грамматика содержит n правил А→αi, где 1≤i≤n, других правил с левой частью А нет, нетерминал А несаморекурсивный, то каждое правило В→βАχ, можно заменить на n правил вида В→βαiχ, где 1≤i≤n, а правила с левой частью А исключить из грамматики. Если n=1, то правило А→α единственное с правой частью А. Такое правило называется *одиночным*. Левая часть одиночного правила является несаморекурсивным нетерминалом (иначе он бесплодный). Замена всех вхождений нетерминалов, являющихся левыми частями одиночных правил, называется *одиночной заменой*.

В правиле грамматики назовём самое левое вхождение символа в его правую часть *краем* правила (ε-правило края не имеет). Если заменяемый нетерминал является краем правила, то такая замена называется *заменой края*.

**Примеры**.

а) замена края.

Исходная грамматика: Результат замены края:

1. А→а 1. А→а

2. А→Вс 2\_1. А→аАс

3. В→аА 2\_2. А→bВс

4. B→bВ 3. В→аА

4. B→bВ

Заменяемый нетерминал В (край второго правила) саморекурсивный, поэтому третье и четвёртое правило остаются в множестве правил грамматики.

Исходная грамматика: Результат замены края:

1. А→сА 1. А→сА

2. А→Вс 2\_1. А→аАс

3. В→аА 2\_2. А→bс

4. B→b

Заменяемый нетерминал В (край второго правила) несаморекурсивный, поэтому третье и четвёртое правило исключаются из множества правил грамматики.

б) одиночная замена.

Исходная грамматика: Результат одиночной замены :

1. А→аВВ 1. А→а аАb аАb

2. А→вВ 2. А→bаАb

3. А→с 3. А→с

4. B→аАb

**Выделение нетерминала**

Преобразование, обратное замене, назовём *выделением нетерминала*.

Если грамматика содержит n правил вида В→βαiχ , где 1≤i≤n, n правил А→αi , где 1≤i≤n и других правил с левой частью А нет, то n правил вида В→βαiχ , где 1≤i≤n можно заменить одним правилом В→βАχ .

Если грамматика содержит n правил вида В→βαiχ , где 1≤i≤n, то их можно заменить одним правилом В→βАχ , где А – новый нетерминал, и ввести n правил А→αi , где 1≤i≤n.

Если в правых частях правил грамматики встречается цепочка α , то её можно заменить на новый нетерминал А и ввести правило А→α.

**Левая факторизация**

Особым случаем выделения нетерминала является левая факториза-ция.

Если n≥2 правил грамматики имеют одинаковые левые части, допустим нетерминал А, и правые части начинаются одним или несколькими одинаковыми символами (имеют общий префикс α), т.е. i-ое правило имеет вид А→αβi, где 1≤i≤n, то можно общий префикс α вынести в отдельное правило А→αВ, где В – новый нетерминал, и добавить n правил вида В→βi, где 1≤i≤n. Такое преобразование называется *левой факторизацией.* Результат применения левой факторизации неоднозначный, зависит от выбора префикса, выносимого в отдельное правило (например, в качестве префикса можно взять один символ или общий префикс наибольшей длины) и количества правил, участвующих в факторизации.

**Пример 1**.

|  |  |  |
| --- | --- | --- |
| Исходная грамматика | Первый шаг | Второй шаг |
| S→abBa  S→abBb  S→abA  B→bB  B→b  A→a | S→abC  C→Ba  C→Bb  C→A  B→bD  D→B  D→ε  A→a | S→abC  C→BE  C→A  E→a  E→b  B→bD  D→B  D→ε  A→a |

**Пример 2**.

|  |  |  |
| --- | --- | --- |
| Исходная грамматика | Первый шаг | Второй шаг |
| S→abBa  S→abBb  S→abA  B→bB  B→b  A→a | S→abBE  S→abA  E→a  E→b  B→bD  D→B  D→ε  A→a | S→abC  C→BE  C→A  E→a  E→b  B→bD  D→B  D→ε  A→a |

**Пример 3**.

|  |  |  |  |
| --- | --- | --- | --- |
| Исходная грамматика | Первый шаг | Второй шаг | Третий шаг |
| S→abBa  S→abBb  S→abA  B→bB  B→b  A→a | S→aC  C→bBa  C→bBb  C→bA  B→bD  D→B  D→ε  A→a | S→aC  C→bE  E→Ba  E→Bb  E→A  B→bD  D→B  D→ε  A→a | S→aC  C→bE  E→BF  E→A  F→a  F→b  B→bD  D→B  D→ε  A→a |

**Устранение левой рекурсии**

Правило А→χ называется *рекурсивным*, если существует вывод χ⇒\*αАβ . Если α=ε и β≠ε, то правило А→χ называется *леворекурсивным.* Правило называется *самолеворекурсивным*, если его край совпадает с левой частью. Самолеворекурсивное правило также является и леворекурсивным.

1. Исключение самолеворекурсивных правил.

Предположим, что нетерминал А имеет m самолеворекурсивных правил А→Аα i , где 1≤i≤m, и n правил А→β j , где 1≤j≤n, которые не являются самолеворекурсивными и других правил с левой частью А нет. Эти правила заменяются следующими:

А→β jВ, где 1≤j≤n, В – новый нетерминал,

В→α iВ, где 1≤i≤m,

В→ε

2. Исключение леворекурсивных правил.

*Алгоритм исключения леворекурсивных правил.*

1. Обозначить нетерминалы грамматики А1, А2,…,Аn, где n – количество нетерминалов.

2. Для каждого нетерминала грамматики Аi, где 1≤i≤n, выполнить п.3 и 4.

3. Для каждого правила вида Аi→Аjα, где 1≤j≤i-1, выполнить замену края (новые правила необходимо учитывать при выполнении п.3).

4. Исключить самолеворекурсивные правила для нетерминала Аi (новые нетерминалы далее не рассматривать).

Алгоритм применим, если грамматика не имеет циклов (цепных правил) и ε-правил. Цепные правила и ε-правила могут быть удалены предварительно. Получающаяся грамматика без левой рекурсии может иметь ε-правила.

**Пример**.

Устранить левую рекурсию в грамматике:

А1→А1аА3

А1→А2b

А2→А1c

А2→А3a

А3→А1b

А3→c

Рассматриваем нетерминал А1.

Правил вида А1→А0α в грамматике нет, т.к. нет нетерминала А0, поэтому замену края (п.3) не выполняем.

Исключаем самолеворекурсивное правило А1→А1аА3 , получаем грамматику:

А1→А2bB1

А2→А1c

А2→А3a

А3→А1b

А3→c

B1→аА3B1

B1→ε

Рассматриваем нетерминал А2.

Выполняем замену края в правиле А2→А1с , получаем грамматику:

А1→А2bB1

А2→А2bB1c

А2→А3a

А3→А1b

А3→c

B1→аА3B1

B1→ε

Исключаем самолеворекурсивное правило А2→А2bB1c , получаем грамматику:

А1→А2bB1

А2→А3aB2

А3→А1b

А3→c

B1→аА3B1

B1→ε

B2→bB1cB2

B2→ε

Рассматриваем нетерминал А3.

Выполняем замену края в правиле А3→А1b , получаем грамматику:

А1→А2bB1

А2→А3aB2

А3→А2bB1b

А3→c

B1→аА3B1

B1→ε

B2→bB1cB2

B2→ε

Выполняем замену края в правиле А3→А2bB1b , получаем грамматику: А1→А2bB1

А2→А3aB2

А3→ А3aB2bB1b

А3→c

B1→аА3B1

B1→ε

B2→bB1cB2

B2→ε

Исключаем самолеворекурсивное правило А3→А3aB2bB1b, получаем грамматику:

А1→А2bB1

А2→А3aB2

А3→cB3

B1→аА3B1

B1→ε

B2→bB1cB2

B2→ε

B3→aB2bB1bB3

B3→ε

Грамматика без леворекурсивных правил получена. Далее можем выполнить две одиночные замены и получим грамматику:

А1→ cB3aB2bB1

А3→cB3

B1→аА3B1

B1→ε

B2→bB1cB2

B2→ε

B3→aB2bB1bB3

B3→ε

***17 Регулярные языки и конечные распознаватели. Использование конечных распознавателей в трансляторах***

**Правосторонние грамматики**

В 1959 г. американский учёный-лингвист Н.Хомский предложил классифицировать формальные языки по типу правил порождающей грамматики. *Регулярный язык* – это язык, который может быть задан регулярной (*правосторонней*) грамматикой, которая может иметь правила двух видов: *А→α* или *А→αВ*, где *А* и *В* – нетерминалы, а *α* - цепочка терминалов, возможно, пустая. Другими словами, *в правой части правил может быть не более одного нетерминала и, если он есть, то занимает самую крайнюу правую позицию.* Очевидно, что правосторонняя грамматика является подклассом КС-грамматик.

Грамматику, которая может иметь правила двух видов: *А→tВ* и *А→ε* , где *А* и *В* – нетерминалы, а *t* – терминал, назовём *автоматной правосторонней* или *правосторонней грамматикой специального вида*. Любую правостороннюю грамматику можно преобразовать в автоматную правостороннюю.

*Алгоритм преобразования правосторонней грамматики в автоматную правостороннюю.*

1. Если имеются правила вида *А→α , α* - непустая цепочка термина-лов, то ввести новый нетерминал *Е* и добавить правило *Е→ε .* Затем каждое из правил вида *А→α* заменить правилом *А→αЕ.*

2. Пока в грамматике есть правила вида *А→αВ* , где *α* - цепочка более чем из одного терминала, выполнять выделение нетерминала следующим образом: правило *А→αВ* заменить правилами *А→tC* и *С→βВ* , где *t –* первый терминал в цепочке *α* , *С* – новый нетерминал, *β -* цепочка *α* без первого терминала.

3. Устранить цепные правила.

**Пример**.

Привести правостороннюю грамматику к автоматному виду.

1. S→aA 4. A→abS

2. S→b 5. A→cA

3. S→A 6. A→ε

1. В грамматике есть правило 2 вида *А→α,* поэтому введём новый нетерминал *Е* , добавим правило 7 и изменим правило 2 :

1. S→aA 4. A→abS

2. S→bЕ 5. A→cA

3. S→A 6. A→ε

7. Е→ε

2. Выполним выделение нетерминала в правиле 4, обозначив цепочку *bS* ннетерминал *В* :

1. S→aA 4.2. В→bS

2. S→bЕ 5. A→cA

3. S→A 6. A→ε

4.1. A→aВ 7. Е→ε

3. Устраним цепное правило 3 :

1. S→aA 4.1. A→aВ

2. S→bЕ 4.2. В→bS

3.1. S→aB 5. A→cA

3.1. S→cA 6. A→ε

3.1. S→ε 7. E→ε

Автоматная правосторонняя грамматика получена.

**Левосторонние грамматики**

*Левосторонняя* грамматика может иметь правила двух видов: *А→α* или *А→αВ*, где *А* и *В* – нетерминалы, а *α* - цепочка терминалов, возможно, пустая. Грамматику, которая может иметь правила двух видов: *А→Вt* и *А→ε*, где *А* и *В* – нетерминалы, а *t* – терминал, назовём *автоматной левосторонней* или *левосторонней грамматикой специального вида*. Левостороннюю грамматику можно преобразовать в автоматную левостороннюю.

*Алгоритм преобразования левосторонней грамматики в автоматную левостороннюю.*

1. Если имеются правила вида *А→α , α* - непустая цепочка термина-лов, то ввести новый нетерминал *Е* и добавить правило *Е→ε .* Затем каждое из правил вида *А→α* заменить правилом *А→Еα.*

2. Пока в грамматике есть правила вида *А→Вα* , где *α* - цепочка более чем из одного терминала, выполнять выделение нетерминала следующим образом: правило *А→Вα* заменить правилами *А→Сt* и *С→Вβ* , где *t –* последний терминал в цепочке *α* , *С* – новый нетерминал, *β -* цепочка *α* без последнего терминала.

3. Устранить цепные правила.

**Пример**.

Привести левостороннюю грамматику к автоматному виду.

1. S→Aa 4. A→Sab

2. S→b 5. A→Ac

3. S→A 6. A→ε

1. В грамматике есть правило 2 вида *А→α,* поэтому введём новый нетерминал *Е* , добавим правило 7 и изменим правило 2 :

1. S→Aa 4. A→Sab

2. S→Eb 5. A→Ac

3. S→A 6. A→ε

7. Е→ε

2. Выполним выделение нетерминала в правиле 4, обозначив цепочку *Sa* как нетерминал *В* :

1. S→Aa 4.2. В→Sa

2. S→Еb 5. A→Ac

3. S→A 6. A→ε

4.1. A→Ba 7. Е→ε

3. Устраним цепное правило 3 :

1. S→Aa 4.1. A→Ba

2. S→Eb 4.2. В→Sb

3.1. S→Ba 5. A→Ac

3.1. S→Ac 6. A→ε

3.1. S→ε 7. E→ε

Автоматная левосторонняя грамматика получена.

**Связь правосторонних и левосторонних грамматик**

Класс левосторонних грамматик эквивалентен классу правосторонних грамматик, т.е. любой язык, заданный правосторонней грамматикой, может быть задан левосторонней грамматикой и любой язык, заданный левосторон-ней грамматикой, может быть задан правосторонней грамматикой, следовательно, левосторонняя грамматика порождает регулярный язык.

Покажем, что любую левостороннюю грамматику можно преобразовать в эквивалентную ей правостороннюю грамматику и любую правостороннюю грамматику можно преобразовать в эквивалентную ей левостороннюю грамматику. Будем считать, что в исходной грамматике исключены ε-правила выполнением соответствующих преобразований.

*Алгоритм преобразования левосторонней грамматики в правостороннюю.*

1. Устранить левую рекурсию.

2. Выполнить замену края.

3. В каждом i-ом правиле вида *А→αВβ*, где *α* и *β* - непустые цепочки терминалов, выделить новый нетерминал *Ci* , определяемый правилом *Ci→Вβ* , а правило *А→αВβ* заменить на *А→α Ci.*

4. Выполнить замену края.

5. В правых частях правил грамматики подцепочки, представляющие собой правую часть правила, определяющего нетерминал *Ci* в правилах, полученных в п.3, заменить на *Ci* .

