Теория автоматов и формальных языков

Презентация к лекциям

Блог: anshamshev.wordpress.com

Структура курса

- **Введение.**
- **Основные понятия теории автоматов.**
- **Конечные автоматы.**
- **Регулярные выражения** и **регулярные языки.**
- **Контекстно-свободные грамматики и языки и автоматы с магазинной памятью.**
- **Нормальные формы кс-грамматик. Проблема неоднозначности для языков и грамматик.**
- **Языки и грамматики в целом.**
- **Алгоритмически неразрешимые проблемы автоматов и формальных грамматик.**
- **Машина Тьюринга, модификации машины. Универсальная машина Тьюринга**

Объём курса

- 5 семестр:
  - 16 часов лекций
  - 16 часов практики
  - Расчётно-графическая работа
  - Зачёт

Список литературы

- Джон Хопкрофт и др. Введение в теорию автоматов, языков и вычислений
- Ахо А., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2001.
- Рейуорд-Смит В. Дж. Теория формальных языков. Вводный курс. М.: Радио и связь, 1988.

История конечных автоматов

- Теория автоматов занимается изучением абстрактных вычислительных устройств, или "машин". В 1930-е годы, задолго до появления компьютеров, А. Тьюринг исследовал абстрактную машину, которая, по крайней мере в области вычислений, обладала всеми возможностями современных вычислительных машин. Целью Тьюринга было точно описать границу между тем, что вычислительная машина может делать, и тем, чего она не может. Полученные им результаты применимы не только к абстрактным *машинам Тьюринга,* но и к реальным современным компьютерам.
- В 1940-х и 1950-х годах немало исследователей занимались изучением простейших машин, которые сегодня называются "конечными автоматами". Такие автоматы вначале были предложены в качестве модели функционирования человеческого мозга. Однако вскоре они оказались весьма полезными для множества других целей, которые будут описаны ниже. А в конце 1950-х лингвист Н. Хомский занялся изучением формальных "грамматик". Не будучи машинами в точном смысле слова, грамматики, тем не менее, тесно связаны с абстрактными автоматами и служат основой некоторых важнейших составляющих программного обеспечения, в частности, компонентов компиляторов.

История конечных автоматов

- В 1969 году С. Кук развил результаты Тьюринга о вычислимости и невычислимости. Ему удалось разделить задачи на те, которые могут быть эффективно решены вычислительной машиной, и те, которые, в принципе, могут быть решены, но требуют для этого так много машинного времени, что компьютер оказывается бесполезным для решения практически всех экземпляров задачи, за исключением небольших. Задачи последнего класса называют "трудно разрешимыми" ("труднорешаемыми") или "NP-трудными". Даже при экспоненциальном росте быстродействия вычислительных машин ("закон Мура") весьма маловероятно, что удастся достигнуть значительных успехов в решении задач этого класса.
- Все эти теоретические построения непосредственно связаны с тем, чем занимаются ученые в области информатики сегодня. Некоторые из введенных понятий, такие, например, как конечные автоматы и некоторые типы формальных грамматик, используются при проектировании и создании важных компонентов программного обеспечения. Другие понятия, например, машина Тьюринга, помогают нам уяснить принципиальные возможности программного обеспечения. В частности, теория сложности вычислений позволяет определить, можно ли решить ту или иную задачу "в лоб" и написать соответствующую программу для ее решения, или же следует искать решение данной трудно разрешимой задачи в обход, используя приближенный, эвристический, или какой-либо другой метод.

Введение в теорию конечных автоматов

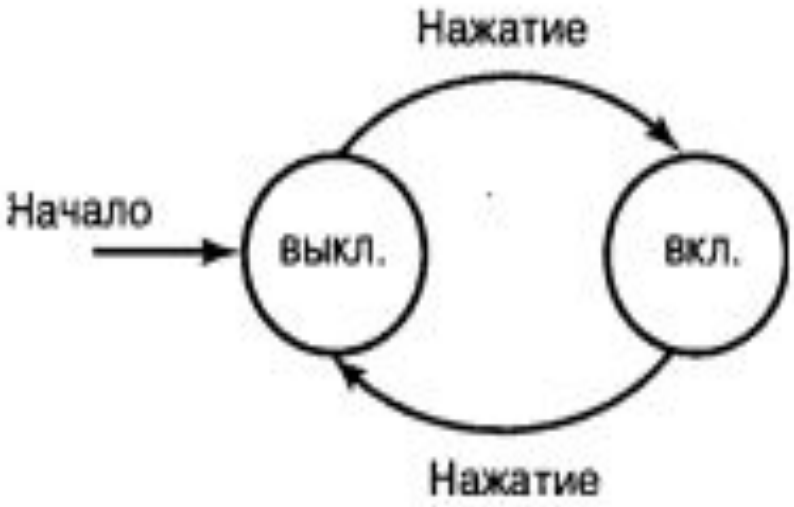
Основы теории формальных доказательств

Конечные автоматы являются моделью для многих компонентов аппаратного и программного обеспечения. Ниже будут рассмотрены примеры их использования, а сейчас просто перечислим наиболее важные из них:

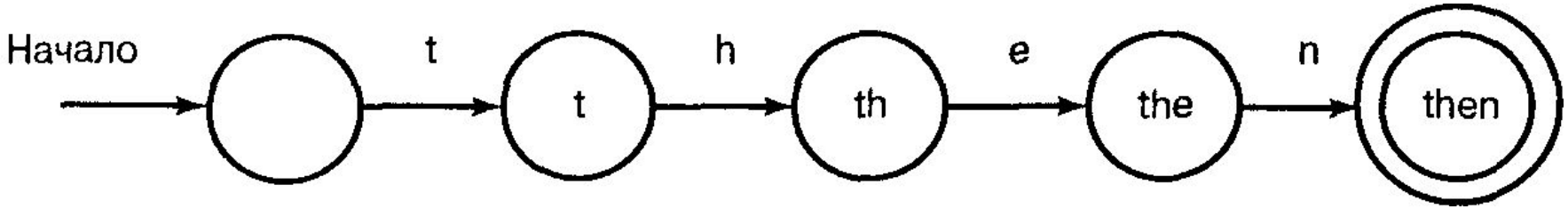
- Программное обеспечение, используемое для разработки и проверки цифровых схем.
- "Лексический анализатор" стандартного компилятора.
- Программное обеспечение для сканирования таких больших текстовых массивов с целью поиска заданных слов, фраз или других последовательностей символов (шаблонов).
- Программное обеспечение для проверки различного рода систем (протоколы связи или протоколы для защищенного обмена информацией), которые могут находиться в конечном числе различных состояний.

Простейший нетривиальный автомат

- Простейшим нетривиальным конечным автоматом является переключа тель "включено-выключено". Это устройство помнит свое текущее состояние, и от этого состояния зависит результат нажатия кнопки. Из состояния "выключено" нажатие кнопки переводит переключатель в состояние "включено", и наоборот.



Конечный автомат, распознающий слово



- Входным сигналам соответствуют буквы. Можно считать, что данный лексический анализатор всякий раз просматривает по одному символу компилируемой программы. Каждый следующий символ рассматривается как входной сигнал для данного автомата. Начальное состояние автомата соответствует пустой цепочке, и каждое состояние имеет переход по очередной букве слова then в состояние, соответствующее следующему префиксу. Состояние, обозначенное словом "then", достигается, когда по буквам введено все данное слово. Поскольку функция автомата заключается в распознавании слова then, то последнее состояние будем считать единственным допускающим.

Структурные представления автоматов

- Следующие системы записи не являются автоматными, но играют важную роль в теории автоматов и ее приложениях.
- *Грамматики. Они являются полезными моделями при проектировании программного обеспечения, обрабатывающего данные рекурсивной структуры. Наиболее известный пример — "синтаксический анализатор". Этот компонент компилятора работает с такими рекурсивно вложенными конструкциями в типичных языках программирования, как выражения: арифметические, условные и т.п.*
- *Регулярные выражения. Они также задают структуру данных, в частности, текстовых цепочек. Шаблоны описываемых ими цепочек представляют собой то же самое, что задают конечные автоматы. Стиль этих выражений сущ*ественно отличается от стиля, используемого в грамматиках.

Основные понятия теории автоматов

- Алфавиты и цепочки
- Языки
- Проблемы

Алфавит

- *Алфавитом* называют конечное непустое множество символов. Условимся обозначать алфавиты символом ∑. Наиболее часто используются следующие алфавиты.
  - ∑ = *{0,1} — бинарный или двоичный алфавит.*
  - ∑ = *{a, b,* …, z} — множество строчных букв английского алфавита.
- Множество ASCII-символов или множество всех печатных ASCII-символов.

Цепочки

Цепочка, или иногда *слово,* — это конечная последовательность символов некоторого алфавита. Например, 01101 — это цепочка в бинарном алфавите ∑ = {0, 1}. Цепочка 111 также является цепочкой в этом алфавите.

Пустая цепочка

Пустая цепочка— это цепочка, не содержащая ни одного символа. Эту цепочку, обозначаемую ε, можно рассматривать как цепочку в любом алфавите.

Длина цепочки

Часто оказывается удобным классифицировать цепочки по их *длине,* т.е. по числу позиций для символов в цепочке. Например, цепочка 01101 имеет длину 5. Обычно говорят, что длина цепочки — это "число символов" в ней. Это определение широко распространено, но не вполне корректно. Так, в цепочке 01101 всего 2 символа, но число *позиций* в ней — пять, поэтому она имеет длину 5. Все же следует иметь в виду, что часто пишут "число символов", имея в виду "число позиций".

Длину некоторой цепочки w обычно обозначают |w|. Например, |011| = 3, а | ε | = 0.

Степени алфавита

- Если ∑ — некоторый алфавит, то можно выразить множество всех цепочек определенной длины, состоящих из символов данного алфавита, используя знак степени. Определим Σ^k как множество всех цепочек длины *k,* состоящих из символов алфавита ∑.
- **Пример** 13. Заметим, что Σ^0 = { ε } независимо от алфавита ∑, т.е. ε — единственная цепочка длины 0.
- Если X = {0, 1}, то $\sum^1$ = {0, 1}, $\sum^2$= {00, 01, 10, 11}, $\sum^3$ = {000, 001, 010, 011, 100, 101, 110, 111} и так далее. Отметим, что между ∑ и $\sum^1$ есть небольшое различие. Дело в том, что ∑ есть алфавит, и его элементы 0 и 1 являются символами, а $\sum^1$ является множеством цепочек, и его элементы — это цепочки 0 и 1, каждая длиной 1. Разные обозначения для этих множеств не вводятся, полагая, что из контекста будет понятно, является {0, 1} или подобное ему множество алфавитом или же множеством цепочек.
- Как правило, строчными буквами из начальной части алфавита (или цифрами) обозначаются символы, а строчными буквами из конца алфавита, например *w, x, y* или z — цепочки.

- Множество всех цепочек над алфавитом ∑ принято обозначать ∑*. Так, например, {0,1}* = {в, 0, 1, 00, 01, 10, 11, 000,...}. По-другому это множество можно записать в виде

$$\Sigma^* = \Sigma^0 \bigcup \Sigma^1 \bigcup \Sigma^2 \bigcup \dots$$

- Иногда необходимо исключать из множества цепочек пустую цепочку. Множество всех непустых цепочек в алфавите ∑ обозначают через ∑⁺. Таким образом, имеют место следующие равенства

$$\Sigma^+ = \Sigma^1 \bigcup \Sigma^2 \bigcup \Sigma^3 \bigcup \dots$$

$$\Sigma^* = \Sigma^+ \bigcup \{\varepsilon\}$$

Конкатенация цепочек

- Пусть *x и y* — цепочки. Тогда *xy* обозначает их *конкатенацию* (соединение), т.е. цепочку, в которой последовательно записаны цепочки *x* и *y.* Более строго, если *x* — цепочка из i символов: $x = a_1 a_2 \cdots a_i$, а *y*— цепочка из *j* символов: $y = b_1 b_2 \cdots b_j$, то *xy*— это цепочка длины *i+j:* $xy = a_1 a_2 \cdots a_i b_1 b_2 \cdots b_j$
- **Пример 14.** Пусть *x* = 01101 и *y*= 110. Тогда xy = 01101110, а *yx*= 11001101. Для любой цепочки *w* справедливы равенства ε*w* = *w*ε = *w.* Таким образом, ε является *единицей (нейтральным элементом) относительно операции конкатенации,* поскольку результат ее конкатенации с любой цепочкой дает ту же самую цепочку.

Языки

- Множество цепочек, каждая из которых принадлежит ∑*, где ∑ — некоторый фиксированный алфавит, называют *языком.* В рамках такого определения часто задаются формальные языки. Если ∑ — алфавит, и $L \subseteq \sum^*$, то *L* — это *язык над* ∑, или *в* ∑. Отметим, что язык в ∑ не обязательно должен содержать цепочки, в которые входят все символы ∑. Поэтому, если известно, что *L* является языком в ∑, то можно утверждать, что L — это язык над любым алфавитом, содержащим ∑.
- Термин "язык" может показаться странным. Однако оправданием служит то, что **и** обычные языки можно рассматривать как множества цепочек. Возьмем в качестве примера английский язык, где набор всех литературных английских слов есть множество цепочек *в* алфавите (английских же букв). Еще один пример —любой другой язык программирования, в котором правильно написанные программы представляют собой подмножество множества всех возможных цепочек, а цепочки состоят из символов алфавита данного языка. Этот алфавит является подмножеством символов ASCII. Алфавиты для разных языков программирования могут быть различными, хотя обычно они состоят из прописных и строчных букв, цифр, знаков пунктуации и математических символов.

Примеры языков из теории автоматов

- – Язык, состоящий из всех цепочек, в которых n единиц следуют за n нулями для некоторого n > 0:{ ε, 01,0011,000111, ...}.
  – Множество цепочек, состоящих из 0 и 1 и содержащих поровну тех и других: {ε, 01, 10, 0011, 1001, ...}.
  – Множество двоичных записей простых чисел: {10, 11, 101, 111,1011, ...}.
  – ∑* — язык для любого алфавита ∑.
  – ∅ — пустой язык в любом алфавите.
  – {ε} — язык, содержащий одну лишь пустую цепочку. Он также является языком в любом алфавите. Заметим, что ∅ <> {ε}; первый не содержит вообще никаких цепочек, а второй состоит из одной цепочки.
- Единственное существенное ограничение для множеств, которые могут быть языками, состоит в том, что все алфавиты конечны. Таким образом, хотя языки и могут содержать бесконечное число цепочек, но эти цепочки должны быть составлены из символов некоторого фиксированного конечного алфавита.

Проблемы

- В теории автоматов *проблема* — это вопрос о том, является ли данная цепочка элементом определенного языка. Все, что называется "проблемой" в более широком смысле слова, может быть выражено в виде проблемы принадлежности некоторому языку. Точнее, если ∑ — некоторый алфавит, и *L* — язык в то проблема *L* формулируется следующим образом:

• Дана цепочка w из ∑*, требуется выяснить, принадлежит цепочка w языку *L,* или нет.

- Задачу проверки простоты данного числа можно выразить в терминах принадлежности языку L_p, который состоит из всех двоичных цепочек, выражающих простые числа. Таким образом, ответ "да" соответствует ситуации, когда данная цепочка из нулей и единиц является двоичным представлением простого числа. В противном случае ответом будет "нет". Для некоторых цепочек принять решение довольно просто. Например, цепочка 0011101 не может быть представлением простого числа по той причине, что двоичное представление всякого целого числа, за исключением 0, начинается с 1. Однако решение данной проблемы для цепочки 11101 не так очевидно и требует значительных затрат таких вычислительных ресурсов, как время и/или объем памяти.

Проблемы

- В приведенном определении "проблемы" есть одно слабое место. Дело в том, что обычно под проблемами подразумевают не вопросы разрешения (истинно нечто, или нет), а запросы на обработку или преобразование некоторых входных данных (желательно, наилучшим способом). Например, задача анализатора в компиляторе языка C — определить, принадлежит ли данная цепочка символов ASCII множеству L_C всех правильных программ на C — в точности соответствует нашему определению. Но в задачи анализатора входят также формирование дерева синтаксического анализа, заполнение таблицы имен и, возможно, другие действия.
- Тем не менее, определение "проблем" как языков выдержало проверку временем и позволяет нам успешно решать многие задачи, возникающие в теории сложности. В рамках этой теории определяются нижние границы сложности определенных задач. Особенно важны тут методы доказательства того, что определенные типы задач не могут быть решены за время, меньшее по количеству, чем экспонента от размера их входных данных. Оказывается, что в этом смысле задача, сформулированная в терминах теории языков (т.е. требующая ответа "да" или "нет"), так же трудна, как и исходная задача, требующая "найти решение".
- Таким образом, если можно доказать, что трудно определить, принадлежит ли данная цепочка множеству L_X всех правильных программ на языке X, следовательно, переводить программы с языка L_X объектный код, по крайней мере, не легче. Этот метод, позволяющий показать трудность одной задачи с использованием предполагаемого эффективного алгоритма ее решения для эффективного решения другой, заведомо сложной задачи, называется "сведением" второй задачи к первой.

Конечные автоматы – задача-пример

- Рассмотрим развернутый пример реальной проблемы, в решении которой важную роль играют конечные автоматы. Изучим протоколы, поддерживающие операции с "электронными деньгами"— файлами, которые клиент использует для платы за товары в Internet, а продавец получает с гарантией, что "деньги" — настоящие. Для этого продавец должен знать, что эти файлы не были подделаны или скопированы и отосланы продавцу, хотя клиент сохраняет копию этого файла и вновь использует ее для оплаты.
- Невозможность подделки файла должна быть гарантирована банком и стратегией шифрования. Таким образом, третий участник, банк, должен выпускать и шифровать "денежные" файлы так, чтобы исключить возможность подделки. Но у банка есть и другая важная задача: хранить в своей базе данных информацию о всех выданных им деньгах, годных к платежу. Это нужно для того, чтобы банк мог подтвердить, что полученный магазином файл представляет реальные деньги и может быть переведен на счет магазина. Мы не будем останавливаться на криптографическом аспекте проблемы, а также на том, каким образом банк может хранить и обрабатывать биллионы "электронных денежных счетов".
- Однако для того, чтобы использовать электронные деньги, необходимо составить протоколы, позволяющие производить с этими деньгами различные действия в зави симости от желания пользователя. Поскольку в монетарных системах всегда возможно мошенничество, нужно проверять правильность использования денег, какая бы система шифрования ни применялась. Иными словами, нужно гарантировать, что произойти могут только предусмотренные события. Это не позволит нечистому на руку пользователю украсть деньги у других или их "напечатать". В конце раздела приводится очень простой пример (плохого) протокола расчета электронными деньгами, моделируемого конечными автоматами, и показывается, как конструкции на основе автоматов можно использовать для проверки протоколов.

Основные участники задачи

Есть три участника: клиент, магазин и банк. Для простоты предположим, что есть всего один "денежный" файл ("деньги"). Клиент может принять решение передать этот файл магазину, который затем обменяет его в банке (точнее, потребует, чтобы банк взамен его выпустил новый файл, принадлежащий уже не клиенту, а магазину) и доставит товар клиенту. Кроме того, клиент имеет возможность отменить свой файл, т.е. попросить банк вернуть деньги на свой счет, причем они уже не могут быть израсходованы.

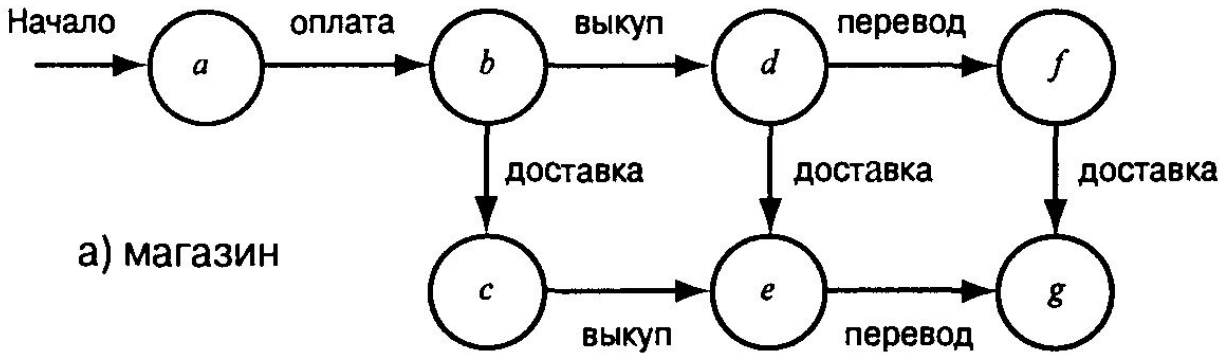
Взаимодействие трех участников ограничено, таким образом, следующими пятью событиями:

- Клиент может совершить *оплату (pay)* товара, т.е. переслать денежный файл в магазин.
- Клиент может выполнить *отмену (cancel)* денег. Они отправляются в банк вместе с сообщением о том, что их сумму следует добавить к банковскому счету клиента.
- Магазин может произвести *доставку (ship)* товара клиенту.
- Магазин может совершить *выкуп (redeem*) денег. Они отправляются в банк вместе с требованием передать их сумму магазину.
- Банк может выполнить *перевод (transfer*) денег, создав новый, надлежащим образом зашифрованный, файл и переслав его магазину.

