Теоретический минимум по курсу “Систем программирования”, составленный студентами 214 и 215 групп:

Мелтонян Артур
(набор текста, верстка, синхронизация работы)
Мартиросян Артур
(набор текста)
Титов Владислав
(набор текста)
Савостин Петр
(набор текста)
Лыков Александр
(набор текста)

Система программирования - это комплекс программных инструментов и библиотек, который поддерживает весь технологический цикл создания программного продукта.

Программа создается для решения отдельной задачи автором программы и используется в некоторой конкретной операционной среде.

Программным продуктом называется такая программа, которая работает без авторского присутствия в рамках некоторого набора операционных сред.

Системный (интегрированный) программный продукт есть комплекс программных продуктов.

Этапы жизненного цикла:

1. фазы разработки
2. фазы сопровождения
3. фазы использования.

Процессы разработки и сопровождения включают в себя этапы:

1. Анализ требований
2. Проектирование
3. Написание текста программ
4. Компоновка или интеграция программного комплекса
5. Верификация, тестирование и отладка
6. Документирование
7. Внедрение
8. Тиражирование
9. Сопровождение, повторяющее все предыдущие этапы.

Анализа требований во многом не формализован, но он влияет на всю разработку и качество конечного продукта. На этом этапе необходимо выяснить потребности конечного пользователя.

Проектирование разбивается на два подэтапа: проектирование структуры системы (проектирование “в большом”), и проектирование совокупности взаимосвязанных подсистем (проектирование “в малом”). На этапе проектирования осуществляется процесс, который называют управлением сложностью.

Написание текста программ представляет собой непосредственное программирование. Этот этап заканчивается, когда в наличии оказывается текст программы на некотором языке программирования.

Компоновка представляет собой интеграционный процесс связывания отдельных частей программы в одну большую систему программного обеспечения. Правильная организация процессов верификации, тестирования и документирования имеет особое значение для последующего внедрения, тиражирования и сопровождения создаваемых программ.

Верификация является процессом проверки на правильность программы. **Тестирование** программы есть процесс обнаружения дефектов в созданных программах. Во время тестирования выявляется несоответствие программы исходным требованиям и спецификациям по тестам, то есть программам с заранее известными ответами. **Отладка** программ – это выявление причин дефектов, а также их устранение. Отладка начинается после обнаружения дефектов.

Внедрение есть работа по привлечению заказчика к использованию созданного программного продукта. На этом этапе возникает множество организационных проблем по обучению пользователей, тестированию работоспособности и устойчивости программы при работе в организации и операционной среде.

Сопровождение является продолжением контактов пользователей и разработчиков. Сопровождение является зеркальным отражением разработки и внутри себя содержит все этапы фазы разработки.

Модели разработки:

1. **Каскадная** - процесс разработки может иметь вид последовательно выполняемых слабо связанных между собой этапов.
2. **Возвратная** - замыкание процесса и проведение повторного анализа требований после проведения его тестирования.
3. **Каскадно-возвратная** – возможны возвраты на предыдущие этапы.

Требований к системам программирования:

1. требование согласованности интерфейсов и непротиворечивости результатов работы компонентов этих систем
2. работы с системами программирования и отдельными их компонентами
3. полнота набора системных компонентов.

Общая схема работы систем программирования

1. средства интеграции компонентов системы программирования
2. редакторы текстов, в том числе макрогенераторы
3. компиляторы и ассемблеры
4. библиотеки
5. редакторы связей
6. средства конфигурирования и управления версиями
7. отладчики и средства тестирования
8. средства тестирования
9. профилировщики
10. справочные системы.

Интегрированная среда разработки - комплекс программных средств, поддерживающих полный жизненный цикл программного продукта.

Компоненты классической системы программирования

1. **Редакторы текстов:**
    a) **Текстовые редакторы** являются наиболее частыми начальным элементами процесса создания программы. Они позволяют готовить и вносить изменения в тексты исходных программ, однако в современных системах программирования их функции стали еще более широкими.
    b) **Пакетные редакторы** не требуют непосредственного присутствия программиста для своей работы. Они получают на вход исправляемый текст и пакетное задание на редактирование, в котором указано, какие фрагменты текста надо из текста исключить, какие переставить местами.
    c) **Диалоговые редакторы** отличаются от пакетных редакторов тем, что для них готовить задание на редактирование не требуется. Пользователь указывает редактору, какой текст он собирается редактировать, далее непосредственно вводятся редактирующие приказы.