В результате устранения левой рекурсии (п.1 алгоритма) можем получить правила вида:

1.  *А→Bα1*  - в правой части нетерминал и цепочка терминалов;

2. *А→α2*  - в правой части цепочка терминалов;

3.  *B→α3B’*  - в правой части цепочка терминалов и новый нетерминал;

4.  *B’→α4B’*

5.  *B’→ε*

Правила вида 1-3 определяют нетерминалы исходной грамматики, а правила вида 4 и 5 определяют нетерминалы, введённые в грамматику при устранении левой рекурсии. Среди получаемых правил только правила вида 1 не соответствуют правилам правосторонней грамматики. В соответствии с п.2 алгоритма выполним замену края в правилах вида 1. В результате замены края можем получить правила вида 2 и правила вида:

6. *A→α3B’α1*

Правило вида 6 не соответствует правилам правосторонней грамматики. По п.3 алгоритма выполним выделение нетерминала С и получим правила вида:

7. *A→α3С*

8. *C→B’α1*

В правилах вида 8 выполним замену края (п.4 алгоритма), используя правила вида 4 и 5. В результате вместо правил вида 8 получим правила вида:

9. *С→α4B’α1*

10. *С→α1*

В правилах вида 9 есть подцепочка *B’α1* , соответствующая правой части правила вида 8, и, заменяя её на левую часть правила вида 8 (п.5 алгоритма), получаем правостороннюю грамматику.

**Пример**.

Преобразовать левостороннюю грамматику в правостороннюю.

1. A→Ba 3. B→Bca

2. A→ab 4. B→Ac

5. B→b

1. В результате устранения левой рекурсии получим:

1. A→Ba 5. B’→caB’

2. A→ab 6. В’→acB’

3. B→abcB’ 7. B’→ε

4. B→bB’

2. В результате замены края в правиле 1 получим:

1. A→ abcB’a 5. B→bB’

2. A→bB’a 6. B’→caB’

3. A→ab 7. В’→acB’

4. B→abcB’ 8. B’→ε

3. В результате выделения нетерминала *С* в правилах 1 и 2 получим:

1. A→ abcС 5. B→bB’

2. A→bС 6. B’→caB’

3. A→ab 7. В’→acB’

4. B→abcB’ 8. B’→ε

9. С→B’a

4. В результате замены края в правиле 9 получим:

1. A→ abcС 6. B’→caB’

2. A→bС 7. В’→acB’

3. A→ab 8. B’→ε

4. B→abcB’ 9. С→ caB’a

5. B→bB’ 10. С→ acB’a

11. С→ a

5. В результате замены подцепочки *B’a* в правилах 9 и 10 на нетерминал *С*  в соответствии с правилом *9. С→B’a* п.3 получим правостороннюю грамматику:

1. A→ abcС 6. B’→caB’

2. A→bС 7. В’→acB’

3. A→ab 8. B’→ε

4. B→abcB’ 9. С→ caС

5. B→bB’ 10. С→ acС

11. С→ a

При преобразовании правосторонней грамматики в левостороннюю будем использовать операцию обращения языка.

Если язык L задан порождающей грамматикой G , то язык L-1 – обращение языка L , можно задать грамматикой G’ , полученной из грамматики G обращением правой части каждого правила (изменения порядка следования символов в правой части правила на обратный).

Если язык L задан правосторонней грамматикой G , то его обращение – язык L-1 , можно задать левосторонней грамматикой G’ . Если же язык L задан левосторонней грамматикой G , то его обращение можно задать правосторонней грамматикой G’ . Естественно, что обращение обращения языка L есть язык L, т.е. L= L-1-1. Имея алгоритм преобразования левосторонней грамматики в правостороннюю и учитывая вышесказанное, получаем простой алгоритм преобразования правосторонней грамматики в левостороннюю.

*Алгоритм преобразования правосторонней грамматики в левостороннюю.*

1. Получить левостороннюю грамматику G’ путём обращения правых частей правил грамматики G. Грамматика G’ порождает обращение языка, заданного исходной грамматикой.

2. Преобразовать левостороннюю грамматику G’ в эквивалентную ей правостороннюю грамматику G’’ .

3. Получить правостороннюю грамматику G’’’ путём обращения правых частей правил грамматики G’’. Грамматика G’’’ порождает обращение языка, заданного грамматикой G’’, т.е. исходный язык.

**Пример**.

Преобразовать правостороннюю грамматику в левостороннюю.

1. A→aB 3. B→acB

2. A→ba 4. B→cA

5. B→b

1. Получим левостороннюю грамматику путём обращения правых частей правил исходной грамматики:

1. A→Ba 3. B→Bca

2. A→ab 4. B→Ac

5. B→b

2. В результате преобразования левосторонней грамматики в правостороннюю (процесс преобразования рассмотрен в предыдущем примере) получим:

1. A→ abcС 6. B’→caB’

2. A→bС 7. В’→acB’

3. A→ab 8. B’→ε

4. B→abcB’ 9. С→ caС

5. B→bB’ 10. С→ acС

11. С→ a

3. В результате обращения правых частей правил получим левостороннюю грамматику, эквивалентную исходной правосторонней:

1. A→Сcba 6. B’→B’ac

2. A→Сb 7. В’→B’ca

3. A→ba 8. B’→ε

4. B→B’cba 9. С→ Сac

5. B→B’b 10. С→ Сca

11. С→ a

Другие алгоритмы взаимного преобразования право- и левосторонних грамматик можно получить, если дать следующие интерпретации нетерминалам  *А* и *В* в правилах вида *A→αB,* где *α* - произвольная цепочка терминалов, возможно, пустая:

1) цепочки, определяемые нетерминалом  *А* , начинаются с цепочки *α* , за которой следуют цепочки, определяемые нетерминалом  *B* ;

2) цепочки, определяемые нетерминалом  *B* , это цепочки, определяемые нетерминалом  *А* , за которым следует цепочка *α* .

Первая интерпретация соответствует правилу *A→αB* правосторонней грамматике, а вторая – правилу *B→Aα* левосторонней грамматике.

Если в правосторонней (левосторонней) грамматике есть ε-правило *А→ε* , то в левосторонней (правосторонней) грамматике нетерминал *А*  - начальный. Если же в правосторонней (левосторонней) грамматике n>1 ε-пра-вил *Аi→ε* , то в левосторонней (правосторонней) грамматике должен быть начальный нетерминал *S* , отличный от нетерминалов правосторонней (левосторонней) грамматики, и n правил *S→ Аi*.

Если в правосторонней (левосторонней) грамматике нетерминал *А*  - начальный, то в левосторонней (правосторонней) грамматике должно быть ε-правило *А→ε* .

*Алгоритм преобразования правосторонней грамматики в левостороннюю.*

1. Все правила вида *A→αB* в правосторонней грамматике заменить правилами *B→Aα* левосторонней грамматики.

2. Если в правосторонней грамматике есть ε-правило *А→ε* , то в левосторонней грамматике нетерминал *А*  - начальный. Если же в правосторонней грамматике n>1 ε-пра-вил *Аi→ε* , то в левосторонней грамматике должен быть начальный нетерминал *S* , отличный от нетерминалов правосторонней грамматики, и n правил *S→ Аi*.

3. В левостороннюю грамматику добавить правило *А→ε* , где *А -* начальный нетерминал правосторонней грамматики.

*Алгоритм преобразования левосторонней грамматики в левостороннюю.*

1. Все правила вида *В→Аα* в левосторонней грамматике заменить правилами *А→αВ* правосторонней грамматики.

2. Если в левосторонней грамматике есть ε-правило *А→ε* , то в правосторонней грамматике нетерминал *А*  - начальный. Если же в левосторонней грамматике n>1 ε-пра-вил *Аi→ε* , то в правосторонней грамматике должен быть начальный нетерминал *S* , отличный от нетерминалов левосторонней грамматики, и n правил *S→ Аi*.

3. В правостороннюю грамматику добавить правило *А→ε* , где *А -* начальный нетерминал левосторонней грамматики.

**Пример 1**.

Преобразование правосторонней грамматики в левостороннюю.

|  |  |  |  |
| --- | --- | --- | --- |
| Правосторонняя  грамматика | Преобразование правил вида *A→αB* | Определение начального нетерминала | Добавление ε-правила |
| A→aB  A→baE  B→acB  B→A  B→ε  E→ε | B→Aa  E→Aba  B→Bac  A→B | S→B  S→E  B→Aa  B→Bac  A→B  E→Aba | S→B  S→E  B→Aa  B→Bac  A→B  А→ε  E→Aba |

**Пример 2**.

Преобразование левосторонней грамматики в правостороннюю.

|  |  |  |  |
| --- | --- | --- | --- |
| Левосторонняя  грамматика | Преобразование правил вида *В→Аa* | Определение начального нетерминала | Добавление ε-правила |
| A→Ba  A→Eba  B→Bac  B→A  B→ε  E→ε | B→aA  E→baA  B→acB  A→B | S→B  S→E  B→aA  B→acB  A→B  E→baA | S→B  S→E  B→aA  B→acB  A→B  А→ε  E→baA |

В примерах в первом столбце исходная грамматика, в последнем – результирующая.

Грамматики, заведомо порождающие регулярные языки (определяется по виду правил), назовём *регулярными*. Из сказанного выше следует, что к классу регулярных грамматик относятся право- и левосторонние и право- и левосторонние автоматные грамматики.

**Конечные распознаватели. Основные понятия**

Определить принадлежность заданной (входной) цепочки заданному регулярному языку можно с помощью конечного распознавателя. *Конечный распознаватель* – это конечный автомат Рабина-Скотта, который определяется пятёркой: A=<S, X, s0, δ, F>, где

S – конечное непустое множество состояний;

X – конечное непустое множество входных символов;

s0∈S – начальное состояние;

δ: S×X→S – функция переходов, которая паре (состояние, входной символ) ставит в соответствие новое состояние (состояние перехода);

F⊆S – множество допускающих состояний.

Конечный распознаватель можно представить как устройство с одним входом, которое в момент времени t может находиться в некотором состоя-нии st∈S, а на вход поступает символ xt∈X из входной цепочки. Будем говорить, что распознаватель в момент времени t находится в состоянии st и обрабатывает символ xt. В соответствии с функцией переходов распознаватель перейдёт в новое состояние st+1=δ( st, xt), st+1∈S и в следующий момент времени t+1 будет находиться в состоянии st+1 и обрабатывать следующий символ xt+1 входной цепочки. В начальный момент времени t=0 распознаватель находится в начальном состоянии s0 и обрабатывает первый символ входной цепочки. На каждом шаге распознаватель обрабатывает новый (следующий за предыдущим) символ цепочки. Если после обработки последнего символа входной цепочки распознаватель окажется в допускаю-щем состоянии sk∈F, то входная цепочка *принадлежит* языку (*цепочка допускается распознавателем*), иначе – не принадлежит языку (*цепочка отвергается распознавателем*). Другими словами, *конечный распознаватель допускает входную цепочку, если она переводит его из начального состояния в одно из допускающих (позволяет “связать” начальное состояние с одним из допускающих)*. Множество (возможно, бесконечное) всех цепочек, допускаемых конечным распознавателем А, образует *язык L(A), допускаемый распознавателем A*.

Конечный распознаватель можно задать таблицей, строки которой соответствуют входным символам, а столбцы – состояниям. Первый столбец соответствует начальному состоянию, а столбцы, соответствующие допускающим состояниям будем отмечать символом “1”. Если распознаватель из состояния sj при обработке символа xi переходит в состояние sk, то в клетке таблице на пересечении строки xi и столбца sj записывается состояние sk.

**Пример.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | 1 |
|  | s0 | s1 | s2 | s3 | s4 |
| ц | s2 | s2 | s2 | s4 | s4 |
| . | s3 | s3 | s4 |  |  |
| + | s1 |  |  |  |  |
| - | s1 |  |  |  |  |

Этой таблицей представлен конечный распознаватель, имеющий шесть состояний S={s0, s1, s2, s3, s4, Error} и четыре входных символа X={ц, ., +, -}. Состоянию Error не соответствует столбец таблицы. Это особое состояние ошибки, из которого невозможен переход в какое либо состояние, отличное от него, а, следовательно, из него недостижимо допускающее состояние. Если в процессе обработки входной цепочки распознаватель переходит в состояние ошибки, то цепочка отвергается независимо от необработанной её части. Переходам в состояние ошибки соответствуют пустые клетки. Допускающее состояние одно F={s4}, обозначено символом “1” в верхней строке.

Конечный распознаватель можно задать ориентированным графом. Вершинам графа соответствуют состояния, а дугам – переходы. В вершинах записываются состояния, а на дугах – входые символы. Если распознаватель из состояния sj при обработке символа xi переходит в состояние sk, то из вершины sj проводится дуга, отмеченная символом si, в вершину sk. Вершина, соответствующая состоянию ошибки, и дуги, ведущие в состояние ошибки, в графе не изображаются (подразумеваются). Начальное состояние отмечается стрелочкой, а допускающие состояния выделяются жирной линией.

**Пример**.

Граф распознавателя, представленного выше таблицей.

ц

ц

ц . ц

+

- .

ц

.

Конечный распознаватель называется *полностью определённым*, если определено состояние перехода для каждой пары (состояние, входной символ). В противном случае распознаватель называется *неполностью определённым* или *частичным*. В дальнейшем будем рассматривать только полностью определённые распознаватели.

Конечный распознаватель, который не может одновременно находиться более чем в одном состоянии, называется *детерминированным.* Определённый выше конечный распознаватель является детерминированным, т.к. имеет только одно начальное состояние (находится в одном состоянии в начальный момент времени) и функция переходов задаёт единственное следующее состояние (состояние перехода) для любой пары (состояние, входной символ).

Конечный распознаватель, который может одновременно находиться более чем в одном состоянии, называется *недетерминированным.* Недетерминированный распознаватель может иметь несколько начальных состояний и может перейти из состояния при обработке входного символа более чем в одно состояние. Недетерминированный конечный распознаватель не подпадает под данное ранее определение конечного распознавателя.

*Конечный недетерминированный распознаватель* определяется пятёркой: A=<S, X, S0, δ, F>, где

S – конечное непустое множество состояний;

X – конечное непустое множество входных символов;

S0⊆ S – множество начальных состояний;

δ: S×X→2S, где 2S обозначает булеан S, т.е. множество всех подмножеств множества S – функция переходов, которая паре (состояние, входной символ) ставит в соответствие подмножество состояний;

F⊆S – множество допускающих состояний.

Недетерминированный распознаватель, также как и детерминированный, можно задать таблицей или графом. В клетках таблицы недетерминированного распознавателя записываются множества состояний переходов, а столбцы, соответствующие начальным состояниям, отмечаются символом “↓”. В графе недетерминированного распознавателя из одной вершины могут выходить различные дуги, отмеченные одним и тем же входным символом.

**Пример**.

Таблица недетерминированного распознавателя.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ↓ | ↓ | ↓ |  |  | 1 |
|  | s0 | s1 | s2 | s3 | s4 | s5 |
| ц | s1,s3 | s1,s3 |  |  | s5 | s5 |
| . | s4 |  | s4 | s5 |  |  |
| + | s1,s2 |  |  |  |  |  |
| - | s1,s2 |  |  |  |  |  |

**Пример**.

Граф недетерминированного распознавателя.

. . ц

+,-

ц

ц,+,-

ц

ц ц .

Недетерминированный распознаватель может из состояния перейти в пустое множество состояний. Если при обработке входной цепочки множество текущих состояний в некоторый момент времени окажется пустым, то распознаватель прекращает работу и цепочка отвергается.