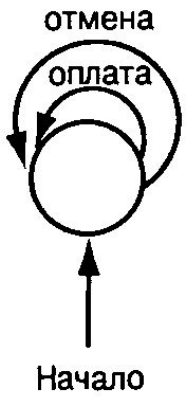
Протокол работы с деньгами

- Во избежание недоразумений участники должны вести себя осторожно. В нашем случае можно резонно предположить, что клиенту доверять нельзя. Клиент, в частности, может попытаться скопировать денежный файл и после этого уплатить им несколько раз или уплатить и отменить его одновременно, получая, таким образом, товар бесплатно.
- Банк должен вести себя ответственно, иначе он не банк. В частности, он должен проверять, не посылают ли на выкуп два разных магазина один и тот же денежный файл, и не допускать, чтобы одни и те же деньги и отменялись, и выкупались. Магазин тоже должен быть осторожен. Он, например, не должен доставлять товар, пока не убедится, что получил за него деньги, действительные к оплате.
- Протоколы такого типа можно представить в виде конечных автоматов. Каждое состояние представляет ситуацию, в которой может находиться один из участников. Таким образом, состояние "помнит", что одни важные события произошли, а другие — еще нет. Переходы между состояниями в рассматриваемом случае совершаются, когда происходит одно из пяти описанных выше событий. События будем считать "внешними" по отношению к автоматам, представляющим трех наших участников, несмотря на то, что каждый из них может инициировать одно или несколько из этих событий. Оказывается, важно не то, кому именно позволено вызывать эти события, а то, какие последовательности событий могут произойти.

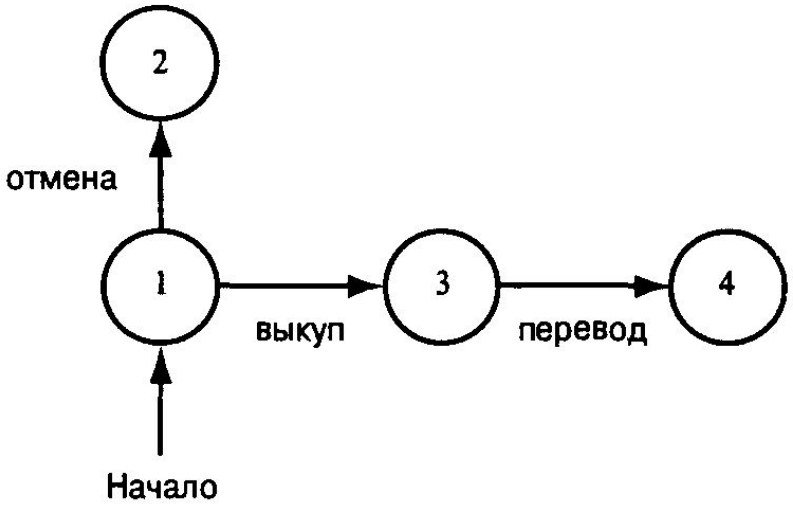
Конечные автоматы участников



а) магазин

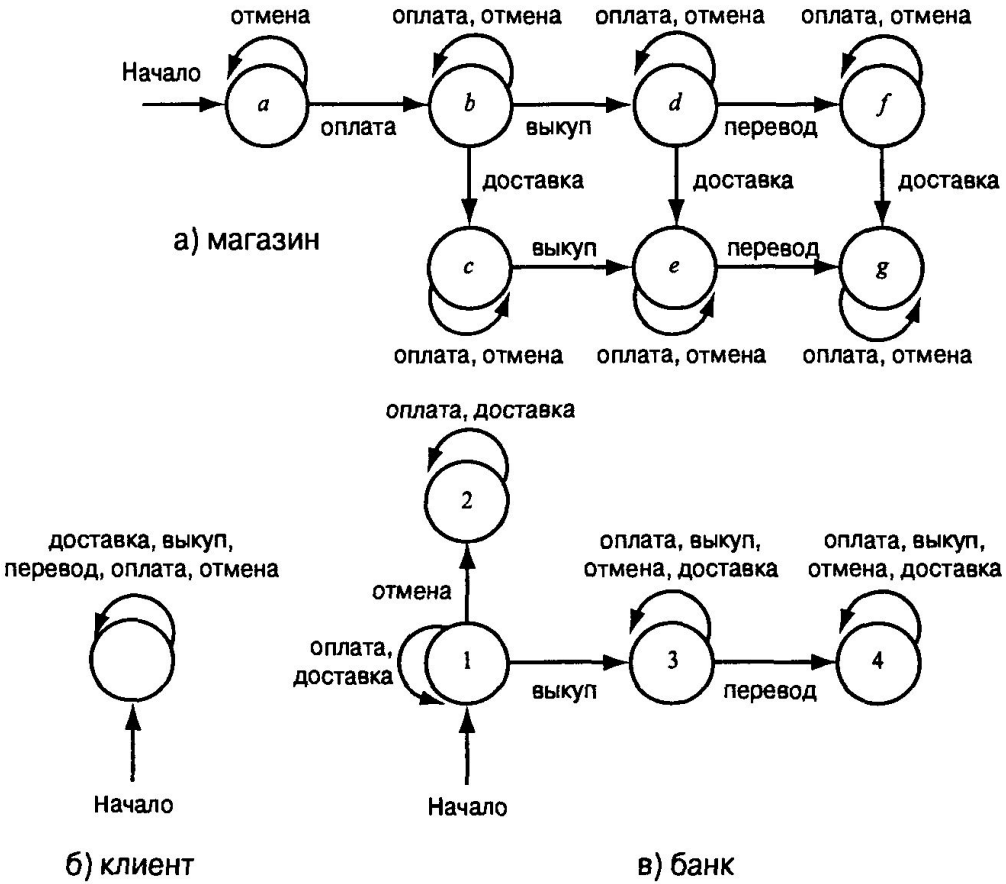


б) клиент



в) банк

Возможность игнорирования действий



Типы игнорируемых действий:

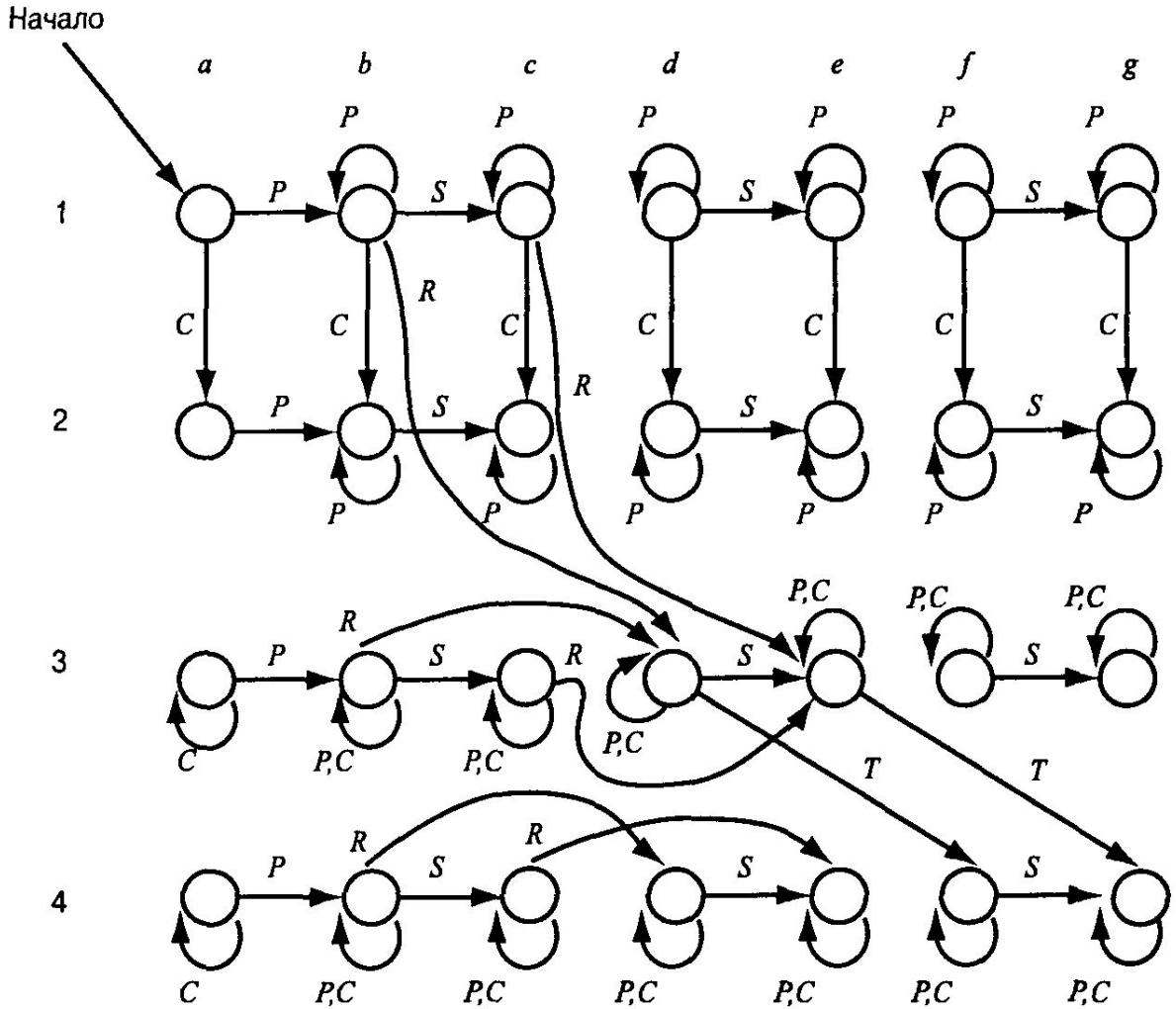
1. *Действия, не затрагивающие данного участника.*
2. *Действия, которые не следует допускать во избежание смерти автомата*

Правило построения дуг

- Чтобы правильно построить дуги в автомате-произведении, нужно проследить "параллельную" работу автоматов банка и магазина. Каждый из двух компонентов автомата-произведения совершает, в зависимости от входных действий, различные переходы. Важно отметить, что если, получив на вход некоторое действие, один из этих двух автоматов не может совершить переход ни в какое состояние, то автомат-произведение "умирает", поскольку также не может перейти ни в какое состояние.
- Придадим строгость правилу переходов из одного состояния в другое. Рассмотрим автомат-произведение в состоянии (i, *x).* Это состояние соответствует ситуации, когда банк находится в состоянии i, а магазин — в состоянии *x.* Пусть Z означает одно из входных действий. Мы смотрим, имеет ли автомат банка переход из состояния i с меткой Z. Предположим, что такой переход есть, и ведет он в состояние *j* (которое может совпадать с i, если банк, получив на вход Z, остается в том же состоянии). Затем, глядя на автомат магазина, мы выясняем, есть ли у него дуга с меткой Z, ведущая в некоторое состояние *y.* Если *j* и *y* существуют, то автомат-произведение содержит дугу из состояния *(i, x)* в состояние (j, *y)* с меткой Z. Если же либо состояния *j,* либо состояния *y* нет (по той причине, что банк или магазин для входного действия Z не имеет, соответственно, перехода из состояния *i* или x), то не существует и дуги с меткой Z, выходя щей из состояния (i, *x).*

Автомат, определяющий систему в целом

- Обы ствия
подс ении
так н *ения.*
Сост
пары ть
сост ание
банк
прои ацию,
когд
мага
мага нк —
семь
прои



Проверка протокола при помощи автомата

- Из автомата-произведения можно узнать кое-что интересное. Так, из начального состояния (1, a) — комбинации начальных состояний банка и магазина — можно попасть только в десять из всех 28 состояний.
- Однако реальной целью анализа протоколов, подобных данному, с помощью автоматов является ответ на вопрос: "возможна ли ошибка данного типа?". Простейший пример: нас может интересовать, возможно ли, что магазин доставит товар, а оплаты за него так и не получит, т.е. может ли автомат-произведение попасть в состояние, в котором магазин уже завершил доставку (и находится в одном из состояний в столбцах *c, e* или *g),* и при этом перехода, соответствующего входу T, никогда ранее не было и не будет.
- К примеру, в состоянии (3, *e)* товар уже доставлен, но переход в состояние (4, *g),* со ответствующий входу *T, в* конце концов произойдет. В терминах действий банка это означает, что если банк попал в состояние 3, то он уже получил запрос на *выкуп* и обработал его. Значит, он находился в состоянии 1 перед получением этого запроса, не получал требования об *отмене* и будет игнорировать его в будущем. Таким образом, в конце концов банк переведет деньги магазину.
- Однако в случае состояния (2, e) возникает проблема. Состояние достижимо, но единственная выходящая дуга ведет в него же. Это состояние соответствует ситуации, когда банк получил сообщение об *отмене* раньше, чем запрос на *выкуп.* Но магазин получил сообщение об *оплате,* т.е. наш пройдоха-клиент одни и те же деньги и потратил, и отменил. Магазин же, по глупости, доставил товар прежде, чем попытался выкупить деньги. Теперь, если магазин выполнит запрос на *выкуп,* то банк даже не подтвердит получение соответствующего сообщения, так как после *отмены,* находясь в состоянии 2, банк не будет обрабатывать запрос на *выкуп.*

Детерминированные конечные автоматы (ДКА)

- *Детерминированный конечный автомат* состоит из следующих компонентов.
  - Конечное множество *состояний,* обозначаемое обычно как *Q.*
  - Конечное множество *входных символов,* обозначаемое обычно как ∑
  - *Функция переходов, аргументами которой являются текущее состояние и входной символ, а значением — новое состояние. Функция переходов обычно обозначается как δ. Представляя нестрого автомат в виде графа, δ изображается отмеченными дугами, соединяющими состояния. Если q— состояние и a— входной символ, то δ (q, a) — это состояние p, для которого существует дуга, отмеченная символом a и ведущая из q в p.*
  - *Начальное состояние, одно из состояний в Q.*
  - Множество *заключительных,* или *допускающих,* состояний *F.* Множество *F* является подмножеством *Q.*
- В дальнейшем детерминированный конечный автомат часто обозначается как *ДКА.* Наиболее компактное представление ДКА— это список пяти вышеуказанных его компонентов. В доказательствах ДКА часто трактуется как пятерка $A=(Q, \sum, \delta, q_0, F)$, где *A* — имя ДКА, *Q* — множество состояний, ∑ — множество входных символов, δ — функция переходов, q_0— начальное состояние и *F*— множество допускающих состояний.

Обработка цепочек при помощи ДКА

- "Язык" ДКА— это множество всех его допустимых цепочек. Пусть $a_1a_2...a_n$ — последовательность входных символов. ДКА начинает работу в начальном состоянии q_0. Для того чтобы найти состояние, в которое *A* перейдет после обработки первого символа a_1, необходимо выполнить функцию переходов δ. Пусть, например, $S(q_0, a_1) = q_1$. Для следующего входного символа a_2 находим $\delta(q_1, a_2)$. Пусть это будет состояние q_2. Аналогично находятся и последующие состояния $q_3q_4..q_n$, где $\delta(q_{i-1}, a_i) = q_i$, для каждого *i*. Если q_n принадлежит множеству *F*, то входная последовательность $a_1a_2...a_n$ допускается, в противном случае она "отвергается" как недопустимая.

Пример обработки цепочек слайд 1/3

Пример. Определить формально ДКА, допускающий цепочки из 0 и 1, которые содержат в себе подцепочку 01. Этот язык можно описать следующим образом:

{w | *w* имеет вид x01y, где x и *y* — цепочки, состоящие только из 0 и 1}.

Можно дать и другое, эквивалентное описание, содержащее хну слева от вертикальной черты:

{x01y | x и y — некоторые цепочки, состоящие из 0 и 1}.

Примерами цепочек этого языка являются цепочки 01, 11010 и 1000111. В качестве примеров цепочек, *не принадлежащих* данному языку, можно взять цепочки ε, 0 и 111000.

Пример обработки цепочек слайд 2/3

- Что можно сказать об автомате, допускающем цепочки данного языка *L?* Во-первых, что алфавитом его входных символов является ∑ = {0, 1}. Во-вторых, имеется некоторое множество *Q* состояний этого автомата. Один из элементов этого множества, скажем, q_0, является его начальным состоянием. Для того чтобы решить, содержит ли входная по следовательность подцепочку 01, автомат *A* должен помнить следующие важные факты относительно прочитанных им входных данных:

1. Была ли прочитана последовательность 01? Если это так, то всякая читаемая далее последовательность допустима, т.е. с этого момента автомат будет находиться лишь в допускающих состояниях.
2. Если последовательность 01 еще не считана, то был ли на предыдущем шаге считан символ 0? Если это так, и на данном шаге читается символ 1, то последовательность 01 будет прочитана, и с этого момента автомат будет находиться только в допускающих состояниях.
3. Действительно ли последовательность 01 еще не прочитана, и на предыдущем шаге на вход либо ничего не подавалось (состояние начальное), либо был считан символ 1? В этом случае *A* не перейдет в допускающее состояние до тех пор, пока им не будут считаны символы 0 и сразу за ним 1.

Пример обработки цепочек слайд 3/3

- Каждое из этих условий можно представить как некоторое состояние. Условию (3) соответствует начальное состояние q_0. Конечно, находясь в самом начале процесса, нужно последовательно прочитать 0 и 1. Но если в состоянии q_0 читается 1, то это нисколько не приближает к ситуации, когда прочитана последовательность 01, поэтому нужно оставаться в состоянии q_0 . Таким образом, $\delta(q_0, 1) = q_0$. Однако если в состоянии q_0 читается 0, то мы попадаем в условие (2), т.е. 01 еще не прочитаны, но уже прочитан 0. Пусть q_2 обозначает ситуацию, описываемую условием (2). Переход из q_0 по символу 0 имеет вид $\delta(q_0, 0) = q_2$.
- Рассмотрим теперь переходы из состояния q_2. При чтении 0 мы попадаем в ситуацию, которая не лучше предыдущей, но и не хуже. 01 еще не прочитаны, но уже прочитан 0, и теперь ожидается 1. Эта ситуация описывается состоянием q_2, поэтому определим $\delta(q_2, 0) = q_2$. Если же в состоянии q_2 читается 1, то становится ясно, что во входной последовательности непосредственно за 0 следует 1. Таким образом, можно перейти в допускающее состояние, которое обозначается q_1 и соответствует приведенному выше условию (1), т.е. $\delta(q_2, 1) = q_1$
- Наконец, нужно построить переходы в состоянии q_1. В этом состоянии уже прочитана последовательность 01, и, независимо от дальнейших событий, мы будем находиться в этом же состоянии, т.е. $\delta(q_1, 0) = \delta(q_1, 1) = q_1$.
- Таким образом, $Q = \{q_0, q_1 q_2\}$. Ранее упоминалось, что q_0 — начальное, а q_1 — единственное допускающее состояние автомата, т.е. $F = \{q_1\}$. Итак, полное описание автомата Л, допускающего язык/, цепочек, содержащих 01 в качестве подцепочки, имеет вид $A = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\})$, где δ — функция, описанная выше.