Существует несколько вариантов взаимодействия программ с аппаратурой:

1. Первый из этих способов почти не требует системы программирования и связан с **кодированием программ непосредственно на машинном языке**. Такой подход становится приемлемым только в тех случаях, когда над всеми другими соображениями по поводу способов записи программ превалируют соображения эффективности.
2. Второй вариант, в котором введется **программирование на языке программирования**, не совпадающем с машинным языком вычислительной системы, требует наличия системы программирования. В состав системы программирования включаются несколько компонентов, важнейшим из которых является компонент, ответственный за преобразование исходной программы к виду, в котором она может быть понята вычислительной системой - транслятор.

2. **Трансляторы, компиляторы, интерпретаторы, ассемблеры:**

Транслятор - программами, которые переводят исходную программу, написанную на некотором исходном (входном) языке, в другую программу (на объектном языке), эквивалентную первой, в зависимости от методов трансляции различают типы трансляторов:

a) Если исходная запись программы ведется на языках, языках, близких к машинному представлению программ, то такие трансляторы – **ассемблеры.**
b) Если исходная запись программы ведется на языках высокого уровня, а объектным языком является автокод, язык ассемблера или машинный язык некоторой вычислительной машины, то такие трансляторы – **компиляторы**.
c) Если транслятор некоторого исходного языка выполняет действия, которые этой программой предписываются (для этого интерпретатору необходимо передавать еще и входные данные программы), то такой транслятор – **интерпретатор**. Он не порождает объектную программу, которая впоследствии должна выполняться, а выполняет ее сам. При интерпретации некоторой программы она размещается там, где обычно размещаются исходные данные выполняемых программ.

d) Иногда интерпретатор сначала производит преобразование исходной программы в некоторое внутреннее представление, которое затем программно интерпретируется. Такой подход называется **смешанной стратегией трансляции**. Как и языки ассемблеров, язык внутреннего представления программ в интерпретаторах разрабатывается в таком виде, чтобы на второй фазе (фазе интерпретации) легко его расшифровывать и тратить минимум времени на анализ каждого отдельного предложения внутреннего языка при его выполнении.

Проход – это процесс последовательного чтения компилятором данных из внешней памяти, их обработки и записи результата во внешнюю память. Во время одного прохода может выполняться сразу несколько фаз компиляции, но случается, что одна фаза компиляции выполняется за несколько проходов.

Основные компоненты компилятора и фазы компиляции:

1. Начальные установки
2. Фазы анализа программ
   a. Лексический анализ
   b. Синтаксический анализ
   c. Семантический анализатор
3. Фазы оптимизации программ
4. Фазы синтеза программ
   a. Распределение памяти
   b. Распределение регистров
   c. Генерация команд и машинно-зависимая оптимизация.

Задачи лексического анализатора:

1. просмотр всего текста исходной программы и выделение в нем последовательность лексем
2. уменьшает длину программы, устраняя из ее исходного представления несущественные пробельные символы и комментарии.

Задачи синтаксического анализатора:

1. установить, имеет ли цепочка лексем структуру, заданную синтаксисом языка
2. зафиксировать эту структуру.

Задачи семантического анализатора – проверка контекстных условий:

1. каждое имя, используемое в программе, должно быть описано, причем только один раз
2. в операторе присваивания типы переменной и выражения должны совпадать, либо относиться к некоторым семантически близким типам
3. в условном операторе и в операторе цикла в качестве условия возможно только логическое выражение
4. операнды операций отношения должны быть целочисленными, либо иметь какие-либо другие, но точно известные типы
5. при вызове функции число фактических параметров и их типы должны соответствовать числу и типам формальных параметров.

Основные свойства языков внутреннего представления программ:

1. языки внутреннего представления позволяют фиксировать синтаксическую структуру исходной программы
2. текст на языках внутреннего представления можно автоматически генерировать во время синтаксического анализа
3. конструкции языков внутреннего представления относительно просто транслируются в объектный код, либо достаточно эффективно интерпретируются.

Способы внутреннего представления программ:

1. связные списочные структуры, представляющие синтаксическое дерево
2. инфиксная запись (операции записываются между своими операндами)
3. префиксная запись (операции записываются перед своими операндами)
4. постфиксная запись (операции записываются после своих операндов)
5. язык ассемблера и машинные коды
6. многоадресный код с явно именуемым результатом
7. многоадресный код с неявно именуемым результатом.