Цепочка *допускается* недетерминированным конечным распознавателем, если после обработки последнего символа входной цепочки он будет находиться во множестве состояний, содержащих в себе хотя бы одно из допускающих. Если распознаватель представить в виде графа, то допустимой цепочке соответствует путь, помеченный символами этой цепочки, из началь-ного состояния хотя бы в одно из допускающих.

Очевидно, что детерминированные конечные распознаватели являются подклассом недетерминированных конечных распознавателей.

Недетерминированный распознаватель, который может перейти на некотором шаге из состояния в состояние без обработки символа входной цепочки, называется *недетерминированным конечным распознавателем с ε-переходами*.

*Конечный недетерминированный распознаватель с ε-переходами* определяется пятёркой: A=<S, X, S0, δ, F>, где

S – конечное непустое множество состояний;

X – конечное непустое множество входных символов;

S0⊆ S – множество начальных состояний;

δ: S×{X∪ε}→2S, где 2S обозначает булеан S, т.е. множество всех подмножеств множества S – функция переходов, которая паре (состояние, входной символ) или паре (состояние, ε) ставит в соответствие подмножество состояний;

F⊆S – множество допускающих состояний.

Таблица недетерминированного распознавателя с ε-переходами отличается от таблицы недетерминированного распознавателя без ε-переходов наличием дополнительной строки, соответствующей пустому символу ε. В графе распознавателя ε-переходу соответствует дуга, отмеченная символом ε.

**Пример**.

Таблица недетерминированного распознавателя с ε-переходами.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | ↓ |  |  |  |  |  | 1 |
|  | s0 | s1 | s2 | s3 | s4 | s5 | s6 |
| ц |  | s1,s3 |  |  | s5 | s5 |  |
| . |  |  | s4 | s5 |  |  |  |
| + | s1,s2 |  |  |  |  |  |  |
| - | s1,s2 |  |  |  |  |  |  |
| ε | s1,s2 |  |  |  |  | s6 |  |

**Пример**.

Граф недетерминированного распознавателя с ε-переходами.

ц

ц

+,-,ε . ц

ε

+,-,ε ц

.

Недетерминированные конечные распознаватели являются подклассом недетерминированных конечных распознавателейс ε-переходами.

Два конечных распознавателя (детерминированных, недетерминированных с ε-переходами или без них) называются *эквивалентными*, если равны допускаемые ими языки. Процедуру получения недетерминированного конечного распознавателя без ε-переходов, эквивалентного заданному недетерминированному конечному распознавателю с ε-переходами назовём *устранением ε-переходов*. Процедуру получения детерминированного конечного распознавателя, эквивалентного заданному недетерминированному конечному распознавателю назовём *преобразованием недетерминированного конечного распознавателя в детерминированный*.

**Преобразование недетерминированных конечных распознавателей с ε-переходами в детерминированные**

Преобразование недетерминированных конечных распознавателей с ε-переходами в детерминированные выполняется в два этапа:

1) устранением ε-переходов;

2) преобразованием недетерминированного конечного распознавателя в детерминированный.

При устранении ε-переходов будем использовать понятие “ε-замы-кание“. *ε-замыканием состояния s∈S (ε(s))* назовём множество всех состояний, которые достижимы из s по ε-переходам. Само состояние s принадлежит ε-замыканию состояния s (s∈ε(s)). Если из состояния s нет ε-переходов, то ε-замыкание состояния s состоит только из одного состояния s (ε(s)={s}). Для недетерминированного конечного распознавателя с ε-переходами (см. пример выше) ε-замыкания следующие: ε(s0)={s0,s1,s2}, ε(s1)={s1}, ε(s2)={s2}, ε(s3)={s3}, ε(s4)={s4}, ε(s5)={s5,s6}, ε(s6)={s6}.

*Алгоритм устранения ε-переходов.*

1. Множеством состояний недетерминированного конечного распознавателя без ε-переходов является множество ε-замыканий недетерминированного конечного распознавателя с ε-переходами.

2. По входному символу x∈X недетерминированный конечный распознаватель без ε-переходов переходит из состояния, соответствующего ε(si), в множество состояний, соответствующих ε-замыканиям состояний переходов из каждого состояния sj∈ε(si) по входному символу x.

3. Начальными состояниями недетерминированного конечного распознавателя без ε-переходов являются состояния, соответствующие ε-замыка-ниям состояний sj∈ε(s0), где s0∈S0 , т.е. ε-замыкания состояний, достижимых из начальных по ε-переходам.

4. Допускающими состояниями недетерминированного конечного распознавателя без ε-переходов являются состояния, соответствующие ε-замыканиям состояний ε(s), в которые входят допускающие состояния недетерминированного конечного распознавателя с ε-переходами.

5. Состояния недетерминированного конечного распознавателя без ε-переходов, недостижимые из начальных, исключаются из множества состояний.

Выполнение алгоритма устранения ε-переходов будем представлять построением таблицы недетерминированного конечного распознавателя без ε-переходов. Для недетерминированного конечного распознавателя с ε-переходами (см. пример выше) имеем:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | ↓ | ↓ | ↓ |  |  | 1 | 1 |
|  | ε(s0) {s0,s1,s2} | ε(s1) {s1} | ε(s2) {s2} | ε(s3)  {s3} | ε(s4)  {s4} | ε(s5)  {s5,s6} | ε(s6)  {s6} |
| ц | {s1},{s3} | {s1},{s3} |  |  | {s5} | {s5,s6} |  |
| . | {s4} |  | {s4} | {s5} |  |  |  |
| + | {s1},{s2} |  |  |  |  |  |  |
| - | {s1},{s2} |  |  |  |  |  |  |

Обозначив ε(si) через si и исключив недостижимое из начальных состояние ε(s6) получим таблицу недетерминированного конечного распознавателя без ε-переходов:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ↓ | ↓ | ↓ |  |  | 1 |
|  | s0 | s1 | s2 | s3 | s4 | s5 |
| ц | s1,s3 | s1,s3 |  |  | s5 | s5 |
| . | s4 |  | s4 | s5 |  |  |
| + | s1,s2 |  |  |  |  |  |
| - | s1,s2 |  |  |  |  |  |

*Алгоритм 1 преобразования недетерминированного конечного распознавателя в детерминированный*.

1. Состояние детерминированного конечного распознавателя представляет собой подмножество состояний недетерминированного конечного распознавателя, а множеством состояний детерминированного конечного распознавателя является множество всех подмножеств состояний недетерминированного конечного распознавателя.

2. Начальным состоянием детерминированного конечного распознавателя является состояние, соответствующие множеству S0 начальных состояний недетерминированного конечного распознавателя.

3. По входному символу x∈X детерминированный конечный распознаватель переходит из состояния, соответствующего подмножеству Si⊆S, в состояние, соответствующее подмножеству Sj⊆S, элементами которого являются состояния переходов из каждого состояния sk∈Si по входному символу x.

4. Допускающими состояниями детерминированного конечного распознавателя являются состояния, соответствующие подмножествам состояний недетерминированного конечного распознавателя, в которые входит хотя бы одно допускающее состояние недетерминированного конечного распознавателя.

5. Состояния детерминированного конечного распознавателя, недостижимые из начального, исключаются из множества состояний.

Данный алгоритм является неэффективным, т.к. в процессе его выполнения состояния перехода определяются и для состояний, недостижимых из начального, которые затем исключаются из множества состояний. Это требует неоправданных затрат памяти для хранения состояний, соответствующих всем подмножествам состояний (2|S|) недетерминированного конечного распознавателя, и времени на их обработку. Алгоритм можно существенно упростить, если множество состояний детерминированного конечного распознавателя формировать в процессе определения состояний перехода.

*Алгоритм 2 преобразования недетерминированного конечного распознавателя в детерминированный*.

1. Начальным состоянием детерминированного конечного распознавателя сделать состояние, соответствующие множеству S0 начальных состояний недетерминированного конечного распознавателя и включить его в множество S’ состояний детерминированного конечного распознавателя (S’={S0}).

3. Пока в множестве S’ есть состояние, для которого состояния перехода не определены, определить их следующим образом: по входному символу x∈X детерминированный конечный распознаватель переходит из состояния, соответствующего подмножеству Si⊆S, в состояние, соответствующее подмножеству Sj⊆S, элементами которого являются состояния переходов из каждого состояния sk∈Si по входному символу x, состояние перехода Sj включить в множество S’ состояний детерминированного конечного распознавателя.

4. Допускающими состояниями детерминированного конечного распознавателя являются состояния, соответствующие подмножествам состояний недетерминированного конечного распознавателя, в которые входит хотя бы одно допускающее состояние недетерминированного конечного распознавателя.

Выполнение алгоритма преобразования недетерминированного конечного распознавателя в детерминированный будем представлять построением таблицы детерминированного конечного распознавателя. Построение таблицы начинается с первого столбца, который отмечается множеством начальных состояний недетерминированного конечного распознавателя. Множества, соответствующие состояниям перехода записываются в соответствующие строки. Если получено множество, которым не отмечен ни один столбец таблицы, то добавляется новый столбец и отмечается этим множеством. Построение таблицы заканчивается, когда будут определены состояния перехода для всех столбцов. Для недетерминированного конечного распознавателя (см. пример выше) имеем:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | ↓ |  |  |  | 1 |
|  | {s0,s1,s2} | {s1,s3} | {s4} | {s1,s2} | {s5} |
| ц | {s1,s3} | {s1,s3} | {s5} | {s1,s3} | {s5} |
| . | {s4} | {s5} |  | {s4} |  |
| + | {s1,s2} |  |  |  |  |
| - | {s1,s2} |  |  |  |  |

Обозначив состояние, соответствующее i-му столбцу таблицы через si-1, получим таблицу детерминированного конечного распознавателя:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | ↓ |  |  |  | 1 |
|  | s0 | s1 | s2 | s3 | s4 |
| ц | s1 | s1 | s4 | s1 | s4 |
| . | s2 | s4 |  | s2 |  |
| + | s3 |  |  |  |  |
| - | s3 |  |  |  |  |

**Эквивалентность состояний конечных детерминированных распознавателей**

Цепочки, позволяющие связать состояние si с каким либо допускающим, назовём цепочками, допускаемыми состоянием si. Язык, образованный множеством цепочек, допускаемых состоянием si, обозначим через L(si). *Два состояния si и sj эквивелентны*, если равны допускаемые ими языки L(si)=L(sj). Эквивалентными могут быть состояния одного распознавателя или состояния различных распознавателей. Если *два состояния неэквивалентны*, то любая цепочка, под действием которой одно из них переходит в допускающее состояние, а другое – в отвергающее, называется *цепочкой, различающей эти два состояния*. Два состояния эквивалентны тогда и только тогда, когда не существует различающей их цепочки. Далее будем рассматривать эквивалентность состояний одного распознавателя или различных распознавателей с одним и тем же множеством входных символов.

Алгоритм проверки эквивалентности двух состояний основан на следующем факте:

состояния si и sj эквивалентны тогда и только тогда, когда выполня-ются следующие два условия:

1) *условие подобия* - состояния si и sj должны быть либо оба допускающими, либо оба отвергающими;

2) *условие преемственности* – для всех входных символов состояния si и sj должны переходить в эквивалентные состояния.

В процессе выполнения алгоритма строится *дерево проверки эквивалентности состояний*, в вершинах которого записываются пары состояний (в корне – пара проверяемых на эквивалентность состояний), а дуги отмечаются входными символами. Вершины строящегося дерева определяются как внутренние, граничные, дублирующие, конечные или различающие. *Граничная* – это вершина, являющаяся листом и ещё не обработанная алгоритмом. После обработки эта вершина станет либо внутренней, имеющей сыновей, либо дублирующей, либо конечной, либо различающей. *Дублирующая* – это вершина, являющаяся листом и содержащая такую пару состояний, которая есть хотя бы в одной из внутренних вершин. *Конечная* – это вершина, которая содержит два одинаковых состояния. *Различающая* – это вершина, которая содержитодно допускающее и одно отвергающее состояние.

*Алгоритм проверки эквивалентности двух состояний*.

1. Построить вершину - корень дерева, записать в неё пару проверяемых состояний. Если в корне одно состояние допускающее, а другое отвергающее, то пара проверяемых состояний является неэквивалентной (нарушено условие подобия), конец алгоритма. В противном случае корень считать граничной вершиной и выполнить п.2.

2. Пока есть граничные вершины, обрабатывать их в порядке построения следующим образом.

Для каждого входного символа создать вершину, в которую провести дугу, отмеченную этим символом, из обрабатываемой вершины; в созданную вершину записать состояния переходов из состояний, записанных в обрабатываемой вершине, под действием входного символа, отмечающего входящую дугу.

Если в созданной вершине одно состояние допускающее, а другое отвергающее, то сделать эту вершину различающей; пара проверяемых состояний является неэквивалентной, конец алгоритма.

Если в созданной вершине содержится такая пара состояний, которая есть хотя бы в одной из внутренних вершин, то сделать эту вершину дублирующей.

Если в созданной вершине содержится пара одинаковых состояний, то сделать эту вершину конечной.

В остальных случаях созданную вершину считать граничной.

Обработанные граничные вершины считать внутренними.

3. Если в множестве вершин дерева нет различающей, то пара проверяемых состояний является эквивалентной, конец алгоритма.

Если в результате выполнения алгоритма построено дерево, в котором нет различающей вершины, то все пары состояний, записанные в вершинах дерева, являются эквивалентными. Если же в результате выполнения алгоритма построено дерево, в котором есть различающая вершина, то неэквивалентными будут все пары состояний, записанные в вершинах на пути от корня к различающей вершине (для них не выполняется условие преемственности). Цепочка, соответствующая пути от корня к различающей вершине, представляет собой цепочку, различающую проверяемые на эквивалентность состояния, причём это будет кратчайшая из различающих состояния цепочек, т.к. дерево проверки строится в ширину.

Если в множестве состояний распознавателя есть единственное состояние ошибки sош∈S, которое не допускает ни одной цепочки (L(sош)=∅), а любое другое состояние si∈S, допускающее или отвергающее, допускает хотя бы одну цепочку (L(si)≠∅), то состояния sош и si неэквивалентны, и в этом случае вершину, содержащую пару состояний sош и si , будем считать различающей. Если α - цепочка, соответствующая пути от корня к различающей вершине, содержащей пару состояний sош и si , то цепочка αβ недопускается одним из проверяемых на эквивалентность состояний не при каких β и допускается другим состоянием при некоторых β. При построении дерева проверки эквивалентности состояний вершины, содержащие два состояния ошибки, и дуги, ведущие в эти вершины, не будем включать в дерево.

**Пример**.

Граф конечного детерминированного распознавателя.

ц .

.

+ ц ц

. ц ц

- . ц

ц ц . ц

Дерево проверки эквивалентности состояний s1 и s2 .

1

ц .

2 3

ц .

