### **Сети ЭВМ и телекоммуникации**

1. **Модель взаимодействия открытых систем ISO.**

Модель взаимодействия открытых систем OSI (Open Systems Interconnection) — базовая основопологащая модель, описывающая структуру передачи данных от одного приложения другому. Используется как абстрактная схема описания уровневого подхода описания работы Модель OSI состоит из семи концептуальных уровней. Каждый из этих уровней соответствует конкуретной задаче, соответствует выполнению определенной части некоего алгоритма. Благодаря модели OSI становится более понятна парадигма взаимодействия сетевого оборудования и программного обеспечения.

* Уровень 0. **Среда**. Данный уровень представляет посредников, соединяющих конечные компоненты сетевой структуры: кабели, радиолинии и т.д. Поскольку этот уровень де-факто не является элементом схемы, он указывает только на среду.
* Уровень 1. **Физический**. Включает физические аспекты передачи двоичной информации по линии связи. Детально описывает, например, напряжения, частоты, природу передающей среды. Этому уровню вменяется в обязанность поддержание связи и приём-передача битового потока. Безошибочность желательна, но не требуется.
* Уровень 2. **Канальный**. Обеспечивает безошибочную передачу блоков данных первый через уровень, который при передаче может искажать данные. Этот уровень должен определять начало и конец кадра в битовом потоке, формировать из данных, передаваемых физическим уровнем, кадры или последовательности кадров, включать процедуру проверки наличия ошибок и их исправления. Этот уровень (и только он) оперирует такими элементами, как битовые последовательности, методы кодирования, маркеры. Он несёт ответственность за правильную передачу данных (пакетов) на участках между непосредственно связанными элементами сети. Обеспечивает управление доступом к среде передачи.
* Уровень 3. **Сетевой**. Этот уровень пользуется возможностями, предоставляемыми вторым уровнем, для обеспечения связи любых двух точек в сети. Этот уровень осуществляет проводку сообщений по сети, которая может иметь много линий связи, или по множеству совместно работающих сетей, что требует маршрутизации, т.е. определения пути, по которому следует пересылать данные. Маршрутизация производится на этом же уровне. Выполняет обработку адресов, а также мультиплексирование и демультиплексирование. Основной функцией программного обеспечения на этом уровне является выборка информации из источника, преобразование её в пакеты и правильная передача в точку назначения.
* Уровень 4. **Транспортный**. Регламентирует пересылку данных между процессами, выполняемыми на компьютерах сети. Завершает организацию передачи данных: контролирует на сквозной основе поток данных, проходящий по маршруту, определённому третьим уровнем: правильность передачи блоков данных, правильность доставки в нужный пункт назначения, их комплектность, сохранность, порядок следования. Собирает информацию из блоков в её прежний вид. Или же оперирует с дейтаграммами, то есть ожидает отклика-подтверждения приёма из пункта назначения, проверяет правильность доставки и адресации, повторяет посылку дейтаграммы, если не пришёл отклик.
* Уровень 5. **Сеансовый**. Координирует взаимодействие связывающихся процессов: устанавливает связь, взаимодействует, восстанавливает аварийно оконченные сеансы. Он координирует не компьютеры и устройства, а процессы в сети, поддерживает их взаимодействие. То есть управляет сеансами связи между процессами прикладного уровня. Этот же уровень ответственен за картографию сети. Фактически он преобразовывает адреса, удобные для людей, в реальные сетевые адреса, например, в Internet это соответствует преобразованию региональных (доменных) компьютерных имён в числовые адреса глобальной, и наоборот.
* Уровень 6. **Представления данных**. Этот уровень имеет дело с синтаксисом и семантикой передаваемой информации. Здесь устанавливается взаимопонимание двух сообщающихся компьютеров относительно того, как они представляют и понимают по получении передаваемую информацию. На данном этапе решаются такие задачи, как перекодировка текстовой информации и изображений, сжатие и распаковка, поддержка сетевых файловых систем (NFS), абстрактных структур данных.
* Уровень 7. **Прикладной**. Обеспечивает интерфейс между пользователем и сетью, делает доступными для человека всевозможные услуги. На этом уровне реализуется, по крайней мере, пять прикладных служб: передача файлов, удалённый терминальный доступ, электронная передача сообщений, справочная служба и управление сетью. В конкретной реализации определяется пользователем согласно его необходимости и требованиям.

1. Физические основы передачи данных. Цифровые и аналоговые сигналы.
2. Система доменных имён DNS.

**DNS** (англ. Domain Name System — система доменных имён) — компьютерная распределённая система для получения информации о доменах. Основой DNS является представление об иерархической структуре доменного имени и зонах. Каждый сервер, отвечающий за имя, может делегировать ответственность за дальнейшую часть домена другому серверу, что позволяет возложить ответственность за актуальность информации на серверы различных организаций (людей), отвечающих только за «свою» часть доменного имени.

Особенности системы DNS:

* *Распределённость администрирования*Ответственность за разные части иерархической структуры несут разные люди или организации.
* *Распределённость хранения информации*Каждый узел сети в обязательном порядке должен хранить только те данные, которые входят в его зону ответственности и (возможно) адреса корневых DNS-серверов.
* *Кеширование информации*Узел может хранить некоторое количество данных не из своей зоны ответственности для уменьшения нагрузки на сеть.
* *Иерархическая структура*  
  Все узлы объединены в дерево, и каждый узел может или самостоятельно определять работу нижестоящих узлов, или делегировать (передавать) их другим узлам.
* *Резервирование*  
  За хранение и обслуживание своих узлов (зон) отвечают (обычно) несколько серверов, разделённые как физически, так и логически, что обеспечивает сохранность данных и продолжение работы даже в случае сбоя одного из узлов.

Первоначально в локальных сетях из небольшого числа компьютеров применялись плоские имена, состоящие из последовательности символов без разделения их на отдельные части, например MYCOMP. Для установления соответствия между символьными именами и числовыми адресами использовались широковещательные запросы. Однако для больших территориально распределенных сетей, работающих на основе протокола TCP/IP такой способ оказался неэффективным. Поэтому для установления соответствия между доменным именем и IP-адресом используется специальная система доменных имен (DNS, Domain Name System), которая основана на создаваемых администраторами сети таблиц соответствия.

Запись доменного имени начинается с самой младшей составляющей, затем после точки следует следующая по старшинству символьная часть имени и так далее. Последовательность заканчивается корневым именем, например: company.yandex.ru.

Построенная таким образом система имен позволяет разделять административную ответственность по поддержке уникальности имен в пределах своего уровня иерархии между различными людьми или организациями.

Корневой домен управляется центральными органами Интернета: IANA и Internic.

Домены верхнего уровня назначаются для каждой страны, а также для различных типов организаций. Имена этих доменов должны следовать международному стандарту ISO 3166. Для обозначения стран используются двухбуквенные аббревиатуры, например ru (Российская Федерация), us (США), it (Италия), fr (Франция).

Для различных типов организаций используются трехбуквенные аббревиатуры:

* net – сетевые организации;
* org – некоммерческие организации;
* com - коммерческие организации;
* edu – образовательные организации;
* gov – правительственные организации.

Администрирование каждого домена возлагается на отдельную организацию, которая делегирует администрирование поддоменов другим организациям.

Для получения доменного имени необходимо зарегистрироваться в соответствующей организации, которой организация InterNIC делегировала свои полномочия по распределению доменных имен.

Регистратором доменных имен в зоне ru до 2005 г. являлся Российский научно-исследовательский институт развития общественных сетей ( РосНИИРОС ). В настоящее время регистрация доменов осуществляется одним из действующих регистраторов.

DNS, использующая распределенную базу отображений "доменное имя – IP-адрес". Сервер домена хранит только имена, которые заканчиваются на следующем ниже по дереву уровне. Это позволяет распределять более равномерно нагрузку по разрешению имен между всеми DNS-серверами. Каждый DNS-сервер помимо таблицы отображения имен содержит ссылки на DNS-серверы своих поддоменов.

Существуют две схемы разрешения DNS-имен.

Нерекурсивная процедура:

1.DNS-клиент обращается к корневому DNS-серверу с указанием полного доменного имени;

2.DNS-сервер отвечает клиенту, указывая адрес следующего DNS-сервера, обслуживающего домен верхнего уровня, заданный в следующей старшей части имени;

3.DNS-клиент делает запрос следующего DNS-сервера, который отсылает его к DNS-серверу нужного поддомена и т.д., пока не будет найден DNS-сервер, в котором хранится соответствие запрошенного имени IP-адресу. Сервер дает окончательный ответ клиенту.

Рекурсивная процедура:

1.DNS-клиент запрашивает локальный DNS-сервер, обслуживающий поддомен, которому принадлежит клиент;

2.Далее

3.Если локальный DNS-сервер знает ответ, он возвращает его клиенту

4.Если локальный сервер не знает ответ, то он выполняет итеративные запросы к корневому серверу. После получения ответа сервер передает его клиенту.

Таким образом, при рекурсивной процедуре клиент фактически перепоручает работу своему серверу. Для ускорения поиска IP-адресов DNS-серверы широко применяют кэширование (на время от часов до нескольких дней) проходящих через них ответов.

Ключевыми понятиями DNS являются:

* **Домен** (англ. ***domain*** — область) — узел в дереве имён, вместе со всеми подчинёнными ему узлами (если таковые имеются), то есть именованная ветвь или поддерево в дереве имен. Структура доменного имени отражает порядок следования узлов в иерархии; доменное имя читается слева направо от младших доменов к доменам высшего уровня (в порядке повышения значимости), корневым доменом всей системы является точка (‘.’), ниже идут домены первого уровня (географические или тематические), затем — домены второго уровня, третьего и т. д. (например, для адреса ru.wikipedia.org домен первого уровня — org, второго wikipedia, третьего ru). На практике точку в конце имени часто опускают, но она бывает важна в случаях разделения между относительными доменами и FQDN (англ. Fully Qualifed Domain Name, полностью определённое имя домена).
* **Поддомен** (англ. ***subdomain***) — подчиненный домен. (например, wikipedia.org — поддомен домена org, а ru.wikipedia.org — домена wikipedia.org). Теоретически такое деление может достигать глубины 127 уровней, а каждая метка может содержать до 63 символов, пока общая длина вместе с точками не достигнет 254 символов. Но на практике регистраторы доменных имён используют более строгие ограничения. Например, если у вас есть домен вида mydomain.ru, вы можете создать для него различные поддомены вида mysite1.mydomain.ru, mysite2.mydomain.ru и т. д.
* **Ресурсная запись** — единица хранения и передачи информации в DNS. Каждая ресурсная запись имеет имя (то есть привязана к определенному Доменному имени, узлу в дереве имен), тип и поле данных, формат и содержание которого зависит от типа.
* **Зона** — часть дерева доменных имен (включая ресурсные записи), размещаемая как единое целое на некотором сервере доменных имен (DNS-сервере, см. ниже), а чаще — одновременно на нескольких серверах (см. ниже). Целью выделения части дерева в отдельную зону является передача ответственности (см. ниже) за соответствующий Домен другому лицу или организации, так называемое Делегирование (см. ниже). Как связная часть дерева, зона внутри тоже представляет собой дерево. Если рассматривать пространство имен DNS как структуру из зон, а не отдельных узлов/имен, тоже получается дерево; оправданно говорить о родительских и дочерних зонах, о старших и подчиненных. На практике, большинство зон 0-го и 1-го уровня (‘.’, ru, com, …) состоят из единственного узла, которому непосредственно подчиняются дочерние зоны. В больших корпоративных доменах (2-го и более уровней) иногда встречается образование дополнительных подчиненных уровней без выделения их в дочерние зоны.
* **Делегирование** — операция передачи ответственности за часть дерева доменных имен другому лицу или организации. За счет делегирования в DNS обеспечивается распределенность администрирования и хранения. Технически делегирование выражается в выделении этой части дерева в отдельную зону, и размещении этой зоны на DNS-сервере (см. ниже), управляемом этим лицом или организацией. При этом в родительскую зону включаются «склеивающие» ресурсные записи (NS и А), содержащие указатели на DNS-сервера дочерней зоны, а вся остальная информация, относящаяся к дочерней зоне, хранится уже на DNS-серверах дочерней зоны.
* **DNS-сервер** — специализированное ПО для обслуживания DNS, а также компьютер, на котором это ПО выполняется. DNS-сервер может быть ответственным за некоторые зоны и/или может перенаправлять запросы вышестоящим серверам.
* **DNS-клиент** — специализированная библиотека (или программа) для работы с DNS. В ряде случаев DNS-сервер выступает в роли DNS-клиента.
* **Авторитетность** (англ. ***authoritative***) — признак размещения зоны на DNS-сервере. Ответы DNS-сервера могут быть двух типов: авторитетные (когда сервер заявляет, что сам отвечает за зону) и неавторитетные (англ. Non-authoritative), когда сервер обрабатывает запрос, и возвращает ответ других серверов. В некоторых случаях вместо передачи запроса дальше DNS-сервер может вернуть уже известное ему (по запросам ранее) значение (режим кеширования).
* **DNS-запрос** (англ. ***DNS query***) — запрос от клиента (или сервера) серверу. Запрос может быть рекурсивным или нерекурсивным (см. Рекурсия).