Оптимизация программы - это изменение компилируемой программы с целью получения более эффективной объектной программы. Используются два критерия эффективности результирующей программы: **скорость выполнения** программы и **объем памяти**, необходимый для выполнения программы. Принципиально различаются два основных вида оптимизирующих преобразований: **машинно-независимые преобразования** исходной программы и **машинно-зависимые преобразования** результирующей объектной программы.

Машинно-независимые преобразования исходной программы производятся в основном над ее внутренним представлением и основаны на известных математических и логических преобразованиях.

1. Удаление недостижимых фрагментов программы:
   If (1) a := 1; else a := 0; →
   a := 1;
2. Оптимизация линейных участков программы:

   a) Удаление бесполезных присваиваний:
      a := b; a := c; →
      a := c;
   b) Исключение избыточных вычислений:
      m = m + b * c; n = n + b * c; →
      t = b * c; m = m + t; n = n + t;
   c) Перестановка операций:
      a = 2 * b * 3 * c; →
      a = (2 * 3) * (b * c);
   d) Арифметические преобразования:
      a = b * c + b * d; →
      a = b * (c + d);
   e) Логические преобразования:
      a = true || a ; →
      a = true;

f) Свертка объектного кода:
   a = 2 + 2; b = 6 * a; →
   a = 4; b = 24;

3. Оптимизация циклов:
   a) Вынесение инвариантных вычислений из циклов:
      for (i = 1; i <= 10; i++) a [i] = b * c * a [i]; →
      d = b * c; for (i = 1; i <= 10; i++) a [i] = d * a [i];
   b) Замена операций с индуктивными переменными:
      for (i = 1; i <= N; i++) a [i] = i * 10; →
      t = 10; i = 1; while (i <= N) { a [i] = t; t = t + 10; i++; }
   c) Слияние циклов:
      for (i =1; i <= N; i++) for (j = 1; j <= M; j++) a [i] [j] = 0; →
      k = N * M; for (i = 1; i <= k; i++) a [i] = 0; (только в объектном коде!)
   d) Развертывание циклов, кратность выполнения которых известна на этапе компиляции:
      for (i = 1; i <= 3; i++) a [i] = i; →
      a [1] = 1; a [2] = 2; a [3] = 3;
4. Подстановка кода функции вместо ее вызова в объектный код.

Машинно-зависимые преобразования результирующей объектной программы зависят от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. При этом может учитываться объем кэш-памяти, методы организации работы процессора.

1. Распределение регистров процессора:
   использование регистров общего назначения и специальных регистров для хранения значения операндов и результатов вычислений позволяет увеличить быстродействие программы, доступных регистров всегда ограниченное количество, поэтому перед компилятором встает вопрос их оптимального распределения и использования при выполнении вычислений.
2. Оптимизация передачи параметров в процедуры и функции:
   обычно параметры процедур и функций передаются через стек, можно уменьшить код и время выполнения результирующей программы за счет оптимизации передачи параметров в процедуру или функцию, передавая их через регистры процессора, недостатки данного подхода: такие функции не могут быть использованы в качестве библиотечных и этот метод не может быть использован, если где-то в функции требуется
   выполнить операции с адресами параметров.
3. Оптимизация кода для процессоров, допускающих распараллеливание вычислений:
   при возможности параллельного выполнения нескольких операций компилятор должен порождать объектный код таким образом, чтобы в нем было максимально возможное количество соседних операций, все операнды которых не зависят друг от друга, для этого надо найти оптимальный порядок выполнения операций для каждого оператора (переставить их).
   a + b + c + d + e + f; →
   (a + b) + (c + d) + (e + f)

Распределение памяти - это процесс, в результате которого отдельным элементам исходной программы ставятся в соответствие адрес, размер и атрибуты области памяти, необходимой для размещения лексических единиц.

Область памяти - это блок ячеек памяти, выделяемых для данных и каким-то образом объединенных логически.

Распределение памяти выполняется после фазы анализа текста исходной программы **на этапе подготовки к генерации объектного модуля** (перед генерацией кода объектного модуля).

По **способу использования** области памяти делятся на **глобальные** и **локальные**, а по **способу распределения** – на **статические** и **динамические**, динамический способ распределения делится на **стековую дисциплину** и **дисциплину произвольного распределения**, дисциплина произвольного распределения делится на **распределяемую программистом** и **распределяемую компилятором**.