4 5

ц

6

Вершины пронумерованы в порядке их построения. Вершина 4 – дуб-лирующая, т.к. содержит такую же пару состояний, что и вершина 2 (порядок состояний не важен), вершины 3 и 6 – конечные, остальные – внутренние, следовательно состояния s1 и s2 – эквивалентные. Пары состояний, записанные в вершинах 2 и 5 – эквивалентные.

Дерево проверки эквивалентности состояний s2 и s5 .

1

ц .

2 3

В вершине 3 состояние s3 – отвергающее, а s7 – допускающее, следовательно вершина 3 – различающая, состояния s2 и s5 – неэквивалентны, цепочка, состоящая из одного символа “точка” , является различающей для состояний s2 и s5.

**Эквивалентность конечных детерминированных распознавателей**

Конечные распознаватели А и А’ эквивалентны, если равны допускаемые ими языки L(A)=L(A’). Множество цепочек, допускаемых состоянием s0 , образует язык L(s0). Естественно, что L(A)=L(s0), если s0 – начальное состояние распознавателя А и L(A’)=L(s0’), если s0’ - начальное состояние распознавателя А’ . Следовательно, для проверки истинности равенства L(A)=L(A’), т.е. проверки эквивалентности распознавателей А и А’, достаточно проверить истинность равенства L(s0)=L(s0’), т.е. проверить эквивалентность начальных состояний распознавателей А и А’. Таким образом, проверка эквивалентности конечных распознавателей сводится к проверки эквивалентности состояний. При выполнении алгоритма проверки эквивалентности состояний различных распознавателей нужно учитывать то, что в дереве проверки не может быть конечных вершин – вершин, содержащих одинаковые состояния.

**Пример 1**.

Граф конечного распознавателя А.

ц

ц

ц . ц

+

- .

ц

.

Граф конечного детерминированного распознавателя А’.

ц .

.

+ ц ц

. ц ц

- . ц

ц ц . ц

Дерево проверки эквивалентности распознавателей А и А’.

ц + \_ .

ц . ц . ц . ц

ц . ц ц

ц

В дереве проверки эквивалентности нет различающих вершин, следовательно эквивалентны начальные состояния проверяемых на эквива-лентность распознавателей А и А’ и эквивалентны распознаватели А и А’.

**Пример 2**.

Граф конечного распознавателя А.

ц

ц

ц . ц

+

- .

ц

.

Граф конечного распознавателя А’.

ц

ц

ц ц

+ .

- .

ц

.

Дерево проверки эквивалентности распознавателей А и А’.

1

ц + \_ .

2 3 4 5

ц .

6 7

В вершине 7 состояние s4 распознавателя А –допускающее, а состояние s3 распознавателя А’ – отвергающее, следовательно распознаватели А и А’ – неэквивалентны, цепочка ц. является различающей, она допускается распознавателем А и отвергается распознавателем А’.

**Минимизация конечных детерминированных распознавателей**

Конечный детерминированный распознаватель, содержащий наимень-шее число состояний по сравнению с любым, эквивалентным ему распознавателем, называется *минимальным.* Множество всех эквивалентных между собой распознавателей называется *классом эквивалентных распознавателей.* Класс эквивалентных распознавателей может содержать несколько распознавателей с одинаковым, наименьшим числом состояний по сравнению с любым другим распознавателем из этого класса, но все они *изоморфны*, т.е. различаются только названиями состояний. Процесс получения минимального распознавателя, эквивалентного заданному, называется *минимизацией конечного распознавателя.*

Конечный распознаватель является минимальным, если он не содержит эквивалентных между собой состояний и состояний, недостижимых из начального. Поэтому процесс минимизации выполняется в два этапа:

- поиск и устранение состояний, недостижимых из начального;

- поиск и замена эквивалентных между собой состояний одним состоянием.

Первый этап минимизации.

*Алгоритм поиска состояний, недостижимых из начального.*

1. Определить множество состояний, достижимых из начального, используя алгоритм обхода графа (например в ширину или в глубину).

2. В результате исключения из множества состояний распознавателя состояний, достижимых из начального, получим множество состояний, недостижимых из начального.

*Алгоритм устранения состояний, недостижимых из начального*.

1. Исключить из множества состояний распознавателя состояния, недостижимые из начального.

При табличном способе задания распознавателя исключаются стобцы, соответствующие состояниям, недостижимым из начального, а при графовом способе – вершины, соответствующие исключаемым состояниям, и выходящие из них дуги.

Второй этап минимизации.

Второй этап минимизации можно выполнить разными способами.

Первый способ заключается в нахождении пары эквивалентных состояний и исключении одного из них до тех пор, пока есть эквивалентные состояния.

*Алгоритм поиска пары эквивалентных состояний.*

1. Перебирая пары состояний, проверять их на эквивалентность с помощью дерева проверки (см. выше).

2. Поиск заканчивается, если проверяемые состояния эквивалентны или перебраны все пары состояний.

*Алгоритм исключения одного из пары эквивалентных состояний (si,sj).*

1. Исключить состояние si.

2. Перебрать все состояния. Если из некоторого состояния sk  существует переход под действием входного символа x в состояние si, то состоянием перехода состояния sk под действием входного символа x сделать состояние sj.

Второй способ заключается в нахождении классов эквивалентных состояний, которые в дальнейшем становятся состояниями минимального распознавателя. *Класс эквивалентных состояний* – это множество всех эквивалентных между собой состояний конечного распознавателя.

Состояния, для которых не существует различающих цепочек длины k , назаваются *k-эквивалентными*. Множество всех k-эквивалентных между собой состояний называются *классом k-эквивалентных состояний*.

Для нахождения классов эквивалентных состояний нужно последовательно находить классы 0-, 1-, 2-, …,k-эквивалентных состояний, пока классы k-эквивалентных состояний не совпадают с классами k-1-эквивалентных состояний. При совпадении классов k- и k-1-эквивалентных состояний полученные классы представляют собой классы эквивалентных состояний.

Множество всех допускающих состояний образуют класс 0-экви-валентных состояний, т.к. все они допускают пустые цепочки. Множество всех отвергающих состояний образуют другой класс 0-эквивалентных состояний, т.к. все они отвергают пустые цепочки.

Для того, чтобы состояния были k-эквивалентными, они должны быть k-1-эквивалентными и под действием одинаковых входных символов должны переходить в k-эквивалентные состояния.

Процесс нахождения классов эквивалентных состояний можно представить построением таблиц переходов из состояний в классы k-эквива-лентных состояний.

**Пример**.

Таблица переходов конечного распознавателя.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ↓ |  |  |  | 1 |  |  | 1 | 1 |  |
|  | s0 | s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8 |  |
| ц | s6 | s5 | s6 | s4 | s4 | s6 | s5 | s8 |  |  |
| . | s3 | s3 | s3 |  |  | s7 | s8 |  |  |  |
| + | s1 |  |  |  |  |  |  |  |  |  |
| - | s2 |  |  |  |  |  |  |  |  |  |

Последний столбец в таблице переходов соответствует состоянию ошибки, а пустые клетки – переходам в состояния ошибки.

Отвергающие состояния {s0,s1,s2,s3,s5,s6} объединяем в класс К1 0-эквивалентных состояний, а допускающие состояния {s4,s7,s8} – в класс К2.

Таблица переходов в классы 0-эквивалентных состояний.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | K1 | | | | | | | К2 | | |
|  | s0 | s1 | s2 | s3 | s5 | s6 |  | s4 | s7 | s8 |
| ц | К1 | К1 | К1 | К2 | К1 | К1 | К1 | К2 | К2 | К2 |
| . | К1 | К1 | К1 | К1 | К2 | К2 | К1 | К1 | К1 | К1 |
| + | К1 | К1 | К1 | К1 | К1 | К1 | К1 | К1 | К1 | К1 |
| - | К1 | К1 | К1 | К1 | К1 | К1 | К1 | К1 | К1 | К1 |

Распознаватель из состояния s0 под действием входного символа ц переходит в состояние s6 , которое принадлежит классу K1 , поэтому в таблице на пересечении столбца s0 и строки ц записано K1.

Распознаватель из состояния s1 под действием входного символа + переходит в состояние ошибки , которое принадлежит классу K1 , поэтому в таблице на пересечении столбца s1 и строки + записано K1.

Распознаватель из состояния s3 под действием входного символа ц переходит в состояние s4 , которое принадлежит классу K2 , поэтому в таблице на пересечении столбца s3 и строки ц записано K2.

Распознаватель из состояния ошибки под действием любого входного символа переходит в состояние ошибки, которое принадлежит классу K1 , поэтому в таблице в столбце, соответствующем состоянию ошибки, записаны K1 для всех входных символов.

По таблице переходов в классы 0-эквивалентных состояний определяем классы 1-эквивалентных состояний. Состояния будут 1-эквивалентными, если они 0-эквивалентны и под действием одинаковых входных сигналов переходят в состояния, принадлежащие одинаковым классам 0-эквивалентных состояний, т.е. 1-эквивалентным состояниям в таблице переходов в классы 0-эквивалентных состояний соответствуют одинаково заполненные столбцы. Объединяем 1-эквивалентные состояния в классы и строим таблицу переходов в классы 1-эквивалентных состояний для определения 2-эквивалентных состояний.

Таблица переходов в классы 1-эквивалентных состояний.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | K1 | | | | K2 | K3 | | K4 | | |
|  | s0 | s1 | s2 |  | s3 | s5 | s6 | s4 | s7 | s8 |
| ц | K3 | K3 | K3 | K1 | K4 | K3 | K3 | K4 | K4 | K4 |
| . | K2 | K2 | K2 | K1 | K1 | K4 | K4 | K1 | K1 | K1 |
| + | K1 | K1 | K1 | K1 | K1 | K1 | K1 | K1 | K1 | K1 |
| - | K1 | K1 | K1 | K1 | K1 | K1 | K1 | K1 | K1 | K1 |

В классе К1 2-эквивалентными состояниями будут s0, s1 и s2. Состояние ошибки – отдельный класс 2-эквивалентных состояний. Остальные классы 1-эквивалентных состояний также являются и 2-эквивалентными. Классы 2-эквивалентных состояний не совпадают с классами 1-эквивалентных состояний, поэтому строим таблицу переходов в классы 2-эквивалентных состояний для определения классов 3-эквивалентных состояний.

Таблица переходов в классы 2-эквивалентных состояний.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | K1 | | | K2 | K3 | K4 | | K5 | | |
|  | s0 | s1 | s2 |  | s3 | s5 | s6 | s4 | s7 | s8 |
| ц | K4 | K4 | K4 | K2 | K5 | K4 | K4 | K5 | K5 | K5 |
| . | K3 | K3 | K3 | K2 | K2 | K5 | K5 | K2 | K2 | K2 |
| + | K1 | K2 | K2 | K2 | K2 | K2 | K2 | K2 | K2 | K2 |
| - | K1 | K2 | K2 | K2 | K2 | K2 | K2 | K2 | K2 | K2 |

В классе К1 3-эквивалентными состояниями будут s1 и s2. Состояние s0 – отдельный класс 3-эквивалентных состояний. Остальные классы 2-экви-валентных состояний также являются и 3-эквивалентными. Классы 3-эквива-лентных состояний не совпадают с классами 2-эквивалентных состояний, поэтому строим таблицу переходов в классы 3-эквивалентных состояний для определения классов 4-эквивалентных состояний.

Таблица переходов в классы 3-эквивалентных состояний.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | K1 | K2 | | K3 | K4 | K5 | | K6 | | |
|  | s0 | s1 | s2 |  | s3 | s5 | s6 | s4 | s7 | s8 |
| ц | K5 | K5 | K5 | K3 | K6 | K5 | K5 | K6 | K6 | K6 |
| . | K4 | K4 | K4 | K3 | K3 | K6 | K6 | K3 | K3 | K3 |
| + | K2 | K3 | K3 | K3 | K3 | K3 | K3 | K3 | K3 | K3 |
| - | K2 | K3 | K3 | K3 | K3 | K3 | K3 | K3 | K3 | K3 |

По таблице видно, что классы 4-эквивалентных состояний совпадают с классами 3-эквивалентных состояний, следовательно классы 3-эквивалент-ных состояний представляют собой классы эквивалентных состояний.

*Алгоритм формирования минимального распознавателя.*

1. Множеством состояний минимального распознавателя сделать мно-жество классов эквивалентных состояний исходного распознавателя. Класс эквивалентных состояний исходного распознавателя соответствует состоянию минимального распознавателя.

2. Начальному состоянию минимального распознавателя соответствует класс эквивалентных состояний, содержащий в себе начальное состояние исходного распознавателя.

3. Допускающим состояниям минимального распознавателя соответствуют классы эквивалентных состояний, содержащие в себе допускающие состояние исходного распознавателя.

4. Функция переходов минимального распознавателя определяется следующим образом. Если из состояний класса Ki под действием входного символа x исходный распознаватель переходит в состояния класса Kj, то в минимальном распознавателе состоянием перехода состояния Ki под действием входного символа x сделать состояние Kj.

Если классы эквивалентных состояний определялись с помощью таблиц переходов в классы k-эквивалентных состояний (см. пример выше), то для того, чтобы получить таблицу минимального распознавателя достаточно в последней таблице в каждом классе эквивалентных состояний оставить по одному столбцу, исключить строку, в которой записаны названия состояний исходного распознавателя, и отметить начальное и допускающие состояния.

**Пример** (продолжение).

Таблица переходов минимального распознавателя.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ↓ |  |  |  |  | 1 |
|  | K1 | K2 | K3 | K4 | K5 | K6 |
| ц | K5 | K5 | K3 | K6 | K5 | K6 |
| . | K4 | K4 | K3 | K3 | K6 | K3 |
| + | K2 | K3 | K3 | K3 | K3 | K3 |
| - | K2 | K3 | K3 | K3 | K3 | K3 |

Переобозначим состояния K1 через s0, K2 через s1, K4 через s2, K5 через s3, K6 через s4, состояние ошибки К3 и переходы в него исключим из таблицы.

Окончательная таблица переходов минимального распознавателя.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | ↓ |  |  |  | 1 |
|  | s0 | s1 | s2 | s3 | s4 |
| ц | s3 | s3 | s4 | s3 | s4 |
| . | s2 | s2 |  | s4 |  |
| + | s1 |  |  |  |  |
| - | s1 |  |  |  |  |

**Регулярные множества и регулярные выражения**

Элементами регулярного множества являются цепочки символов, образованные из символов алфавита X, т.е. регулярное множество представляет собой некоторый формальный язык.

*Регулярными множествами* являются:

1. ∅ - пустое множество;

2. {ε} – множество, состоящее из одной пустой цепочки;

3. {x} – множество, состоящее из одной цепочки, представляющей собой один символ алфавита X;

4. если А и В – произвольные регулярные множества, то множества А∪В, АВ, А\*, В\* также являются регулярными множествами;

5. ничто другое не является регулярным множеством.