Система DNS содержит иерархию DNS-серверов, соответствующую иерархии зон. Каждая зона поддерживается как минимум одним авторитетным сервером, на котором расположена информация о домене.  
Имя и IP-адрес не тождественны — один IP-адрес может иметь множество имён, что позволяет поддерживать на одном компьютере множество веб-сайтов (это называется виртуальный хостинг). Обратное тоже справедливо — одному имени может быть сопоставлено множество IP-адресов: это позволяет создавать балансировку нагрузки.  
Для повышения устойчивости системы используется множество серверов, содержащих идентичную информацию, а в протоколе есть средства, позволяющие поддерживать синхронность информации, расположенной на разных серверах. Существует 13 корневых серверов, их адреса практически не изменяются.

Протокол DNS использует для работы TCP- или UDP-порт 53 для ответов на запросы. Традиционно запросы и ответы отправляются в виде одной U.

**Записи DNS**, или **Ресурсные записи** (англ. ***Resource Records, RR***) — единицы хранения и передачи информации в DNS. Каждая ресурсная запись состоит из следующих полей:

* имя (NAME) — доменное имя, к которому привязана или которому «принадлежит» данная ресурсная запись,
* TTL (Time To Live) — допустимое время хранения данной ресурсной записи в кэше неответственного DNS-сервера,
* тип (TYPE) ресурсной записи — определяет формат и назначение данной ресурсной записи,
* класс (CLASS) ресурсной записи; теоретически считается, что DNS может использоваться не только с TCP/IP, но и с другими типами сетей, код в поле класс определяет тип сети,
* длина поля данных (RDLEN),
* поле данных (RDATA), формат и содержание которого зависит от типа записи.

Наиболее важные типы DNS-записей:

* **Запись A** (address record) или запись адреса связывает имя хоста с адресом IP. Например, запрос A-записи на имя referrals.icann.org вернет его IP адрес — 192.0.34.164
* **Запись AAAA** (IPv6 address record) связывает имя хоста с адресом протокола IPv6. Например, запрос AAAA-записи на имя K.ROOT-SERVERS.NET вернет его IPv6 адрес — 2001:7fd::1
* **Запись CNAME** (canonical name record) или каноническая запись имени (псевдоним) используется для перенаправления на другое имя
* **Запись MX** (mail exchange) или почтовый обменник указывает сервер(ы) обмена почтой для данного домена.
* **Запись NS** (name server) указывает на DNS-сервер для данного домена.
* **Запись PTR** (pointer) или запись указателя связывает IP хоста с его каноническим именем. Запрос в домене in-addr.arpa на IP хоста в reverse форме вернёт имя (FQDN) данного хоста (см. Обратный DNS-запрос). Например, (на момент написания), для IP адреса 192.0.34.164: запрос записи PTR 164.34.0.192.in-addr.arpa вернет его каноническое имя referrals.icann.org. В целях уменьшения объёма нежелательной корреспонденции (спама) многие серверы-получатели электронной почты могут проверять наличие PTR записи для хоста, с которого происходит отправка. В этом случае PTR запись для IP адреса должна соответствовать имени отправляющего почтового сервера, которым он представляется в процессе SMTP сессии.
* **Запись SOA** (Start of Authority) или начальная запись зоны указывает, на каком сервере хранится эталонная информация о данном домене, содержит контактную информацию лица, ответственного за данную зону, тайминги (параметры времени) кеширования зонной информации и взаимодействия DNS-серверов.
* **Запись SRV** (server selection) указывает на серверы для сервисов, используется, в частности, для Jabber и Active Directory.

**BIND** (***Berkeley Internet Name Domain***, до этого: Berkeley Internet Name Daemon) — открытая и наиболее распространённая реализация DNS-сервера, обеспечивающая выполнение преобразования DNS-имени в IP-адрес и наоборот.  
BIND поддерживается организацией Internet Systems Consortium. BIND был создан студентами и впервые был выпущен в BSD 4.3.  
В Unix этот сервер является стандартом де-факто.

## Типы DNS-серверов

По выполняемым функциям DNS-серверы делятся на несколько групп, в зависимости от конфигурации конкретный сервер может относиться к нескольким типам;

* **авторитативный DNS-сервер** — сервер, отвечающий за какую-либо зону. – **Мастер** или **первичный сервер** (в терминологии BIND) — сервер, имеющий право на внесение изменений в данные зоны. Обычно для зоны бывает только один мастер сервер. В случае Microsoft DNS-сервера и его интеграции с Active Directory мастер-серверов может быть несколько (так как репликация изменений осуществляется не средствами DNS-сервера, а средствами Active Directory, за счёт чего обеспечивается равноправность серверов и актуальность данных). – **Слейв** или **вторичный сервер**, не имеющий права на внесение изменений в данные зоны и получающий сообщения об изменениях от мастер-сервера. В отличие от мастер-сервера их может быть (практически) неограниченное количество. Слейв так же является авторитативным сервером (и пользователь не может различить мастер и слейв, разница появляется только на этапе конфигурирования/внесения изменений в настройки зоны).
* **Кэширующий DNS-сервер** — сервер, который обслуживает запросы клиентов, (получает рекурсивный запрос, выполняет его с помощью нерекурсивных запросов к авторитативным серверам или передаёт рекурсивный запрос вышестоящему DNS-серверу)
* **Локальный DNS-сервер**; используется для обслуживания DNS-клиентов, исполняющихся на локальной машине. Фактически, это разновидность кэширующего DNS-сервера, сконфигурированная для обслуживания локальных приложений.
* **Перенаправляющий DNS-сервер**; (англ. forwarder, внутренний DNS-сервер) сервер, перенаправляющий полученные рекурсивные запросы вышестоящему кэширующему серверу в виде рекурсивных запросов. Используется преимущественно для снижения нагрузки на кэширующий DNS-сервер.
* **Корневой DNS-сервер** — сервер, являющийся авторитативным за корневую зону. Общеупотребительных корневых серверов в мире всего 13 штук, их доменные имена находятся в зоне root-servers.net и называются a.root-servers.net, b.root-servers.net, … , m.root-servers.net. В определённых конфигурациях локальной сети возможна ситуация настройки локальных корневых серверов.
* **Регистрирующий DNS-сервер**. Сервер, принимающий динамические обновления от пользователей. Часто совмещается с DHCP-сервером. В Microsoft DNS-сервере при работе на контроллере домена сервер работает в режиме регистрирующего DNS-сервера, принимая от компьютеров домена информацию о соответствии имени и IP компьютера и обновляя в соответствии с ней данные зоны домена.

**Прямой запрос**

Прямой (forward) запрос — запрос на преобразование имени (символьного адреса) хоста в IP-адрес.

**Обратный запрос**

Обратный (reverse) запрос — запрос на преобразование IP-адреса в имя хоста.

1. Понятие протокола, стека протоколов.

Протокол передачи данных — набор соглашений интерфейса логического уровня, которые определяют обмен данными между различными программами. Эти соглашения задают единообразный способ передачи сообщений и обработки ошибок при взаимодействии программного обеспечения разнесённой в пространстве аппаратуры, соединённой тем или иным интерфейсом.

Любая связь между устройствами возможна лишь благодаря протоколам. Они делятся на физические протоколы (регулируют то, как именно и какие сигналы будут идти от одного устройства к другому — например, импульсами по 5 вольт 100 раз в секунду или на определённой частоте радиосигналов) и логические протоколы, которые отвечают за смысл и передачу данных, когда связь уже установлена. Так, браузер на компьютере связывается с сервером по протоколу HTTP или HTTPS.

Сетево́й протоко́л — набор правил и действий (очерёдности действий), позволяющий осуществлять соединение и обмен данными между двумя и более включёнными в сеть устройствами.

Стек протоколов — это иерархически организованный набор сетевых протоколов, достаточный для организации взаимодействия узлов в сети. Протоколы работают в сети одновременно, значит работа протоколов должна быть организована так, чтобы не возникало конфликтов или незавершённых операций. Поэтому стек протоколов разбивается на иерархически построенные уровни, каждый из которых выполняет конкретную задачу — подготовку, приём, передачу данных и последующие действия с ними.

Количество уровней в стеке меняется в соответствии с конкретным стеком протоколов. Протоколы нижних уровней часто реализуются комбинацией программных и аппаратных средств, а протоколы верхних уровней, как правило, программными средствами.

Существует достаточное количество стеков протоколов, широко применяемых в сетях. Наиболее популярные стеки протоколов: OSI международной организации по стандартизации, TCP/IP, используемый в сети Internet и во многих сетях на основе операционной системы UNIX, IPX/SPX фирмы Novell, NetBIOS/SMB, разработанный фирмами Microsoft и IBM, DECnet корпорации Digital Equipment, SNA фирмы IBM и некоторые другие.

Сетевые протоколы предписывают правила работы компьютерам, которые подключены к сети. Они строятся по многоуровневому принципу. Протокол некоторого уровня определяет одно из технических правил связи. В настоящее время для сетевых протоколов используется модель [OSI](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%B0%D1%8F_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_OSI" \o "Сетевая модель OSI) (Open System Interconnection — взаимодействие открытых систем, ВОС).

Модель OSI — 7-уровневая логическая модель работы сети. Реализуется группой протоколов и правил связи, организованных в несколько уровней:

* на физическом уровне определяются физические (механические, электрические, оптические) характеристики линий связи;
* на канальном уровне определяются правила использования физического уровня узлами сети;
* сетевой уровень отвечает за адресацию и доставку сообщений;
* транспортный уровень контролирует очерёдность прохождения компонентов сообщения;
* сеансовый уровень координирует связь между двумя прикладными программами, работающими на разных рабочих станциях;
* уровень представления служит для преобразования данных из внутреннего формата компьютера в формат передачи;
* прикладной уровень является пограничным между прикладной программой и другими уровнями, обеспечивая удобный интерфейс связи для сетевых программ пользователя.

Другая модель — [стек протоколов TCP/IP](https://ru.wikipedia.org/wiki/TCP/IP) — содержит 4 уровня:

* канальный уровень (link layer),
* сетевой уровень (Internet layer),
* транспортный уровень (transport layer),
* прикладной уровень (application layer).

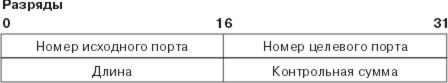
1. Протокол UDP.

Протокол **UDP** (User Datagram Protocol, RFC-768) является одним из основных протоколов, расположенных непосредственно над IP. Он предоставляет прикладным процессам транспортные услуги, немногим отличающиеся от услуг протокола IP. Протокол UDP обеспечивает доставку дейтограмм, но не требует подтверждения их получения. Протокол UDP не требует соединения с удаленным модулем UDP ("бессвязный" протокол). К заголовку IP-пакета UDP добавляет поля *порт отправителя* и *порт получателя*, которые обеспечивают мультиплексирование информации между различными прикладными процессами, а также поля *длина* UDP-дейтограммы и *контрольная сумма*, позволяющие поддерживать целостность данных. Таким образом, если на уровне IP для определения места доставки пакета используется адрес, на уровне UDP - номер порта.

Хотя протокол UDP не гарантирует доставки, по умолчанию предполагается, что вероятность потери пакета достаточно мала.

Иногда возникает необходимость отправить сообщение от одного приложения другому приложению или процессу, выполняемому на другом компьютере, подключенном к сети. **UDP** обеспечивает передачу дейтаграмм между приложениями хостов Internet.

Для отправки дейтаграмм протокол **UDP** применяет протокол **IP**, поэтому **UDP** так же не устанавливает соединения, как и **IP**. Он не гарантирует доставку дейтаграммы и не обеспечивает защиту от дублирования данных. Однако **UDP** позволяет отправителю задать для сообщения исходный и целевой порты и обеспечивает проверку целостности данных и заголовка сообщения с помощью контрольной суммы. Это позволяет отправителю и получателю проверить правильность доставки сообщения.