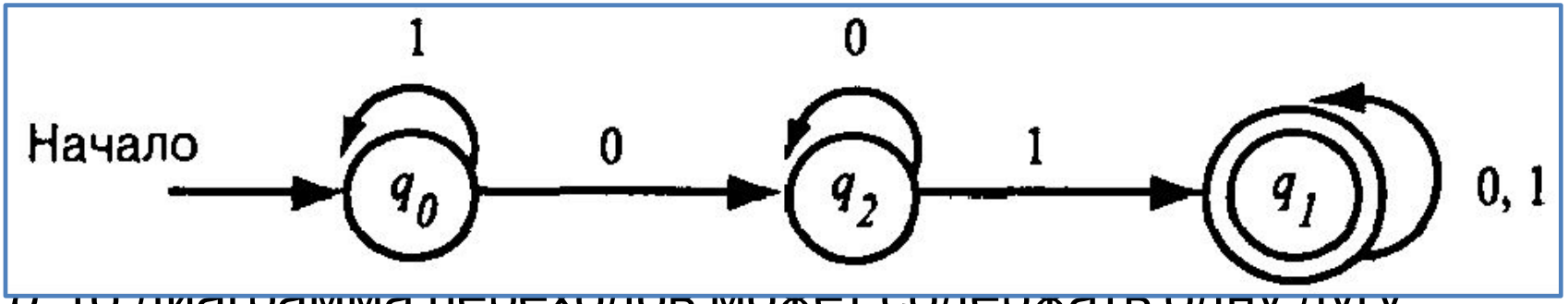
Способы представления ДКА

- *Диаграмма переходов, которая представляет собой граф.*
- *Таблица переходов, дающая табличное представление функции* δ. *Из нее очевидны состояния и входной алфавит.*

Диаграмма переходов

Диаграмма переходов для ДКА вида $A=(Q, \sum, \delta, q_0, F)$ есть граф, определяемый следующим образом:

а) всякому состоянию из Q соответствует некоторая вершина;



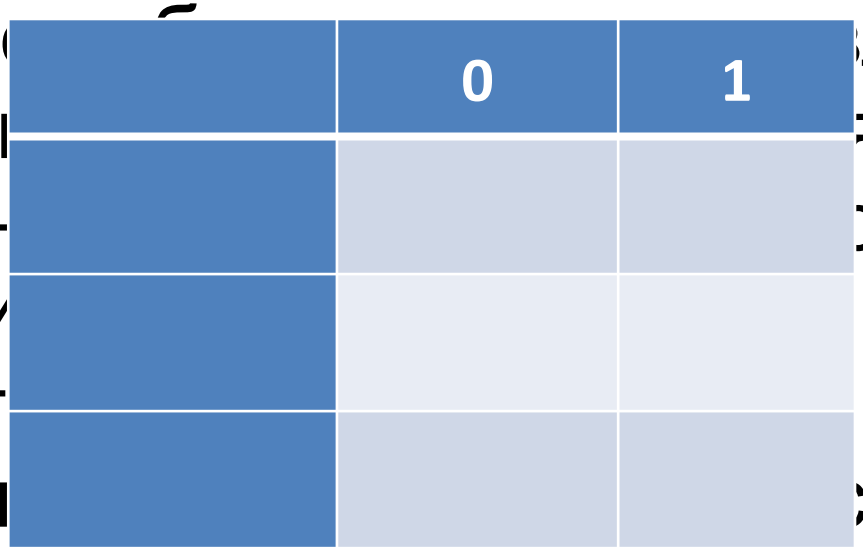
p, то диаграмма переходов может содержать одну дугу, отмеченную списком этих символов;

в) диаграмма содержит стрелку в начальное состояние, отмеченную как *Начало.* Эта стрелка не выходит ни из какого состояния;

г) вершины, соответствующие допускающим состояниям (состояниям из *F),* отмечаются двойным кружком. Состояния, не принадлежащие *F,* изображаются простым (одинарным) кружком.

Таблица переходов

- *Таблица переходов* представляет собой обычно ление функци я двум аргумен ствие одно значени соответ столбцы — входны сечении строки, соответствующей состоянию *q,* и столбца, соответствующего входному символу *a,* находится состояние δ (q, a).



Расширенная функция переходов

- Теперь дадим строгое определение языка ДКА. С этой целью определим *расширенную функцию переходов,* которая описывает ситуацию, при которой отслеживается произвольную последовательность входных символов, начиная с произвольного состояния.

Если $\boldsymbol{\delta}$ — функция переходов, то расширенную функцию, построенную по $\boldsymbol{\delta}$, обозначим $\hat{\delta}$. Расширенная функция переходов ставит в соответствие состоянию *q* и цепочке *w* состояние *p,* в которое автомат попадает из состояния *q,* обработав входную последовательность *w.* Определим $\hat{\delta}$ индукцией по длине входной цепочки следующим образом:

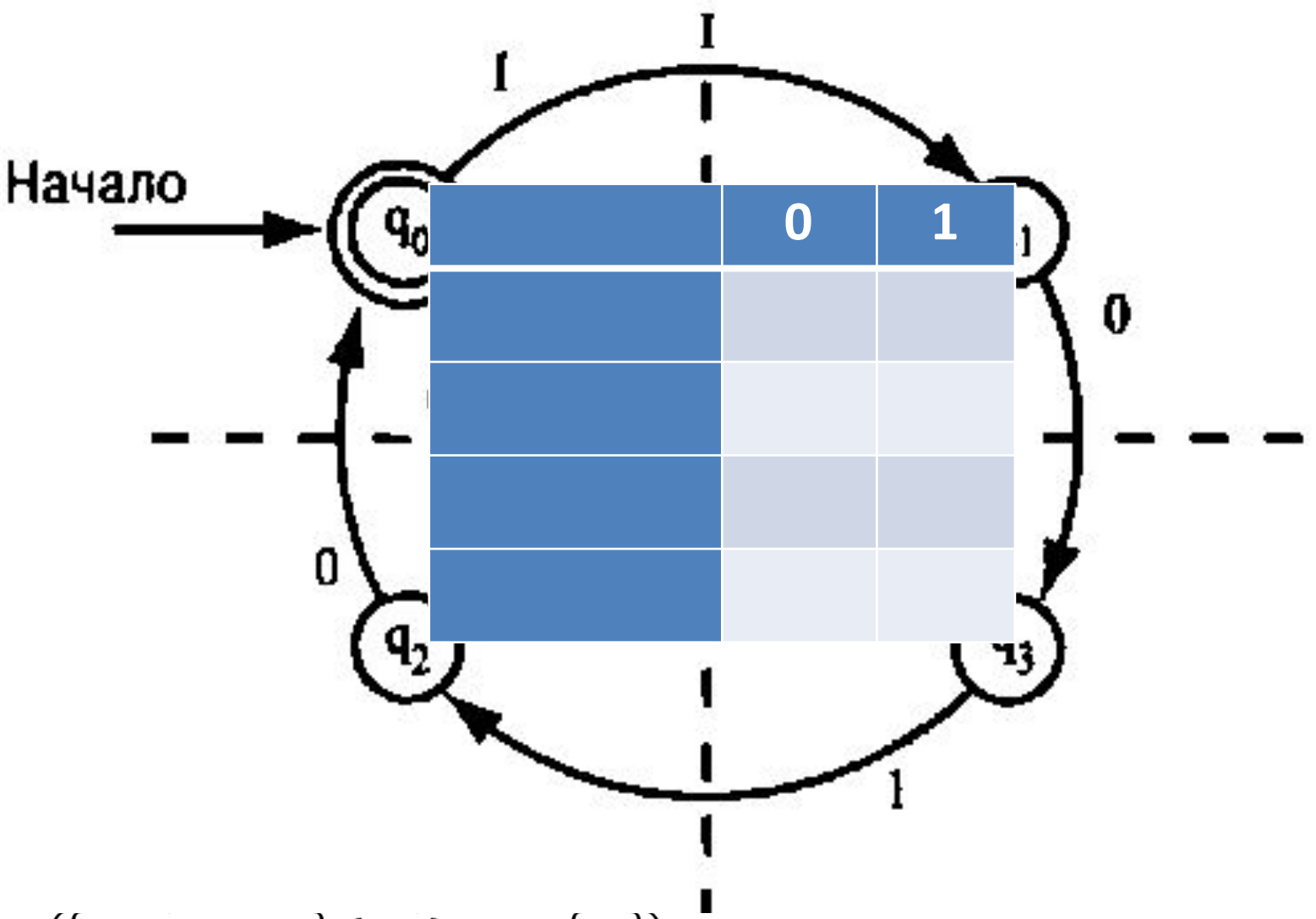
Базис. $\hat{\delta}(q, \varepsilon) = q$, т.е., находясь в состоянии *q* и не читая вход, мы остаемся в состоянии *q.*

Индукция. Пусть w — цепочка вида *xa,* т.е. *a* — последний символ в цепочке, ax — цепочка, состоящая из всех символов цепочки *w,* за исключением последнего. Например, *w* = 1101 разбивается на x: = 110 и *a* = 1. Тогда

$$\hat{\delta}(q, \omega) = \delta\big(\hat{\delta}(q, x), a\big) \quad (1)$$

Выражение (1) может показаться довольно громоздким, но его идея проста. Для того чтобы найти $\hat{\delta}(q, \omega)$, вначале находим $\hat{\delta}(q, x)$ — состояние, в которое автомат попадает, обработав все символы цепочки *w,* кроме последнего. Предположим, что это состояние *p,* т.е. $\hat{\delta}(q, x) = p$. Тогда $\hat{\delta}(q, \omega)$ - это состояние, в которое автомат переходит из *p* при чтении *a* — последнего символа *w.* Таким образом, $\hat{\delta}(q, \omega) = \delta(p, a)$.

Пример



	0	1

$A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$

Построение расширенной функции переходов

Допустим, на вход подается цепочка 110101. Она содержит четное число 0 и 1, поэтому принадлежит данному языку. Таким образом, ожидается, что $\hat{\delta}(q_0, 110101) = q_0$, так как q_0 — единственное допускающее состояние. Проверим это утверждение.

Для проверки требуется найти $\hat{\delta}(q_0, \omega)$ для всех постепенно нарастающих, начиная с ε, префиксов ω цепочки 110101. Результат этих вычислений выглядит следующим образом.

$$\hat{\delta}(q_0, \varepsilon) = q_0$$
$$\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \varepsilon), 1) = \delta(q_0, 1) = q_1$$
$$\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$$
$$\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_2$$
$$\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_3$$
$$\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = q_1$$
$$\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = q_0$$

Язык ДКА

- **Язык ДКА**
- Теперь можно определить *язык* ДКА вида $A = (Q, \Sigma, \delta, q_0, F)$. Этот язык обозначается *L(A)* и определяется как
- $L(A) = \{\omega | \hat{\delta}(q_0, \omega) \text{ принадлежит } F\}$.
- Таким образом, язык — множество цепочек, приводящих автомат из состояния q_0 в одно из допускающих состояний. Если язык *L* есть *L(A)* для некоторого ДКА *A,* то говорят, что *L* является *регулярным языком.*

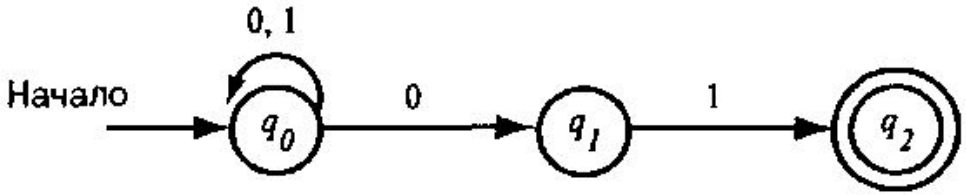
Недетерминированный конечный автомат (НКА)

- "Недетерминированный" конечный автомат, или НКА (NFA — Nondeterministic Finite Automaton), обладает свойством находиться в нескольких состояниях одновременно. Эту особенность часто представляют как свойство автомата делать "догадки" относительно его входных данных. Так, если автомат используется для поиска определенных цепочек символов (например, ключевых слов) в текстовой строке большой длины, то в начале поиска полезно "догадаться", что автомат находится в начале одной из этих цепочек, а затем использовать некоторую последовательность состояний для простой проверки того, что символ за символом появляется данная цепочка.

Неформальное описание НКА

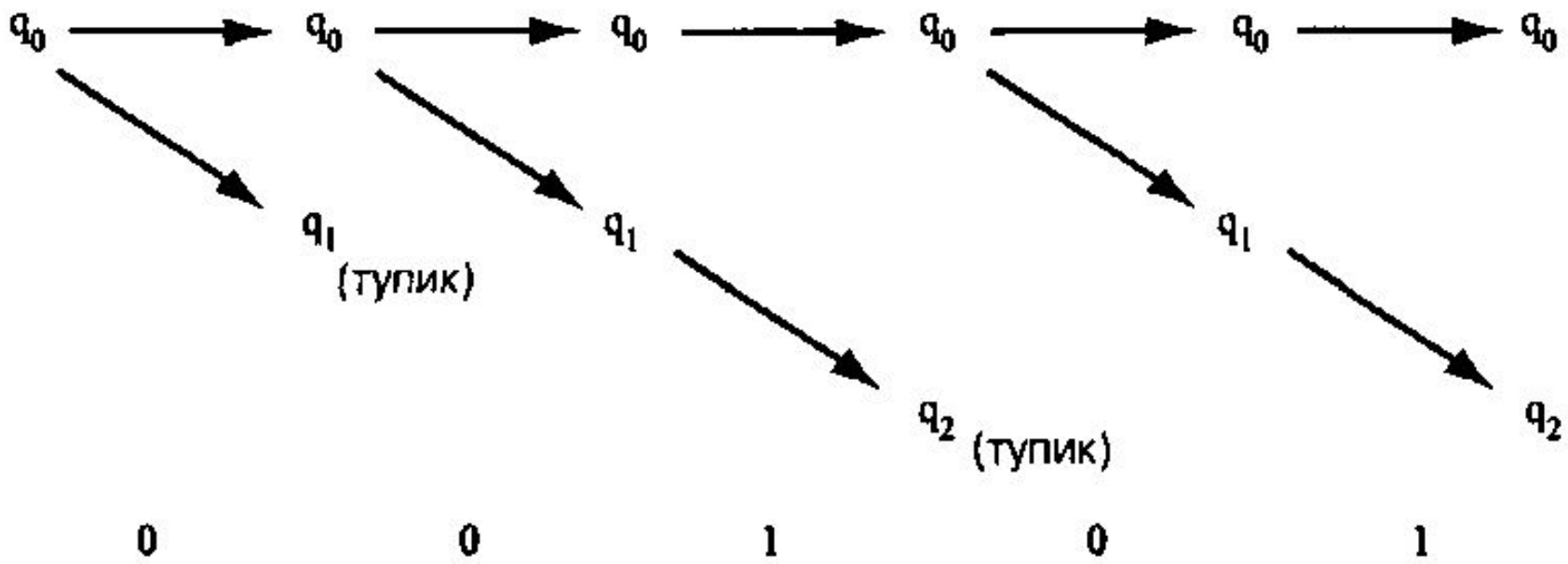
- НКА, как и ДКА, имеют конечное множество состояний, конечное множество входных символов, одно начальное состояние и множество допускающих состояний. Есть также функция переходов, которая, как обычно, обозначается через δ. Различие между ДКА и НКА состоит в типе функции δ. В НКА δ есть функция, аргументами которой являются состояние и входной символ (как и в ДКА), а значением — множество, состоящее из нуля, одного или нескольких состояний (а не одно состояние, как в ДКА).

Пример НКА



- На слайде изображен недетерминированный конечный автомат, допускающий те и только те цепочки из 0 и 1, которые оканчиваются на 01. Начальным является состояние q_0 и можно считать, что автомат находится в этом состоянии (а также, возможно, и в других состояниях) до тех пор, пока не "догадается", что на входе началась замыкающая подцепочка 01. Всегда существует вероятность того, что следующий символ не является начальным для замыкающей подцепочки 01, даже если это символ 0. Поэтому состояние q_0 может иметь переходы в себя как по 1, так и по 0.

Обработка цепочек НКА



Определение НКА

Теперь определим формально понятия, связанные с недетерминированными конечными автоматами, выделив по ходу различия между ДКА и НКА. Структура НКА в основном повторяет структуру ДКА: $A = (Q, \Sigma, \delta, q_0, F)$.

Эти обозначения имеют следующий смысл:

Q - конечное множество *состояний*

Σ - конечное множество *входных символов*

δ – функция переходов - это функция, аргументами которой являются состояние из *Q* и входной символ из Σ, а значением — некоторое подмножество множества *Q.* Заметим, что единственное различие между НКА и ДКА состоит в типе значений функции 8. Для НКА — это множество состояний, а для ДКА — одиночное состояние.

q_0 – начальное состояние – один из элементов Q

F – множество заключительных или допускающих состояний – подмножество Q

Таблица переходов для НКА

	0	1
	∅	
	∅	∅

Расширенная функция переходов НКА

Для НКА, так же, как и для ДКА, нам потребуется расширить функцию δ до функции $\hat{\delta}$, аргументами которой являются состояние *q* и цепочка входных символов *w*, а значением — множество состояний, в которые НКА попадает из состояния *q*, обработав цепочку *w*.

По сути, $\hat{\delta}(q,w)$ есть столбец состояний, которые получаются при чтении цепочки *w*, при условии, что *q* — единственное состояние в первом столбце. Так, выше показано, что $\hat{\boldsymbol{\delta}}(\boldsymbol{q_0}, \boldsymbol{001}) = \{\boldsymbol{q_0}, \boldsymbol{q_2}\}$. Формально $\hat{\delta}$ для НКА определяется следующим образом:

Базис $\hat{\boldsymbol{\delta}}(\boldsymbol{q}, \boldsymbol{\varepsilon}) = \{\boldsymbol{q}\}$:, т.е., не прочитав никаких входных символов, НКА находится только в том состоянии, в котором начинал.

Индукция. Предположим, цепочка ***w*** имеет вид ***w=xa,*** где ***a*** — последний символ цепочки *w*, *ax* — ее оставшаяся часть. Кроме того, предположим, что $\hat{\boldsymbol{\delta}}(\boldsymbol{q}, \boldsymbol{x}) = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \cdots, \boldsymbol{p_k}\}$.

Пусть $\bigcup_{i=1}^{k} \boldsymbol{\delta}(\boldsymbol{p_i}, \boldsymbol{a}) = \{\boldsymbol{r_1}, \boldsymbol{r_2}, \cdots, \boldsymbol{r_m}\}$

Тогда $\boldsymbol{\delta}(\boldsymbol{q}, \boldsymbol{w}) = \{\boldsymbol{r_1}, \boldsymbol{r_2}, \cdots, \boldsymbol{r_m}\}$.

Говоря менее формально, для того, чтобы найти $\hat{\boldsymbol{\delta}}(\boldsymbol{q}, \boldsymbol{w})$, нужно найти $\boldsymbol{\delta}(\boldsymbol{q}, \boldsymbol{x})$, а затем совершить из всех полученных состояний все переходы по символу *a*.