Локальная память - это область памяти, которая выделяется в начале выполнения некоторого фрагмента результирующей программы (блока, функции, оператора) и может быть освобождена по завершении выполнения данного фрагмента. Доступ к локальной области памяти всегда запрещен за пределами того фрагмента программы, в котором она выделяется.

Глобальная память - это область памяти, которая выделяется один раз при инициализации результирующей программы и действует все время выполнения программы. Как правило, глобальная область памяти доступна из любой части исходной программы.

Статическая память - это область памяти, размер которой известен на этапе компиляции. Для статической памяти компилятор порождает некоторый адрес (как правило, относительный), и дальнейшая работа с ней происходит относительно этого адреса.

Динамическая память - это область памяти, размер которой становится известным только на этапе выполнения результирующей программы. Для динамической памяти компилятор порождает фрагмент кода, который отвечает за распределение памяти (ее выделение и освобождение). Динамические области памяти можно разделить на динамические области памяти, **выделяемые пользователем (явно выделяемые)** и **непосредственно компилятором (неявно)**.

Методы явного динамического распределения памяти:

1. Явное выделение блоков фиксированного размера
2. Явное выделение блоков переменного размера.

Неявное выделение динамической памяти

1. Неявное выделение блоков фиксированного размера
2. Неявное выделение блоков переменного размера.

При неявном динамическом выделении и освобождении блоков памяти, выделяемые блоки обычно имеют следующую структуру:

1. **Размер блока** (если блок фиксированного размера, то эта информация в нем не хранится)
2. **Счетчик ссылок, пометка** (обычно есть либо одно, либо другое).

Счетчик ссылок подсчитывает количество указателей в программе, которые ссылаются на этот блок (например, при p = q, один счётчик (для блока, на который указывал указатель p) уменьшается на единицу, а другой увеличивается), если счётчик равен нулю, то блок не используется и его можно освободить. Существует проблема циклических ссылок, когда счётчики всегда больше 0.

Пометка фиксирует, задействован данный блок или не задействован, т.е. у программы есть хотя бы один указатель, ссылающийся на этот блок. В некоторый момент начинает работать «сборщик мусора». Он помечает все блоки как недостижимые, а затем начинает анализ текущих указателей программы, что является очень трудоемким процессом. Блоки, на которые ничего не указывает, считаются свободными. Отсутствует проблема циклических ссылок.

При **генерации** объектного кода компилятор переводит текст программы во внутреннем представлении в текст программы на выходном языке (как правило, машинном).

Генерация объектного кода происходит на основе:

1. Определенной на фазе компиляции синтаксической структуры программы.
2. Информации, хранящейся в таблице идентификаторов.
3. Результата распределения памяти.

Часто для построения кода результирующей программы компиляторы используют синтаксически управляемый перевод.

Редактор связей (компоновщик) предназначен для связывания между собой (по внешним данным) объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав системы программирования.

Редактор связей выполняет следующее:

1. связывает между собой по внешним данным объектные модули, порождаемые компилятором и составляющие единую программу
2. связывает файлы статически подключаемых библиотек с целью получения единого исполняемого модуля
3. готовит таблицу трансляции относительных адресов для загрузчика
4. готовит таблицу точек вызова функций динамически подключаемых библиотек.

Загрузчик преобразовывает условные адреса разделов памяти в истинные (абсолютные).

Настраивающий загрузчик – это загрузчик, который выполняет трансляцию адресов в момент запуска программы.

Типы библиотек

1. Библиотеки функций (или подпрограмм)
2. Библиотеки классов
3. Библиотеки компонент.

Библиотеки функций представляют собой откомпилированные объектные модули, а необходимые фрагменты библиотеки функций включаются в исполняемый файл на этапе работы редактора связей**.**
Различают:

1. библиотеки для языков программирования (например, функции ввода-вывода, работа со строками)
2. библиотеки для решения задач в конкретной проблемной области (например, функции, реализующие алгоритмы линейной алгебры).

Библиотеки классов включаются в программу на этапе компиляции и компилируются со всей программой вместе. Недостаток библиотеки классов - все ее классы должны быть написаны на том же языке программирования, на котором пишется программа, куда интегрируются библиотечные классы.
Различают:

1. конкретные классы
2. абстрактные классы, иерархии классов
3. шаблоны классов, иерархии шаблонов классов.