*Заголовок пакета в протоколе UDP*

На рисунке показаны первые 32 бита стандартного заголовка пакета UDP. Первые 16 бит содержат номер исходного порта и длину. Вторые 16 бит содержат номер целевого порта и контрольную сумму.

Длина сообщения равна числу байт в UDP-дейтограмме, включая заголовок. Поле UDP контрольная сумма содержит код, полученный в результате контрольного суммирования UDP-заголовка и поля данные. Не трудно видеть, что этот протокол использует заголовок минимального размера (8 байт). Таблица номеров UDP-портов приведена ниже (4.4.2.1). Номера портов от 0 до 255 стандартизованы и использовать их в прикладных задачах не рекомендуется. Но и в интервале 255-1023 многие номера портов заняты, поэтому прежде чем использовать какой-то порт в своей программе, следует заглянуть в RFC-1700. Во второй колонке содержится стандартное имя, принятое в Internet, а в третей - записаны имена, принятые в UNIX.

Для надежной доставки дейтаграмм с помощью UDP в приложении должны быть предусмотрены процедуры проверки. Для надежной доставки потоков данных предназначен протокол TCP.

UDP решает проблему задержек в передаче данных, позволяя информации перемещаться молниеносно, даже если это означает возможность потери некоторых данных по пути. Это ключевой момент для приложений, где каждая секунда на счету, например, в онлайн-играх или видеоконференциях.

**Схема вычисления контрольных сумм**

Модуль IP передает поступающий IP-пакет модулю UDP, если в заголовке этого пакета указан код протокола UDP. Когда модуль UDP получает дейтограмму от модуля IP, он проверяет контрольную сумму, содержащуюся в ее заголовке. Если контрольная сумма равна нулю, это означает, что отправитель ее не подсчитал. ICMP, IGMP, UDP и TCP протоколы имеют один и тот же алгоритм вычисления контрольной суммы (RFC-1071). Но вычисление контрольной суммы для UDP имеет некоторые особенности. Во-первых, длина UDP-дейтограммы может содержать нечетное число байт, в этом случае к ней добавляется нулевой байт, который служит лишь для унификации алгоритма и никуда не пересылается. Во-вторых, при расчете контрольной суммы для UDP и TCP добавляются 12-байтные псевдо-заголовки, содержащие IP-адреса отправителя и получателя, код протокола и длину дейтограммы (см. рис. 4.4.2.2). Как и в случае IP-дейтограммы, если вычисленная контрольная сумма равна нулю, в соответствующее поле будет записан код 65535.

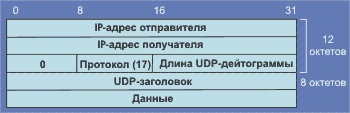


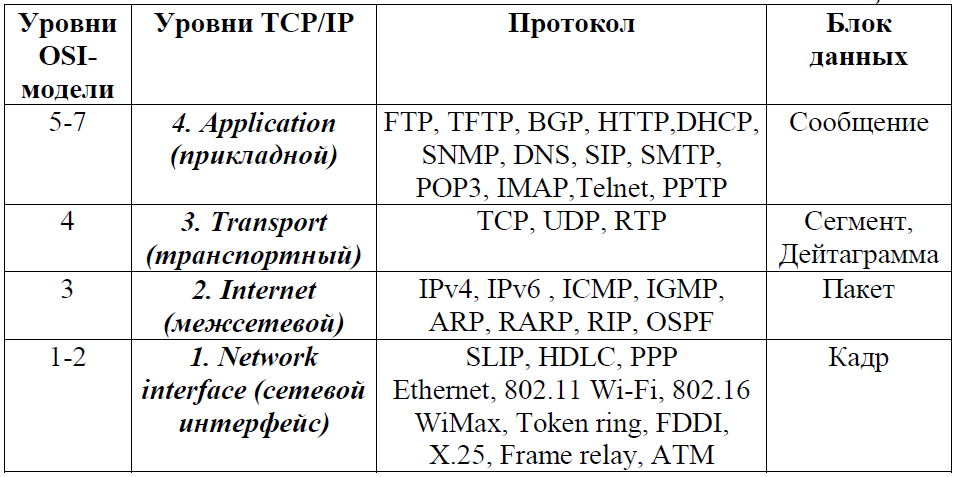
Рис. 4.4.2.2. Псевдозаголовок, используемый при расчете контрольной суммы

Если контрольная сумма правильная (или равна 0), то проверяется порт назначения, указанный в заголовке дейтограммы. Если прикладной процесс подключен к этому порту, то прикладное сообщение, содержащиеся в дейтограмме, становится в очередь к прикладному процессу для прочтения. В остальных случаях дейтограмма отбрасывается. Если дейтограммы поступают быстрее, чем их успевает обрабатывать прикладной процесс, то при переполнении очереди сообщений поступающие дейтограммы отбрасываются модулем UDP. Следует учитывать, что во многих посылках контрольное суммирование не охватывает адреса отправителя и места назначения. При некоторых схемах маршрутизации это приводит к зацикливанию пакетов в случае повреждения его адресной части (адресат не признает его "своим").

Может возникнуть вопрос, зачем вычислять и проверять контрольную сумму, если подтверждение доставки и повторная пересылка в рамках протокола не предусмотрены. Дело в том, что UDP используется не только для мультимедийных задач но и некоторыми другими протоколами (DNS, SNMP и др.), где повторные запросы и отклики могут выполняться на прикладном уровне.

Так как максимальная длина IP-дейтограммы равна 65535 байтам, максимальная протяженность информационного поля UDP-дейтограммы составляет 65507 байт. На практике большинство систем работает с UDP-дейтограммами с длиной 8192 байта или менее (Ethernet допускает 1508 байт). Детальное описание форматов, полей пакетов и пр. читатель может найти в RFC-768. Смотри также RFC-2147 (IPv6 Jumbo), RFC-2508 (компрессия заголовков) и RFC-3828 (Lightweight UDP).

1. Стек сетевых протоколов TCP/IP.



Стек протоколов TCP/IP (Transmission Control Protocol/Internet Protocol, протокол управления передачей/протокол интернета) — сетевая модель, описывающая процесс передачи цифровых данных. Она названа по двум главным протоколам, по этой модели построена глобальная сеть интернет.

**Канальный уровень (link layer)**

Предназначение канального уровня — дать описание тому, как происходит обмен информацией на уровне сетевых устройств, определить, как информация будет передаваться от одного устройства к другому. Информация здесь кодируется, делится на пакеты и отправляется по нужному каналу, т.е. среде передачи.

Этот уровень также вычисляет максимальное расстояние, на которое пакеты возможно передать, частоту сигнала, задержку ответа и т.д. Все это — физические свойства среды передачи информации. На канальном уровне самым распространенным протоколом является Ethernet, который мы рассмотрим в конце статьи.

**Межсетевой уровень (internet layer)**

Глобальная сеть интернет состоит из множества локальных сетей, взаимодействующих между собой. Межсетевой уровень используется, чтобы описать обеспечение такого взаимодействия.

Межсетевое взаимодействие — это основной принцип построения интернета. Локальные сети по всему миру объединены в глобальную, а передачу данных между этими сетями осуществляют магистральные и пограничные маршрутизаторы.

Именно на межсетевом уровне функционирует протокол IP, позволивший объединить разные сети в глобальную. Как и протокол TCP, он дал название модели, рассматриваемой в статье.

**Транспортный уровень (transport layer)**

Постоянные резиденты транспортного уровня — протоколы TCP и UDP, они занимаются доставкой информации.

**TCP (протокол управления передачей)** — надежный, он обеспечивает передачу информации, проверяя дошла ли она, насколько полным является объем полученной информации и т.д. TCP дает возможность двум конечным устройствам производить обмен пакетами через предварительно установленное соединение. Он предоставляет услугу для приложений, повторно запрашивает потерянную информацию, устраняет дублирующие пакеты, регулируя загруженность сети. TCP гарантирует получение и сборку информации у адресата в правильном порядке.

**UDP (протокол пользовательских датаграмм)** — ненадежный, он занимается передачей автономных датаграмм. UDP не гарантирует, что всех датаграммы дойдут до получателя. Датаграммы уже содержат всю необходимую информацию, чтобы дойти до получателя, но они все равно могут быть потеряны или доставлены в порядке отличном от порядка при отправлении.

UDP обычно не используется, если требуется надежная передача информации. Использовать UDP имеет смысл там, где потеря части информации не будет критичной для приложения, например, в видеоиграх или потоковой передаче видео. UDP необходим, когда делать повторный запрос сложно или неоправданно по каким-то причинам.

Протоколы транспортного уровня не интерпретируют информацию, полученную с верхнего или нижних уровней, они служат только как канал передачи, но есть исключения. RSVP (Resource Reservation Protocol, протокол резервирования сетевых ресурсов) может использоваться, например, роутерами или сетевыми экранами в целях анализа трафика и принятия решений о его передаче или отклонении в зависимости от содержимого.

**Прикладной уровень (application layer)**

В модели TCP/IP отсутствуют дополнительные промежуточные уровни (представления и сеансовый) в отличие от OSI. Функции форматирования и представления данных делегированы библиотекам и программным интерфейсам приложений (API) — своего рода базам знаний, содержащим сведения о том, как приложения взаимодействуют между собой. Когда службы или приложения обращаются к библиотеке или API, те в ответ предоставляют набор действий, необходимых для выполнения задачи и полную инструкцию, каким образом эти действия нужно выполнять.

Протоколы прикладного уровня действуют для большинства приложений, они предоставляют услуги пользователю или обмениваются данными с «коллегами» с нижних уровней по уже установленным соединениям. Здесь для большинства приложений созданы свои протоколы. Например, браузеры используют HTTP для передачи гипертекста по сети, почтовые клиенты — SMTP для передачи почты, FTP-клиенты — протокол FTP для передачи файлов, службы DHCP — протокол назначения IP-адресов DHCP и так далее.

1. Сетевые устройства: повторитель, мост, коммутатор, маршрутизатор, концентратор, шлюз.

**Коммутатор в компьютерных сетях** — применяется для соединения нескольких узлов компьютерной сети. Компьютерная сеть (вычислительная сеть, сеть передачи данных) — система связи компьютеров и/или компьютерного оборудования (серверы, маршрутизаторы и другое оборудование). Для передачи информации могут быть использованы различные физические явления, как правило — различные виды электрических сигналов, световых сигналов или электромагнитного излучения.

**PAN (**Personal Area Network) — персональная сеть, предназначенная для взаимодействия различных устройств, принадлежащих одному владельцу.

**LAN** (Local Area Network) — локальные сети, имеющие замкнутую инфраструктуру до выхода на поставщиков услуг. Термин «LAN» может описывать и маленькую офисную сеть, и сеть уровня большого завода, занимающего несколько сотен гектаров. Зарубежные источники дают даже близкую оценку — около шести миль (10 км) в радиусе. Локальные сети являются сетями закрытого типа, доступ к ним разрешен только ограниченному кругу пользователей, для которых работа в такой сети непосредственно связана с их профессиональной деятельностью.

**CAN** (Campus Area Network — кампусная сеть) — объединяет локальные сети близко расположенных зданий.

**MAN** (Metropolitan Area Network) — городские сети между учреждениями в пределах одного или нескольких городов, связывающие много локальных вычислительных сетей.

**WAN** (Wide Area Network) — глобальная сеть, покрывающая большие географические регионы, включающие в себя как локальные сети, так и прочие телекоммуникационные сети и устройства. Пример WAN — сети с коммутацией пакетов (Frame relay), через которую могут «разговаривать» между собой различные компьютерные сети. Глобальные сети являются открытыми и ориентированы на обслуживание любых пользователей.

**По типу функционального взаимодействия**Клиент-сервер, Смешанная сеть, Одноранговая сеть, Многоранговые сети

**По типу сетевой топологии**Шина, Кольцо, Двойное кольцо, Звезда, Решётка, Дерево

**По типу среды передачи**Проводные (телефонный провод, коаксиальный кабель, витая пара, волоконно-оптический кабель)

Беспроводные (передачей информации по радиоволнам в определенном частотном диапазоне)

**По функциональному назначению**Сети хранения данных, Серверные фермы, Сети управления процессом, Сети SOHO, домовые сети,

**По скорости передач**низкоскоростные (до 10 Мбит/с),среднескоростные (до 100 Мбит/с),высокоскоростные (свыше 100 Мбит/с);

**Сетевой концентратор или хаб** (жарг. от англ. hub — центр деятельности) — сетевое устройство, предназначенное для объединения нескольких устройств Ethernet в общий сегмент сети. Устройства подключаются при помощи витой пары, коаксиального кабеля или оптоволокна. Термин концентратор (хаб) применим также к другим технологиям передачи данных: USB, FireWire и пр.