Пример. Используем $\hat{\boldsymbol{\delta}}$ для описания того, как НКА обрабатывает цепочку 00101.

$$\hat{\boldsymbol{\delta}}(\boldsymbol{q_0}, \boldsymbol{\varepsilon}) = \{\boldsymbol{q_0}\}$$

$$\hat{\boldsymbol{\delta}}(\boldsymbol{q_0}, \boldsymbol{0}) = \boldsymbol{\delta}(\boldsymbol{q_0}, \boldsymbol{0}) = \{\boldsymbol{q_0}, \boldsymbol{q_1}\}$$

$$\hat{\boldsymbol{\delta}}(\boldsymbol{q_0}, \boldsymbol{00}) = \boldsymbol{\delta}(\boldsymbol{q_0}, \boldsymbol{0}) \bigcup \boldsymbol{\delta}(\boldsymbol{q_1}, \boldsymbol{0}) = \{\boldsymbol{q_0}, \boldsymbol{q_1}\} \bigcup \emptyset = \{\boldsymbol{q_0}, \boldsymbol{q_1}\}$$

$$\hat{\boldsymbol{\delta}}(\boldsymbol{q_0}, \boldsymbol{001}) = \boldsymbol{\delta}(\boldsymbol{q_0}, \boldsymbol{1}) \bigcup \boldsymbol{\delta}(\boldsymbol{q_1}, \boldsymbol{1}) = \{\boldsymbol{q_0}\} \bigcup \{\boldsymbol{q_2}\} = \{\boldsymbol{q_0}, \boldsymbol{q_2}\}$$

$$\hat{\boldsymbol{\delta}}(\boldsymbol{q_0}, \boldsymbol{0010}) = \boldsymbol{\delta}(\boldsymbol{q_0}, \boldsymbol{0}) \bigcup \boldsymbol{\delta}(\boldsymbol{q_2}, \boldsymbol{0}) = \{\boldsymbol{q_0}, \boldsymbol{q_1}\} \bigcup \emptyset = \{\boldsymbol{q_0}, \boldsymbol{q_1}\}$$

$$\hat{\boldsymbol{\delta}}(\boldsymbol{q_0}, \boldsymbol{00101}) = \boldsymbol{\delta}(\boldsymbol{q_0}, \boldsymbol{1}) \bigcup \boldsymbol{\delta}(\boldsymbol{q_1}, \boldsymbol{1}) = \{\boldsymbol{q_0}\} \bigcup \{\boldsymbol{q_2}\} = \{\boldsymbol{q_0}, \boldsymbol{q_2}\}$$

Язык НКА

- По описанию НКА допускает цепочку *w,* если в процессе чтения этой цепочки символов можно выбрать хотя бы одну последовательность переходов в следующие состояния так, чтобы прийти из начального состояния в одно из допускающих.
- Тот факт, что при другом выборе последовательности переходов по символам цепочки w автомат может попасть в недопускающее состояние или вообще не попасть ни в какое (т.е. последовательность состояний "умирает"), отнюдь не означает, что w не является допустимой для НКА в целом. Формально, если $A = (Q, \Sigma, \delta, q_0, F)$ некоторый НКА, то $L(A) = \{\omega | \hat{\partial}(q_0, \omega) \cap F = \emptyset\}$.
- Таким образом, *L(A)* есть множество цепочек *w* из Σ^*, для которых среди состояний $\hat{\delta}(q_0, \omega)$ есть хотя бы одно допускающее.

Эквивалентность ДКА и НКА

- Для многих языков, в частности, для языка цепочек, оканчивающихся на 01, построить соответствующий НКА гораздо легче, чем ДКА. Несмотря на это, всякий язык, который описывается некоторым НКА, можно также описать и некоторым ДКА. Кроме того, этот ДКА имеет, как правило, примерно столько же состояний, сколько и НКА, несмотря на то, что часто содержит больше переходов. Однако в худшем случае наименьший ДКА может содержать 2^n состояний, в то время как НКА для того же самого языка имеет всего n состояний.
- В доказательстве того, что ДКА обладают всеми возможностями НКА, используется одна важная конструкция, называемая *конструкцией подмножеств,* поскольку включает построение всех подмножеств множества состояний НКА. Вообще, в доказательствах утверждений об автоматах часто по одному автомату строится другой. Конструкция подмножеств важна в качестве примера того, как один автомат описывается в терминах состояний и переходов другого автомата без знания специфики последнего.

Конструкция подмножеств

Построение подмножеств начинается, исходя из НКА $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Целью является описание ДКА $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$, у которого *L(N) = L(D).* Отметим, что входные алфавиты этих двух автоматов совпадают, а начальное состояние *D* есть множество, содержащее только начальное состояние *N.* Остальные компоненты *D* строятся следующим образом:

- Q_D есть *булеан* множества Q_N. Отметим, что если Q_N содержит n состояний, то Q_D будет содержать уже 2^n состояний. Однако часто не все они достижимы из начального состояния автомата *D.*
- F_D есть множество подмножеств S множества Q_N, для которых $S \cap F_N \neq \emptyset$, т.е. F_D состоит из всех множеств состояний *N,* содержащих хотя бы одно допускающее состояние *N.*
- Для каждого множества $S \subseteq Q_N$ и каждого входного символа *a* из Σ $\delta_D(S, a) = \bigcup \delta_N(p, a)$.
- Таким образом, для того, чтобы найти $\delta_D(S, a)$, мы рассматриваем все состояния *p* из *S,* ищем те состояния *N,* в которые можно попасть из состояния *p* по символу a, а затем берем объединение множеств найденных состояний по всем состояниям *p.*

Пример конструкции подмножеств

- Пусть N— автомат, допускающий цепочки, которые оканчиваются на 01. Поскольку множество состояний *N* есть $\{q_0, q_1, q_2\}$, то конструкция подмножеств дает ДКА с $2^3 = 8$ состояниями, отвечающими всем подмножествам, составленным из этих трех состояний. На слайде приведена таблица переходов для полученных восьми состояний. Объясним вкратце, как были получены элементы этой таблицы.

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\varnothing$	$\{q_2\}$
$^*\{q_2\}$	$\varnothing$	$\varnothing$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$^*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$^*\{q_1, q_2\}$	$\varnothing$	$\{q_2\}$
$^*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Пример конструкции подмножеств

- Показанная на предыдущем слайде таблица, элементами которой являются множества, соответствует детерминированному конечному автомату, поскольку состояния построенного ДКА *сами являются множествами.* Для ясности можно переобозначить состояния. Например, 0 обозначить как *A*, $\{q_0\}$ — как *B* и т.д. Таблица переходов для ДКА на данном слайде определяет в точности тот же автомат, что и таблица выше, и из ее вида понятно, что элементами таблицы являются одиночные состояния ДКА.

	0	1
A	*A*	*A*
→ *B*	*E*	*B*
C	*A*	*D*
**D*	*A*	*A*
E	*E*	*F*
**F*	*E*	*B*
**G*	*A*	*D*
**H*	*E*	*F*

Пример конструкции подмножеств

Начиная в состоянии *B,* из всех восьми состояний можно попасть только в состояния *B, E* и *F.* Остальные пять состояний из начального недостижимы, и поэтому их можно исключить из таблицы. Часто можно избежать построения элементов таблицы переходов для всех подмножеств, что требует экспоненциального времени. Для этого выполняется следующее "ленивое вычисление" подмножеств:

- Базис. Точно известно, что одноэлементное множество, состоящее из начального состояния M, является достижимым.
- Индукция. Предположим, установлено, что множество состояний *S* является достижимым. Тогда для каждого входного символа *a* нужно найти множество состояний $\delta_D(S, a)$. Найденные таким образом множества состояний также будут достижимы.

Пример конструкции подмножеств

Нам известно, что $\{q_0\}$ есть одно из состояний ДКА *D.* Находим, что $\delta_D(\{q_0\},0)=\{q_0,q_1\}$ и $\delta_D(\{q_0\},1)=\{q_0\}$. Оба эти факта следуют из диаграммы переходов для автомата; как видно, по символу 0 есть переходы из q_0 в q_0 и q_1 а по символу 1 — только в q_0. Таким образом, получена вторая строка таблицы переходов ДКА.

Одно из найденных множеств, $\{q_0\}$, уже рассматривалось. Но второе, $\{q_0,q_1\}$, — новое, и переходы для него нужно найти: $\delta_D(\{q_0,q_1\},0)=\{q_0,q_1\}$ и $\delta_D(\{q_0,q_1\},1)=\{q_0,q_2\}$. Проследить последние вычисления можно, например, так:

$$\delta_D(\{q_0,q_1\},1)=\delta_N(q_0,1)\bigcup\delta_N(q_1,1)=\{q_0\}\bigcup\{q_2\}=\{q_0,q_2\}$$

Теперь получена пятая строка таблицы и одно новое состояние $\{q_0, q_2\}$. Аналогичные вычисления показывают, что

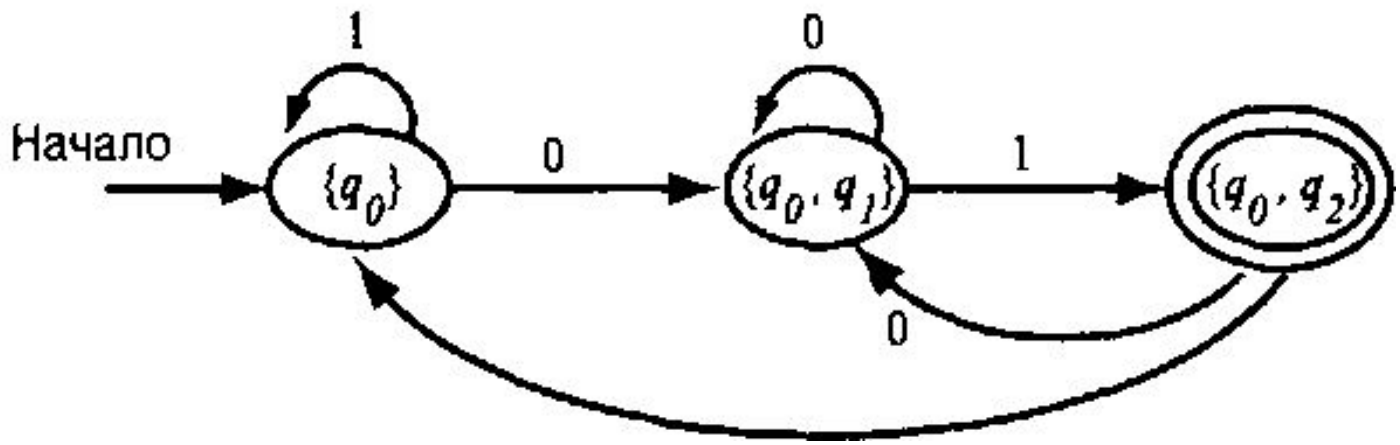
$$\delta_D(\{q_0,q_2\},0)=\delta_N(q_0,0)\bigcup\delta_N(q_2,0)=\{q_0,q_1\}\bigcup\{\emptyset\}=\{q_0,q_1\}$$

$$\delta_D(\{q_0,q_2\},1)=\delta_N(q_0,1)\bigcup\delta_N(q_{21},1)=\{q_0\}\bigcup\{\emptyset\}=\{q_0\}$$

Эти вычисления дают шестую строку таблицы, но при этом не получено ни одного нового множества состояний.

Пример конструкции подмножеств

- Итак, конструкция подмножеств сошлась; известны все допустимые состояния и со ответствующие им переходы. Полностью ДКА показан ниже. Заметим, что он имеет лишь три состояния. Это число случайно оказалось равным числу состояний НКА, по которому строился этот ДКА. Но ДКА ниже имеет шесть переходов, а НКА автомат — лишь четыре



Теоремы конструкции подмножеств

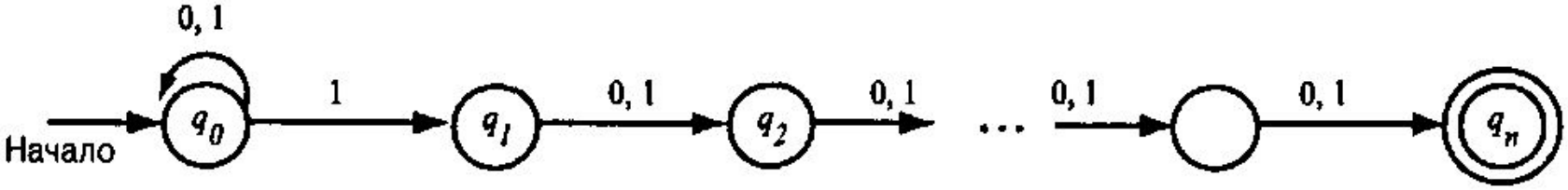
- Теорема. Если ДКА $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ построен по $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ посредством конструкции подмножеств, то ***L(D) = L(N).***
- Теорема. Язык ***L*** допустим некоторым ДКА тогда и только тогда, когда он допускается некоторым НКА.

Плохая конструкция подмножеств слайд 1

- В рассмотренном выше примере число состояний ДКА и число состояний НКА одинаково. Как уже было сказано, ситуация, когда количества состояний НКА и построенного по нему ДКА примерно одинаковы, на практике встречается довольно часто. Однако при переходе от НКА к ДКА возможен и экспоненциальный рост числа состояний, т.е. все 2^n состояний, которые могут быть построены по НКА, имеющему n состояний, оказываются достижимыми. В следующем примере данный предел немного не будет достигнут, но будет ясно, каким образом наименьший ДКА, построенный по НКА с n + 1 состояниями, может иметь 2^n состояний.

Плохая конструкция подмножеств слайд 2

- Рассмотрим НКА, показанный ниже. *L(N)* есть множество всех цепочек из 0 и 1, у которых n-м символом с конца является 1. Интуиция подсказывает, что ДКА *D,* допускающий данный язык, должен помнить последние n прочитанных символов. Поскольку всего имеется 2^n последовательностей, состоящих из последних n символов, то при числе состояний *D* меньше 2^n нашлось бы состояние *q,* в которое *D* попадает по прочтении двух разных последовательностей, скажем, $a_1 a_2 \cdots a_n$ и $b_1 b_2 \cdots b_n$.
- Поскольку последовательности различны, они должны различаться символом в некоторой позиции, например, $a_i \neq b_i$. Предположим (с точностью до симметрии), что $a_i = 1$ и $b_i = 0$. Если i=1, то состояние *q* должно быть одновременно и допускающим, и недопускающим, поскольку последовательность $a_1 a_2 \cdots a_n$ *(n*-й символ с конца есть 1), а $b_1 b_2 \cdots b_n$ — нет. Если же *i* > 1, то рассмотрим состояние *p,* в которое *D* попадает из состояния *q* по прочтении цепочки из i-1 нулей. Тогда *p* вновь должно одновременно и быть, и не быть допускающим, так как цепочка $a_i a_{i+1} \cdots a_n 00 \cdots 0$ допустима, а $b_i b_{i+1} \cdots b_n 00 \cdots 0$ - нет.



Плохая конструкция подмножеств слайд 3

- Теперь рассмотрим, как работает НКА *N*, показанный выше. Существует состояние q_0, в котором этот НКА находится всегда, независимо от входных символов. Если следующий символ — 1, то *N* может "догадаться", что эта 1 есть *n*-й символ с конца. Поэтому одновременно с переходом в q_0 НКА *N* переходит в состояние q_1. Из состояния q_x по любому символу *N* переходит в состояние q_2. Следующий символ переводит *N* в состояние q_3 и так далее, пока *n* - 1 последующий символ не переведет *N* в допускающее состояние q_n. Формальные утверждения о работе состояний N выглядят следующим образом:
  - *N* находится в состоянии q_0 по прочтении любой входной последовательности ω.
  - *N* находится в состоянии q_i (i = 1,2, ..., *n)* по прочтении входной последовательности ω тогда и только тогда, когда i-й символ с конца ω есть 1, т.е. ω имеет вид $x1a_1a_2\cdots a_{i-1}$, где a_j — входные символы.

Пример использования автомата: поиск цепочек в тексте

В век Internet и электронных библиотек с непрерывным доступом обычной является следующая проблема. Задано некоторое множество слов, и требуется найти все документы, в которых содержится одно (или все) из них. Популярным примером такого процесса служит работа поисковой машины, которая использует специальную технологию поиска, называемую *обращенными индексами* (inverted indexes). Для каждого слова, встречаю щегося в Internet (а их около 100,000,000), хранится список адресов всех мест, где оно встречается. Машины с очень большим объемом оперативной памяти обеспечивают постоянный доступ к наиболее востребованным из этих списков, позволяя многим людям одновременно осуществлять поиск документов.

В методе обращенных индексов конечные автоматы не используются, но этот метод требует значительных затрат времени для копирования содержимого сети и переписывания индексов. Существует множество смежных приложений, в которых применить технику обращенных индексов нельзя, зато можно с успехом использовать методы на основе автоматов. Те приложения, для которых подходит технология поиска на основе автоматов, имеют следующие отличительные особенности:

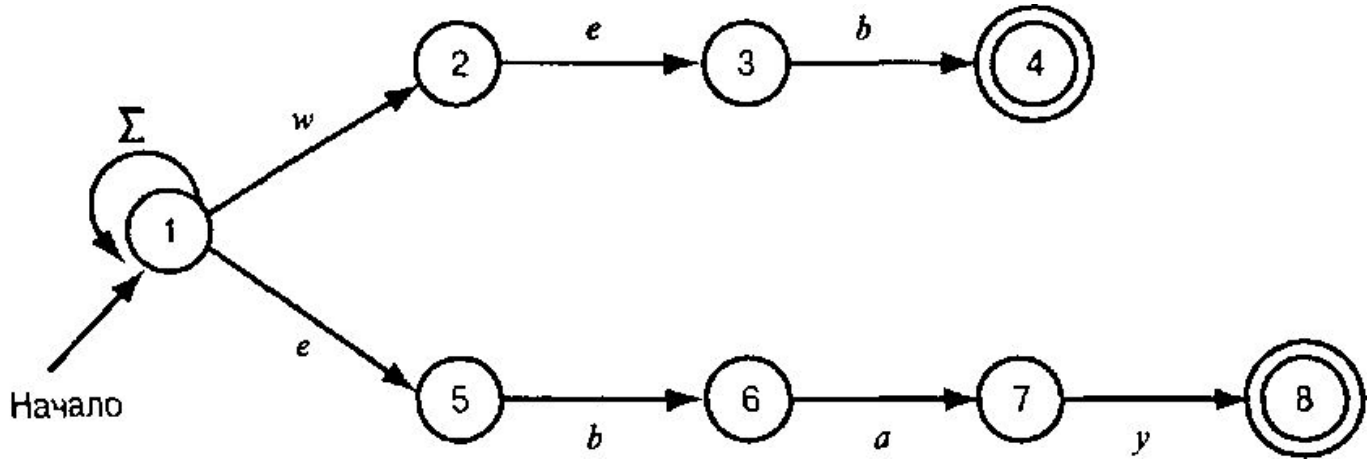
- Содержимое хранилища текста, в котором производится поиск, быстро меняется.
- Документы, поиск которых осуществляется, не могут быть каталогизированы или генерируются «на лету».

Недетерминированный поисковый автомат

- Пусть дано множество слов, которые в дальнейшем будут называться *ключевыми словами,* и нужно отыскать в тексте места, где встречается любое из этих слов. В подобных приложениях бывает полезно построить недетерминированный конечный автомат, который, попадая в одно из допускающих состояний, дает знать, что встретил одно из ключевых слов. Текст документа, символ за символом, подается на вход НКА, который затем распознает в нем ключевые слова. Существует простая форма НКА, распознающего множество ключевых слов.
  - Есть начальное состояние с переходом в себя по каждому входному символу, например, печатному символу ASCII при просмотре текста. Начальное состояние можно представлять себе, как "угадывание" того, что ни одно из ключевых слов еще не началось, даже если несколько букв одного из ключевых слов уже прочитано.
  - Для каждого ключевого слова $a_1 a_2 \cdots a_k$ имеется *k* состояний, скажем $q_1 q_2 \cdots q_k$,
- Для входного символа a_1 есть переход из начального состояния в q_1, для входного символа a_2 — переход из q_1 в q_2 и т.д. Состояние q_k является допускающим и сигнализирует о том, что ключевое слово $a_1 a_2 \cdots a_k$ обнаружено.

Пример такого автомата

- Предположим, что необходимо построить НКА, распознающий слова web и ebay. Диаграмма переходов данного НКА, изображенная ниже, построена с помощью изложенных выше правил. Начальное состояние — это состояние 1, а **Σ** обозначает множество печатаемых символов ASCII. Состояния *2-4* отвечают за распознавание слова web, а состояния 5-8 — за распознавание слова ebay.