Библиотеки компонент - это библиотеки готовых откомпилированных программных модулей, предназначенных для использования в качестве составных частей программ, и которыми можно манипулировать во время разработки программ. Компоненты бывают **локальные** (находящиеся на той электронно-вычислительной машине, где создается программный продукт) и **распределенные** (расположенные на сервере и доступные по сети электронно-вычислительной машины).

Динамически подключаемые библиотеки в отличие от статических библиотек подключаются к программе не во время компиляции программы, а непосредственно в ходе её выполнения. На этапе компоновки программы редактор связей, встречая вызовы функций динамически подключаемых библиотек, вместо процедуры связывания формирует таблицу точек вызова функций динамически подключаемых библиотек для последующей операции динамического связывания. Таким образом, процесс полной компоновки завершается уже на этапе выполнения целевой программы.

Преимущества динамически подключаемые библиотеки:

1. не требуется включать в программу объектный код часто используемых функций, что существенно сокращает объем кода
2. различные программы, выполняемые в некоторой ОС, могут пользоваться кодом одной и той же библиотеки, содержащейся в ОС
3. изменения и улучшения функций библиотек сводится к обновлению только содержимого динамически подключаемых библиотек, а уже существующие тексты программ не требуют перекомпиляции (этот же факт может оказаться недостатком, если при модификации функций меняется логика их работы).

Варианты управления конфигурацией программного комплекса:
1. из командной строки (последовательное обращение к компонентам)
2. использование командных файлов (содержащих последовательность вызовов компонент систем)
3. работа в интегрированных средах с проектами программных комплексов (переход от отладочной к оптимизированной конфигурации)
4. использование систем управления версиями программных комплексов (коллективный доступ к централизованным базам данным программных объектов).

Системы контроля версий:
1. **SCCS** – отслеживание изменений различных версий файлов проекта и разделение доступа к ним при помощи документирования модификаций проекта, контроля полномочий пользователя-разработчика проекта, сопровождения параллельных версий проекта, восстановления старых версий проекта.
2. **RCS** – по сравнению с SCCS имеет более четкий командный интерфейс, хорошие средства сборки редакций проекта, такого как история дерева каталогов с исходным кодом (маркировка времени изменений, имени пользователя-редактора, комментария).
3. **CVS** – по сравнению с RCS имеет систему параллельных версий.

Конфликт – одновременное обращение нескольких разработчиков к одному участку файла с целью его редактирования. Решение этому – временная блокировка одним из разработчиков данного участка.

Цикл отладки над программой:
1. расставить операторы выдачи промежуточных результатов работы
2. исследовать содержимое памяти, занятое командами или данными
3. применить автоматизированные средства отладки и тестирования.

Отладчик – программное средство организации проверочных запусков программ, способствующее локализации и исправлению ошибок.

Задачи отладчика в рамках ИСР:
1. пошаговое выполнение отлаживаемой программы
2. выполнение отлаживаемой программы до достижения ею точки курсора/одной из заранее заданных точек остановки
3. выполнение отлаживаемой программы до становления истинным некоторого логического выражения над переменными и адресами программы
4. трассировка и обратная трассировка работы программы
5. выдача диагностических сообщений в терминах входного языка отлаживаемой программы
6. просмотр и/или изменение значения переменных программы и содержимого областей памяти, занятой ею
7. изменять текст отлаживаемой программы и продолжать отладку без полной перекомпиляции.

Двоичные отладчики – выполнение в виде автономных программных компонентов, символьные – в терминах исходного языка.

Стратегия тестирования — это метод, используемый для отбора тестов, которые должны быть включены в тестовый комплект. Ее эффективность определяется вероятностью обнаружения ошибки тестируемого объекта и зависит от комбинации природ тестов и ошибок. Различают следующие типы стратегий поведения:

1. **Стратегия поведенческого теста** – основана на технических требованиях. Тестирование, основанное на ней, называется функциональным/поведенческим/черного ящика (выполнение всех тестов, вытекающих из требования к программе).
2. **Стратегия структурного теста** – основана на структуре тестируемого объекта и полном доступе к ней. Тестирование, основанное на данной стратегии, называется тестированием белого ящика (выполнение каждой ветви программы/использование всех объектов).
3. **Стратегия гибридного теста** = поведенческая+ структурная. Модули и низкоуровневые компоненты тестируются с помощью структурной стратегии, большие компоненты и системы – с помощью поведенческой.