**Принцип работы**Концентратор работает на физическом уровне сетевой модели OSI, повторяет приходящий на один порт сигнал на все активные порты. В случае поступления сигнала на два и более порта одновременно возникает коллизия, и передаваемые кадры данных теряются. Таким образом, все подключённые к концентратору устройства находятся в одном домене коллизий. Концентраторы всегда работают в режиме полудуплекса, все подключённые устройства Ethernet разделяют между собой предоставляемую полосу доступа.

**Упрощённое описание принципа работы**Хаб работает по следующему принципу: копирует все полученные пакеты во все порты. При этом может возникнуть проблема, при которой по двум и более портам приходят пакеты в одно и то же время. Другая проблема — безопасность — все пакеты доходят до всех компьютеров сети, поэтому существует возможность несанкционированного доступа к информации. И, наконец, ещё одной проблемой является то, что копирование пакетов повышает нагрузку на сеть, причём весьма существенно — весь трафик сегмента сети поступает к каждому из компьютеров и тем самым загружает сеть.

**Повторитель**(репи́тер, от англ. repeater) — сетевое оборудование, Предназначен для увеличения расстояния сетевого соединения путём повторения электрического сигнала «один в один». Бывают однопортовые повторители и многопортовые. В терминах модели OSI работает на физическом уровне.

**Сетевой шлюз** (англ. gateway) — аппаратный маршрутизатор или программное обеспечение для сопряжения компьютерных сетей, использующих разные протоколы (например, локальной и глобальной). Сетевой шлюз конвертирует протоколы одного типа физической среды в протоколы другой физической среды (сети). Например, при соединении локального компьютера с сетью Интернет вы используете сетевой шлюз.

**Роутеры (маршрутизаторы**) являются одним из примеров аппаратных сетевых шлюзов.Сетевые шлюзы работают на всех известных операционных системах. Основная задача сетевого шлюза — конвертировать протокол между сетями. Роутер сам по себе принимает, проводит и отправляет пакеты только среди сетей, использующих одинаковые протоколы. Сетевой шлюз может с одной стороны принять пакет, сформатированный под один протокол (например Apple Talk) и конвертировать в пакет другого протокола (например TCP/IP) перед отправкой в другой сегмент сети. Сетевые шлюзы могут быть аппаратным решением, программным обеспечением или тем и другим вместе, но обычно это программное обеспечение, установленное на роутер или компьютер. Сетевой шлюз должен понимать все протоколы, используемые роутером. Обычно сетевые шлюзы работают медленнее, чем сетевые мосты и коммутаторы**.**

**Сетевой шлюз** — это точка сети, которая служит выходом в другую сеть. В сети Интернет узлом или конечной точкой может быть или сетевой шлюз, или хост Сетевой шлюз может быть специальным аппаратным роутером или программным обеспечением, установленным на обычный сервер или персональный компьютер. Большинство компьютерных операционных систем использует термины, описанные выше.

**Маршрутиза́тор**или ро́утер — специализированный сетевой компьютер, имеющий минимум два сетевых интерфейса и пересылающий пакеты данных между различными сегментами сети, принимающий решения о пересылке на основании информации о топологии сети и определённых правил, заданных администратором. Маршрутизаторы делятся на программные и аппаратные. Маршрутизатор работает на более высоком «сетевом» уровне 3 сетевой модели OSI, нежели коммутатор и сетевой мост.Обычно маршрутизатор использует адрес получателя, указанный в пакетах данных, и определяет по таблице маршрутизации путь, по которому следует передать данные. Если в таблице маршрутизации для адреса нет описанного маршрута, пакет отбрасывается.Существуют и другие способы определения маршрута пересылки пакетов, когда, например, используется адрес отправителя, используемые протоколы верхних уровней и другая информация, содержащаяся в заголовках пакетов сетевого уровня. Нередко маршрутизаторы могут осуществлять трансляцию адресов отправителя и получателя, фильтрацию транзитного потока данных на основе определённых правил с целью ограничения доступа, шифрование/дешифрование передаваемых данных и т. д.

**Мост, сетевой мост, бридж**  — сетевое устройство 2 уровня модели OSI, предназначенное для объединения сегментов (подсети) компьютерной сети разных топологий и архитектур.

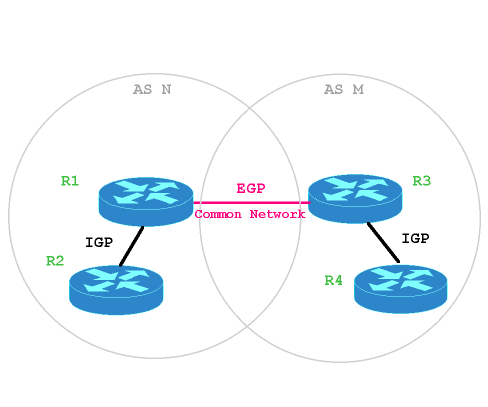
**Различия между коммутаторами и мостами**В общем случае коммутатор (свитч) и мост аналогичны по функциональности; разница заключается во внутреннем устройстве: мосты обрабатывают трафик, используя центральный процессор, коммутатор же использует коммутационную матрицу (аппаратную схему для коммутации пакетов). В настоящее время мосты практически не используются (так как для работы требуют производительный процессор), за исключением ситуаций, когда связываются сегменты сети с разной организацией первого уровня, например, между xDSL соединениями, оптикой, Ethernet’ом. В случае SOHO-оборудования, режим прозрачной коммутации часто называют «мостовым режимом» (bridging).

**Функциональные возможности**Мост обеспечивает: ограничение домена коллизий, задержку фреймов, адресованных узлу в сегменте отправителя, ограничение перехода из домена в домен ошибочных фреймов: ,карликов (фреймов меньшей длины, чем допускается по стандарту (64 байта)), фреймов с ошибками в CRC, фреймов с признаком «коллизия» ,затянувшихся фреймов (размером больше, чем разрешено стандартом)Мосты «изучают» характер расположения сегментов сети путем построения адресных таблиц вида «Интерфейс:MAC-адрес», в которых содержатся адреса всех сетевых устройств и сегментов, необходимых для получения доступа к данному устройству.

1. Внешняя и внутренняя маршрутизация.

Протоколы внешней маршрутизации

Протоколы этого типа используются для определения маршрутов передачи данных между различными автономными системами. Такие протоколы обычно относят к классу Exterior Gateway Protocol. В настоящее время существуют два протокола данного типа:



Border Gateway Protocol

Exterior Gateway Protocol

Особенности внешней маршрутизации

Два маршрутизатора, которые обмениваются информацией о маршрутах, называются внутренними соседями в том случае, если они принадлежат к одной автономной системе и внешними – в том случае, если они принадлежат к различным автономным системам. На рисунке маршрутизаторы R2 R4 являются внутренними для автономных систем AS N и AS M соответственно. R1 и R3 совмещают функции внешнего и внутреннего маршрутизаторов. Маршрутизатор R1 представляет для автономной системы AS M маршруты к сетям, которые находятся в автономной системе AS N. Аналогичную функцию выполняет маршрутизатор R3 по отношению к маршрутам AS M.

Основная особенность протоколов внешней маршрутизации заключается в том, что они представляют метрики маршрутов, которые рассчитываются относительно некоторой общей сети, а не относительно своих интерфейсов.

Протоколы внутренней маршрутизации, используемые для

маршрутизации внутри автономной системы. Данный тип маршрутизации также

называют внутренней маршрутизацией автономной системы. Компании, организации

и даже операторы связи используют протоколы внутренней маршрутизации в своих

внутренних сетях. К протоколам внутренней маршрутизации относятся протоколы

RIP, EIGRP, OSPF и IS-IS.Существует четыре дистанционно-векторных протокола внутренней маршрутизации IPv4: • RIPv1 — устаревшая версия протокола первого поколения; • RIPv2 — простой дистанционно-векторный протокол; • IGRP — запатентованный протокол Cisco первого поколения (на сегодняшний день также устаревший, заменен протоколом EIGRP); • EIGRP — расширенная версия дистанционно-векторного протокола.

Протоколы внешнего шлюза (EGP), используемые для маршрутизации

между автономными системами. Маршрутизацию данного типа также называют

внешней маршрутизацией автономной системы. Взаимодействие между сетями

операторов связи и крупных компаний может осуществляться посредством протокола

внешней маршрутизации. Протокол пограничного шлюза (BGP) является

единственным в настоящее время официальным протоколом маршрутизации EGP,

используемым в Интернете

### **Теория языков программирования и методы трансляции**

1. **Формы Бэкуса-Наура.**

Для определения языка требуется задать множество основных символов языка и описать его синтаксис (грамматику) и семантику. *Синтаксис* языка определяет правила составления корректных цепочек, состоящих из основных символов языка. *Семантика* задает смысловые значения конструкций языка, а также интерпретацию различных синтаксических конструкций [человеком].

Разработка ЯП начинается с определения его синтаксиса. Язык, предназначенный для описания другого языка, называется **метаязыком**. Метаязык задает систему обозначений, понятий языка и образованных из них конструкций, позволяющих представить описываемый язык с помощью определенных ранее понятий и отношений между ними. При этом каждое понятие языка подразумевает некоторую синтаксическую единицу (конструкцию) и определяемые ею свойства программных объектов или процесса обработки данных. Метаязыки используются для задания грамматики ЯП. При описании метаязыка используют два важных понятия.

**Терминал (терминальный символ)** – объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»). В ЯП в качестве терминалов обычно берут все или часть стандартных символов ASCII – латинские буквы, цифры и специальные символы.

**Нетерминал (нетерминальный символ)** – объект, обозначающий какую-либо *сущность* языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения. Это можно трактовать как название структурной единицы.

Обычно терминалы обозначают маленькими буквами, а нетерминалы – большими.

Одним из первых появился *метаязык Хомского,* который вышел из математической логики и имеет такую систему обозначений:

* *символ* ***→*** отделяет левую часть правила от правой; обозначает *порождает* или *это есть*;
* ***нетерминалы*** обозначаются буквой *А* с индексом, указывающим на его номер;
* ***терминалы*** – это символы, используемые в описываемом языке;
* каждое правило определяет порождение одной новой цепочки;
* один и тот же нетерминал может встречаться в нескольких правилах слева.

В качестве примера предлагается рассмотреть описание идентификатора условного ЯП на этом метаязыке. Оно состоит из 65 правил:

*1. A1 → A* *2. A1 → B …* *26. A1 → Z*

*27. A1 → a* *28. A1 → b …* *52. A1 → z*

*53. A2 → 0* *54. A2 → 1 …* *62. A2 → 9*

*63. A3 → A1* *64. A3 → A3A1* *65. A3 → A3A2*

Данное описание идентификатора показывает громоздкость метаязыка, поэтому его можно эффективно использовать только для описания небольших абстрактных языков. Для более компактного описания позже был предложен

*метаязык Хомского-Шутценбергера*. Но наибольшее распространение для описания синтаксиса ЯП получил язык металингвистических формул Бэкуса и его модификации.

Форма Бэкуса-Наура (БНФ)впервые использовалась для описания синтаксиса ЯП Алгол 60. Наряду с новыми обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов, что сделало описание языка нагляднее и позволило применять данную нотацию для описания реальных ЯП.

В БНФ каждое определяемое понятие – это металингвистическая переменная (МЛП)*.* Ее значением может быть любая синтаксическая конструкция из некоторого фиксированного для этого понятия набора конструкций. Каждая металингвистическая форма (МЛФ) определяет одну МЛП и состоит из двух частей – левой и правой.

В левой части записывается определяемая МЛП, которая заключается в угловые скобки '<' и '>' (предполагается, что эти скобки являются метасимволами и не принадлежат алфавиту определяемого языка), например: <двоичное число>, <метка>, <арифметическое выражение>.

В правой части формы записываются все варианты определения конструкции (или *правила*), задаваемой этой формой. Каждый вариант представляет собой цепочку основных символов определяемого языка и МЛП. Варианты (правила) разделяются металингвистической связкой '|', имеющей смысл "или". Левая и правая части формы разделяются метасимволом '::=', означающим "по определению есть".