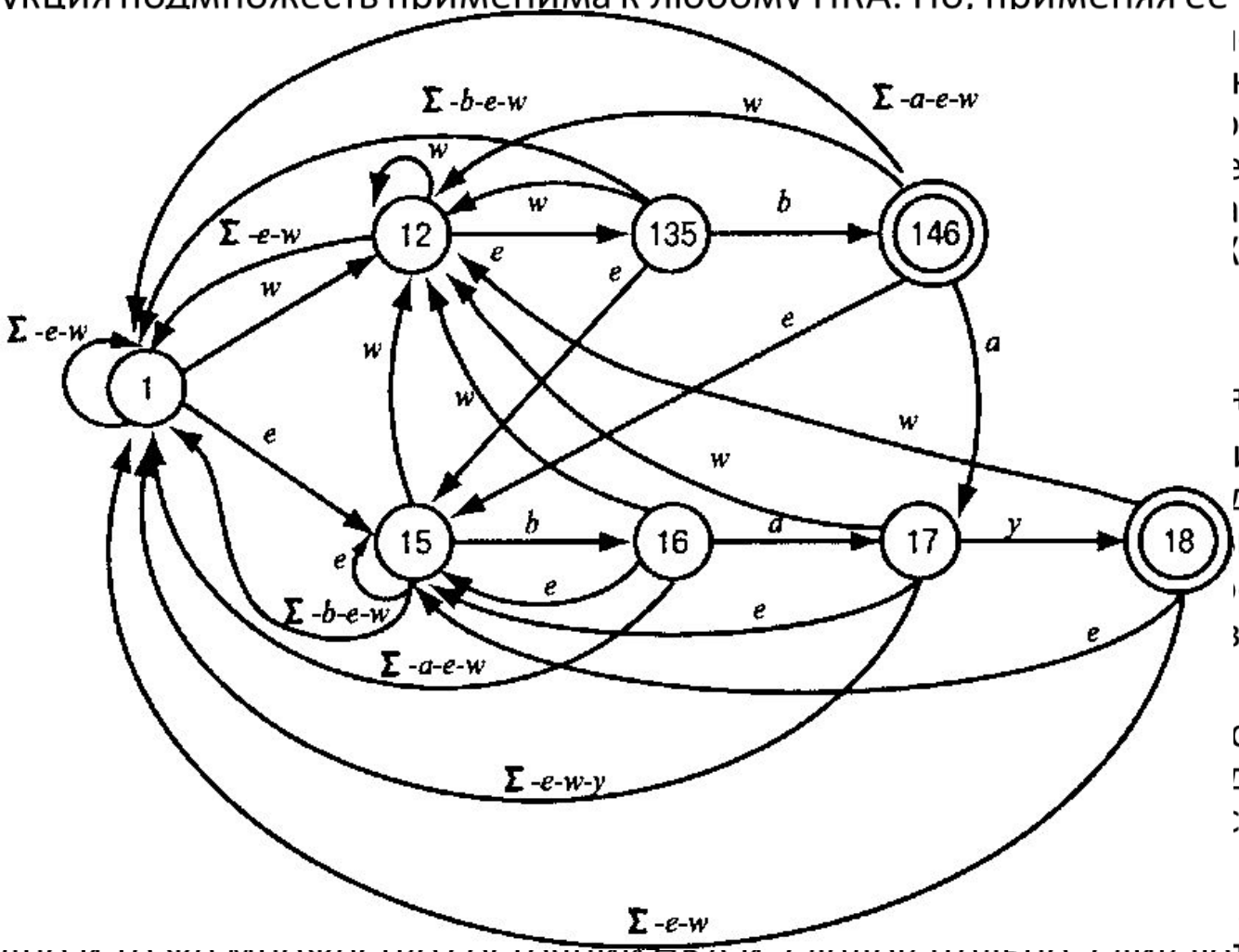
ДКА распознавания слов

Конструкция подмножеств применима к любому НКА. Но, применяя ее к НКА, по-
строе ю
состо ний этого
НКА. жет
прои ечание
обод істо
испо КА.
Прав
следу

а) ний ДКА;

б) из
нача да одним
из со ных со-
стоян фиксом
(окон вида
$a_j a_{j+}$

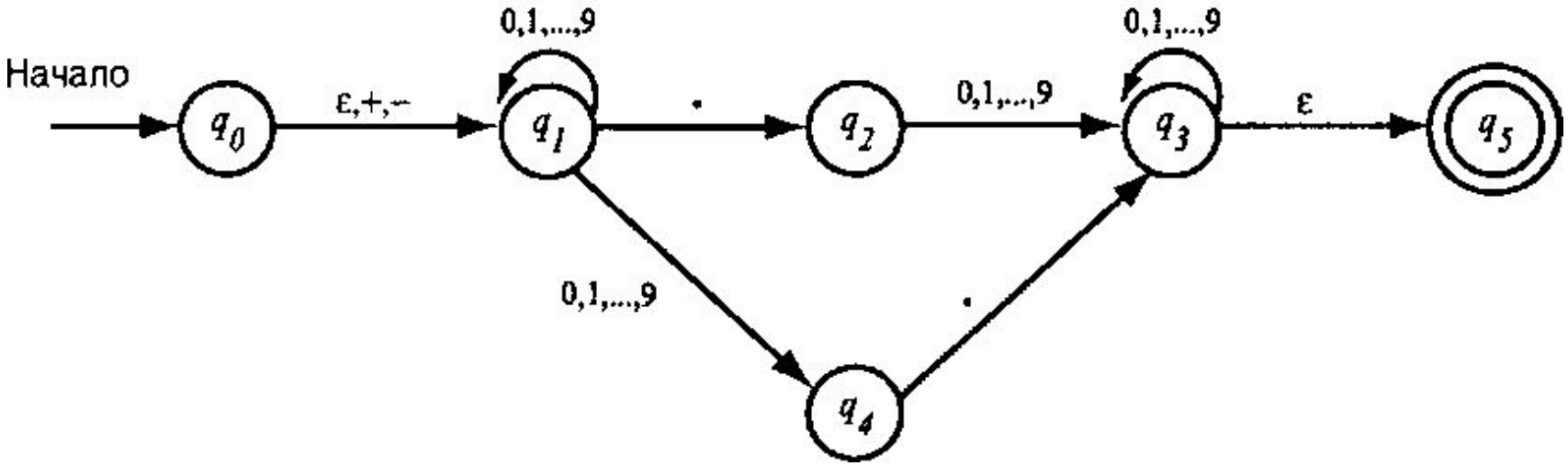
Отме стояние
ДКА. деле дают
одно стояние
ДКА. буквы,
скаж меткой *a*,
дают одно и то же множество состояний НКА и, следовательно, сливаются в одно состояние ДКА.



Конечный автомат с ε – переходом

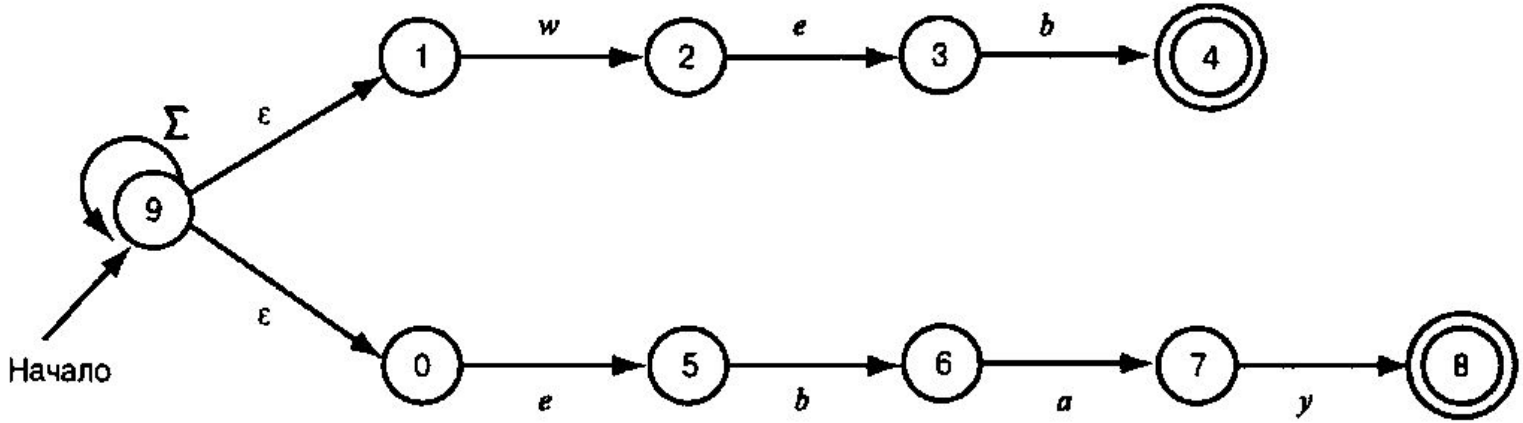
- Рассмотрим еще одно обобщение понятия конечного автомата. Придадим автомату новое "свойство" — возможность совершать переходы по ε, пустой цепочке, т.е. спонтанно, не получая на вход никакого символа. Эта новая возможность, как и недетерминизм, не расширяет класса языков, допустимых конечными автоматами, но дает некоторое дополнительное "удобство программирования". Кроме того, рассмотрев далее регулярные выражения, покажем, что последние тесно связаны с НКА, имеющими ε -переходы. Такие автоматы будем называть ε-*НКА.* Они оказываются полезными при доказательстве эквивалентности между классами языков, задаваемых конечными автоматами и регулярными выражениями.

Использование ε – переходов



- На слайде изображен ε -НКА, допускающий десятичные числа, которые состоят из следующих элементов.
  - Необязательный знак + или -.
  - Цепочка цифр.
  - Разделяющая десятичная точка.
  - Еще одна цепочка цифр. Эта цепочка, как и цепочка (2), может быть пустой, но хотя бы одна из них непуста.

Упрощение поискового автомата



- НКА, распознающий ключевые слова web и ebay, можно реализовать и с помощью ε - переходов, как показано на слайде. Суть в том, что для каждого ключевого слова строится полная последовательность состояний, как если бы это было единственное слово, которое автомат должен распознавать. Затем добавляется новое начальное состояние с ε -переходами в начальные состояния автоматов для каждого из ключевых слов.

Формальная запись ε-НКА

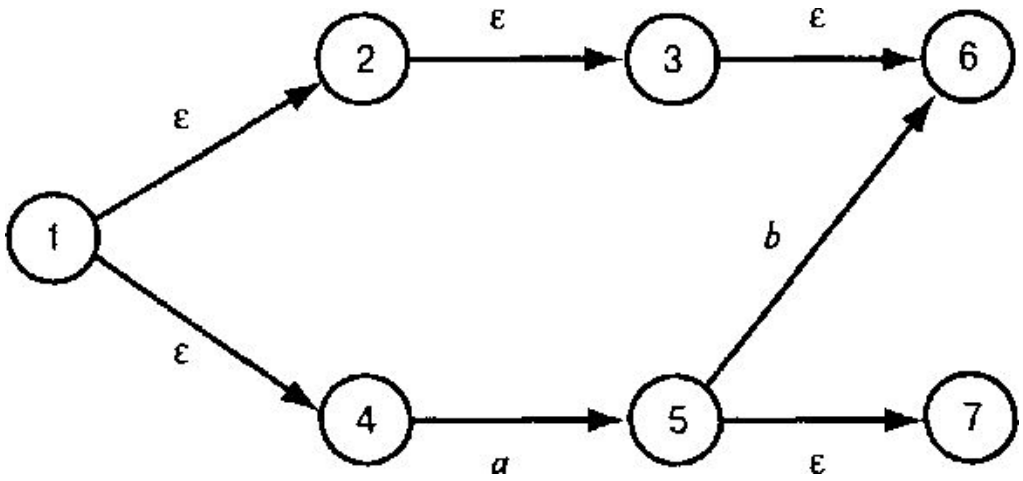
- ε -НКА можно представлять точно так же, как и НКА, с той
- При

	ε	$+, -$	.	$0, 1, \ldots, 9$
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

ε-замыкание

- Дадим формальное определение расширенной функции переходов для ε -НКА, которое приведет к определению допустимости цепочек и языков для данного типа автоматов и в конце концов поможет понять, почему ДКА могут имитировать работу ε -НКА. Однако прежде нужно определить одно из центральных понятий, так называемое ε -*замыкание* состояния. Говоря нестрого, мы получаем ε -замыкание состояния *q,* совершая все возможные переходы из этого состояния, отмеченные ε. Но после совершения этих переходов и получения новых состояний снова выполняются ε -переходы, уже из новых состояний, и т.д. В конце концов, мы находим все состояния, в которые можно попасть из *q* по любому пути, каждый переход в котором отмечен символом ε. Формально определяем ε -замыкание, ECLOSE, рекурсивно следующим образом:
- **Базис.** ECLOSE(q) содержит состояние *q.*
- **Индукция.** Если ECLOSE(q) содержит состояние *p,* и существует переход, отмеченный ε, из состояния p в состояние r, то ECLOSE(q) содержит r. Точнее, если δ есть функция переходов рассматриваемого e-НКА и ECLOSE(q) содержит *p,* то ECLOSE(q) содержит также все состояния из $\delta(p, \varepsilon)$.

Пример ε-замыкания



- Для данного в нем набора состояний, который может быть частью некоторого ε -НКА, мы можем заключить, что ECLOSE(I)= {1,2, 3,4, 6}.
- В каждое из этих состояний можно попасть из состояния 1, следуя по пути, отмеченному исключительно ε. К примеру, в состояние 6 можно попасть по пути 1—>2—>3—>6. Состояние 7 не принадлежит ECLOSE(1), поскольку, хотя в него и можно попасть из состояния 1, в соответствующем пути содержится переход 4—>5, отмеченный не ε. И не важно, что в состояние 6 можно попасть из состояния 1, следуя также по пути 1 —>4—>5—>6, в котором присутствует не е-переход. Существования одного пути, от меченного только ε, уже достаточно для того, чтобы состояние 6 содержалось в ECLOSE(1)

Расширенные переходы и языки ε –НКА

Пусть $\mathrm{E} = (\mathrm{Q}, \Sigma, \delta, \mathrm{q}_0, \mathrm{F})$ — некоторый ε -НКА. Для отображения того, что происходит при чтении некоторой последовательности символов, сначала определим $\hat{\delta}$ — расширенную функцию переходов. Замысел таков: определить $\hat{\delta}(q, w)$ как множество состояний, в которые можно попасть по путям, конкантенации меток вдоль которых дают цепочку w. При этом, как и всегда, символы ε, встречающиеся вдоль пути, ничего не добавляют к w. Соответствующее рекурсивное определение $\hat{\delta}$ имеет следующий вид:

Базис. $\hat{\delta}(q, \varepsilon) = ECLOSE(q)$. Таким образом, если ε — метка пути, то мы можем совершать переходы лишь по дугам с меткой ε, начиная с состояния *q;* это дает нам в точности то же, что и ECLOSE(q).

Индукция. Предположим, что *w* имеет вид *xa,* где *a* — последний символ w. Отметим, что *a* есть элемент Σ и, следовательно, не может быть ε, так как ε не принадлежит Σ. Мы вычисляем $\hat{\delta}(q, w)$ следующим образом:

Пусть $\{\mathrm{p}_1, \mathrm{p}_2, \cdots, \mathrm{p}_\mathrm{k}\}$ есть $\hat{\delta}(q, x)$, т.е. p_i — это все те и только те состояния, в которые можно попасть из *q* по пути, отмеченному *x.* Этот путь может оканчиваться одним или несколькими ε -переходами, а также содержать и другие ε -переходы.

Пусть $\bigcup_{i=1}^{k} \delta(p_i, a)$ есть множество $\{\mathrm{r}_1, \mathrm{r}_2, \cdots, \mathrm{r}_\mathrm{m}\}$, т.е. нужно совершить все переходы, отмеченные символом *a, из* тех состояний, в которые мы можем попасть из *q* по пути, отмеченному x. Состояния r_i — лишь *некоторые* из тех, в которые мы можем попасть из *q* по пути, отмеченному *w.* В остальные такие состояния можно попасть из состояний r_i посредством переходов с меткой ε, как описано ниже в (3).

$\hat{\delta}(q, w) = \bigcup_{j=1}^{m} ECLOSE(\mathrm{r}_\mathrm{i})$. На этом дополнительном шаге, где мы берем замыкание и добавляем все выходящие из *q* пути, отмеченные *w,* учитывается возможность существования дополнительных дуг, отмеченных ε, переход по которым может быть совершен после перехода по последнему "непустому" символу *a.*

Пример

Вычислим $\hat{\delta}(q_0, 5.6)$ для ε -НКА на слайде 69. Для этого выполним следующие шаги.

$\hat{\delta}(q_0, \varepsilon) = ECLOSE(q_0) = \{q_0, q_1\}$.

Вычисляем $\hat{\delta}(q_0, 5)$ следующим образом:

Находим переходы по символу 5 из состояний q_0 и q_1 полученных при вычислении $\hat{\delta}(q_0, \varepsilon)$: $\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$.

Находим ε -замыкание элементов, вычисленных на шаге (1). Получаем: $ECLOSE(q_1) \cup ECLOSE(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$.

Эта двушаговая схема применяется к следующим двум символам.

Вычисляем $\hat{\delta}(q_0, 5)$.

Сначала $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$

Затем $\hat{\delta}(q_0, 5.) = ECLOSE(q_2) \cup ECLOSE(q_4) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$.

Наконец, вычисляем $\hat{\delta}(q_0, 5.6)$.

Сначала $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \emptyset = \{q_3\}$.

Затем $\hat{\delta}(q_0, 5.6) = ECLOSE(q_3) = \{q_3, q_5\}$

Теперь можно определить язык ε -НКА $\mathrm{E} = (\mathrm{Q}, \Sigma, \delta, \mathrm{q}_0, \mathrm{F})$ так, как и было задумано ранее: $\mathrm{L}(\mathrm{E}) = \{\omega | \hat{\delta}(q, w) \cap F \neq \emptyset\}$. Таким образом, язык E— это множество цепочек *w,* которые переводят автомат из начального состояния хотя бы в одно из допускающих.

$\hat{\delta}(q_0, 5.6)$ содержит допускающее состояние q_5, поэтому цепочка 5 . 6 принадлежит языку ε -НКА.

Устранение ε-переходов

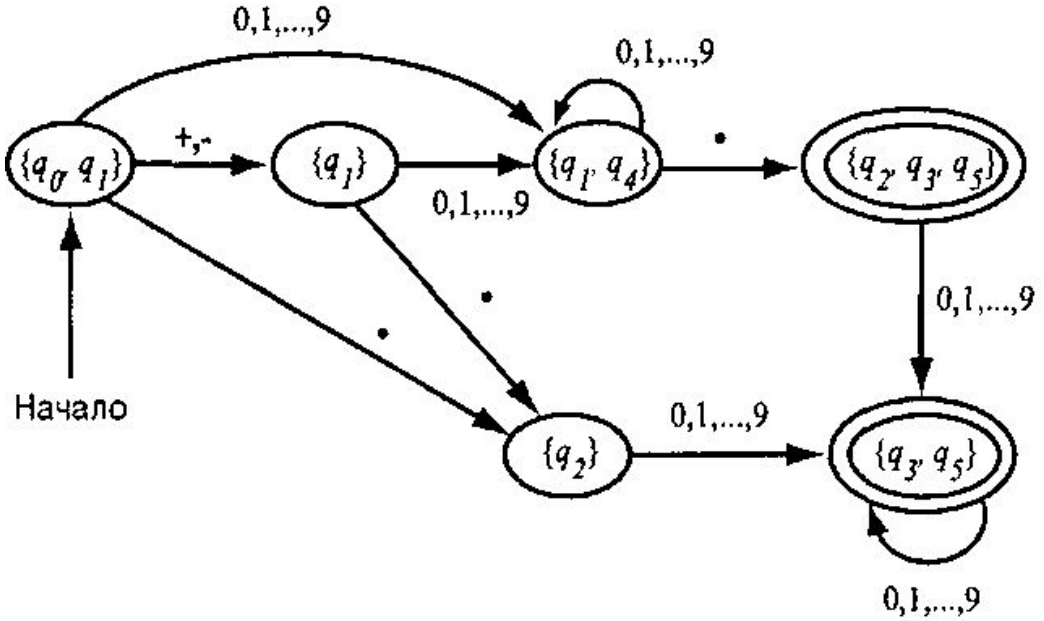
Для всякого ε -НКА *E* можно найти ДКА *D,* допускающий тот же язык, что и *E.* Поскольку состояния *D* являются подмножествами из состояний *E,* то используемая конструкция очень напоминает конструкцию подмножеств. Единственное отличие состоит в том, что нужно присоединить еще и ε -переходы *E,* применив механизм ε -замыкания.

Пусть $\mathrm{E} = (\mathrm{Q_E}, \Sigma, \delta_\mathrm{E}, \mathrm{q_E}, \mathrm{F_E})$- Тогда эквивалентный ДКА $\mathrm{D} = (\mathrm{Q_D}, \Sigma, \delta_\mathrm{D}, \mathrm{q_D}, \mathrm{F_D})$ определяется следующим образом.