Другими словами, множество цепочек, образованных из символов алфавита Х, является регулярным тогда и только тогда, когда оно либо ∅, либо {ε}, либо {х} для некоторого хeХ, либо его можно получить из этих множеств применением конечного числа операций объединения, конкатенации и итерации.

Одним из способов задания регулярного множества является регулярное выражение.

*Регулярными выражениями* являются:

1. 0 – регулярное выражение, обозначающее ∅;

2. ε - регулярное выражение, обозначающее {ε};

3. х - регулярное выражение, обозначающее {x};

4. если а и b – регулярные выражения, обозначающие регулярные множества А и В, то:

а+в - регулярное выражение, обозначающее множество А∪В;

ав - регулярное выражение, обозначающее множество АВ;

а\* - регулярное выражение, обозначающее множество А\*;

в\* - регулярное выражение, обозначающее множество В\*.

При записи регулярных выражений можно использовать круглые скобки, как и в обычных арифметических выражениях, для указания порядка выполнения операций. Для уменьшения числа скобок используются приоритеты операций: итерация самая приоритетная; менее приоритетна конкатенация; самый низкий приоритет у объединения. При отсутствии скобок операции выполняются слева направо с учётом приоритета.

Для сокращения записи регулярного выражения можно использовать систему выражений вида:

d1=r1

d2=r2

...

dn=rn

где di - различные имена, а каждое ri - регулярное выражение над символами Хu{d1,d2,...,di-1}, т.е. символами основного алфавита и ранее определенными символами. Таким образом, для любого ri можно построить регулярное выражение над Х, повторно заменяя имена регулярных выражений на обозначаемые ими регулярные выражения.

Регулярные множества и регулярные выражения весьма близки. Но они представляют собой разные сущности: регулярное множество – это множество цепочек (в общем случае бесконечное), а регулярное выражение – это формула (составленная из конечного числа символов), схематично показывающая, как может быть построено соответствующее её регулярное множество.

**Пример**.

Регулярные выражения r1 и r2 в алфавите Х={ц, .,+,-}:

r1=((ц + (+ + -)ц)ц\*. + (. + (+ + -).)ц)ц\*

r2=цц\*.ц\* + (+ + -)цц\*.ц\* + (+ + -).цц\* + .цц\*

Одно и то же регулярное множество может быть представлено различными регулярными выражениями, например выражениями r1 и r2 (см. пример). Два регулярных выражения называются *эквивалентными*, если они определяют одно и то же регулярное множество (выражения r1 и r2 - эквивалентны, r1 = r2 ).

Для любых регулярных выражений a, b и c справедливы равенства:

1. a+b = b+a 6. a+a\* = a\* 11. a+a = a

2. a+(b+c) = (a+b)+c 7. aa\* = a\*a 12. 0a = a0 = 0

3. a(b+c) = ab+ac 8. a(ba)\* = (ab)\*a 13. 0+a = a+0 = a

4. (a+b)c = ac+bc 9. (a\*)\* = a\* 14. εa = aε = a

5. a(bc) = (ab)c 10. 0\* = ε 15. ε+a\* = a\*+ε = a\*

Эти равенства можно доказать, проверяя равенство соответствующих множеств цепочек. Их можно использовать для преобразования, в том числе и для упрощения регулярных выражений.

**Регулярные выражения и конечные распознаватели**

Любой регулярный язык может быть задан регулярным выражением и может быть распознан конечным распознавателем, следовательно для каждого конечного распознавателя существует регулярное выражение, определяющее тот язык, который допускается распознавателем, и, обратно, для каждого регулярного выражения существует конечный распознаватель, который допускает только то множество цепочек, которое задаёт регулярное выражение. Далее рассмотрим алгоритм получения регулярного выражения, определяющего множество цепочек, допускаемых заданным конечным распознавателем и алгоритм построения конечного распознавателя, допускающего множество цепочек, заданных регулярным выражением.

**Получение регулярного выражения из конечного распознавателя**

Введём в рассмотрение модель *обобщённого графа переходов* как расширение графа конечного распознавателя. В обобщённом графе переходов одна начальная и произвольное количество допускающих вершин, а дуги, в отличие от графа конечного распознавателя, помечены не символами алфавита, а регулярными выражениями. *Цепрочка допускается обобщённым графом переходов*, если эта цепочка принадлежит множеству, описываемому конкатенацией регулярных выражений, которые помечают путь из начальной вершины в допускающую. *Язык, допускаемый* *обобщённым графом переходов*, представляет собой множество всех допускаемых им цепочек. Граф конечного распознавателя является частным случаем обобщённого графа переходов, поэтому все языки допускаемые конечными распознавателями, допускаются и обобщёнными графами переходов.

Обобщённый граф переходов можно представить в *нормализованной форме*, в которой только одна допускающая вершина, из которой не исходит ни одна дуга, а в единственную начальную вершину не входит ни одна дуга. Для этого в обобщённый граф переходов нужно ввести новую допускающую вершину и провести в неё дуги - ε-переходы - из каждой допускающей вершины (за искючением введённой вершины), а исходные допускающие вершины не считать допускающими. Если в начальную вершину входит хотя бы одна дуга, то ввести новую вершину, провести из неё дугу - ε-переход – в начальную вершину и начальной вершиной считать только введённую вершину.

Нормализованный обобщённый граф переходов можно преобразовать в граф, содержащий только две вершины – одну начальную и одну допускающую, которые соединены одной дугой, отмеченной регулярным выражением, описывающим множество, допускаемое исходным графом. Такое преобразование выполняется путём исключения дуг и вершин. При исключении дуг и вершин применяются следующие три правила:

1) правило исключения “параллельных” дуг. Две дуги с одинаковым началом, одинаковым концом и с метками r1 и r2 соответственно, заменяются одной дугой с теми же началом и концом, отмеченной выражением r1 + r2.

r1

=> r1 + r2

r2

r1  r2 =>r1 + r2

2) правило исключения “петель”. Если существует вершина, отличная от начальной и допускающей, являющаяся началом и концом дуги, отмеченной r , то такая дуга (“петля”) исключается, а метки всех выходящих из вершины дуг конкатенируются слева с r\* . Если же ни одна дуга (после исключения “петли”) не выходит из вершины, то вершина исключается вместе со всеми входящими в неё дугами. Вершина исключается и в том случае, если в неё не входит ни одна дуга (исключается вместе со всеми выходящими из неё дугами).

r1 r\*r1

r2 => r\*r2

r

r3 r\*r3

3) правило исключения вершин. Вершина без петли, отличная от начальной и допускающей, исключается. Каждый путь от вершины, из которой выходит дуга, ведущая в исключаемую вершину, до вершины, в которую входит дуга, исходящая из исключаемой вершины, заменяется дугой, отмеченной конкатенацией меток на дугах рассматриваемого пути. Допускаются циклические пути и дуги, соединяющие начальную и конечную вершину пути.

r1r2

r1 r2 r1r3

=>

r4  r3 r4r2

r4r3

r5

r5

r1r2

r1 r2

=> r1r3 r4r2

r4  r3

r4r3

*Алгоритм получения регулярного выражения, определяющего множество цепочек, допускаемых заданным конечным распознавателем*.

1. Преобразовать граф конечного распознавателя в нормализованный обобщённый граф переходов.

2. Во всех возможных случаях применить правило исключения “параллельных” дуг. Во всех возможных случаях применить правило исключения “петель”. Если возможно, применить правило исключения вершины и выполнить п.2, иначе перейти к п.3.

3. Если в полученном графе не оказалось дуг, провести дугу от начальной вершины к допускающей и отметить её символом ∅.

4. Дуга, ведущая от начальной вершины к допускающей, отмечена регулярным выражением, определяющим множество цепочек, допускаемых заданным конечным распознавателем.

**Пример**.

Граф конечного распознавателя.

ц

ц

ц . ц

+

- .

ц

.

1. Преобразование графа конечного распознавателя в нормализованный обобщённый граф переходов добавлением нового допускающего состояния.

ц

ц

ц . ц

+

ε

- .

ц

.

2. Исключение “параллельных” дуг.

ц

ц

ц . ц

+ + - ε

.

ц

.

3. Исключение “петель”.

ц

ц ц\*.

+ + - ц\*

.

ц

.

4. Исключение вершины s1 .

ц

ц\*.

(+ + -) ц

ц\*

(+ + -) .

ц

.

5. Исключение “параллельных” дуг.

ц + (+ + -) ц

ц\*.

ц\*

ц

. + (+ + -) .

6. Исключение вершины s2 .

(ц + (+ + -) ц) ц\*.

ц\*

ц

. + (+ + -) .

7. Исключение вершины s3 .

(ц + (+ + -) ц) ц\*.

ц\*

(. + (+ + -) .) ц

8. Исключение “параллельных” дуг.

(ц + (+ + -) ц) ц\*. + (. + (+ + -) .) ц ц\*

9. Исключение вершины s4 .

((ц + (+ + -) ц) ц\*. + (. + (+ + -) .) ц) ц\*

10. Искомое регулярное выражение

r = ((ц + (+ + -) ц) ц\*. + (. + (+ + -) .) ц) ц\*

**Построение конечного распознавателя по регулярному выражению**

Пусть задано регулярное выражение r . Конечный распознаватель (возможно, недетерминированный), допускающий множество цепочек, определяемых регулярным выражением r, содержащий одну начальную вершину, в которую не входит ни одна дуга, и одну допускающую вершину, из которой не исходит ни одна дуга, представим следующей моделью:

Начальная В прямоугольнике Допускающая

вершина сосредоточны внут- вершина

ренние вершины

Дуга, ведущая из начальной вершины в прямоугольник, соответствует всем дугам, идущим из начальной вершины, а дуга, ведущая из прямоугольника в допускающую вершину, соответствует всем дугам, идущим в допускающую вершину при графовом представлении конечного распознавателя.

В зависимости от вида регулярного выражения r , прямоугольник в модели можно детализировать по следующим правилам:

1) если r=∅, то прямоугольник исключается

2) если r=ε , то прямоугольник заменяется одной дугой, отмеченной символом ε.

ε

3) если r=x , x∈X, то прямоугольник заменяется одной дугой, отмеченной символом x.

x

4) если r=r1+r2, то прямоугольник с меткой r заменяется двумя “параллельно” соединёнными прямоугольниками с метками r1 и r2 . Начальные и допускающие вершины прямоугольников с метками r1 и r2 совмещаются.

5) если r=r1r2, то прямоугольник с меткой r заменяется двумя “последовательно” соединёнными через дополнительную вершину прямоугольниками с метками r1 и r2 . Новая вершина представляет собой совмещение допускающей вершины прямоугольника с меткой r1 с начальной вершиной прямоугольника с меткой r1 .

6) если r=r1\*, то прямоугольник с меткой r заменяется новой вершиной с “петлёй”, на которой находится прямоугольник с меткой r1 . Дуги, соединяющие начальную и допускающую вершину с новой вершиной, отмечаются символом ε . Новая вершина представляет собой совмещение начальной и допускающей вершин для прямоугольника с меткой r1 .

ε ε

*Алгоритм построения детерминированного конечного распознавателя, допускающего множество цепочек, определяемых регулярным выраже-нием r.*

1. Конечный распознаватель, допускающий множество цепочек, определяемых регулярным выражением r , представить моделью, содержащей начальную, допускающую вершину и прямоугольник, в который вписано регулярное выражение r .

2. Пока в модели есть прямоугольники, детализировать их по правилам 1-6 .

3. Если после выполнения п.2 получен недетерминированный конечный распознаватель, то преобразовать его в детерминированный.

**Пример**.

Регулярное выражение r = ((ц + (+ + -) ц) ц\*. + (. + (+ + -) .) ц) ц\*

1. Конечный распознаватель, допускающий множество цепочек, определяемых регулярным выражением r , представим моделью:

2. Регулярное выражение r представим как r=r1r2, где

r1=((ц + (+ + -) ц) ц\*. + (. + (+ + -) .) ц)

r2=ц\*

По правилу 5 получаем:

3. Регулярное выражение r1 представим как r1=r3 + r4, где

r3=(ц + (+ + -) ц) ц\*.

r4=(. + (+ + -) .) ц

Применяя правило 4 к прямоугольнику с меткой r1 и правила 6 и 3 к прямоугольнику с меткой r2, получаем:

ц

ε ε

4. Регулярное выражение r3 представим как r3=r5r6, где

r5=(ц + (+ + -) ц) ц\*

r6=.

Регулярное выражение r4 представим как r4=r7r8, где

r7=. + (+ + -) .

r8=ц

Применяя правила 5 и 3 к прямоугольникам с метками r3 и r4, получаем:

ц

.

ε ε

ц

5. Регулярное выражение r5 представим как r5=r9r10, где

r9=ц + (+ + -) ц

r10=ц\*

Регулярное выражение r7 представим как r7=r11 + r12, где

r11=.

r12= (+ + -) .

Применяя правила 5 и 6 к прямоугольнику с меткой r5 и правила 4 и 3 к прямоугольнику с меткой r7, получаем:

ц

ε ε ц

.

ε ε

ц

.

6. Регулярное выражение r9 представим как r9=r13 + r14, где

r13=ц

r14=(+ + -) ц

Регулярное выражение r12 представим как r12=r15r16, где

r15=+ + -

r16=.

Применяя правила 4 и 3 к прямоугольнику с меткой r9 и правила 5 и 3 к прямоугольнику с меткой r12, получаем:

ц

ц ε ε ц

.

ε ε

ц

.

.

7. Регулярное выражение r14 представим как r14=r17r18, где

r17=+ + -

r18=ц

Применяя правила 5, 4 и 3 к прямоугольнику с меткой r14 и правила 4 и 3 к прямоугольнику с меткой r12, получаем недетерминированный конечный распознаватель:

ц

ц ε ε ц

+ ц

- .

ε ε

+ ц

- .

.

После преобразования недетерминированного конечного распознавателя в детерминированный получим:

ц

ц

ц . ц

+

- .

ц

.

**Конечные распознаватели и регулярные грамматики**

Любой регулярный язык порождается регулярной грамматикой и может быть распознан конечным распознавателем, следовательно для каждого конечного распознавателя существует регулярная грамматика, порождающая тот язык, который допускается распознавателем, и, обратно, для регулярной грамматики существует конечный распознаватель, который допускает только то множество цепочек, которое порождает регулярная грамматика. Далее рассмотрим алгоритмы получения грамматик, порождающих множество цепочек, допускаемых заданным конечным распознавателем и алгоритмы построения конечного распознавателя, допускающего множество цепочек, порождаемое заданной регулярной грамматикой.

**Получение грамматики из конечного распознавателя**

Пусть задан конечный распознаватель. Поставим в соответствие множеству входных символов распознавателя множество терминалов, а множеству состояний – множество нетерминалов. Дадим нетерминалу Si, соответствующему состоянию si такую интерпретацию:

<цепочки, допускаемые состоянием si >

Если в распознавателе существует переход из состояния si в sj под действием входного символа x , <цепочки, допускаемые состоянием si > могут начинаться символом (терминалом) x , за которым следуют <цепочки, допускаемые состоянием sj >, т.е. получаем правило: Si → x Sj .