Способы тестирования:

1. **Автономное** – тестирование отдельных компонент программ до их объединения в единый комплекс (проводится разработчиками).
2. **Комплексное** - проверка всех аспектов работы программы от правильности взаимодействия компонент, до правильности взаимодействия программного комплекса с пользователями.
3. **Пользовательское** – проверка результатов работы программы с прикладной точки зрения.
4. **Техническое** – проверка безопасности и эффективности работы в нормальном и пиковых режимах ее использования (в смысле влияния на важнейшие технические параметры, например, быстродействие).
5. **Регрессивное** – пост-проверка отсутствия изменения/утраты функционала программного комплекса в результате внесенных изменений, не ставящих перед собой такую цель.

Проблемы работы с версиями программы:

1. Обратное восстановление – трудоемкость и ненадежность процесса восстановления в результате нежизнеспособности внесенных изменений
2. Отслеживание изменений – локализация изменений в коде
3. Отслеживание ошибок – наследие ошибок.

Справочная система – совокупность справочных материалов по языку программирования и компонентам системы программирования.

Профилирование – определение времени, затрачиваемого на выполнение отдельных фрагментов программ, как правило, для отдельных участков кода (в системе UNIX профилировщик называется **prof**).
Способы решения проблем профилировщиком:

1. отказ от лишних вычислений
2. корректировка алгоритма
3. отказ от многократных повторных вычислений путем хранения результатов для последующего использования.

Алфавит - это конечное множество символов.
Цепочкой символов в алфавите V - любая конечная последовательность символов этого алфавита.
Пустая цепочка ε - цепочка, которая не содержит ни одного символа.
Конкатенация, или сцепление цепочек α и β - цепочка αβ, то есть результат приписывания цепочки β в конец цепочки α.
Обращением, **или реверс** цепочки α - цепочка α, символы которой записаны в обратном порядке.
N-ая степень цепочки α - конкатенация n цепочек α.
Длина цепочки - это число составляющих ее символов.
$|\alpha|_s$ - число вхождений символа s в цепочку α.
V^*- множество содержащее все цепочки в алфавите V, включая пустую цепочку ε.
V^+- множество содержащее все цепочки в алфавите V, исключая пустую цепочку ε.
Язык в алфавите V — это подмножество множества всех цепочек в этом алфавите.
Порождающая грамматика G — это четверка (T, N, P, S), где
T — алфавит терминальных символов (терминалов);
N — алфавит нетерминальных символов (нетерминалов), $T \cap N = \emptyset$;
P — конечное подмножество множества $(T \cup N)^+ \times (T \cup N)^*$;
элемент (α,β) множества P - правилом вывода и записывается в виде $\alpha \to \beta$;
α - левая частью правила, β - правая часть;
левая часть любого правила из P обязана содержать хотя бы один нетерминал;
S — начальный символ (цель) грамматики, $S \in N$.
Цепочка $\beta = \mu_1\beta_0\mu_2$ **непосредственно выводима из** $\alpha = \mu_1\alpha_0\mu_2$: $\alpha_0 \to \beta_0$ – существует правило $\alpha_0 \to \beta_0$.
Цепочка $\beta = \mu_1\beta_0\mu_2$ **выводима из** $\alpha = \mu_1\alpha_0\mu_2 : \alpha \Longrightarrow \beta$ – существует последовательность цепочек γ, такая что $\alpha_0 = \gamma_0 \to \ldots \to \gamma_n = \beta_0$.
Язык, порождаемый грамматикой G = (T, N, P, S) – множество всех возможных цепочек, выводимых из элементов T^*.
Сентенциальной формой грамматики G называется цепочка, выводимая из начального нетерминала грамматики G.
Эквивалентные грамматики G_1 и G_2, если L(G1) = L(G2).
Почти эквивалентны грамматики G_1 и G_2, если $L(G1) \cup \{ \varepsilon \} = L(G2) \cup \{ \varepsilon \}$.
Классификация грамматик и языков по Хомскому