1. $\mathrm{Q_D}$ есть множество подмножеств $\mathrm{Q_E}$. Точнее, как мы выясним, для *D* допустимыми состояниями являются только ε -*замкнутые подмножества* $\mathrm{Q_E}$, т.е. такие множества *S* с $\mathrm{Q_E}$, для которых $S = \mathrm{ECLOSE(S)}$. Иначе говоря, ε -замкнутые множества состояний *S*— это такие множества, у которых всякий ε -переход из состояния, принадлежащего *S,* приводит снова в состояние из *S.* Заметим, что ∅ есть ε -замкнутое множество.
2. $\mathrm{q_D} = \mathrm{ECLOSE}(\mathrm{q_0})$, т.е., замыкая множество, содержащее только начальное состояние *E,* мы получаем начальное состояние *D.* Заметим, что это правило отличается от использованного ранее в конструкции подмножеств — там за начальное состояние построенного автомата принималось множество, содержащее только начальное состояние данного НКА.
3. $\mathrm{F_D}$ — это такие множества состояний, которые содержат хотя бы одно допускающее состояние автомата *E.* Таким образом, $\mathrm{F_D} = \{\mathrm{S|S\ принадлежит Q_D\ и\ S} \bigcap \mathrm{F_E} \neq \emptyset\}$.
4. $\delta_\mathrm{D}(\mathrm{S}, \mathrm{a})$ для всех a из Σ и множеств *S* из $\mathrm{Q_D}$ вычисляется следующим образом:
   а) пусть $S = \{p_1, p_2, \cdots, p_k\}$,
   б) вычислим $\bigcup_{i=1}^{k} \delta(p_i, a)$; пусть это будет множество $\{r_1, r_2, \cdots, r_m\}$;
   в) тогда $\delta_\mathrm{D}(\mathrm{S}, \mathrm{a}) = \bigcup_{\mathrm{j}=1}^{\mathrm{m}} \mathrm{ECLOSE}(\mathrm{r_j})$.

Пример

- Удалим ε -переходы из ε -НКА (см. слайд 69), который далее называется *E.* По *E* строим ДКА *D,* изображенный слайде. Для того чтобы избежать излишнего нагромождения, состояние ∅ и все переходы в него с рисунка удалены. Поэтому, следует иметь в виду, что у каждого состояния есть еще дополнительные переходы в состояние ∅ по тем входным символам, для которых переход на рисунке отсутствует. Кроме того, у состояния ∅ есть переход в себя по любому входному символу.



Регулярные выражения

Перейдем от "машинного" задания языков с помощью ДКА и НКА к алгебраическому описанию языков с помощью регулярных выражений. Можно показать, что регулярные выражения определяют точно те же языки, что и различные типы автоматов, а именно, регулярные языки. В то же время, в отличие от автоматов, регулярные выражения позволяют определять допустимые цепочки декларативным способом. Поэтому регулярные выражения используются в качестве входного языка во многих системах, обрабатывающих цепочки. Рассмотрим два примера.

- Команды поиска. В таких системах регулярные выражения используются для описания шаблонов, которые пользователь ищет в файле. Различные поисковые системы преобразуют регулярное выражение либо в ДКА, либо в НКА и применяют этот автомат к файлу, в котором производится поиск.
- Генераторы лексических анализаторов, такие как Lex или Flex. Лексический анализатор — это компонент компилятора, разбивающий исходную про грамму на логические единицы *(лексемы),* которые состоят из одного или нескольких символов и имеют определенный смысл. Генератор лексических анализаторов получает формальные описания лексем, являющиеся по существу регулярными выражениями, и создает ДКА, который распознает, какая из лексем появляется на его входе.

Операторы регулярных выражений слайд 1

- Регулярные выражения обозначают (задают, или представляют) языки. В качестве простого примера рассмотрим регулярное выражение **01*+10*.** Оно определяет язык всех цепочек, состоящих либо из одного нуля, за которым следует любое количество единиц, либо из одной единицы, за которой следует произвольное количество нулей.
- Чтобы понять, почему наша интерпретация заданного регулярного выражения правильна, необходимо определить все использованные в этом выражении символы, поэтому рассмотрим следующие три операции над языками, соответствующими операторам регулярных выражений.

Операторы регулярных выражений слайд 2

- *Объединение* двух языков *L* и *M,* обозначаемое *LU M,* — это множество цепочек, которые содержатся либо в *L,* либо в *M,* либо в обоих языках. Например, если *L* = {001, 10, 111} и M= {ε, 001}, то $L \cup M = \{\varepsilon, 10, 001, 111\}$.
- *Конкатенация* языков *L* и *M*— это множество цепочек, которые можно образовать путем дописывания к любой цепочке из L любой цепочки из M. Выше было дано определение конкатенации двух цепочек: результатом ее является запись одной цепочки вслед за другой. Конкатенация языков обозначается либо точкой, либо вообще никак не обозначается, хотя оператор конкатенации часто называют "точкой". Например, если *L* = {001, 10, 111} и M= {ε, 001}, то *L.M,* или просто *LM,* — это {001, 10, 111, 001001, 10001, 111001}. Первые три цепочки в *LM*— это цепочки из *L,* соединенные с ε. Поскольку ε является единицей (нейтральным элементом) для операции конкатенации, результирующие цепочки будут такими же, как и цепочки из *L.* Последние же три цепочки в *LM* образованы путем соединения каждой цепочки из L со второй цепочкой из M, т.е. с 001. Например, 10 из L, соединенная с 001 из *M,* дает 10001 для *LM.*
- *Итерация ("звездочка*", или *замыкание Клини)* языка L обозначается L^* и представляет собой множество всех тех цепочек, которые можно образовать путем конкатенации любого количества цепочек из L. При этом допускаются повторения, т.е. одна и та же цепочка из L может быть выбрана для конкатенации более одного раза. Например, если *L* = {0, 1}, то L^* — это все цепочки, состоящие из нулей и единиц. Если *L* = {0, 11}, то в L^* входят цепочки из нулей и единиц, содержащие четное количество единиц, например, цепочки 011, 11110 или ε, и не входят цепочки 01011 или 101. Более формально язык L^* можно представить как бесконечное объединение $\bigcup_{i \geq 0} L^i$, где $L^0 = \{\varepsilon\}$, $L^1 = L$, $L^i = L \ldots L$ (конкатенация i копий L)

Пример итерации слайд 1

Поскольку идея итерации языка может показаться довольно сложной, рассмотрим несколько примеров. Для начала возьмем L= {0, 11}. $L^0 = \{\varepsilon\}$ независимо от языка *L*.

$L^1 = L$, что означает выбор одной цепочки из *L*. Таким образом, первые два члена в разложении L^* дают $\{L^*, 0, 11\}$.

Далее рассмотрим L^2 Выберем две цепочки из L и, поскольку их можно выбирать с повторениями, получим четыре варианта, которые дают $L^2 = \{00, 011, 110, 111\}$. Аналогично, L^3 представляет собой множество цепочек, образованных троекратным выбором из двух цепочек языка L. Следовательно, L^3 имеет вид $\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$

Для вычисления L^* необходимо вычислить L^i для каждого *i* и объединить все эти языки. Язык L^i содержит 2^i элементов. Хотя каждое множество L^i конечно, объединение бесконечного числа множеств L^1 образует, как правило, бесконечный язык, что справедливо, в частности, и для рассматриваемого примера.

Пример итерации слайд 2

- Пусть теперь *L* — множество всех цепочек, состоящих из нулей. Заметим, что такой язык бесконечен, в отличие от предыдущего примера, где был рассмотрен конечный язык. Однако нетрудно увидеть, что представляет собой L^*. Как всегда, $\mathrm{L}^0 = \{\varepsilon\}$, $\mathrm{L}^1 = \mathrm{L}$. L^2 — это множество цепочек, которые можно образовать, если взять одну цепочку из нулей и соединить ее с другой цепочкой из нулей. В результате получим цепочку, также состоящую из нулей. Фактически, любую цепочку из нулей можно записать как конкатенацию двух цепочек из нулей. Следовательно, $\mathrm{L}^2 = \mathrm{L}$. Аналогично, $\mathrm{L}^3 = \mathrm{L}$ и так далее. Таким образом, бесконечное объединение $\mathrm{L}^* = \mathrm{L}^0 \cup \mathrm{L}^1 \cup \mathrm{L}^2 \cup \cdots$ совпадает *с* L *в* том особом случае, когда язык *L* является множеством всех нулевых цепочек.
- В качестве последнего примера рассмотрим $\emptyset^* = \{\varepsilon\}$. Заметим, что $\emptyset^0 = \{\varepsilon\}$, тогда как $\emptyset^i, i > 1$ будет пустым множеством, поскольку *мы* не можем выбрать ни одной цепочки из пустого множества. Фактически, $\emptyset$ является одним из всего двух языков, итерация которых *не является* бесконечным множеством.

Построение регулярных выражений

- Все алгебры начинаются с некоторых элементарных выражений. Обычно это константы и/или переменные. Применяя определенный набор операторов к этим элементарным выражениям и уже построенным выражениям, можно конструировать более сложные выражения. Обычно необходимо также иметь некоторые методы группирования операторов и операндов, например, с помощью скобок. К примеру, обычная арифметическая алгебра начинается с констант (целые и действительные числа) и переменных и позволяет нам строить более сложные выражения с помощью таких арифметических операторов, как + или *.
- Алгебра регулярных выражений строится по такой же схеме: используются константы и переменные для обозначения языков и операторы для обозначения трех операций регулярных выражений — объединение, точка и звездочка (оператор звёздочка позволяет получить все цепочки, принадлежащие языку).

Определение регулярного выражения

- Регулярные выражения можно определить рекурсивно. В этом определении не только характеризуются правильные регулярные выражения, но и для каждого регулярного выражения E описывается представленный им язык, который обозначается через $L(E)$.
- Базис. Базис состоит из трех частей.
  - Константы ε и ∅ являются регулярными выражениями, определяющими языки {ε} и ∅, соответственно, т.е. $L(\varepsilon) = \{\varepsilon\}$ и $L(\emptyset) = \emptyset$.
  - Если a — произвольный символ, то **a** — регулярное выражение, определяющее язык {a}, т.е. $L(\mathbf{a}) = \{\mathrm{a}\}$. Заметим, что для записи выражения, соответствующего символу, используется жирный шрифт. Это соответствие, т.е. что **a** относится к *a,* должно быть очевидным.
  - Переменная, обозначенная прописной курсивной буквой, например, *L,* представляет произвольный язык.
- Индукция. Индуктивный шаг состоит из четырех частей, по одной для трех операторов и для введения скобок.
  - Если *E* и *F*— регулярные выражения, то *E+F*— регулярное выражение, определяющее объединение языков *L(E)* и *L(F*), т.е. $L(E + F) = L(E) \cup L(F)$.
  - Если *E* и *F* — регулярные выражения, то *EF* — регулярное выражение, определяющее конкатенацию языков *L(E)* и *L(F).* Таким образом, $\mathrm{L(EF)} = \mathrm{L(E).L(F)}$. Заметим, что для обозначения оператора конкатенации — как операции над языками, так и оператора в регулярном выражении — можно использовать точку. Например, регулярное выражение **0.1** означает то же, что и **01**, и представляет язык {01}. Однако мы избегаем использовать точку в качестве оператора конкатенации в регулярных выражениях.
  - Если *E* — регулярное выражение, то E^* — регулярное выражение, определяющее итерацию языка *L(E).* Таким образом, $\mathrm{L}(E^*) = \big(\mathrm{L(E)}\big)^*$.
  - Если E — регулярное выражение, то (E) — регулярное выражение, определяющее тот же язык *L(E),* что и выражение *E.* Формально, $\big(\mathrm{L(E)}\big) = \mathrm{L(E)}$.

Пример регулярного выражения слайд 1

- Напишем регулярное выражение для множества цепочек из чередующихся нулей и единиц. Сначала построим регулярное выражение для языка, состоящего из одной-единственной цепочки 01. Затем используем оператор "звездочка" для того, чтобы построить выражение для всех цепочек вида 0101...01.
- Базисное правило для регулярных выражений говорит, что **0** и **1** — это выражения, обозначающие языки {0} и {1}, соответственно. Если соединить эти два выражения, то получится регулярное выражение **01** для языка {01}. Как правило, если мы хотим написать выражение для языка, состоящего из одной цепочки $\boldsymbol{\omega}$, то используем саму $\boldsymbol{\omega}$ как регулярное выражение. Заметим, что в таком регулярном выражении символы цепочки $\boldsymbol{\omega}$ обычно выделяют жирным шрифтом, но изменение шрифта предназначено лишь для того, чтобы отличить выражение от цепочки, и не должно восприниматься как что-то существенное.
- Далее, для получения всех цепочек, состоящих из нуля или нескольких вхождений 01, используем регулярное выражение $(01)^*$. Заметим, что выражение 01 заключается в скобки, чтобы не путать его с выражением 01*. Цепочки языка 01* начинаются с 0, за которым следует любое количество 1. Причина такой интерпретации объясняется ниже и состоит в том, что операция "звездочка" имеет высший приоритет по сравнению с операцией "точка", и поэтому аргумент оператора итерации выбирается до выполнения любых конкатенаций.

Пример регулярного выражения слайд 2

- Однако $L((01)^*)$ — не совсем тот язык, который нам нужен. Он включает только те цепочки из чередующихся нулей и единиц, которые начинаются с 0 и заканчиваются 1. Необходимо также учесть возможность того, что вначале стоит 1 и/или в конце 0. Одним из решений является построение еще трех регулярных выражений, описывающих три другие возможности. Итак, $(10)^*$ представляет те чередующиеся цепочки, которые начинаются символом 1 и заканчиваются символом 0, $0(10)^*$ можно использовать для цепочек, которые начинаются и заканчиваются символом 0, а $1(01)^*$ — для цепочек, которые и начинаются, и заканчиваются символом 1. Полностью это регулярное выражение имеет следующий вид.
- $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
- Заметим, что оператор + используется для объединения тех четырех языков, которые вместе дают все цепочки, состоящие из чередующихся символов 0 и 1.

Пример регулярного выражения слайд 3

- Однако существует еще одно решение, приводящее к регулярному выражению, которое имеет значительно отличающийся и к тому же более краткий вид. Снова начнем с выражения $(01)^*$. Можем добавить необязательную единицу в начале, если слева к этому выражению допишем выражение ε+1. Аналогично, добавим необязательный 0 в конце с помощью конкатенации с выражением ε + 0. Например, используя свойства оператора +, получим, что
- $L(\varepsilon + 1) = L(\varepsilon) \cup L(1) = \{\varepsilon\}\{1\} = \{\varepsilon, 1\}$.
- Если мы допишем к этому языку любой другой язык *L,* то выбор цепочки ε даст нам все цепочки из *L,* а выбрав 1, получим *1 w* для каждой цепочки *w* из *L.* Таким образом, совокупность цепочек из чередующихся нулей и единиц может быть представлена следующим выражением: $(\varepsilon + 1)(01)^*(\varepsilon + 0)$.
- Обратите внимание на то, что суммируемые выражения необходимо заключать в скобки, чтобы обеспечить правильную группировку операторов.

Приоритеты операторов регулярных выражений

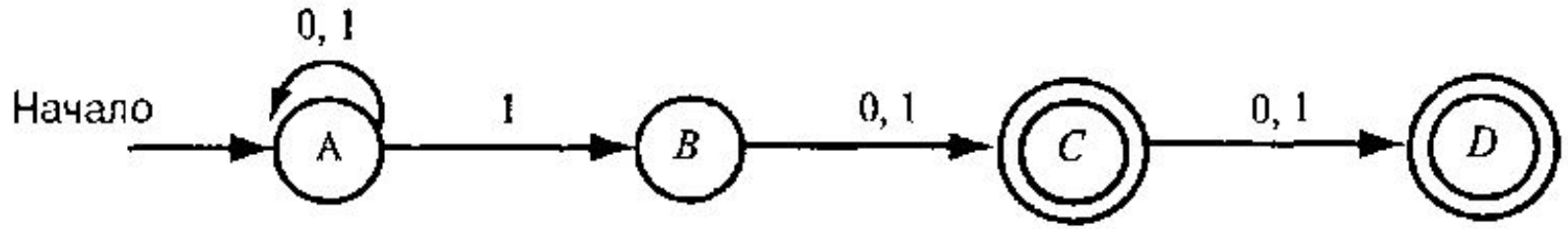
- Для операторов регулярных выражений определен следующий порядок приоритетов:
  - Оператор "звездочка" имеет самый высокий приоритет, т.е. этот оператор применяется только к наименьшей последовательности символов, находящейся слева от него и являющейся правильно построенным регулярным выражением.
  - Далее по порядку приоритетности следует оператор конкатенации, или "точка". Связав все "звездочки" с их операндами, связываем операторы конкатенации с соответствующими им операндами, т.е. все *смежные* (соседние, без промежуточных операторов) выражения группируются вместе. Поскольку оператор конкатенации является ассоциативным, то не имеет значения, в каком порядке мы группируем последовательные конкатенации. Если же необходимо сделать выбор, то следует группировать их, начиная слева. Например, 012 группируется как (01)2.
  - В заключение, со своими операндами связываются операторы объединения (операторы +). Поскольку объединение тоже является ассоциативным оператором, то и здесь не имеет большого значения, в каком порядке сгруппированы последовательные объединения, однако мы будем придерживаться группировки, начиная с левого края выражения.

Связь конечных автоматов и регулярных выражений

- Связь конечных автоматов и регулярных выражений показана в соответствующей презентации

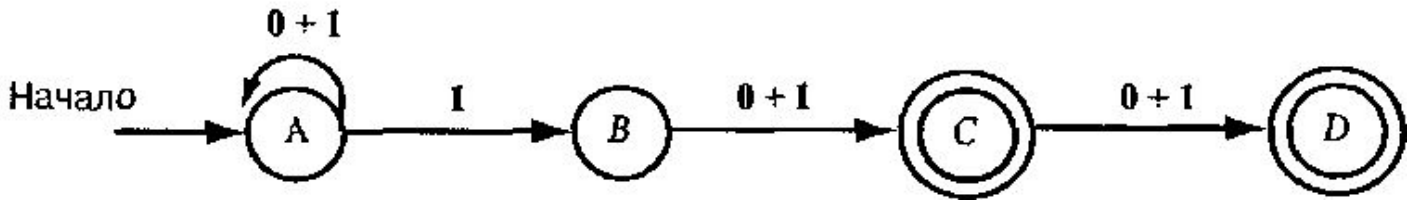
Минимизация НКА регулярными выражениями слайд 1/5

- Рассмотрим НКА, допускающий цепочки из нулей и единиц, у которых либо на второй, либо на третьей позиции с конца



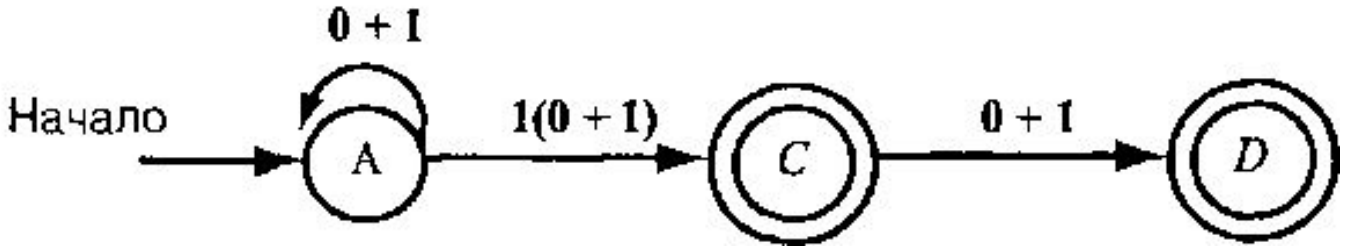
Минимизация НКА регулярными выражениями слайд 2/5

- Вначале преобразуем этот автомат в автомат с регулярными выражениями в качестве меток. Пока исключение состояний не производилось, то все, что нам нужно сделать, это заменить метки "0, 1" эквивалентным регулярным выражением 0+1.