Если же в распознавателе есть ε-переход из состояния si в sj , то <цепочки, допускаемые состоянием si > представляют собой <цепочки, допускаемые состоянием sj > , т.е. получаем правило Si → Sj  .

Если в распознавателе состояние si допускающее, то <цепочка, допускаемая состоянием si > может быть пустой, т.е. получаем правило: Si → ε.

Если каждому переходу и каждому допускающему состоянию распознавателя сопоставить полученные вше правила, то получим автоматную правостороннюю грамматику (правостороннюю грамматику специального вида), возможно, с цепными правилами.

*Алгоритм получения автоматной правосторонней грамматики из конечного распознавателя*.

1. Множеством терминалов грамматики сделать множество входных символов распознавателя.

2. Множеством нетерминалов грамматики сделать множество состояний распознавателя, а начальным нетерминалом – его начальное состояние.

3. Если в распознавателе есть переход из состояния si в sj под действием входного символа x , то в грамматику ввести правило: Si → x Sj .

4. Если в распознавателе есть ε-переход из состояния si в sj , то в грамматику нужно ввести правило Si → Sj  .

5. Если в распознавателе состояние si допускающее, то в грамматику ввести правило: Si → ε.

6. Устранить цепные правила.

Если применять алгоритм к детерминированному или недетерминированному конечному распознавателю без ε-переходов, то п.4 и п.6 не выполняются.

**Пример**.

Граф конечного распознавателя.

ц

ц

ц . ц

+

- .

ц

.

Автоматная правосторонняя грамматика.

S0 → + S1 S1 → ц S2 S3 → ц S4

S0 → - S1 S1 → . S3 S4 → ц S4

S0 → ц S2 S2 → ц S2 S4 → ε

S0 → . S3 S2 → . S4

Нетерминалу Si, соответствующему состоянию si , можно дать другую интерпретацию:

<цепочки, связывающие начальное состояние с состоянием si >

Если в распознавателе существует переход в состояние sj из si под действием входного символа x , то <цепочки, связывающие начальное состояние с состоянием si > могут представлять собой <цепочки, связывающие начальное состояние с состоянием sj >, к которым приписан символ (терминал) x , т.е. получаем правило: Si → Sj x .

Если в распознавателе в состояние sj существует ε-переход из si , то <цепочки, связывающие начальное состояние с состоянием si > представляют собой <цепочки, связывающие начальное состояние с состоянием sj >, т.е. получаем правило: Si → Sj .

Если в распознавателе состояние si начальное, то <цепочка, связывающая начальное состояние с состоянием si > представляет собой пустую цепочку, т.е. получаем правило: Si → ε.

Если в распознавателе состояние si допускающее, то <цепочки, связывающие начальное состояние с состоянием si > представляют собой цепочки, допускаемые распознавателем, поэтому Si  - начальный нетерминал. Если же в распознавателе n>1 допускающих состояний, то каждому допускающему состоянию должен соответствовать свой начальный нетерминал, но так как в грамматике должен быть только один начальный нетерминал, то необходимо ввести новый начальный нетерминал S и n правил вида S→ Si  для каждого допускающего состояния si .

Сопоставив каждому переходу, начальному и допускающему состоянию распознавателя полученные выше правила, получим автоматную левостороннюю грамматику, возможно, с цепными правилами.

*Алгоритм получения автоматной левосторонней грамматики из конечного распознавателя*.

1. Множеством терминалов грамматики сделать множество входных символов распознавателя.

2. Множеством нетерминалов грамматики сделать множество состояний распознавателя.

3. Если в распознавателе состояние si начальное, то в грамматику ввести правило: Si → ε.

4. Если в распознавателе есть переход в состояние si из sj под действием входного символа x , то в грамматику ввести правило: Si → Sj х.

5. Если в распознавателе есть ε-переход в состояние si из sj , то в грамматику ввести правило: Si → Sj .

6. Если в распознавателе n≥1 допускающих состояний, то ввести новый начальный нетерминал S и n правил вида S→ Si  для каждого допускающего состояния si .

7. Устранить цепные правила.

Если применять алгоритм к детерминированному или недетерминированному конечному распознавателю без ε-переходов и с одним допускающим состоянием, то п.5-7 не выполняются и в качестве начального нетерминала нужно принять нетерминал, соответствующий допускающему состоянию.

**Пример**.

Автоматная левосторонняя грамматика, полученная из распознавателя (см. пример выше).

S4 → S4 ц S3 → S0 . S1 → S0 +

S4 → S2 . S2 → S0 ц S1 → S0 -

S4 → S3 ц S2 → S1 ц S0 → ε

S3 → S1 . S2 → S2 ц

**Построение конечных распознавателей по регулярным грамматикам**

К регулярным грамматикам относятся право- и левосторонние грамматики, которые можно преобразовать в автоматные право- и левосторонние грамматики. По автоматным грамматикам можно построить конечные распознаватели, “обратив” алгоритмы получения автоматных грамматик из конечных распознавателей.

*Алгоритм построения конечного распознавателя по автоматной правосторонней грамматике.*

1. Множеством входных символов распознавателя сделать множество терминалов грамматики.

2. Множеством состояний распознавателя сделать множество нетерминалов грамматики, а начальным состоянием – состояние, соответствующее начальному нетерминалу.

3. Если в грамматике есть правило Si → x Sj , то в распознавателе ввести переход из состояния si в состояние sj под действием входного символа х.

4. Если в грамматике есть правило Si → ε, то состояние si сделать допускающим.

В результате выполнения алгоритма может быть получен детерминированный или недетерминированный распознаватель без ε-переходов с одним начальным состоянием.

*Алгоритм построения конечного распознавателя по автоматной левосторонней грамматике.*

1. Множеством входных символов распознавателя сделать множество терминалов грамматики.

2. Множеством состояний распознавателя сделать множество нетерминалов грамматики, а допускающим состоянием – состояние, соответствующее начальному нетерминалу.

3. Если в грамматике есть правило Si → Sj х , то в распознавателе ввести переход в состояние si из состояния sj под действием входного символа х.

4. Если в грамматике есть правило Si → ε, то состояние si сделать начальным.

В результате выполнения алгоритма может быть получен детерминированный или недетерминированный распознаватель без ε-переходов с одним допускающим состоянием.

***18 Лексические анализаторы. Основные функции, проектирование и методы программной реализации***

ЛА-первая фаза программной обработки языков. Читает символы программы на исходном я зыке и строит лексемы (слова) исходного языка. Лексема – структурная единица языка, которая состоит из символов алфавита и не содержит в себе других структурных единиц языка. Лексемами языка программирования являются цепочки символов, представляющие собой идентификаторы, константы, ключевые слова, знаки операции и тд. Каждая лексема имеет тип и, возможно, значение, позволяющее отличить ее от другой лексемы того же типа. Последовательность типов лексем обрабатывается СА. Значением лексемы типа идентификатор может быть ссылка на элемент таблицы идентификаторов, которая хранит основные характеристики идентификаторов. Для переменной это может быть: имя, ее тип, область памяти ну и тп. Не все характеристики идентификаторов могут быть определены ЛА. Например, для переменной, ее имя может быть определено ЛА, а тип- нет. Тип может быть определен СА, а область памяти – ГК.

ЛА может представлять собой подпрограмму – функцию, которая возвращает тип очередной выделенной лексемы, а значение заносит в некоторую глобальную переменную. Подпрограмма ЛА может вызывать СА при параллельном взаимодействии, и тип, и значение сразу обрабатываются СА. При последовательном взаимодействии эта подпрограмма вызывается многократно, пока не будет обработан текст программы или не найдена ошибка. На каждом шаге тип лексем и ее значения заносятся в таблицу лексем. Порядок следования лексем в таблице такой же, как в исходной программе. Например, если ЛА обрабатывает цепочку for i:=15 to n\*(m-k) do таблица лексем будет иметь следующий вид:

|  |  |  |
| --- | --- | --- |
| имя | тип | значение |
| for | key for | - |
| i | id |  |
| := | op := | - |
| 15 | const | 15 |
| to | key to | - |
| n | id |  |
| \* | mul | 1 (по номеру операции) |
| ( | ( | - |
| m | id |  |
| - | add | 2 |
| k | id |  |
| ) | ) | - |

|  |  |
| --- | --- |
| имя | параметры |
| i |  |
| n |  |
| m |  |
| k |  |

Ла может при его вызове из очередной последовательности символов создать лексемму. Такой анализатор называется прямым. Непрямому ЛА задается тип лексемы, которая определяет действие «распознать» на очередном шаге. Такой анализатор возвращает истину, если очередная последовательность обрабатываемых символов – есть лексемы заданного типа, и ложь иначе. Непрямой ЛА (НЛА) вызывается из СА. Если ЛА возвращает ложь, возможно, СА опять обратиться к ЛА с другим типом лексемы, тогда ЛА будет заново обрабатывать ту же последовательность символов с целью формирования лексемы другого типа. НЛА может быть использован при параллельном взаимодействии с СА, когда одна последовательность символов может представлять собой различные лексемы, в зависимости от ее окружения. Кроме основной задачи выделения лексем, ЛА выполняет также и другие задачи, например: удаление из исходной программы комментариев, удаление пустых символов и тп, также обнаружение ошибок (лексических и синтаксических), например:

* 1. ЛА проверяет парность лексем (if…then, (), и тп). Для того, чтобы обнаружить такого типа ошибки, используется массив. Индекс массива соответствует паре лексем. Сначала все элементы массива равны 0, далее, если встречается первый элемент пары соответствующий элемент массива увеличивается на 1, если второй элемент пары - уменьшается на 1. Если после обработки всего текста исходной программы все элементы такого массива равны 0, то этого типа синтаксических ошибок в программе нет, если хотя бы один элемент не равен нулю, то может быть выдано сообщение об ошибке, чего и сколько не хватает.
  2. Сочетаемость/порядок ключевых слов. (за for – to, за while – do и тп). Для обнаружения таких ошибок используется матрица, в которой строки и столбцы соответствуют ключевым словам. Если за i-тым может следовать j – ое ключевое слово, то табл[i,j] =1 и 0 иначе. Для того,чтобы обнаружить ошибку такого типа, нужно знать предыдущее ключевое слово. После прочтения очередного ключевого слова – обращение к элементу матрицы, и если он равен нулю – есть ошибка.
  3. Сочетаемость рядом стоящих лексем. Например, после for может быть идентификатор, но не наоборот. Для обнаружения можно составить матрицу, как в 2.
  4. Согласование местоположения ошибки, лексической или синтаксической и текста исходной программы.
  5. Можно выполнить простейший синтаксический анализ. Например, распознавание унарного минуса, а не бинарного, так как с точки зрения ЛА – это одна и та же лексема.

Принципы построения ЛА.

Лексемы определенного типа представляют собой множество цепочек, то есть формально – язык.Такие языки являются регулярными языками (РЯ), поэтому каждый тип лексемы может быть распознан КР (конечным распознавателем). При реализации непрямого ЛА задается тип лексемы, который и обнаруживается на каждом шаге. По типу лексемы обращаются к КР и определяют, является ли очередная последовательность символов лексемой заданного типа. Если ЛА – прямой, тогда он должен представлять собой один КР, который обнаруживает все типы лексем. Множество КР можно рассматривать как один недетерминированный конечный распознаватель, и, для того, чтобы использовать, нужно преобразовать в детерминированный. Каждый из распознавателей определенного типа лексем представляет собой детерминированный КР, множество символов которых попарно не имеют пересечения. В распознавателе можно объединить в одно все начальные состояния, чтобы получить один детерминированный распознаватель. Задача ЛА – определение границ лексем. Решение этой задачи нужно начинать с разработки языка. Возможные варианты:

1. использовать в качестве границ незначащие символы – пробелы, табуляции и тп. Это требует установки незначащих символов после каждой лексемы. Часто обозначение границ лексем не ограничивается пробелами.
2. Использование принципа выбора лексемы наибольшей длины.

При обработке текста в переменную, в которой будет храниться имя лексемы, последовательно добавляются символы и если очередной символ может быть добавлен в лексему, он добавляется, иначе символ – граница лексемы и с этого символа дальше начинается следующая лексема. Это принцип не всегда правилен.

ЛА может в процессе работы выполнять некоторые действия, например, накапливать имя лексемы, значение константы может вычисляться в процессе анализа или после. Если взаимодействие СА и ЛА последовательное, то ЛА после определения очередной лексемы создает очередной элемент таблицы лексем. ЛА пропускает комментарии.

'{'

'}'

ЛА выделяет ключевые слова. Способы:

1) может составить таблицу ключевых слов

2) Множество ключевых слов образует регулярный язык, для распознавания которого можно построить КР.

Программная реализация ЛА.

Используются методы программной реализации КР. СА обрабатывает последовательность лексем и проверяет на соответствие их требованиям языка. Множество последовательностей лексем задают исходный язык, представляющий собой формальный язык. Для языков высокого уровня этот язык является не регулярным, а КС. Основными причинами регулярности таких языков являются наличие сбалансированных скобок и структурированных операторов. Для того, чтобы убедиться, что язык не является регулярным, можно использовать лемму «о накачке» или о разрастании языка.

Пусть задан РЯ. Если язык содержит конечное число цепочек – точно регулярный. Для такого языка можно построить КР. Граф будет ациклическим, содержать конечный путь от начальной до завершающей вершины, соответствующий цепочкам исходного языка. Если язык содержит бесконечное число цепочек, то граф будет циклическим. Допустим, язык регулярный и КР языка содержит n состояний. Так как количество цепочек конечно, то распознаватель может допускать цепочки большой длины. Пусть КР обрабатывает цепочку, длина которой больше или равна количеству состояний. Тогда хотя бы одна из вершин при обработке встретится дважды. Нужно доказать регулярность языка.

z

y

x

нач

кон

Выбираемую цепочку нужно разбить на 3 такие части (\*). После разбиения определяем, принадлежат ли цепочки исходному языку. Если так разбить нельзя, то язык перерегулярный.