Ниже даётся описание идентификатора условного ЯП с использованием БНФ (каждое правило можно также записать и отдельно).

1. <буква> :: = А|В|С|D|E|…|W|X|Y|Z|a|b|c|d|e|…|w|x|y|z
2. <цифра> :: = 0|1|2|3|4|5|6|7|8|9
3. <идентификатор> ::= <буква>|<идентификатор><буква>|<идентификатор><цифра>

Характерной особенностью многих МЛФ является наличие в них *рекурсии.* Рекурсия имеет место в том случае, когда для определения конструкций ЯП используются МЛП, обозначающие саму определяемую конструкцию. Рекурсия может быть явной или неявной. *Явная рекурсия* используется, например, в определении МЛП «идентификатор». *Неявная рекурсия* имеет место в случае, когда МЛП, обозначающая какую-либо синтаксическую конструкцию, используется на некотором промежуточном шаге определения этой конструкции.

Наличие рекурсивных определений затрудняет чтение и понимание БНФ, хотя и является наиболее удобным способом описания бесконечных языков с помощью конечного числа правил. На практике для описания синтаксиса ЯП часто используют расширения БНФ, позволяющие более естественно представлять альтернативные, необязательные и повторяющиеся части МЛФ. Так, одно из расширений БНФ (РБНФ) разрешает использовать следующие упрощения:

* необязательные элементы СК заключаются в квадратные скобки;
* альтернативные варианты могут в случае необходимости заключаться в круглые скобки для образования многовариантного выбора;
* элементы СК, повторяющиеся нуль и более раз, заключаются в фигурные скобки '{' и '}'.

Любая синтаксическая конструкция, полученная с помощью РБНФ, может быть получена с помощью БНФ и наоборот.

1. **Классификация грамматик по Хомскому.**

Грамматика – это описание способа построения предложений некоторого языка. Определив грамматику языка, мы указываем правила порождения цепочек символов, принадлежащих этому языку. Таким образом, грамматика – это генератор цепочек языка. Она относится ко второму способу определения языков – порождению цепочек символов.

По иерархии Хомского, грамматики делятся на **4 типа**, где каждый последующий тип является более ограниченным подмножеством предыдущего (но и легче поддающимся анализу):

* тип 0 – неограниченные грамматики – возможны любые правила;
* тип 1 – контекстно-зависимые грамматики – левая часть может содержать один нетерминал, окруженный «контекстом» (цепочки символов, в том же виде присутствующие в правой части); сам нетерминал заменяется непустой последовательностью символов в правой части;
* тип 2 – контекстно-свободные грамматики (КСГ) – левая часть состоит из одного нетерминала;
* тип 3 – регулярные грамматики – более простые, эквивалентные конечным автоматам.

Примером использования грамматики нулевого типа является естественный язык или **машина Тьюринга**. КСГ широко применяются для определения грамматической структуры в грамматическом анализе. Регулярные грамматики (в виде регулярных выражений) применяются как шаблоны для текстового поиска, разбивки и подстановки (в том числе – в лексическом анализе).

Грамматика ЯП содержит правила двух типов: первые определяют синтаксические конструкции ЯП (их довольно легко можно формально описать), вторые – семантические ограничения ЯП (обычно излагаются в неформальной форме). Поэтому любое описание (стандарт) ЯП обычно состоит из двух частей: вначале формально излагаются правила построения синтаксических конструкций, а потом на естественном языке дается описание семантических правил. Во многих компиляторах имеется специальная часть – семантический анализатор, проверяющий правильность программы.

1. **Методика построения автоматов. 4 типа автоматов.**

**Абстрактный автомат** – математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. На вход этому устройству поступают символы одного алфавита, на выходе оно выдаёт символы другого (в общем случае) алфавита.

Формально абстрактный автомат определяется как A=(S,X,Y,δ,λ) , где

S – конечное множество состояний автомата,

X, Y – конечные входной и выходной алфавиты соответственно, из которых формируются строки, считываемые и выдаваемые автоматом,

δ : S ´ X →S – функция переходов,

l : S ´ X →Y – функция выходов.

**Конечный автомат** (КА) – абстрактный автомат, число возможных внутренних состояний которого конечно.

Теория автоматов лежит в основе цифровых технологий и программ. Компьютер является частным случаем практической реализации КА. Часть математического аппарата теории автоматов напрямую применяется при разработке ЛА и парсеров для ФЯ, в том числе ЯП, а также при построении компиляторов и разработке самих ЯП.

Существуют различные способы описания КА (задания алгоритма его функционирования). Например, КА может быть задан в виде упорядоченной пятерки элементов некоторых множеств: M = (V,Q,q0,F,δ) , где

V – входной алфавит (конечное множество входных символов), из которого формируются входные слова, воспринимаемые КА;

Q – множество внутренних состояний КА;

q0 – начальное состояние КА (q0⸦Q);

F – множество заключительных (конечных) состояний КА (F⸦Q);

δ – функция переходов, определенная как *δ(q,a)*; значение функции переходов на упорядоченной паре (*q* – состояние, *a* – входной символ или пустая цепочка) есть множество всех состояний, в которые из данного состояния возможен переход по данному входному символу или пустой цепочке.

Конечные автоматы подразделяются на детерминированные (ДКА) и недетерминированные (НКА).

**Детерминированный конечный автомат** – последовательность (кортеж) из пяти элементов (Q,Σ,δ,S0,F), где:

Q – множество состояний автомата

Σ – алфавит языка, который понимает автомат

δ – функция перехода, такая что δ: Q ´ Σ→Q

S0∈Q – начальное состояние

F∈ Q – множество конечных состояний.

**Недетерминированный конечный автомат** – последовательность (кортеж) из пяти элементов (Q,Σ,∆,S,F), где:

Q – множество состояний автомата

Σ – алфавит языка, который понимает автомат

∆ – отношение перехода, ∆={<*q,a,p*>:*q,p*∈ Q, *a*∈Σ ∪{ε}} , где {ε}- пустое слово. То есть, НКА может совершить скачок из состояния *q* в состояние *p*, в отличие от ДКА, через пустое слово, а также перейти из *q* по *a* в несколько состояний (что опять же в ДКА невозможно)

S Í Q – множество начальных состояний

F Í Q – множество конечных состояний.

НКА является обобщением ДКА.

**Автомат с магазинной памятью** (МПА) – это односторонний распознаватель, в потенциально бесконечной памяти которого элементы информации хранятся и используются так же, как и патроны в автомате – в каждый момент доступен только верхний элемент. МПА являются естественной моделью синтаксического анализатора контекстно-свободных языков*.*

Задать МПА – значит описать алгоритм его работы. Автомат считается заданным, если известны:

* алфавит входных сигналов;
* алфавит выходных сигналов;
* алфавит состояний;
* начальное состояние;
* функция переходов;
* функция выходов.

Используются следующие **способы задания автоматов**:

* словесное описание;
* графическое описание (направленные графы);
* табличное описание (таблица переходов и выходов);
* аналитическое описание (задание конкретного вида функций переходов и выходов).

**Преобразователи с магазинной памятью** получаются из МПА, если их снабдить выходной лентой и разрешить на каждом такте выдавать выходную цепочку конечной длины. Формально они задаются восьмеркой:

P = (Q,Σ,Г,∆,δ,q0,Z0,F),

где Q – конечное множество состояний преобразователя;

Σ – конечный входной алфавит;

Γ – конечный алфавит магазинных символов;

∆ – конечный выходной алфавит;

δ – множество переходов преобразователя (отображение множества вида Q´ (Σ ∪ {ε}) ×Τ\* в множество конечных подмножеств множества Q´Г\*´∆\*);

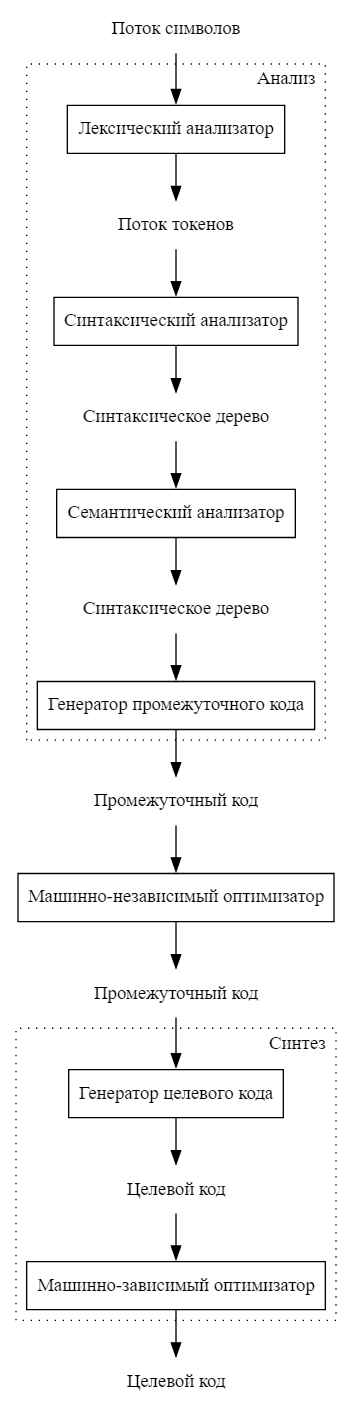
*q0* ∈ Q – начальное состояние автомата;

*Z0* ∈ Г – начальное состояние магазина;

*F* Í *Q* – множество заключительных состояний.

1. **Структура компилятора. Лексический анализатор.**

Процесс компиляции обычно разделяется на две фазы: анализ и синтез.

В фазе анализа происходит чтение исходного текста программы, затем этот текст разбивается на элементарные блоки, на них накладывается грамматическая структура, и создаётся промежуточное представление исходного текста и собирается другая информация об исходном тексте. На этой фазе так же возможен статический анализ исходного текста.

В фазе синтеза, на основе промежуточного представления и прочей информации, строится представление исходной программы в целевом коде. На этой фазе так же возможны преобразования целевого кода, называемые оптимизациями.

Кроме того, между анализом и синтезом может находиться фаза преобразований промежуточного кода, называемая машинно-независимой оптимизацией.

## Лексический анализ

Первая фаза компиляции называется лексическим анализом или сканированием.

Лексический анализатор соответственно так же называется лексером или сканером.

Лексический анализатор сканирует входной поток символов (исходного текста программы) и выделяет значащие последовательности символов, называемые лексемами.

Для каждой лексемы анализатор выводит токен, представляющий из себя комбинацию абстрактного символа (названия типа токена) и произвольного набора атрибутов. Часто в качестве “набора атрибутов” выступает ссылка в глобальную таблицу, называемую таблицей символов.

## Синтаксический анализ

Вторая фаза – синтаксический анализ или разбор, парсинг (от англ. parsing).

Синтаксический анализатор соответственно называется так же парсером.

Парсер строит из токенов, полученных от лексера, древовидное промежуточное представление (часто неявно), отражающее грамматическую структуру исходного кода. Примером такого представления является синтаксическое дерево, где узлы представляют операцию, дочерние узлы – аргументы этой операции.

Например, синтаксическое дерево арифметического выражения 1+2∗31+2∗3 может иметь вид:

## Семантический анализ

Семантический анализатор использует синтаксическое дерево для проверки исходной программы на корректность.

На этом же этапе происходит проверка типов, и информация о типах переменных записывается в атрибуты соответствующих узлов синтаксического дерева.

Если спецификация языка разрешает неявное приведение типов, на этом этапе синтаксическое дерево может быть переписано с добавлением явных операций приведения типов.

## Генерация промежуточного кода

В процессе компиляции, могут создаваться несколько промежуточных представлений, в частности, синтаксическое дерево.

Как правило, после завершения синтаксического и семантического анализа, значительная часть высокоуровневой информации (типы, названия переменных, многие управляющие конструкции и т.п.) далее не требуется, в связи с чем многие компиляторы по достижении этой фазы генерируют более низкоуровневое представление, называемое обычно промежуточным кодом.

Основными требованиями к промежуточному коду являются, с одной стороны, простота его получения из синтаксического дерева, и с другой стороны, простота генерации на его основе машинного кода.

Как следствие, часто в качестве промежуточного кода используется последовательность инструкций для некой абстрактной вычислительной машины.

На этом этапе обычно принимаются решения о распределении памяти для хранения значений переменных.

## Машинно-независимая оптимизация

На фазе машинно-независимой оптимизации, промежуточный код преобразуется с целью “улучшения” без изменений наблюдаемого поведения (в соответствии со спецификацией языка[1](https://wiki.livid.pp.ru/students/sp/lectures/1.html#fn1)). Под “улучшением” обычно понимается “ускорение”, но иногда возможны другие критерии, например “код меньшего размера” или “меньшее потребление памяти”.