Минимизация НКА регулярными выражениями слайд 3/5

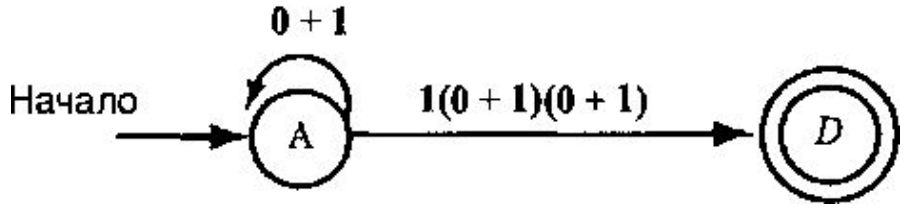
- Исключим сначала состояние B. Поскольку это состояние не является ни начальным, ни допускающим, то его не будет ни в одном из сокращенных автоматов. Мы избавимся от лишней работы, если исключим это состояние до того, как начнем строить два сокращенных автомата, соответствующих двум его допускающим состояниям.
- Существует одно состояние *A,* предшествующее *B,* и одно последующее состояние C. Используя обозначения регулярных выражений диаграммы, получим: $Q_1 = 1, P_1 = 0 + 1, R_{11} = 0$ (потому что из *A* в C дуги нет) и $S = \emptyset$ (поскольку нет петли в состоянии *B).* В результате, выражение над новой дугой из *A* в *C* равно Ø + 1Ø*(Ø+ 1).
- Для сокращения полученного выражения сначала исключаем первый элемент 0, который можно игнорировать при объединении. Выражение приобретает вид 1Ø*(0 + 1). 1Ø*(0+1) эквивалентно выражению 1(0+1), которое использовано для дуги A —> C



Минимизация НКА регулярными выражениями слайд 4/5

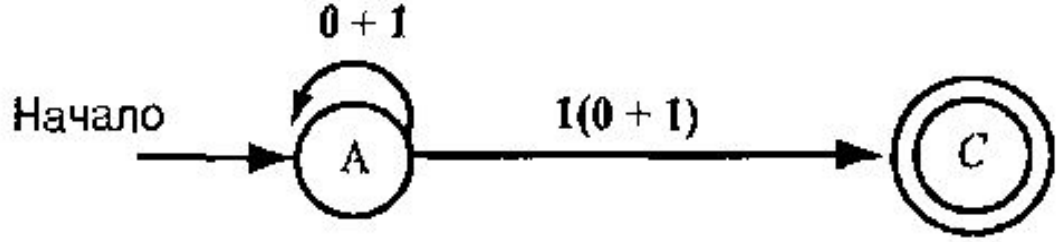
Далее нужно по отдельности исключить состояния C и *D*. Процедура исключения состояния C аналогична описанному выше исключению состояния *B*

В обозначениях обобщенного автомата с двумя состояниями, изображенного на ниже, соответствующие регулярные выражения равны: $R = 0 + 1, S = 1(0 + 1)(0 + 1), T = \emptyset, U = \emptyset$. Выражение *U** можно заменить на ε, т.е. исключить его из конкатенации, поскольку, как показано выше, 0* = ε. Кроме того, выражение *SU*T* эквивалентно ∅, потому что T— один из операндов конкатенации— равен 0. Таким образом, общее выражение $(R + SU^*T)^*SU^*$ в данном случае упрощается до R^*S, или $(0 + 1)^*1(0 + 1)(0 + 1)$. Говоря неформально, язык этого выражения состоит из цепочек, заканчивающихся единицей с двумя последующими символами, каждый из которых может быть либо нулем, либо единицей.



Минимизация НКА регулярными выражениями слайд 5/5

- Теперь снова нужно вернуться к исходному автомату и исключить состояние *D.* Поскольку в этом автомате нет состояний, следующих за *D,* то необходимо лишь удалить дугу, ведущую из *C* в *D,* вместе с состоянием *D.* В результате получится автомат, показанный ниже.
- Этот автомат очень похож на автомат, изображенный на слайде выше; различаются только метки над дугами, ведущими из начального состояния в допускающее. Следовательно, можно применить правило для автомата с двумя состояниями и упростить данное выражение, получив в результате (0 + 1)*1(0 + 1). Это выражение представляет другой тип цепочек, допустимых этим автоматом, — цепочки, у которых 1 стоит на второй позиции с конца.
- Осталось объединить оба построенные выражения, чтобы получить следующее выражение для всего автомата.
- (0 + 1



Построение автомата на основе регулярного выражения слайд 1/3

Все конструируемые автоматы представляют собой ε-НКА с одним допускающим состоянием.

Теорема. Любой язык, определяемый регулярным выражением, можно задать некоторым конечным автоматом.

Доказательство. Предположим, что *L = L(R)* для регулярного выражения *R.* Покажем, что *L = L(E)* для некоторого ε-НКА *E,* обладающего следующими свойствами.

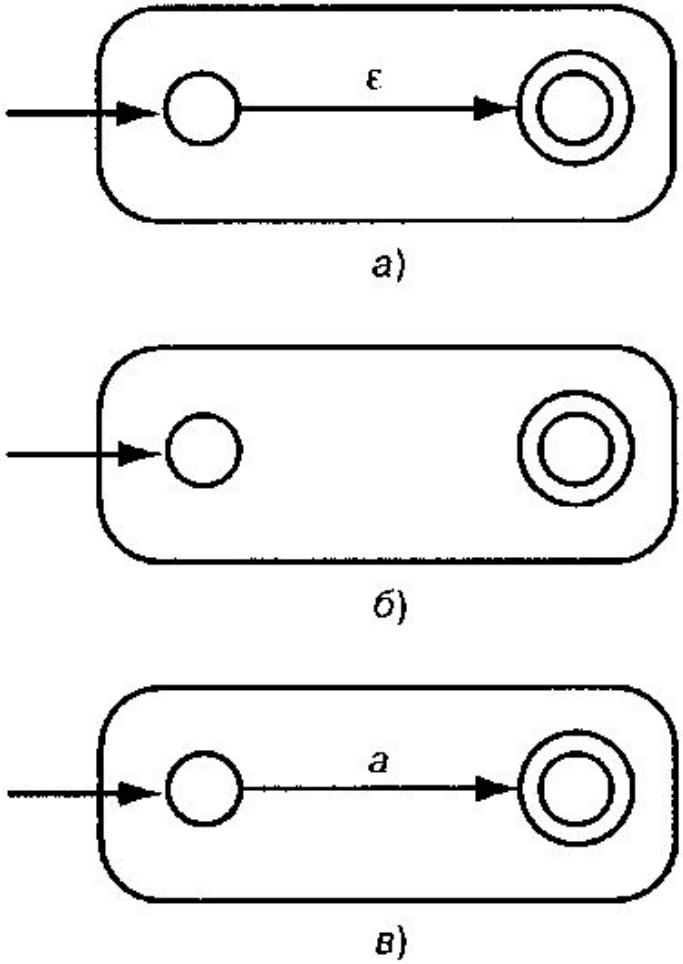
- Он имеет ровно одно допускающее состояние.
- У него нет дуг, ведущих в начальное состояние.
- У него нет дуг, выходящих из допускающего состояния.

Доказательство проводится структурной индукцией по выражению *R,* следуя рекурсивному определению регулярных выражений, данному выше.

Построение автомата на основе регулярного выражения слайд 2/3

Базис. Базис состоит из трех частей, представленных на слайде.

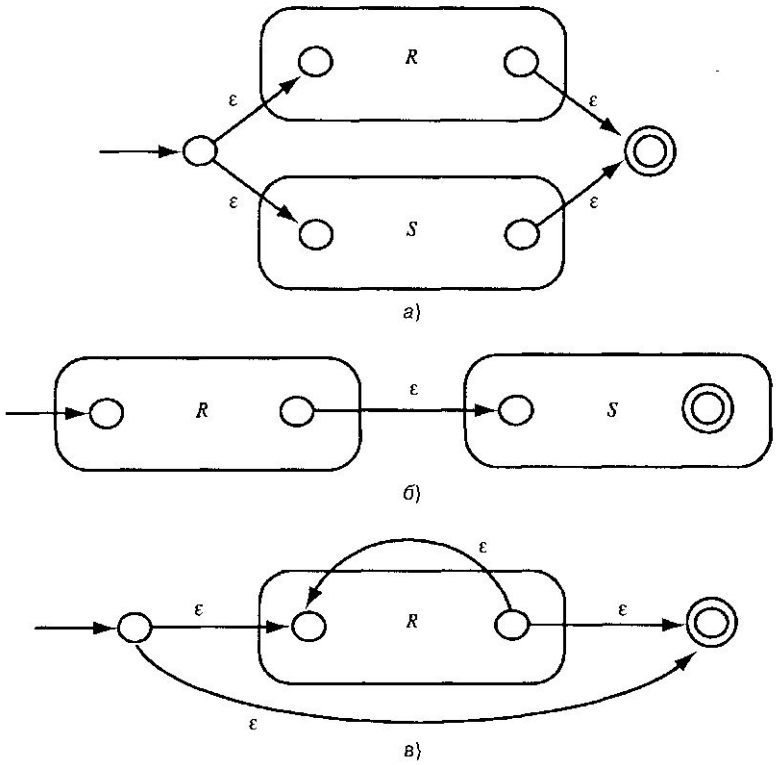
В части (а) рассматривается выражение ε. Языком такого автомата является {ε}, поскольку единственный путь из начального в допускающее состояние помечен выражением ε. В части (б) показана конструкция для $\varnothing$. Понятно, что, поскольку отсутствуют пути из начального состояния в допускающее, языком этого автомата будет $\varnothing$. Наконец, в части (в) представлен автомат для регулярного выражения а. Очевидно, что язык этого автомата со стоит из одной цепочки *а* и равен *L(*а). Кроме того, все эти автоматы удовлетворяют условиям (1), (2) и (3) индуктивной гипотезы.



Построение автомата на основе регулярного выражения слайд 3/3

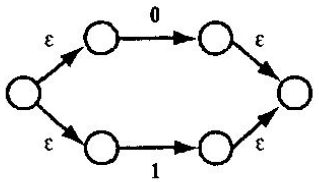
Предположим, что утверждение теоремы истинно для непосредственных подвыражений данного регулярного выражения, т.е. языки этих подвыражений являются также языками ε-НКА с единственным допускающим состоянием. Возможны четыре случая.

1. Данное выражение имеет вид *R + S* для некоторых подвыражений *R* и *S*. Тогда ему соответствует автомат, представленный на рис. а. Язык автомата, представленного на рис. *а,* равен $L(R) \cup L(S)$.
2. Выражение имеет вид *RS* для некоторых подвыражений *R* и *S*. Автомат для этой конкатенации представлен на рис. б. Путями автомата, представленного на рис. б, будут те и только те пути, которые помечены цепочками из языка *L(R)L(S).*
3. Выражение имеет вид *R** для некоторого подвыражения *R*. Используем автомат, представленный на рис. в.
4. Выражение имеет вид *(R)* для некоторого подвыражения *R*. Автомат для *R* может быть автоматом и для *(R),* поскольку скобки не влияют на язык, задаваемый выражением.

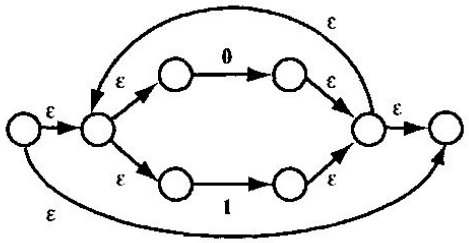


Пример построения автомата

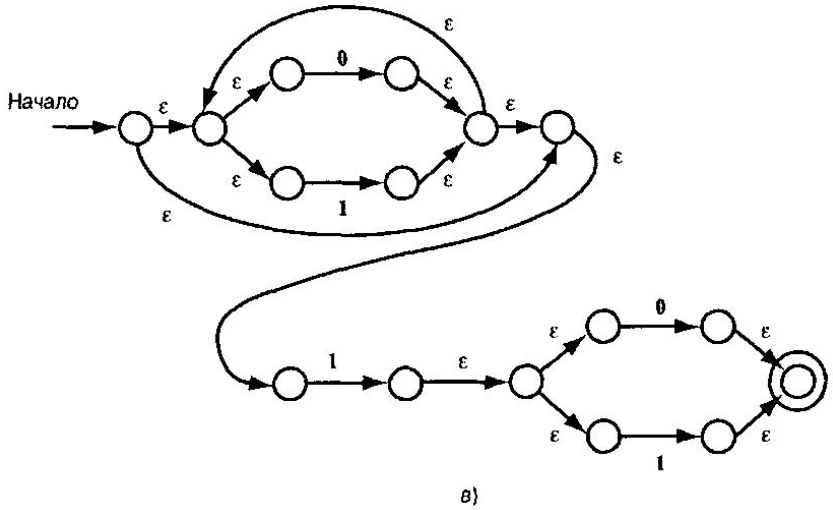
- Преобразуем регулярное выражение (0 + 1)*1(0 + 1) в ε-НКА.
- Построим сначала автомат для 0 + 1. Для этого используем два автомата, построенные согласно предыдущему слайду *:* один с меткой 0 на дуге, другой— с меткой 1. Эти автоматы соединены с помощью конструкции объединения. Результат изображен на рис. *а.*
- Далее, применим к автомату конструкцию итерации. Полученный автомат изображен на рис. б. На последних двух шагах применяется конструкция конкатенации. Сначала автомат, представленный на рис. б, соединяется с автоматом, допускающим только цепочку 1. Последний получается путем еще одного применения базисной конструкции с меткой 1 на дуге. Отметим, что для распознавания цепочки 1 необходимо создать *новый* автомат; здесь нельзя использовать автомат для 1, являю щийся частью автомата, изображенного на рис. *а.* Третьим автоматом в конкатенации будет еще один автомат для выражения 0+1. Опять-таки, необходимо создать копию автомата, поскольку нельзя использовать автомат для 0+1, представляющий собой часть автомата.
- Полный автомат показан на рис. в.



а)



б)



в)

Алгебра регулярных выражений

- В примере выше возникла необходимость упрощения регулярных выражений для того, чтобы их размер не превышал разумные пределы. Было показано, почему одно выражение можно было заменить другим. Во всех рассмотренных ситуациях основной вывод заключался в том, что эти выражения оказывались *эквивалентными,* т. е. задавали один и тот же язык. Ниже будет показан ряд алгебраических законов, позволяющих рассматривать вопрос эквивалентности двух регулярных выражений на более высоком уровне. Вместо исследования определенных регулярных выражений, рассмотрим пары регулярных выражений с переменными в качестве аргументов.
- Два выражения с переменными являются *эквивалентными,* если при подстановке любых языков вместо переменных оба выражения представляют один и тот же язык.

Коммутативность и ассоциативность

Коммутативность— это свойство операции, заключающееся в том, что при перестановке ее операндов результат не меняется. *Ассоциативность*— это свойство операции, позволяющее перегруппировывать операнды, если оператор применяется дважды. Например, ассоциативный закон умножения имеет вид: (xy)xz = x(yz). Для регулярных выражений выполняются три закона такого типа:

- *L + M= M+L,* т.е. *коммутативный закон для объединения* утверждает, что два языка можно объединять в любом порядке.
- *(L + M) + N = L + (M + N).* Этот закон— *ассоциативный закон объединения* — говорит, что для объединения трех языков можно сначала объединить как два первых, так и два последних из них. Обратите внимание на то, что вместе с коммутативным законом объединения этот закон позволяет объединять любое количество языков в произвольном порядке, разбивая их на любые группы, и результат будет одним и тем же.
- *(LM)N = L(MN).* Этот *ассоциативный закон конкатенации* гласит, что для конкатенации трех языков можно сначала соединить как два первых, так и два последних из них.

В предложенном списке нет "закона" *LM=ML,* гласящего, что операция конкатенации является коммутативной, поскольку это утверждение неверно.

Пример. Рассмотрим регулярные выражения 01 и 10. Эти выражения задают языки {01} и {10}, соответственно. Поскольку эти языки различаются, то общий закон *LM=ML* для них не выполняется. Если бы он выполнялся, то можно было бы подставить регулярное выражение 0 вместо *L* и 1 вместо M и получить неверное заключение 01 = 10.

Нулевые и единичные элементы

- *Единичным (нейтральным) элементом (единицей)* операции называется элемент, для которого верно следующее утверждение: если данная операция применяется к единичному элементу и некоторому другому элементу, то результат равен другому элементу. Например, 0 является единичным элементом операции сложения, поскольку 0 + x = x + 0 = x, а 1 — единичным элементом для умножения, потому что $1 * x = x * 1 = x$.

Нулевым элементом (нулем, аннулятором) операции называется элемент, для которого истинно следующее: результатом применения данной операции к нулевому и любому другому элементу является нулевой элемент. Например, 0 является нулевым элементом умножения, поскольку $0 * x = x * 0 = 0$. Операция сложения нулевого элемента не имеет.

Для регулярных выражений существует три закона, связанных с этими понятиями.

- $\emptyset + L = L + \emptyset = L$. Этот закон утверждает, что ∅ является единицей объединения.
- $\varepsilon L = L\varepsilon = L$. Этот закон гласит, что ε является единицей конкатенации.
- ∅L=L∅=∅. Этот закон утверждает, что ∅ является нулевым элементом конкатенации.

Эти законы являются мощным средством упрощения выражений. Например, если необходимо объединить несколько выражений, часть которых упрощена до ∅, то ∅ можно исключить из объединения. Аналогично, если при конкатенации нескольких выражений некоторые из них можно упростить до ε, то ε можно исключить из конкатенации. Наконец, если при конкатенации любого количества выражений хотя бы одно из них равно ∅, то результат такой конкатенации — ∅.

Дистрибутивные законы

Дистрибутивный закон связывает две операции и утверждает, что одну из операций можно применить отдельно к каждому аргументу другой операции.

Поскольку умножение коммутативно, то не имеет значения, с какой стороны, слева или справа от суммы, применяется умножение. Для регулярных выражений существует аналогичный закон, но, поскольку операция конкатенации некоммутативна, то мы сформулируем его в виде следующих двух законов.

- L(M+ N) = LM+ LN. *Этот закон называется* левосторонним дистрибутивным законом конкатенации относительно объединения.
- (M + N)L = ML + NL. *Этот закон называется* правосторонним дистрибутивным законом конкатенации относительно объединения.

Теорема. Если *L, M и N*— произвольные языки, то $\mathrm{L(M \cup N) = LM \cup LN}$

Пример. Рассмотрим регулярное выражение $0 + 01^*$. В объединении можно "вынести за скобки 0", но сначала необходимо представить выражение 0 в виде конкатенации 0 с чем-то еще, а именно с ε. Используем свойства единичного элемента для конкатенации, меняя 0 на *0ε, в* результате чего получим выражение $0\varepsilon + 01^*$. Теперь можно применить левосторонний дистрибутивный закон, чтобы заменить это выражение на $0(\varepsilon + 1^*)$. Далее, учитывая, что ε принадлежит ***L(***1^****),*** заметим, что $\varepsilon + 1^* = 1^*$, и окончательно упростим выражение до 01^*.

Идемпотентность

Операция называется *идемпотентной,* если результат применения этой операции к двум одинаковым значениям как операндам равен этому значению. Обычные арифметические операции не являются идемпотентными. В общем случае $x + x \neq x$ и $x * x \neq x$ (хотя существуют *некоторые* значения x, для которых это равенство выполняется, например 0 + 0 = 0). Однако объединение и пересечение являются идемпотентными операциями, поэтому для регулярных выражений справедлив следующий закон:

- *L+ L = L.*

Этот закон — *закон идемпотентности операции объединения* — утверждает, что объединение двух одинаковых выражений можно заменить одним таким выражением.

Законы оператора итерации

Существует ряд законов, связанных с операцией итерации и ее разновидностями + и ? в стиле UNIX. Перечислим эти законы и вкратце поясним, почему они справедливы.

- $(\mathrm{L}^*)^* = \mathrm{L}^*$. Этот закон утверждает, что при повторной итерации язык уже итерированного выражения не меняется. Язык выражения $(\mathrm{L}^*)^*$ содержит все цепочки, образованные конкатенацией цепочек языка L^*.
- $\emptyset^* = \varepsilon$. Итерация языка 0 состоит из одной-единственной цепочки ε.
- $\varepsilon^* = \varepsilon$. Легко проверить, что единственной цепочкой, которую можно образовать конкатенацией любого количества пустых цепочек, будет все та же пустая цепочка.
- $\mathrm{L}^+ = \mathrm{LL}^* = \mathrm{L}^*\mathrm{L}$. Напомним, что L^+ по определению равно L + LL + LLL + ... Поскольку $\mathrm{L}^* = \varepsilon + L + LL + LLL + \cdots$, *то* $\mathrm{LL}^* = \mathrm{L}\varepsilon + LL + LLL + \cdots$ Если учесть, что $\mathrm{L}\varepsilon = L$, то очевидно, что бесконечные разложения для LL^* и для L^+ совпадают. Это доказывает, что $\mathrm{L}^+ = \mathrm{LL}^*$. Доказательство того, что *L* = L*L,* аналогично.
- L* = L^+ + ε. Это легко доказать, поскольку в разложении L^+ присутствуют те же члены, что и в разложении L^*, за исключением цепочки ε.
- *L? = ε + L.* В действительности это правило является определением оператора "?".