L={0n1n|n>0} 000…0111…1 000…0111…1 000…0111…1 – разрастание, перерегулярный

Для распознавания КС языков используются распознаватели с кассетной памятью(МП-распознаватели)

***19 Нисходящий анализ методом рекурсивного спуска***

Для СА (синтаксический анализатор) используется автомат с магазинной (стековой) памятью (сокращенно МП-автомат) – это кортеж из семи элементов P=(Q,T,Г,d,q0,Z0,F) , где

Q- конечное множество символов состояний, представляющих всевозможные состояния управляющего устройства;

T- конечный входной алфавит;

Г - конечный алфавит магазинных символов;

d- функция переходов - отображение множества Q x (Tu{e}) x Г в множество конечных подмножеств Q x Г\* , т.е.d:Q x (Tu{e}) x Г -> {Q x Г\*} ;

q0 eQ - начальное состояние управляющего устройства;

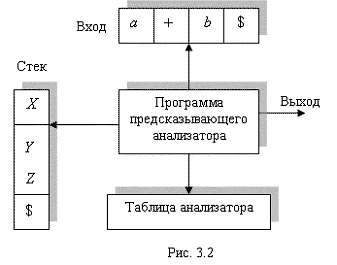
Z0 eГ- символ, находящийся в стеке в начальный момент (начальный символ);

F cQ- множество заключительных состояний

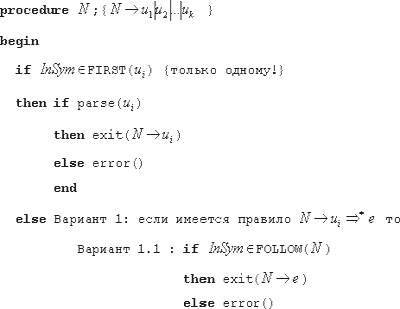
Конфигурацией (текущее состояние) МП-автомата называется тройка (q,w,u) eQ x T\* x Г\* , где

q- текущее состояние управляющего устройства;

w- неиспользованная часть входной цепочки; первый символ цепочки w находится под входной головкой; если we, то считается, что вся вхоная лента прочитана;

u- содержимое стека; самый левый символ цепочки u считается верним символом стека; если u=e, то стек считается пустым.

Начальной конфигурацией МП-автомата P называется конфигурация вида (q0,w,Z0), где weT\* , т.е. управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую нужно распознать, а в стеке есть только начальный символ Z0 (в лекциях мы как правило использовали символ S). Заключительная конфигурация - это конфигурация вида (q,e,u), где qeF, ueГ\* .

 В таком автомате определены операции:

1) Над входной цепочкой:

- сдвиг

- держать

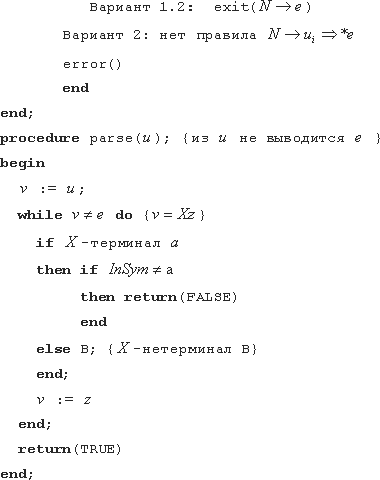
2) Над магазином:

- втолкнуть

- вытолкнуть

- не изменять

3) Над состоянием:

 - перейти в новое состояние

- не изменять

Рассмотрим работу нисходящего распознавателя: входную цепочку можно представить в виде 2 частей: a b -|, где

a - обработанная часть цепочки (изначально пустая)

b – необработанная часть

Магазин содержит цепочку j, которая представляет собой символы языка и считается что j => \*b (если так то считается что цепочка допускается). В начальный момент в магазине находится начальный нетерминал (S), и символ дна $.

Цепочку b можно представить в виде: t1 b’ -|

Магазин можно представить в виде: t2 j’ $

Если t1=t2, то t1 и t2 считаются обработанными и отбрасываются. Если t2 – нетерминал то его надо заменить на правую часть правила грамматики, которое определяет этот нетерминал.

Можно предложить вариант предсказывающего анализатра, когда каждому нетерминалу сопоставляется, вообще говоря, рекурсивная процедура и стек образуется неявно при вызовах этих процедур.

Процедуры рекурсивного спуска могут быть записаны, как это изображено на рисунке. В процедуре N для случая, когда имеется альтернатива N -> ui -> \*e (не может быть более одной альтернативы, из которой выводится e), приведены два варианта 1.1 и 1.2. В варианте 1.1 делается проверка, принадлежит ли следующий входной символ FOLLOW(N). Если нет - выдается ошибка. Во втором варианте этого не делается, так что анализ ошибки откладывается на процедуру, вызвавшую N.

**Блок 2. Вопрос 2. Системы баз данных. Архитектура систем баз данных. Современные СУБД**

Система БД – компьютеризированная система хранения записей. Саму БД можно рассматривать как подобие электронной картотеки, т.е. хранилище или контейнер для некоторого набора занесённых в компьютер файлов данных. Пользователям этой системы предоставляется возможность выполнять следующие операции:

- добавлять пустые файлы в БД;

- вставлять новые данные в существующие файлы;

- получать данные из существующих файлов;

- изменять данные в существующих файлах;

- удалять данные из существующих файлов;

- удалять существующие файлы из БД;

СБД существуют как на малых компьютерах, так и на крупных. Системы на больших машинах, как правило, многопользовательские, на малых – однопользовательские.

Однопользовательская система – это система, в которой одновременно может получить доступ к базе данных не более одного человека.

Многопользовательская система – система, в которой к базе данных могут получить доступ сразу несколько пользователей.

Данные в БД являются интегрированными и разделяемыми.

Под понятием интегрированности данных подразумевается возможность представить БД как объединение нескольких отдельных файлов данных, полностью или частично исключающее избыточность хранения информации.

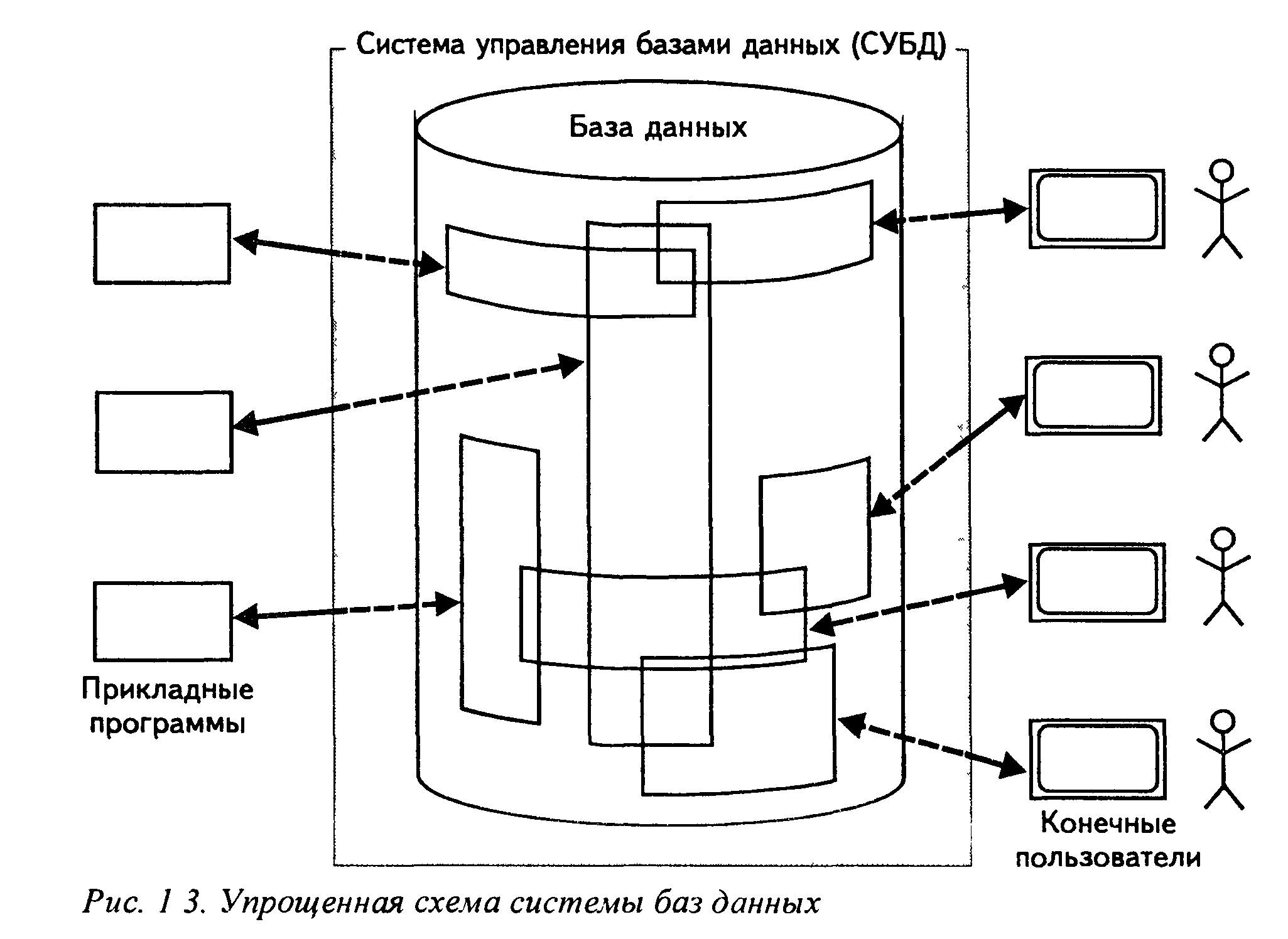
Под понятием разделяемости данных подразумевается возможность использования отдельных элементов, хранимых в БД несколькими различными пользователями.

К аппаратному обеспечению системы относится следующее:

- тома вторичной (внешней) памяти, используемые для хранения информации, а также соответствующие устройства ввода-вывода, контроллеры устройств, каналы ввода-вывода и т.д.

- аппаратный процессор (или процессоры) вместе с основной (первичной) памятью, предназначенные для поддержки работы программного обеспечения системы БД.

**Архитектура СБД. Три уровня архитектуры.**



1. Внутренний уровень (физический)

Наиболее близок к физическому хранилищу информации, т.е. связан со способом сохранения информации на физическом устройстве. Внутреннее представление отделено от физического уровня, поиском в нем рассматриваемой физической записи.

1. Концептуальный уровень (общ лог)

Промежуточный уровень между внешним и внутренним. Связан с обобщенным представлением пользователя о структуре бд. Концепт представление – представление всей инф БД в более абстракт форме по сравнению с физическим способом хранения. Оно определяется с помощью концептуальной схемы, которая включает определенные для каждого существующего типа концептуальные записи.

1. Внешний уровень (пользоват логич)

Наиболее близок к пользовательскому, т.е. связан со способом представления данных для отдельного пользователя. Система управления данными поддерживает языки управления данными.

Существует отображение одного уровня на другой:

1. Концепт внутр.

Устанавливает соответствие между концептуальным уровнем и хранимой бд. Чтобы обеспечить независимость данных, результаты внесения любых изменений в схему хранения не должны быть видны на концептуальном уровне.

1. Внешний-концепт

Определяет соответствие между внешним и концептуальным уровнем. При этом в одно и то же время может существовать несколько представлений. Одно и то же представление может принадлежать нескольким пользователям.

**Архитектура «клиент-сервер»**

На высоком уровне систему БД можно рассматривать как систему с очень простой структурой, состоящей из двух частей – сервера и набора клиентов (внешний компьютер или внешний интерфейс).

Сервер – сама СУБД. Он поддерживает все основные функции субд, а именно определение данных, обработку данных, защиту данных, поддержание целостности и т.д. В частности он предоставляет полную поддержку внешн, концепт и внутр уровней.

Клиент – различные приложения, которые выполняются поверх субд (напис польз, встроенн прилож, сторонн поставщ ПО).

Распределенная обработка – разные машины можно соединить в коммуникационную сеть. Клиент и сервер могут выполнять и зачастую на отдельных машинах, обеспечивая таким образом простой вариант распределения обработки данных. В общем случае каждый сервер может обслуживать много клиентов, а каждый клиент может работать с многими серверами. Если система обеспечивает полную прозрачность доступа (т.е. клиент работает так, как будто он имеет дело с одним сервером на единственной машине) – в таком случая мы имеем настоящую распределенную систему БД.

**СУБД -** Программное обеспечение, которое управляет всем доступом к БД.

1. Пользователь выдает запрос на доступ к данным, применяя определенный подъязык данных.
2. Субд перехватывает запрос и анализирует его
3. Субд просматривает внешние сходства для этого польз, соответственно отображается внешн-концепт, концепт схож, отображается концепт-внутр и определяется структура хранения.
4. Выполняет необходимые операции в хранилище бд.

Функции СУБД.

1. Определение данных – в структуру СУБД входит процессор языка определения данных.
2. Обработка данных – субд обрабатывает запрос пользователя на выбор, изменение, удаление существующих данных или добавление новых. Эти функции выполняет процессор языка манипуляции данных (или компилятор).
3. Оптимизация и выполнение – в состав субд входит оптимизатор – предназначен для выполнения запросов. Оптимизированные запросы выполняются подоправ менеджера времен управления.
4. Защита и сохранность целостности данных
5. Восстановление данных при поддержке параллельных схем.(менеджер транзакций).
6. Словарь данных. Метаданные – данные о данных, определяющие объект БД.
7. Производит. – максим возможность оптимизации.

Основное назначение субд = предоставление пользователю интерфейса для управления БД.

**Блок 2. Вопрос 4. Инфологическая модель данных. Диаграмма «Сущность-связь»**

**Этапы проектирования базы данных**

**Инфологическая модель данных –** описание предметной области, выполняемое с использованием естественного языка, математических формул, таблиц, графиков и других средств, понятных всем людям, работающих над проектированием БД (обобщенное неформальное описание данных). Выполняется на первом этапе проектирования.

Инфологическая модель данных является человеко-ориентированной моделью и полностью независима от физических параметров среды хранения данных. Такой средой хранения данных может быть память человека, а не компьютер. Поэтому инфологическая модель не изменяется до тех пор, пока какие-то изменения в реальном мире не потребуют внесения в нее соответствующих изменений так, чтобы эта модель продолжала отражать предметную область.

Цель инфологического этапа проектирования состоит в получении семантических (концептуальных) моделей, отражающих предметную область и информационные потребности пользователей. Поэтому этот этап называют еще как **семантическое моделирование**. Семантическое моделирование представляет собой моделирование структуры данных, опираясь на смысл этих данных.

Понятие **“Предметная область”** - базисное в теории БД и не имеет строгого определения. Оно вытекает из понятий “объект” и “предмет”. *Предметная область* **(ПО)** – часть реального мира, подлежащая изучению с целью организации управления и, в конечном итоге, автоматизации. ПО представляется множеством *фрагментов*, которые характеризуются множеством *объектов*, множеством процессов, использующих объекты, а также множеством пользователей, характеризуемых единым взглядом на предметную область.

*Объектом* называется явление внешнего мира. Это либо нечто реально существующее - человек, товар, изделие, либо процесс - учет рождаемости, получение товаров, выпуск изделий. Каждый объект обладает огромным количеством свойств.

Примеры.

Объект "*Человек*" обладает свойствами: рост, имя, дата рождения … ,

объект - "*Изделие*" обладает свойствами: качество, дата изготовления, внешний вид….

Между объектами существуют многочисленные связи. Например:

* *Человек* покупает, продает, производит *Изделие*
* *Изделие* создается, покупается, продается *Человеком*.