Часто, алгоритм первичной генерации промежуточного кода достаточно простой, поэтому без фазы оптимизации, код оказывается достаточно неэффективным.

Объём работы, проделываемый различными компиляторами на этом этапе может сильно отличаться. Большинство распространённых на рынке компиляторов являются “оптимизирующими” и значительная часть времени компиляции уходит именно на оптимизацию (обычно есть способ отключить оптимизацию при необходимости).

## Генерация целевого кода

Генератор целевого кода, получая на вход промежуточный код, отображает каждую команду промежуточного кода в одну или несколько команд целевого.

Кроме того, генератор целевого кода занимается задачей распределения регистров исполнительного устройства.

## Машинно-зависимая оптимизация

Шаг машинно-зависимой оптимизации преобразует, как правило, уже целевой код. Основными способами оптимизации на данном этапе могут быть различные эквивалентные замены последовательностей машинных команд на более быстрые аналоги, не меняющие поведения перестановки команд или блоков команд, приводящие к ускорению и т.п.

Большинство решений машинно-зависимой оптимизации принимаются на основе модели исполнительного устройства, встроенной в компилятор. Например, в компилятор может быть включена информация об относительном времени выполнения различных инструкций определённого процессора (или семейства процессоров).

**Лексический анализ** («токенизация») – процесс аналитического разбора входной последовательности символов на распознанные группы – ***лексемы*** – с целью получения на выходе идентифицированных последовательностей, называемых **токенами** (подобно группировке букв в словах). В прикладном программировании понятия «лексема» и «токен» идентичны, но более сложные токенизаторы дополнительно классифицируют лексемы по различным типам («идентификатор, оператор», «часть речи» и т. п.).

Лексический анализ используется в компиляторах и интерпретаторах [исходного кода](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) ЯП, и в различных парсерах [слов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B2%D0%BE) естественного языка. Как правило, он производится с точки зрения определённого ФЯ или набора языков. Язык, а точнее его грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

Лексический анализатор (ЛА) обычно работает в две стадии –сканирование и оценка. На стадии сканирования ЛА обычно реализуется в виде конечного автомата, определяемого регулярными выражениями. В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах. Например, токен «целое число» может содержать любую последовательность десятичных цифр. Во многих случаях первый непробельный символ может использоваться для определения типа следующего токена, после чего входные символы обрабатываются один за другим пока не встретится символ, не входящий во множество допустимых символов для данного токена. В некоторых языках правила разбора лексем несколько более сложные и требуют возвратов назад по читаемой последовательности.

Чтобы получить токен со значением, соответствующим типу (например, целое или дробное число), выполняется **оценка** этой строки – проход по символам и вычисление значения. Токен с типом и подготовленным значением передаётся на вход синтаксического анализатора.

Результатом лексического анализа является список лексем-дескрипторов и таблицы. В таблицах хранятся значения выделенных в программе лексем.

**Дескриптор** − это пара вида: ( <тип лексемы> , < указатель>),

где <тип лексемы> − это, как правило, числовой код класса лексемы, который означает, что лексема принадлежит одному из конечного множества классов слов, выделенных в ЯП;

<указатель>− это может быть либо начальный адрес области основной памяти, в которой хранится адрес этой лексемы, либо число, адресующее элемент таблицы, в которой хранится значение этой лексемы.

1. **Синтаксические деревья и неоднозначность. Минимизация конечных автоматов.**

Рассмотрим грамматику c правилами вывода

(**1.1.1**)

*B → B+B*⏐*B\*B*⏐*V*⏐*C*

*V → a*⏐*b*⏐*c*⏐*... x*⏐*y*⏐*z*

*C → 0*⏐*1*⏐*2*⏐*...8*⏐*9*

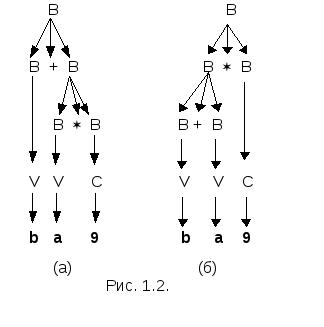
и вывод цепочки *b+a\*9*

(**1.1.2***) B⇒B+B⇒B+B*\**B⇒V+B*\**B⇒V+V*\**B⇒V+V*\**C⇒b+V*\**C⇒b+a*\**C⇒b+a*\**9*

Такая запись не очень удобна, так как по ней трудно определить в какой части сентенциальной формы проводилась замена и какой нетерминал породил тот или иной символ. Более наглядна запись в виде ***дерева вывода***или ***синтаксического дерева***, представленного на рис 1.2 (a).

Для того чтобы понять что выведено, применяем левый обход дерева. Идем от корня по крайней левой ветви, дойдя до терминала (конца ветви), выписываем его, возвращаемся до ближайшего разветвления и идем по самой левой из тех, которые еще не пройдены. Если все ветви данного узла уже исчерпаны, возвращаемся к предыдущему разветвлению, если оно есть. Продолжая таким образом, получим в результате *b+a\*9*. Кроме вывода (1.1.2) по данному дереву можно получить целую серию выводов, например,

(**1.1.3**) *B⇒B+B⇒B+B*\**B⇒B+B*\**C⇒V+V*\**9⇒B+V*\**9⇒B+a*\**C⇒V+a*\**C⇒b+a*\**9*

Заметим, что эти выводы отличаются лишь ***порядком*** применения правил и что синтаксическое дерево и грамматика не определяют точный порядок вывода. На каждом шаге вывода имеется некоторый произвол в выборе заменяемого нетерминала. На данном этапе эти различия порядка для нас несущественны и мы считаем выводы эквивалентными, если им соответствует одно и то же дерево. Более важным здесь является то, что цепочка *b+a*\**9* в данной грамматике имеет два дерева вывода (рисунки 1.2 (а) и (б)). Сентенциальная форма *B+B*\**B* имеет два синтаксических дерева и две основы: *B+B* и *B*\**B*. Грамматика неоднозначна и при разборе сентенциальной формы можно выбрать любую из основ. Нельзя сказать, что выполняется раньше: умножение или сложение. Из рис. 1.2 (б) следует, что *b+a*\**9* имеет два подвыражения*b+a* и *9*, хотя по смыслу необходимо иметь подвыражения *b* и *a*\**9*.

*Цепочка, порождаемая грамматикой,****неоднозначна****, если для ее вывода существует более одного синтаксического дерева.****Грамматика неоднозначна****, если она порождает неоднозначные цепочки, в противном случае она****однозначна****.*

Здесь речь идет о неоднозначной грамматике, а не языке. Изменяя неоднозначную грамматику можно получить однозначную грамматику для того же самого языка. Ниже приведена однозначная грамматика арифметических выражений

(**1.1.4**) *<врж> → <терм>*⎜*+ <терм>*⎜*− <терм>*⎜*<врж> + <терм>*⎜*<врж> − <терм>*

<*терм> → <множ>*⎜*<терм>*\* *<множ>*⎜*<терм> / <множ>*

<*множ> → (<врж>)* ⎜*i*⎜*k*

В этой грамматике *i* - любой идентификатор (имя переменной), а *k* - любая константа. Единственное дерево вывода для выражения *i+i*\**k* представлено на рис. 1.3. (a). В соответствии с предложенной грамматикой, эта, да и все остальные цепочки, порождаемые грамматикой (1.1.4) однозначны.

Определим теперь, что в выражении *i+i*\**k* должно выполняться раньше: сложение или умножение. Операндами для *+*, согласно дереву, является *<врж>*, из которого выводится *i*, и *<терм>*, порождающий *i*\**k*. Это означает, что умножение должно выполняться первым и образовать *<терм>* для сложения; следовательно, умножение предшествует сложению. Сделать наоборот можно используя только скобки, как показано на рис. 1.3 (б). Грамматику арифметических выражений (1.1.4) следует предпочесть грамматике (1.1.1) ввиду ее однозначности и учета приоритета операций.

Пусть дан автомат, распознающий определенный язык. Требуется найти эквивалентный автомат с наименьшим количеством состояний. Если в ДКА существуют два эквивалентных состояния, то при их объединении мы получим эквивалентный ДКА, так как распознаваемый язык не изменится.

Процесс минимизации КА начинается с поиска и удаления всех недостижимых состояний. Затем нужно найти такое разбиение множества состояний автомата, чтобы каждое подмножество содержало неразличимые состояния, т.е. если *s* и *t* принадлежат некоторому подмножеству, то для всех *a* из Σ *δ(s,a)* и *δ(t,a)* также принадлежат этому подмножеству.

Разобьём множество состояний на два подмножества: F и S–F. В дальнейшем нужно пытаться разбить каждое из подмножеств, соблюдая указанное выше условие. Если возникает ситуация, при которой невозможно разбить никакое множество состояний, то процесс разбиения заканчивается. В результате будет получен некоторый набор множеств состояний S1,…,Sk. Каждое из Si содержит только неразличимые состояния. Наконец, в множество состояний минимизированного автомата вносится по одному представителю каждого из множеств Si. На этом процесс завершается.

Пример. Дан исходный ДКА (рис.5).

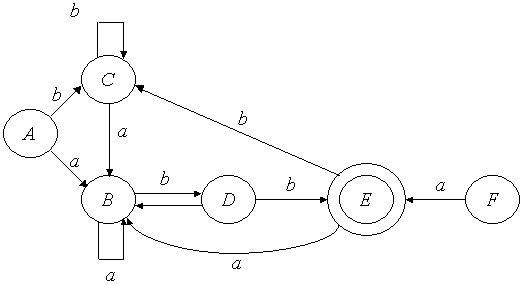


Рис.5. Исходный ДКА

Рассмотрим процесс минимизации данного автомата. Согласно алгоритму, вначале удаляются недостижимые состояния – в примере состояние F очевидно недостижимо и потому не попадёт в минимизированный автомат.

Затем множество состояний автомата разбивается на классы эквивалентности. Укажем такую последовательность разбиений:

1. E, ABCD
2. E, ABC, D, так как *δ(D,b)=E*.
3. E, AC, B, D, так как *δ(B,b)=D.*

Таким образом, состояния A и C неразличимы. Поэтому получаем следующий автомат (рис.6.):

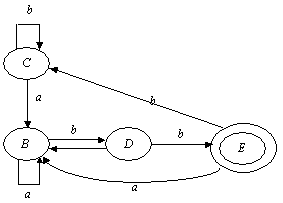


Рис.6. Минимизированный автомат

1. **Эквивалентные преобразования грамматик.**

**Сентенциальная форма** – последовательность терминалов и нетерминалов, выводимых из начального символа.

Пусть дана КСГ c алфавитом терминалов {*a*, *b*} и начальным символом *S*:

*S* → *aS | A | a*

*B* → *b*

Для данной грамматики нетерминалы *A* и *B* не могут встречаться в сентенциальных формах выводов терминальных цепочек из *S*. Другими словами, они не принимают участия в порождении цепочек языка, то есть являются в этом смысле **бесполезными**. Любую КСГ можно привести к форме, не содержащей бесполезных символов.

Символ (терминал или нетерминал) **недостижимым** в КСГ, если он не появляется ни в одной сентенциальной форме этой грамматики.

Нетерминальный символ называется **бесплодным** в КСГ, если множество выводимых из этого символа терминальных цепочек пусто.

КСГ называется **приведенной**, если в ней нет недостижимых и бесплодных символов.

Символ X называется **полезным** в грамматике G=(N,T,P,S), если существует некоторое порождение (пошаговый процесс, в котором на каждом шаге из цепочки, уже полученной на предыдущем шаге, можно путем применения к ней правил замены получить новую цепочку) вида S→αXβ→w, где w∈N\*.

X может быть как переменной, так и терминалом, а выводимая цепочка αXβ – первой или последней в порождении. Если символ X не является полезным, то называется бесполезным.

Символ X называется порождающим*,* если X→w для некоторой терминальной цепочки w.

Символ X называется достижимым*,* если существует порождение S→αXβ для некоторых α и β.

Полезный символ является одновременно и порождающим и достижимым.

Если сначала удалить из грамматики непорождающие символы, а потом недостижимые (**но не наоборот!**), то останутся только полезные. По сути, это – **приведение** грамматики.

**Алгоритм удаления непорождающих символов**

**Шаг 0**. Создаём пустое множество порождающих нетерминалов.

**Шаг 1**. Находим правила, не содержащие нетерминалов в правых частях и добавляем нетерминалы, встречающихся в левых частях таких правил, в это множество.