1. **Тип 0:** любая порождающая грамматика является грамматикой типа 0.
2. **Тип 1:**
   Грамматика G = (T, N, P, S) называется **неукорачивающей**, если правая часть каждого правила из P не короче левой части.
   Грамматика G = (T, N, P, S) называется **контекстно-зависимой**, если каждое правило из P имеет вид $\alpha \to \beta$, где $\alpha = \varepsilon_1 \alpha_0 \varepsilon_2$, $\beta = \varepsilon_1 \beta_0 \varepsilon_2$, $\alpha_0 \in N$, $\beta_0 \in (T \cup N)^+$, $\varepsilon_1, \varepsilon_2 \in (T \cup N)^*$.
3. **Тип 2:**
   Грамматика G = (T, N, P, S) называется **контекстно-свободной**, если каждое правило из P имеет вид $\alpha \to \beta$, где $\alpha \in N$, $\beta \in (T \cup N)^*$

4. **Тип 3:**
   Грамматика G = (T, N, P, S) называется **праволинейной**, если каждое правило из P имеет вид $\alpha \to w\,\beta$ либо $\alpha \to w$, где $\alpha, \beta \in N$, $w \in T^*$.
   Грамматика G = (T, N, P, S) называется **леволинейной**, если каждое правило из P имеет вид $\alpha \to \beta\, w$ либо $\alpha \to w$, где $\alpha, \beta \in N$, $w \in T^*$.

Иерархия Хомского:

1. **Иерархия грамматик:**
   НЕУКОРАЧИВАЮЩИЕ РЕГУЛЯРНЫЕ ГРАММАТИКИ ⊂
   НЕУКОРАЧИВАЮЩИЕ КОНЕТКСТНО СВОБОДНЫЕ ГРАММАТИКИ⊂
   КОНТЕКСТНО-ЗАВИСИМЫЕ ГРАММАТИКИ ⊂
   ТИПА 0 ГРАММАТИКИ
2. **Иерархия языков:**
   РЕГУЛЯРНЫЕ ЯЗЫКИ ⊂
   КОНЕТКСТНО СВОБОДНЫЕ ЯЗЫКИ ⊂
   КОНТЕКСТНО-ЗАВИСИМЫЕ ЯЗЫКИ ⊂
   ТИПА 0 ЯЗЫКИ

Контекстно-свободная грамматика G называется **неоднозначной**, если существует хотя бы одна цепочка α из L(G), для которой может быть построено два или более различных деревьев вывода, иначе – **однозначная**.
Язык, порождаемый грамматикой, называется **неоднозначным**, если он не может быть порожден никакой однозначной грамматикой, иначе – однозначный.
Недостижимый символ $x \in T \cup N$ в грамматике G = (T, N, P, S), если он не появляется ни в одной сентенциальной форме этой грамматики.
Бесплодный символ $x \in N$ в грамматике G = (T, N, P, S), если множество $\{ \alpha \in T^* \mid x \Longrightarrow \alpha \}$ пусто.
Детерминированный конечный автомат (ДКА) — это пятерка
(K, T, δ, H, S), где
K — конечное множество состояний;
T — конечное множество допустимых входных символов;
δ — отображение множества $K \times T$ в K, определяющее поведение автомата;
$H \in K$ — начальное состояние;
$S \in K$ — множество заключительных состояний
Недетерминированный конечный автомат (НКА) — это пятерка
(K, T, δ, H, S), где
K — конечное множество состояний;
T — конечное множество допустимых входных символов;
δ — отображение множества $K \times T$ в множество подмножеств K;
$H \in K$ — конечное множество начальных состояний;
$S \in K$ — конечное множество заключительных состояний.
follow(A) в грамматике G = (T, N, P, S) — это множество терминальных символов, которые могут появляться в сентенциальных формах грамматики непосредственно справа от A (или от цепочек, выводимых из A).
first (α) в грамматике G = (T, N, P, S) — это множество терминальных символов, которыми начинаются цепочки, выводимые в G из цепочки $\alpha \in (T \cup N)^*$

Критерий применимости метода рекурсивного спуска: пусть G - контекстно-свободная грамматика, тогда метод рекурсивного спуска применим к G, если и только если для любой пары альтернатив $X \to \alpha \mid \beta$ выполняются:

1. $first(\alpha) \cap first(\beta) = \emptyset$
2. справедливо не более чем одно из двух соотношений: $\alpha \Longrightarrow \varepsilon$, $\beta \Longrightarrow \varepsilon$
3. если $\beta \Longrightarrow \varepsilon$, то $first(X) \cap follow(X) = \emptyset$.

Мы специально печатали не все, все выучить невозможно. (с) Великие цитаты Великих всех