***Предмет*** – модель реального объекта, в котором зафиксированы лишь выделенные для ИС свойства и связи. Совокупность отобранных предметов образует ***объектное ядро*** предметной области, а совокупность их взаимосвязей - ***структуру фрагмента действительности***. Т.о. понятие “Предметная область” соответствует точке зрения потребителя на объектное ядро: в ней выделены только те объекты, свойства объектов и связи между объектами, которые представляют ценность для ИС и должны быть сохранены в БД.

Все действия по выявлению ядра предметной области производятся на этапе анализа ИС.

Объектное ядро системы в течение ЖЦ ИС не остается постоянным: пропадают и возникают объекты, меняются их свойства и взаимосвязи. Зафиксированные во времени цепочки этих изменений называются *траекториями предметной области,* а совокупность общих свойств траекторией – *семантикой предметной области*

Имеется целый ряд методик моделирования предметной области. Одна из наиболее популярных в настоящее время методик базируется на использовании графических диаграмм, включающих небольшое число разнородных компонентов ERD (Entity-Relationship Diagrams). В русскоязычной литературе эти диаграммы называют "объект – отношение" либо "сущность - связь".

Модель ERD была предложена в 1976 г. **Питером Пин-Шэн Ченом**. В дальнейшем многими авторами были разработаны свои варианты подобных моделей: нотация (notation – система обозначения, записи) Мартина, нотация IDEF1X, нотация Баркера), но все они базируются на графических диаграммах, предложенных Ченом.

На использовании разновидностей ER-модели основано большинство современных подходов к проектированию реляционных баз данных.

По сути, все варианты диаграмм сущность-связь исходят из одной идеи - рисунок всегда нагляднее текстового описания. Все такие диаграммы используют графическое изображение сущностей предметной области, их свойств (атрибутов), и взаимосвязей между сущностями.

Мы познакомимся с ER-диаграммами в нотации Баркера, как довольно легкой в понимании основных идей.

*Основные понятия ER-диаграмм*

Основными понятиями ER-модели являются сущность, связь и атрибут.

Для большей выразительности и лучшего понимания имя сущности может сопровождаться примерами конкретных объектов этого типа.

*Определение 1*. ***Сущность*** - это реальный или представляемый объект, информация о котором должна сохраняться и быть доступна. Сущностями могут быть люди, места, самолеты, рейсы, вкус, цвет и т.д.

Каждая сущность должна иметь наименование, выраженное существительным в единственном числе. При этом имя сущности - это имя типа, а не некоторого конкретного экземпляра этого типа. Понятие тип сущности относится к набору однородных личностей, предметов, событий или идей, выступающих как целое.

Примерами сущностей могут быть такие классы объектов как "Поставщик", "Сотрудник", "Накладная".

Каждая сущность в модели изображается в виде прямоугольника, содержащего имя сущности:



*Определение 2*. ***Экземпляр сущности*** - это конкретный представитель данной сущности.

Например, представителем сущности "Сотрудник" может быть "Сотрудник Иванов".

Экземпляры сущностей должны быть *различимы*, т.е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности.

*Определение 3*. ***Атрибут сущности*** - это поименованная характеристика сущности. Его наименование должно быть уникальным для конкретного типа сущности, но может быть одинаковым для различного типа сущностей (например, ЦВЕТ может быть определен для многих сущностей: СОБАКА, АВТОМОБИЛЬ, КРАСКА и т.д.). Атрибуты используются для определения того, какая информация должна быть собрана о сущности. Примерами атрибутов для сущности АВТОМОБИЛЬ являются ТИП, МАРКА, НОМЕРНОЙ ЗНАК, ЦВЕТ и т.д.

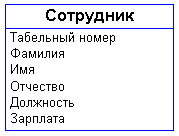
Здесь также существует различие между типом атрибута и экземпляром. Тип атрибута ЦВЕТ имеет много экземпляров или значений: Красный, Синий, Банановый, Белая ночь и т.д., однако каждому экземпляру сущности присваивается только одно значение атрибута.

Абсолютное различие между типами сущностей и атрибутами отсутствует. Атрибут является таковым только в связи с типом сущности. В другом контексте атрибут может выступать как самостоятельная сущность. Например, для автомобильного завода цвет – это только атрибут продукта производства, а для лакокрасочной фабрики цвет – тип сущности.

Каждый атрибут обеспечивается именем, уникальным в пределах сущности. Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с характеризующими прилагательными).

Примерами атрибутов сущности "Сотрудник" могут быть такие атрибуты как "Табельный номер", "Фамилия", "Имя", "Отчество", "Должность", "Зарплата" и т.п.

Атрибуты изображаются в пределах прямоугольника, определяющего сущность:



Атрибуты могут классифицироваться по принадлежности к одному из трех различных типов: описательные, указывающие, вспомогательные.

*Описательные* атрибуты представляют факты, внутренне присущие каждому экземпляру сущности.

*Указывающие* атрибуты используются для присвоения имени или обозначения экземплярам сущности.

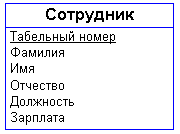
*Вспомогательные* атрибуты используются для связи экземпляра одной сущности с экземпляром другого. Атрибуты подчиняются строго определенным правилам.

*Определение 4*. ***Ключ сущности*** - минимальный набор атрибутов, по значениям которых можно однозначно найти требуемый экземпляр сущности. Минимальность означает, что исключение из набора любого атрибута не позволяет идентифицировать сущность по оставшимся.

Например, для сущности *Расписание* ключом является атрибут *Номер\_рейса* или набор: *Пункт\_отправления*, *Время\_вылета* и *Пункт\_назначения* (при условии, что из пункта в пункт вылетает в каждый момент времени один самолет).

Сущность может иметь несколько различных ключей.

Ключевые атрибуты изображаются на диаграмме подчеркиванием:



*Определение 5*. ***Связь*** - это некоторая ассоциация между *двумя* сущностями. Одна сущность может быть связана с другой сущностью или сама с собою. Связи позволяют по одной сущности находить другие сущности, связанные с нею.

Если бы назначением базы данных было только хранение отдельных, не связанных между собой данных, то ее структура могла бы быть очень простой. Однако одно из основных требований к организации базы данных – это обеспечение возможности отыскания одних сущностей по значениям других, для чего необходимо установить между ними определенные связи. А так как в реальных базах данных нередко содержатся сотни или даже тысячи сущностей, то теоретически между ними может быть установлено более миллиона связей. Наличие такого множества связей и определяет сложность инфологических моделей.

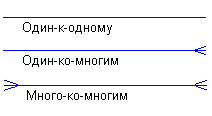
Например, связи между сущностями могут выражаться следующими фразами - "СОТРУДНИК может иметь несколько ДЕТЕЙ", "каждый СОТРУДНИК обязан числиться ровно в одном ОТДЕЛЕ".

Графически связь изображается линией, соединяющей две сущности:

****

Каждая связь имеет два конца и одно или два наименования. Наименование обычно выражается в неопределенной глагольной форме: "иметь", "принадлежать" и т.п. Каждое из наименований относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности.

Каждая связь может иметь один из следующих ***типов связи***:

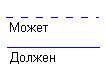


Связь типа ***один-к-одному*** означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой). Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, неправильно разделенную на две.

Связь типа ***один-ко-многим*** означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой). Это наиболее часто используемый тип связи. Левая сущность (со стороны "один") называется ***родительской***, правая (со стороны "много") - ***дочерней***. (см. рис. графического изображения связи)

Связь типа ***много-ко-многим*** означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Тип связи много-ко-многим является *временным* типом связи, допустимым на ранних этапах разработки модели. В дальнейшем этот тип связи должен быть заменен двумя связями типа один-ко-многим путем создания промежуточной сущности.

Каждая связь может иметь одну из двух ***модальностей связи***:



Модальность "***может***" означает, что экземпляр одной сущности *может быть связан* с одним или несколькими экземплярами другой сущности, *а может быть и не связан* ни с одним экземпляром.

Модальность "***должен***" означает, что экземпляр одной сущности *обязан быть связан не менее чем с одним* экземпляром другой сущности.

Связь может иметь *разную модальность* с разных концов.

Описанный графический синтаксис позволяет *однозначно* читать диаграммы, пользуясь следующей схемой построения фраз:

<Каждый экземпляр СУЩНОСТИ 1> <МОДАЛЬНОСТЬ СВЯЗИ> <НАИМЕНОВАНИЕ СВЯЗИ> <ТИП СВЯЗИ> <экземпляр СУЩНОСТИ 2>.

Каждая связь может быть прочитана как слева направо, так и справа налево. Например, связь, представленная на рисунке выше 4 читается так:

Слева направо: "каждый сотрудник может иметь несколько детей".

Справа налево: "Каждый ребенок обязан принадлежать ровно одному сотруднику".

*Нормальные формы ER-схем*

Как и в реляционных схемах баз данных, в ER-диаграмах вводится понятие нормальных форм, причем их смысл очень близко соответствует смыслу реляционных нормальных форм. Приведем только очень краткие и неформальные определения трех первых нормальных форм.

В *первой нормальной* форме ER-диаграммы устраняются повторяющиеся атрибуты или группы атрибутов, т.е. производится выявление неявных сущностей, "замаскированных" под атрибуты.

Во *второй нормальной* форме устраняются атрибуты, зависящие только от части уникального идентификатора (ключа сущности). Эта часть уникального идентификатора определяет отдельную сущность.

В *третьей нормальной* форме устраняются атрибуты, зависящие от атрибутов, не входящих в уникальный идентификатор (ключ сущности). Эти атрибуты являются основой отдельной сущности.

При правильном определении сущностей, полученные таблицы будут сразу находиться в 3НФ. Основное достоинство метода состоит в том, модель строится методом последовательных уточнений первоначальных диаграмм.

**Блок 2. Вопрос 7. Поиск информации в БД. Полнотекстовый поиск**

**Поиск информации в базах данных** – это процесс отбора из них множества описаний объектов, удовлетворяющих сформулированным в запросе условиям. При этом в качестве результатов поиска могут выдаваться не все признаки объектов, а только часть их – в соответствии с условиями запроса. Объект может выбираться из массива по значению одного идентифицирующего его (ключевого) признака или по сочетанию значений нескольких ключевых признаков. Он может также выбираться по сочетанию любых других (неключевых) признаков, если это сочетание однозначно выделяет его из множества всех объектов массива.

Различаются первичные и производные (в частности, обобщенные) признаки объектов. Первичные признаки назначаются при первоначальном описании объектов, а производные являются функциями первичных. Поиск может вестись как по первичным, так и по производным признакам. Чаще всего в процессе поиска информации выбирается не один объект, а множество объектов. Оно может быть задано различными способами:

1) перечнем значений ключевых признаков или сочетаний ключевых признаков;

2) значением или интервалом (перечнем) значений одного неключевого признака;

3) булевой функцией значений или интервалов (перечней) значений любых признаков объекта (как ключевых, так и неключевых);

4) отношением между признаками, выраженным с помощью арифметических и логических операции (операций типа “И”, “ИЛИ”, “НЕ”), а также отношений =, >, < и их отрицаний. Условия выборки признаков у найденных объектов задаются в виде перечней наименований этих признаков.

Важной проблемой, возникающей при реализации процедур поиска информации, является проблема отождествления признаков объектов и установления парадигматических отношений между ними (отношений типа род-вид, целое-часть и др.). Общее решение этой проблемы связано с возможностью распознавания смыслового тождества и парадигматических отношений различных форм наименований понятий на основе их морфологического, синтаксического и семантического анализа.

Поиск информации может выполняться за один или несколько шагов. В первом случае он ведется по одному запросу, во втором – по серии запросов. При многошаговом поиске возможны три основных способа организации процесса выполнения запросов:    
·         композиция запросов – запросы выполняются в строго определенной последовательности, а результаты поиска по предыдущему запросу используются в качестве исходных данных для формирования следующего за ним запроса. При этом первый запрос в серии запросов определяется полностью, а остальные – не полностью и доопределяются в процессе поиска.  
·          объединение запросов– когда результаты поиска по нескольким запросам объединяются в одну общую выдачу.  
·          разветвление запросов – когда после выполнения очередного запроса есть возможность перехода к одному из нескольких запросов в зависимости от выполнения тех или иных условий. Перечисленные способы организации  процесса выполнения запросов могут применяться в различных сочетаниях, что позволяет строить различные процедуры многошагового поиска.

SQLиспользует специальные операторы IN, BETWEEN, LIKE, иISNULL.

Оператор IN определяет набор значений, в котором данное значение может или не может быть включено

SELECT \*

FROM Salespeople

WHERE city IN ('Barcelona','London');

BETWEEN определяет диапазон, значений. Вы должны ввести ключ слово BETWEEN с начальным значением, ключ AND и конечное значение.

SELECT \* FROM Salespeople

WHERE comm BETWEEN .10 AND .12;

ОПЕРАТОР LIKE применим только к полям типа CHAR или VARCHAR. Он ищет поле символов, чтобы видеть, совпадает ли с условием часть его строки. Имеются два типа групп символов использования с LIKE:

* символ подчерк ( \_ ) замещает любой одиночный символ.
* знак процента (%) замещает последов люб числа символов.

Пример:SELECT

FROM Customers

WHERE cname LIKE 'G%';

**Полнотекстовый поиск.**

Полнотекстовые индексы обозначаются как индексы типа FULLTEXT. Эти индексы могут быть созданы в столбцах VARCHAR и TEXT во время создания таблицы командой CREATE TABLE или добавлении позже с помощью команды ALTER TABLE или CREATE INDEX. Загрузка больших массивов данных в таблицу будет происходить намного быстрее, если таблица не содержит индекс FULLTEXT, который затем создается командой ALTER TABLE (или CREATE INDEX). Загрузка данных в таблицу, уже имеющей индекс FULLTEXT, будет более медленной.

Полнотекстовый поиск выполняется с помощью ф-ции MATCH().

CREATE TABLE articles (

id INT UNSIGNED AUTO\_INCREMENT NOT NULL PRIMARY KEY,

title VARCHAR(200),

body TEXT,

FULLTEXT (title,body)

);

SELECT \* FROM articles

WHERE MATCH (title,body) AGAINST ('database');

Ф-ция MATCH() выполняет поиск в естественном языке, сравнивая строку с содержанием текста (совокупность одного или более столбцов, включенных в индекс FULLTEXT). Строка поиска задается как аргумент в выражении AGAINST(). Поиск выполняется без учета регистра символов.

В логическом режиме полнотекстового поиска поддерживаются след операторы:

+ это слово должно присут.

- что это слово не должно присут

( ) группирует слова в подвыражения.

~ воздейстует как оператор отрицания, обуславливает негативный вклад данного слова в релевантность строки. Им отмечают нежелательные слова.

\* оператор усечения. Он должен добавляться в конце слова.

" соответствует только строкам, содержащим эту фразу, написанную буквально.