**Шаг 2**. Если найдено такое правило, что все нетерминалы, стоящие в его правой части, уже входят в множество, то добавим в это множество нетерминалы, стоящие в его левой части.

**Шаг 3**. Повторяем предыдущий шаг, если множество порождающих нетерминалов изменилось.

В результате получаем множество всех порождающих нетерминалов грамматики, а все нетерминалы, не попавшие в него, являются непорождающими.

### Алгоритм удаления недостижимых символов

**Шаг 0.** Создаём множество достижимых нетерминалов, состоящее из единственного элемента: {S}.

**Шаг 1.** Если найдено правило, в левой части которого стоит нетерминал, содержащийся в множестве, добавим в множество все нетерминалы из правой части.

**Шаг 2.** Повторяем предыдущий шаг, если множество достижимых нетерминалов изменилось.

Получаем множество всех достижимых нетерминалов, а нетерминалы, не попавшие в него, являются недостижимыми.

Некоторые применяемые на практике алгоритмы разбора по КСГ требуют, чтобы в грамматиках не было правил с пустой правой частью. Любую КСГ, не порождающую пустую цепочку, можно преобразовать в эквивалентную, не имеющую правил с пустыми правыми частями.

**Алгоритм исключения правил с пустой правой частью**

**Шаг 1.** Создаём множество Х нетерминалов грамматики G=(N,T,P,S), из которых выводима пустая цепочка.

**Шаг 2.** Удаляем из множества правил P все правила с пустой правой частью.

**Шаг 3.** Если S∈X, то вводим новый начальный символ S′, а в множество правил Pдобавляем правило S′ → S | ε.

**Шаг 4.** Для любого нетерминала A∈Xправило вида B→α1Aα2A...αnAαn+1,

где αi∈((N − {A}) ∪ T)\*, заменить 2*n* правилами, соответствующими всем возможным комбинациям вхождений А между αi:

B→α1α2...αnαn+1

B→α1α2...αnAαn+1

…

B→α1α2A...αnAαn+1

B→α1Aα2A...αnAαn+1

Замечание: если α*i*=ε для всех i=1,…,n+1, то получившееся на данном шаге правило B→ε не включаем в множество *P*

**Шаг 5.** Удаляем бесполезные символы и правила, их содержащие.

Пример. Рассмотрим грамматику с правилами S→*а*S*b*S | *b*S*a*S | ε.

Применяя алгоритм, получаем грамматику:

S'→S | ε

S→ *а*S*b*S | *b*S*a*S | *a*S*b* | *ab*S | *ab* | *b*S*a* | *ba*S | *ba*.

Цепное правило имеет вид A→B, где A и B являются нетерминалами. Для КСГ, содержащей цепные правила, можно построить эквивалентную ей грамматику, не содержащую цепных правил.

**Алгоритм удаления цепных правил из грамматики:**

**Шаг 1.** Находим все цепные пары в грамматике G.

**Шаг 2.** Для каждой цепной пары (A,B) добавляем в грамматику G′ все правила вида A→α, где B→α – нецепное правило из G.

**Шаг 3.** Удаляем все цепные правила.

Пример. Рассмотрим грамматику с правилами:

S→A|AB

A→SA|BB|B|bB|b

B→b|aA

Цепные правила: S→A и A→B

Цепные пары: (S,S), (A,A), (B,B)

Добавляем пары (S,A), (A,B), (S,B) для правила A→B и пары (S,A).

Меняем правила (жирным шрифтом выделены добавления):

S→**SA|BB|B|bB|b**|AB для (S,A),

A→SA|BB|**b|aA**|bB|b для (A,B),

S→SA|BB|**b|aA**|bB|b|AB для (S,B)

Удаляем повтор в A: A→SA|BB|aA|bB|b|

Удаляем повтор в S: S→SA|BB|aA|bB|b|AB

Правило вида A→αA , где A∈N , α∈(T∪N)\* , называется **праворекурсивным**, а правило вида A→ Aα – **леворекурсивным**. Для каждой КСГ G, содержащей леворекурсивные правила, можно построить эквивалентную грамматику G', не содержащую леворекурсивных правил.

**Алгоритм удаления леворекурсивных** **правил из грамматики:**

Пусть исходная грамматика G содержит правила:

A→Aα1 | Aα2 | ... |Aαn|β1 |β2 |...|βm ,

где ни одна цепочка β не начинается с A и αi,βj∈(T∪N)\*; i=1,…,n; j=1,…,m.

Введём новый нетерминал A' и преобразуем правила:

A → β1 |β2 |...|βm |β1A' |β2A'|...|βmA',

A'→ α1 |α2 |...|αn|α1A' |α2A'|...|αnA'.

Заменяя все правила с левой рекурсией в G описанным способом, получим грамматику G', такую что L(G)=L(G'), поскольку каждая цепочка, выведенная в грамматике G, может быть построена в грамматике G' и наоборот.

Рассмотрим построение выводов в G и G'. В грамматике G вывод цепочки имеет вид:

A→ Aα1 Aα1α1 Aα1α1α1 β1 α1α1α1.

В грамматике G' эта же цепочка выводится следующим образом:

A→ β1A'β1α1A'β1α1α1A'β1α1α1α1.

### **Пример. Рассмотрим грамматику с правилами:**

A→Sα|Aα

S→Aβ

Здесь есть непосредственная левая рекурсия A→Aα.

Добавим нетерминал A′ и добавим правила:

A→SαA′

A′→αA′

Получаем новую грамматику с правилами:

A→SαA′|Sα

A′→αA′|α

S→Aβ

1. **Алгоритмы генерации объектного кода. Триады, тетрады.**

Возможны различные формы внутреннего представления синтаксических конструкций исходной программы в компиляторе. На этапе синтаксического разбора часто используется форма, именуемая деревом вывода (методы его построения рассматривались в предыдущих лабораторных работах). Но формы представления, используемые на этапах синтаксического анализа, оказываются неудобными в работе при генерации и оптимизации объектного кода. Поэтому перед оптимизацией и непосредственно генерацией объектного кода внутреннее представление программы преобразуется в одну из соответствующих форм записи.

Примерами таких форм записи являются:

* обратная польская запись операций;
* тетрады операций;
* триады операций;
* собственно команды ассемблера.

Команды ассемблера удобны тем, что при их использовании внутреннее представление программы полностью соответствует объектному коду и сложные преобразования не требуются. Однако использование команд ассемблера требует дополнительных структур для отображения их взаимосвязи. Кроме того, внутреннее представление программы получается зависимым от результирующего кода, а это значит, что при ориентации компилятора на другой результирующий код потребуется перестраивать как само внутреннее представление программы, так и методы его обработки в алгоритмах оптимизации (при использовании триад или тетрад этого не требуется).

Для построения внутреннего представления объектного кода (в дальнейшем - просто кода) по дереву вывода может использоваться простейшая рекурсивная процедура. Эта процедура прежде всего должна определить тип узла дерева - он соответствует типу операции, символ которой находится в листе дерева для текущего узла. Этот лист является средним листом узла дерева для бинарных операций и крайним левым листом - для унарных операций. После определения типа процедура строит код для узла дерева в соответствии с типом операции. Если все узлы следующего уровня для текущего узла есть листья дерева, то в код включаются операнды, соответствующие этим листьям, и получившийся код становится результатом выполнения процедуры. Иначе процедура должна рекурсивно вызвать сама себя для генерации кода нижележащих узлов дерева и результат выполнения включить в свой порожденный код.

Поэтому для построения внутреннего представления объектного кода по дереву вывода в первую очередь необходимо разработать формы представления объектного кода для четырех случаев, соответствующих видам текущего узла дерева вывода:

* оба нижележащих узла дерева - листья (терминальные символы грамматики);
* только левый нижележащий узел является листом дерева;
* только правый нижележащий узел является листом дерева:
* оба нижележащих узла не являются листьями дерева.

Рассмотрим построение двух видов внутреннего представления по дереву вывода:

* построение ассемблерного кода по дереву вывода;
* построение списка триад по дереву вывода.

Существуют три формы записи выражений – префиксная, инфиксная и постфиксная. При префиксной записи операция записывается перед своими операндами, при инфиксной – между операндами, а при постфиксной – после операндов. Общепринятая запись арифметических выражений является примером инфиксной записи. Запись математических функций и функций в ЯП является префиксной (другие ее примеры – команды ассемблера, триады и тетрады).

**Тетрады** представляют собой запись операций в форме из четырех составляющих: операция, два операнда и результат операции. Например, тетрады могут выглядеть так:

<операция1>(<операнд1>,<операнд2>,<результат>).

Тетрады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме тетрад, они вычисляются одна за другой последовательно. Каждая тетрада в последовательности вычисляется так: операция, заданная тетрадой, выполняется над операндами и результат ее выполнения помещается в переменную, заданную результатом тетрады. Если какой-то из операндов (или оба операнда) в тетраде отсутствует (например, если тетрада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи и ее реализации).

**Триады** представляют собой запись операций в форме из трех составляющих – операция и два операнда:

<операция>(<операнд1>,<операнд2>).

Особенностью триад является то, что один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады. Поэтому триады при записи последовательно нумеруют для указания ссылок одних триад на другие.

Еще одно представление трехадресного кода состоит в использовании списка указателей на триады вместо списка самих триад. Такая реализация называется **косвенными триадами**.

1. **Трансляторы и компиляторы. Определения, примеры, методы работы.**

Язык программирования задается описанием и реализуется в виде специальной программы – ***транслятора***. Существует два вида трансляторов – ***компиляторы*** и ***интерпретаторы***. Соответственно и языки программирования делятся на два класса – компилируемые и интерпретируемые.

Текст программы, записанный на каком-либо компилируемом языке, при помощи специальной программы компилятора преобразуется (компилируется) в набор инструкций для данного типа процессора (машинный код) и далее записывается в исполняемый файл, который может быть запущен на выполнение как отдельная программа. То есть компилятор переводит программу с языка высокого уровня на низкоуровневый язык, понятный процессору.

Если же программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет (интерпретирует) ее текст без предварительного перевода. То есть компилятор переводит программу на машинный язык сразу и целиком, создавая при этом отдельную программу, а интерпретатор переводит на машинный язык прямо во время исполнения программы. Можно сказать, что процессор компьютера — это аппаратный интерпретатор машинного кода.

Как правило, скомпилированные программы выполняются быстрее и не требуют для выполнения дополнительных средств, так как уже переведены на машинный язык. Вместе с тем при каждом изменении текста программы требуется ее перекомпиляция, что создает трудности для разработчиков. Кроме того, скомпилированная программа может выполняться только на том же типе компьютеров и, как правило, под той же операционной системой, на которые был рассчитан компилятор. Чтобы создать исполняемый файл для машины другого типа требуется новая компиляция.

Интерпретируемые языки обладают некоторыми специфическими дополнительными возможностями, кроме того, программы на них можно запускать сразу же после изменения, что облегчает разработку. Программа на интерпретируемом языке нередко может быть запущена на разных типах машин и операционных систем без дополнительных усилий. Однако интерпретируемые программы выполняются заметно медленнее, чем компилируемые, и они не могут выполняться без дополнительной программы-интерпретатора.

**Компилятор** — это вид транслятора, преобразующий исходный код с какого-либо [языка программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) на [машинный язык](https://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)[[6]](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80#cite_note-%D0%A1%D0%B5%D0%B1%D0%B5%D1%81%D1%82%D0%B0-6).

Процесс компиляции, как правило, состоит из нескольких этапов:

* [лексический анализ](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
* [синтаксический анализ](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
* [семантический анализ](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
* создание на основе результатов анализов промежуточного кода;
* [оптимизация](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80%D0%B0) промежуточного кода;
* создание [объектного кода](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C), в данном случае [машинного](https://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).

Программа может использовать сервисы, предоставляемые [операционной системой](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%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), и сторонние [библиотеки](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (например, библиотеки для работы с файлами и библиотеки для создания графического интерфейса). Для добавления в [объектный файл](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C) [машинного кода](https://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) из других [объектных файлов](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C) (кода [статических](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B8) [библиотек](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))) и информации о [динамических](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B8) [библиотеках](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) выполняется *связывание* ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *link*) или *компоновка*. Связывание или компоновка выполняется [редактором связей или компоновщиком](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA). Компоновщик может быть отдельной программой или частью [компилятора](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80). Компоновщик создаёт [исполняемый файл](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D0%B8%D0%BC%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C). Исполняемый файл (программа) запускается следующим образом:

* по запросу пользователя в ядре [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%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) создаётся объект «[процесс](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))»;
* [загрузчик программ](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D1%87%D0%B8%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC) [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%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) выполняет следующие действия:
* читает [исполняемый файл](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB);
* загружает его в [память](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C);
* загружает в [память](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) [динамические библиотеки](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B8);
* выполняет связывание машинного кода программы с динамическими библиотеками (динамическое связывание);
* передаёт управление программе.

Достоинства компиляции:

* компиляция программы выполняется один раз;
* наличие компилятора на устройстве, для которого компилируется программа, не требуется.

Недостатки компиляции:

* компиляция — медленный процесс;
* при внесении изменений в исходный код, требуется повторная компиляция.

[Ассемблер](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80) — компилятор, преобразующий текст с языка [ассемблера](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0) на [машинный язык](https://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). [Язык ассемблера](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0) — [язык](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), близкий к [машинному языку](https://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), язык [низкого уровня](https://ru.wikipedia.org/wiki/%D0%9D%D0%B8%D0%B7%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F).

Интерпретация — процесс чтения и выполнения [исходного кода](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4). Реализуется программой — [интерпретатором](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80).

Интерпретатор может работать двумя способами:

1. читать код и исполнять его сразу (*чистая интерпретация*[[6]](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80#cite_note-%D0%A1%D0%B5%D0%B1%D0%B5%D1%81%D1%82%D0%B0-6));
2. читать код, создавать в [памяти](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) промежуточное представление кода ([байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) или [p-код](https://ru.wikipedia.org/wiki/P-%D0%BA%D0%BE%D0%B4)), выполнять промежуточное представление кода (*смешанная реализация*[[6]](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80#cite_note-%D0%A1%D0%B5%D0%B1%D0%B5%D1%81%D1%82%D0%B0-6)).

В первом случае трансляция не используется, а во втором — используется трансляция исходного кода в промежуточный код.

Этапы работы интерпретатора:

1. [лексический анализ](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
2. [синтаксический анализ](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
3. [семантический анализ](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7);
4. создание промежуточного представления кода (при чистой интерпретации не выполняется);
5. исполнение.

Интерпретатор моделирует машину ([виртуальную машину](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0)), реализует цикл выборки-исполнения команд машины. Команды машины записываются не на машинном языке, а на языке [высокого уровня](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F). Интерпретатор можно назвать исполнителем языка [виртуальной машины](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0).

Чистая интерпретация применяется, обычно, для языков с простой структурой, например, [языков сценариев](https://ru.wikipedia.org/wiki/%D0%A1%D1%86%D0%B5%D0%BD%D0%B0%D1%80%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA), языков [APL](https://ru.wikipedia.org/wiki/APL_(%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)" \o "APL (язык программирования)) и [Лисп](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BF).

Примеры интерпретаторов, создающих [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4): [Perl](https://ru.wikipedia.org/wiki/Perl" \o "Perl), [PHP](https://ru.wikipedia.org/wiki/PHP" \o "PHP), [Python](https://ru.wikipedia.org/wiki/Python" \o "Python), [Erlang](https://ru.wikipedia.org/wiki/Erlang" \o "Erlang).

Достоинства интерпретаторов по сравнению с компиляторами:

* возможность работы в интерактивном режиме;
* отсутствие необходимости перекомпиляции [исходного кода](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) после внесения изменений и при переносе кода на другую платформу.

Недостатки интерпретаторов по сравнению с компиляторами:

* низкая производительность (машинный код исполняется процессором, а интерпретируемый код — интерпретатором; машинный код самого интерпретатора исполняется процессором);
* необходимость наличия интерпретатора на устройстве, на котором планируется интерпретация программы;
* обнаружение ошибок синтаксиса на этапе выполнения (актуально для чистых интерпретаторов).

Сравнение чистого интерпретатора и интерпретатора, создающего [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4):

* чистый интерпретатор проще в реализации, так как для него не нужно писать код транслятора;
* интерпретатор, создающий [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4), может выполнять его [оптимизацию](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) и добиваться большей производительности, чем чистый интерпретатор;
* интерпретатор, создающий [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4), потребляет больше ресурсов системы (трансляция в [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) занимает процессорное время; [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) занимает место в [памяти](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C)).

1. **Польская инверсная запись.**

**Обратная польская запись** (ОПЗ) – форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Она также именуется как обратная бесскобочная запись, постфиксная нотация, польская инверсная запись (ПОЛИЗ).

Отличительная особенность ОПЗ – все аргументы (или операнды) расположены перед знаком операции. В общем виде при работе с ОПЗ руководствуются следующими правилами:

* Запись набора операций состоит из последовательности операндов и знаков операций. Операнды в выражении при письменной записи разделяются пробелами.
* Выражение читается слева направо. Когда в нём встречается знак операции, то выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их записи. Результат операции заменяет в выражении последовательность её операндов и её знак, после чего выражение вычисляется дальше по тому же правилу.
* Результатом вычисления выражения становится результат последней вычисленной операции.

Вычисления на стеке. Алгоритм Дейкстры для преобразования выражений из инфиксной нотации в ОПЗ.

**Объектно-ориентированное программирование**

1. Сетевое программирование. Классы веб-программирования: WebClient, WebRequest, WebResponse, Uri, IPAddress, Dns. Сокеты..
2. Классы. Конструкторы. Геттеры и сеттеры. Анонимные методы и делегаты.
3. Наследование. Абстрактный класс. Производный класс. Переопределение методов. Интерфейсы. Многопоточное программирование. Классы потоков. Фоновые потоки. Операции потоков.
4. ОО средства языка С++. Специальные методы классов — конструкторы и деструкторы. Привести пример на языке С++.
5. Совместное использование функций. Перегрузка функций, виртуальные и статические методы. Привести пример на языке С++.
6. Абстрактные классы и чистые виртуальные функции, перегрузка операторов. Привести пример на языке С++.
7. Параллельное программирование. Планирование и приоритеты потоков, синхронизация потоков, блокировка потоков.

### **Базы данных**

1. Основные функции СУБД.
2. Функциональные зависимости. Декомпозиция отношений.
3. Модель сущность-связь. ERD.
4. SQL.Создание и модификация структуры таблиц БД.
5. SQL. Выборка, проекция, соединение и сортировка.
6. SQL. Манипуляции данными.
7. SQL. Триггеры. Процедуры. Функции. Агрегатные функции.

### **Компьютерная графика и компьютерное моделирование**

1. Понятие системы координат. Многомерные пространства и проекции.
2. Системы математического описания цвета. Модели CMYK, RGB, CIE LAB.
3. Возможности и ограничения аппаратных средств в представлении цвета. Разрешающая способность устройств ввода и вывода.
4. Двумерные преобразования. Композиция двумерных преобразований.
5. Графические примитивы, математическое описание и возможность отрисовки в программной среде.
6. Анимированные изображения. Покадровая анимация.

### **Операционные системы**

1. Понятие операционной системы в узком и широком смысле.
2. История развития ОС.
3. Мультипрограммные пакетные ОС.
4. Принцип модульности. Соглашение о связях в ОС.
5. Командный язык ОС. Исполнение команд. Формат команд.
6. Управление виртуальной памятью. Страничная, сегментная и сегментно-страничная память.
7. Понятие процесса и ресурса. Планирование процессов.
8. Дисциплины диспетчеризации процессов.
9. Современные ОС для ПК. Отличия и преимущества.
10. История развития отечественных ОС

### **Защита информации**

1. Принципы и методы шифрования с открытым ключом (ассиметричного).
2. Цифровая подпись.
3. Дискреционное, мандатное и ролевое разграничение доступа.

### **Структуры и алгоритмы обработки данных**

1. Включённые в стандартные библиотеки языков программирования структуры данных. Описание и принцип работы.

2. Массивы. Сортировка вставками. Метод «пузырька». Сортировка выбором. Быстрая сортировка.

3. Бинарные деревья. Рекурсивный и итеративный методы обхода дерева.

4. Хеш-таблицы. Хеширование делением. Хеширование умножением. Универсальное хеширование. Хеширование с разрешением коллизий методом цепочек.

5. Структуры динамических множеств: стеки, очереди, связанные списки, деревья. Основные операции над элементами множеств.

**Технология разработки программного обеспечения**

1. Классификация моделей процесса создания ПО
2. Обработка исключений. Сборки dll.
3. Проектирование информационных систем и ПО средствами UML.
4. Системы управления версиями ПО.
5. Отказоустойчивые и критические системы.
6. Подходы к созданию защиты систем с конфиденциальной информацией.
7. Спецификация ПО. Типы, виды, назначение, примеры. Языки спецификаций.
8. Техническое задание. Определения, правила написания.
9. Управление качеством. Тестирование и отладка программного средства.
10. Критерии качества программного продукта. Обеспечение примитивов качества ПО.
11. Жизненный цикл программного продукта. Определения, описание, примеры, характерные особенности.
12. Документирование программных средств. Типы и виды документации. Описание.
13. Руководство программным проектом. Классические методы анализа. Измерения, меры и метрики.
14. Правила «хорошего тона» при написании ПО.
15. Авторские права и лицензирование ПО.
16. Методологии разработки ПО: Waterfall, V-model, Agile, XP, Scrum, Kanban, DevOps.
17. Задачи и проблемы сопровождения.

### **Программирование под платформу .NET**

1. Основные компоненты платформы .NET. CLR, IL,JIT-компилятор, FCL.
2. Объектно-ориентированное программирование на языке C#.
3. C#. Типы данных. Переменные, операции, выражения.
4. C#. Операторы цикла, условный, выбора, перехода.
5. C#. Классы. Структуры. Перечисления. Конструкторы.
6. C#. Индексаторы. Семейства. Клонирование. Итераторы.
7. C#. Интерфейсы. Делегаты. События.
8. C#. Классы - прототипы, параметризованные коллекции.

### **Функциональное и логическое программирование**

1. Функциональное программирование.
2. Логическое программирование.

### **Системы искусственного интеллекта**

1. Классификация и представление знаний. Модели представления знаний.
2. Понятия фазификации, дефазификации и лингвистической переменной. Операции с нечёткими множествами. Обобщённая процедура нечёткого логического вывода.
3. Алгоритмы нечёткого логического вывода. Методы дефазификации.
4. Основные понятия и определения интеллектуального анализа данных. Средства построения интеллектуальных систем поддержки принятия решений.
5. Аналитические и обучаемые модели. Алгоритмы машинного обучения.
6. Теория игр в искусственном интеллекте.
7. Классификатор «2А» систем искусственного интеллекта. Тест Тьюринга.
8. Методы интеллектуального анализа данных. Нейронные сети. Эволюционные алгоритмы.
9. Архитектура системы поддержки принятия решения. OLTP и OLAP системы.

### **Литература**

1. Симонович С.В. Информатика. Базовый курс.
2. Альфред В. Ахо и др. Структуры данных и алгоритмы.
3. Кормен Т. и др. Алгоритмы: построение и анализ.
4. Финогенов К.Г. Основы языка Ассемблера.
5. Юров В. и др. ASSEMBLER.Учебный курс.
6. Гордеев А. В. и др. Системное программное обеспечение.
7. Олифер В.Г., Олифер Н.А. Сетевые операционные системы.
8. Олифер В.Г., Олифер Н.А. Компьютерные сети.
9. Сергиенко А.Б. Цифровая обработка сигналов.
10. Пол А. ООП на С++.
11. Мамаев М. и др. Технология защиты информации в Интернете. Специальный справочник.
12. Гмурман . Теория вероятности и математическая статистика.
13. Турчак Л.И. Основы численных методов.
14. Красс М.С. и др. Основы математики и её приложения в экономическом образовании.
15. Круглов В.В., Дли М.И., Голубов Р.Ю. Нечёткая логика
16. и искусственные нейронные сети. 2001.
17. Гаврилова Т.А., Хорошевский В.Ф. Базы знаний интеллектуальных систем. 2001.
18. Круглов В.В., Борисов В.В. Искусственные нейронные сети: теория и практика. 2002.
19. Чулюков В.А., Астахова И.Ф., Потапов А.С. Системы искусственного интеллекта. Практический курс. 2008.
20. Цилькер Б.Я., Орлов С.А. Организация ЭВМ и систем: Учебник для вузов. – СПб.: Питер, 2011. – 688 с.
21. Мюллер С. Модернизация и ремонт ПК, 19-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2011. – 1072 с.