Установление законов для регулярных выражений

- Каждый из вышеперечисленных законов был доказан, формально или неформально. Однако есть еще бесконечно много законов для регулярных выражений, которые можно было бы сформулировать. Существует ли некая общая методика, с помощью которой можно было бы легко доказывать истинность таких законов? Оказывается, что вопрос о справедливости некоторого закона сводится к вопросу о равенстве двух определенных языков.
- Любое регулярное выражение с переменными можно рассматривать как *конкретное* регулярное выражение, не содержащее переменных, если считать, что каждая переменная — это просто отдельный символ. Например, в выражении $(L + M)^*$ переменные L и M можно заменить символами *a* и *b,* соответственно, и получить регулярное выражение $(\mathrm{a} + \mathrm{b})^*$.
- Язык конкретного выражения дает представление о виде цепочек любого языка, образованного из исходного выражения в результате подстановки произвольных языков вместо переменных. Например, при анализе выражения $(L + M)^*$ можно отметить, что любая цепочка *w,* составленная из последовательности цепочек, выбираемых либо из *L,* либо из M, принадлежит языку $(L + M)^*$. Можно прийти к тому же заключению, рассмотрев язык конкретного выражения $\mathrm{L}\big((\mathrm{a} + \mathrm{b})^*\big)$, который, очевидно, представляет собой множество всех цепочек, состоящих из символов *a* и *b.* В одну из цепочек этого множества можно подставить любую цепочку из *L* вместо символа *a* и любую цепочку из *M* вместо символа b. При этом различные вхождения символов *a* и *b* можно заменять различными цепочками. Если такую подстановку осуществить для всех цепочек из $(\mathrm{a} + \mathrm{b})^*$, то в результате получим множество всех цепочек, образованных конкатенацией цепочек из *L* и/или *M* в любом порядке.

Теорема законов регулярных выражений

- Теорема. Пусть *E*— регулярное выражение с переменными $\mathrm{L}_1, \mathrm{L}_2, \cdots, \mathrm{L}_\mathrm{m}$. Построим конкретное регулярное выражение C, заменив каждое вхождение L_i символом $a_i, i = 1, 2, \cdots, m$. Тогда для произвольных языков $\mathrm{L}_1, \mathrm{L}_2, \cdots, \mathrm{L}_\mathrm{m}$ любую цепочку *w* из *L(E)* можно представить в виде $w = \mathrm{W}_1\mathrm{W}_2 \cdots \mathrm{W}_\mathrm{K}$, где каждая W_I принадлежит одному из этих языков, например L_{ji}, а цепочка $a_{j1}a_{j2} \cdots a_{jk}$ принадлежит языку *L(C).*
- Говоря менее формально, можно построить *L(E),* исходя из каждой цепочки языка *L(C),* скажем, $a_{j1}a_{j2} \cdots a_{jk}$, и заменяя в ней каждый из символов a_{ji} любой цепочкой из соответствующего языка L_{ji}.

Свойства регулярных языков

- Одними из важнейших свойств регулярных языков являются "свойства замкнутости". Эти свойства позволяют создавать распознаватели для одних языков, построенных из других с помощью определенных операций. Например, пересечение двух регулярных языков также является регулярным. Таким образом, при наличии автоматов для двух различных регулярных языков можно (механически) построить автомат, который распо знает их пересечение. Поскольку автомат для пересечения языков может содержать намного больше состояний, чем любой из двух данных автоматов, то "свойство замкнутости" может оказаться полезным инструментом для построения сложных автоматов.

Доказательство нерегулярности

В предыдущих разделах было установлено, что класс языков, известных как регулярные, имеет не менее четырех различных способов описания. Это языки, допускаемые ДКА, НКА и ε-НКА; их можно также определять с помощью регулярных выражений.

Не каждый язык является регулярным. В этом разделе предлагается мощная техника доказательства нерегулярности некоторых языков, известная как "лемма о накачке". Ниже приводится несколько примеров нерегулярных языков. Ниже лемма о накачке используется вместе со свойствами замкнутости регулярных языков для доказательства нерегулярности других языков.

Лемма о накачке для регулярных языков

- Рассмотрим язык $L_{01} = \{0^n 1^n | n \geq 1\}$. Этот язык состоит из всех цепочек вида 01, 0011, 000111 и так далее, содержащих один или несколько нулей, за которыми следует такое же количество единиц. Утверждается, что язык L_{01}, нерегулярен. Неформально, если бы L_{01} был регулярным языком, то допускался бы некоторым ДКА *A,* имеющим какое-то число состояний *k.* Предположим, что на вход автомата поступает *k* нулей. Он находится в некотором состоянии после чтения каждого из *к* + 1 префиксов входной цепочки, т.е. ε, 0, 00, ..., 0^k. Поскольку есть только *k* различных состояний, то прочитав два различных префикса, например, 0^i и 0^j, автомат должен находиться в одном и том же состоянии, скажем, *q.*
- Допустим, что, прочитав i или *j* нулей, автомат *A* получает на вход 1. По прочтении i единиц он должен допустить вход, если ранее получил i нулей, и отвергнуть его, получив *j* нулей. Но в момент поступления 1 автомат *A* находится в состоянии *q* и не способен "вспомнить", какое число нулей, i или j, было принято. Следовательно, его можно "обманывать" и заставлять работать неправильно, т.е. допускать, когда он не должен этого делать, или наоборот.
- Приведенное неформальное доказательство можно сделать точным. Однако к заключению о нерегулярности языка L_{01} приводит следующий общий результат.

Формулировка леммы

Теорема *(лемма о накачке для регулярных языков).* Пусть *L* — регулярный язык. Существует константа n (зависящая от L), для которой каждую цепочку w из языка L, удовлетворяющую неравенству $|w| \geq n$, можно разбить на три цепочки *w* = *xyz* так, что выполняются следующие условия.

- $y \neq \varepsilon$.
- $|xy| \leq n$.
- Для любого *к* > 0 цепочка $xy^k z$ также принадлежит *L.*

Это значит, что всегда можно найти такую цепочку *y* недалеко от начала цепочки *w,* которую можно "накачать". Таким образом, если цепочку y повторить любое число раз или удалить (при *к* = 0), то результирующая цепочка все равно будет принадлежать языку *L.*

Игровое представление леммы

Выше говорилось о том, что любую теорему, утверждение которой содержит несколько чередующихся кванторов "для всех" ("для любого") и "существует", можно представить в виде игры двух противников. Лемма о накачке служит примером теорем такого типа, так как содержит четыре разных квантора: **"для любого** регулярного языка *L* **существует** *n,* при котором **для всех** *w* из *L,* удовлетворяющих неравенству H > *n,* **существует** цепочка *xyz,* равная *w,* удовлетворяющая ...". Применение леммы о накачке можно представить в виде игры со следующими правилами.

- Игрок 1 выбирает язык *L,* нерегулярность которого нужно доказать.
- Игрок 2 выбирает n, но не открывает его игроку 1; первый игрок должен построить игру для всех возможных значений n.
- Игрок 1 выбирает цепочку *w,* которая может зависеть от *n,* причем ее длина должна быть не меньше n.
- Игрок 2 разбивает цепочку *w* на x, *y* и z, соблюдая условия леммы о накачке, т.е. $y \neq \varepsilon$ и $|xy| \leq n$. Опять-таки, он не обязан говорить первому игроку, чему равны *x, y* и z, хотя они должны удовлетворять условиям леммы.
- Первый игрок "выигрывает", если выбирает *k,* которое может быть функцией от n, *x, y* и z и для которого цепочка $xy^k z$ не принадлежит *L.*

Пример доказательства нерегулярности

Пример. Покажем, что язык L_{eq}, состоящий из всех цепочек с одинаковым числом нулей и единиц (расположенных в произвольном порядке), нерегулярен. В терминах игры, описанной выше, мы являемся игроком 1 и должны иметь дело с любыми допустимыми ходами игрока 2. Предположим, что n — это та константа, которая согласно лемме о накачке должна существовать, если язык L_{eq} регулярен, т.е. "игрок 2" выбирает *n*. Мы выбираем цепочку $w = 0^n 1^n$, которая наверняка принадлежит L_{eq}.

Теперь "игрок 2" разбивает цепочку *w* на *xyz*. Известно лишь, что $y \neq \varepsilon$ и $|xy| \leq n$. Но эта информация очень полезна, и игра "выигрывается" следующим образом. Поскольку $|xy| \leq n$, и цепочка *xy* расположена в начале цепочки *w*, то она состоит только из нулей. Если язык L_{eq} регулярен, то по лемме о накачке цепочка *xz* принадлежит L_{eq} (при *к* = 0 в лемме). Цепочка *xz* содержит *n* единиц, так как все единицы цепочки w попадают в z. Но в *xz* нулей меньше *n*, так как потеряны все нули из *y*. Поскольку $y \neq \varepsilon$, то вместе *x* и *z* содержат не более *n* - 1 нулей. Таким образом, предположив, что язык L_{eq} регулярен, приходим к ошибочному выводу, что цепочка xz принадлежит L_{eq}. Следовательно, методом от противного доказано, что язык L_{eq} нерегулярен.

Пример доказательства нерегулярности 2

Пример. Докажем нерегулярность языка L_{pr}, образованного всеми цепочками из единиц, длины которых — простые числа. Предположим, что язык L_{pr} регулярен. Тогда должна существовать константа n, удовлетворяющая условиям леммы о накачке. Рассмотрим некоторое простое число $p \geq n + 2$. Такое *p* должно существовать, поскольку множество простых чисел бесконечно. Пусть $w = 1^p$.

Согласно лемме о накачке можно разбить цепочку *w = xyz* так, что $y \neq \varepsilon$ и $|xy| \leq n$. Пусть $y = |m|$. Тогда $|xy| = p - m$. Рассмотрим цепочку $xy^{p-m}z$, которая по лемме о накачке должна принадлежать языку L_{pr}, если он действительно регулярен. Однако

$$|xy^{p-m}z| = |xy| + (\mathrm{p} - \mathrm{m})|\mathrm{y}| = \mathrm{p} - \mathrm{m} + (\mathrm{p} - \mathrm{m})\mathrm{m} = (\mathrm{m} + 1)(\mathrm{p} - \mathrm{m})$$

Очевидно, что число $|xy^{p-m}z|$ не простое, так как имеет два множителя m+1 и p-m. Однако нужно еще убедиться, что ни один из этих множителей не равен 1, потому что тогда число *(m+1)(p-m)* будет простым. Из неравенства $y \neq \varepsilon$ следует *m* > 1 и *m* + 1 > 1. Кроме того, $m = |y| \leq |xy| \leq \mathrm{n}$, а *p* > *п* + 2, поэтому *p-m >2.*

Начав с предположения, что предлагаемый язык регулярен, пришли к противоречию, доказав, что существует некоторая цепочка, которая не принадлежит этому языку, тогда как по лемме о накачке она должна ему принадлежать. Таким образом, язык L_{pr} нерегулярен.

Свойства замкнутости регулярных языков

- Свойства замкнутости регулярных языков позволяют создавать новые регулярные языки при помощи определённых операций над языками, регулярность которых уже доказана. Свойства замкнутости описываются в соответствующей презентации

Свойства разрешимости регулярных языков

- Сформируем важные вопросы, связанные с регулярными языками. Сначала нужно разобраться, что значит задать вопрос о некотором языке. Типичный язык бесконечен, поэтому бессмысленно предъявлять кому-нибудь цепочки этого языка и задавать вопрос, требующий проверки бесконечного множества цепочек. Гораздо ра зумнее использовать одно из конечных представлений языка, а именно: ДКА, НКА, ε-НКА или регулярное выражение.
- Очевидно, что представленные таким образом языки будут регулярными. В действи тельности не существует способа представления абсолютно произвольных языков. Ниже предлагаются конечные методы описания более широких классов, чем класс регулярных языков, и можно будет рассматривать вопросы о языках из них. Однако алгоритмы разрешения многих вопросов о языках существуют только для класса регулярных языков. Эти же вопросы становятся "неразрешимыми" (не существует алгоритмов ответов на эти вопросы), если они поставлены с помощью более "выразительных" обозначений (используемых для выражения более широкого множества языков), чем представления, разработанные для регулярных языков.
- Начнем изучение алгоритмов для вопросов о регулярных языках, рассмотрев спосо бы, которыми одно представление языка преобразуется в другое. В частности, рассмотрим временную сложность алгоритмов, выполняющих преобразования. Затем рассмотрим три основных вопроса о языках.
- Является ли описываемый язык пустым множеством?
- Принадлежит ли некоторая цепочка ***w*** представленному языку?
- Действительно ли два разных описания представляют один и тот же язык? (Этот во прос часто называют "эквивалентностью" языков.)

Преобразования типов представлений языков

- Выше было показано, что каждое из четырех представлений регулярных языков можно преобразовать в любое из остальных трех. Хотя существуют алгоритмы для любого из этих преобразований, иногда интересует не только осуществимость некоторого преобразования, но и время, необходимое для его выполнения. В частности, важно отличать алгоритмы, которые занимают экспоненциальное время (время как функция от размера входных данных) и, следовательно, могут быть выполнены только для входных данных сравнительно небольших размеров, от тех алгоритмов, время выполнения которых является линейной, квадратичной или полиномиальной с малой степенью функцией от размера входных данных. Последние алгоритмы "реалистичны", так как их можно выполнить для гораздо более широкого класса экземпляров задачи. Рассмотрим временную сложность каждого из преобразований.

Преобразование НКА в ДКА слайд 1/2

- Время выполнения преобразования НКА или ε-НКА в ДКА может быть экспоненциальной функцией от количества состояний НКА. Начнем с того, что вычисление ε-замыкания *n* состояний занимает время $O(n^3)$. Необходимо найти все дуги с меткой ε, ведущие от каждого из n состояний. Если есть n состояний, то может быть не более n^2 дуг. Разумное использование системных ресурсов и хорошо спроектированные структуры данных гарантируют, что время исследования каждого состояния не превысит $O(n^2)$. В действительности для однократного вычисления всего е-замыкания можно использовать такой алгоритм транзитивного замыкания, как алгоритм Уоршалла.
- После вычисления ε-замыкания можно перейти к синтезу ДКА с помощью конструкции подмножеств. Основное влияние на расход времени оказывает количество состояний ДКА, которое может равняться 2^n. Для каждого состояния можно вычислить переходы за время $O(n^3)$, используя ε-замыкание и таблицу переходов НКА для каждого входного символа. Предположим, нужно вычислить $\delta(\{q_1, q_2, \cdots, q_n\}, a)$ для ДКА. Из каждого состояния q_i, можно достичь не более *n* состояний вдоль путей с меткой ε, и каждое из этих состояний может иметь не более, чем *n* дуг с меткой *a*. Создав массив, проиндексированный состояниями, можно вычислить объединение не более n множеств, состоящих из не более, чем n состояний, за время, пропорциональное n^2.

Преобразование НКА в ДКА слайд 2/2

- Таким способом для каждого состояния q, можно вычислить множество состояний, достижимых из q_i, вдоль пути с меткой a (возможно, включая дуги, отмеченные ε). Поскольку $k \leq n$, то существует не более n таких состояний q_i и для каждого из них вычисление достижимых состояний занимает время $O(n^2)$. Таким образом, общее время вычисления достижимых состояний равно $O(n^3)$. Для объединения множеств достижимых состояний потребуется только $O(n^2)$ дополнительного времени, следовательно, вычисление одного перехода ДКА занимает время $O(n^3)$.
- Заметим, что количество входных символов считается постоянным и не зависит от n. Таким образом, как в этой, так и в других оценках времени работы количество входных символов не рассматривается. Размер входного алфавита влияет только на постоянный коэффициент, скрытый в обозначении "О большого".
- Итак, время преобразования НКА в ДКА, включая ситуацию, когда НКА содержит е- переходы, равно $O(n^3 2^n)$. Конечно, на практике обычно число состояний, которые строятся, намного меньше 2^n. Иногда их всего лишь n. Поэтому можно установить оценку времени работы равной $O(n^3 s)$, где s — это число состояний, которые в действительности есть у ДКА.

Преобразование ДКА в НКА

- Это простое преобразование, занимающее время $O(n)$ для ДКА с *n* состояниями. Все, что необходимо сделать, — изменить таблицу переходов для ДКА, заключив в скобки {} состояния, а также добавить столбец для ε, если нужно получить ε-НКА. Поскольку число входных символов (т.е. ширина таблицы переходов) считается постоянным, копирование и обработка таблицы занимает время $O(n)$.

Преобразование автомата в регулярное выражение

- Рассмотрев алгоритм преобразования, можно отметить, что на каждом из n этапов (где n — число состояний ДКА) размер конструируемого регулярного выражения может увеличиться в четыре раза, так как каждое выражение строится из четырех выражений предыдущего цикла. Таким образом, простая запись n^3 выражений может занять время $O\left(n^3 4^n\right)$. Оптимизация алгоритма уменьшает постоянный коэффициент, но не влияет на экспоненциальность этой задачи (в наихудшем случае).
- Аналогичная конструкция требует такого же времени работы, если преобразуется НКА, или даже ε-НКА. Однако использование конструкции для НКА имеет большое значение. Если сначала преобразовать НКА в ДКА, а затем ДКА — в регулярное выражение, то на это потребуется время $\mathrm{O}\left(\mathrm{n}^3 4^{\mathrm{n}^3 2\mathrm{n}}\right)$, которое является дважды экспоненциальным.

Преобразование регулярного выражения в автомат

- Для преобразования регулярного выражения в ε-НКА потребуется линейное время. Необходимо эффективно проанализировать регулярное выражение, используя метод, занимающий время $O(n)$ для регулярного выражения длины n. В результате получим дерево с одним узлом для каждого символа регулярного выражения (хотя скобки в этом дереве не встречаются, поскольку они только управляют разбором выражения).
- Полученное дерево заданного регулярного выражения можно обработать, конструируя ε-НКА для каждого узла. Правила преобразования регулярного выражения, представленные выше, никогда не добавляют более двух состояний и четырех дуг для каждого узла дерева выражения. Следовательно, как число состояний, так и число дуг результирующего ε-НКА равны $O(n)$. Кроме того, работа по созданию этих элементов в каждом узле дерева анализа является постоянной при условии, что функция, обрабатывающая каждое поддерево, возвращает указатели в начальное и допускающие состояния этого автомата.
- Приходим к выводу, что построение ε-НКА по регулярному выражению занимает время, линейно зависящее от размера выражения. Можно исключить ε-переходы из ε-НКА с ***n*** состояниями, преобразовав его в обычный НКА за время $O(n^3)$ и не увеличив числа состояний. Однако преобразование в ДКА может занять экспоненциальное время.

Проверка пустоты регулярных языков

- На первый взгляд ответ на вопрос "является ли регулярный язык *L* пустым?" кажется очевидным: язык 0 пуст, а все остальные регулярные языки — нет. Однако при постановке задачи явный перечень цепочек языка *L* не приводится. Обычно задается некоторое представление языка *L,* и нужно решить, обозначает ли оно язык $\varnothing$, или нет.
- Если язык задан с помощью конечного автомата любого вида, то вопрос пустоты состоит в том, есть ли какие-нибудь пути из начального состояния в допускающие. Если есть, то язык непуст, а если все допускающие состояния изолированы от начального, то язык пуст. Ответ на вопрос, можно ли перейти в допускающее состояние из начального, является простым примером достижимости в графах, подобным вычислению ε-замыкания.

Алгоритм проверки слайд 1/2

- Искомый алгоритм можно сформулировать следующим рекурсивным образом:
- Базис. Начальное состояние всегда достижимо из начального состояния.
- Индукция. Если состояние q достижимо из начального, и есть дуга из q в состояние p с любой меткой (входным символом или ε, если рассматривается e-НКА), то p также достижимо.
- Таким способом можно вычислить все множество достижимых состояний. Если среди них есть допускающее состояние, то ответом на поставленный вопрос будет "нет" (язык данного автомата *не* пуст), в противном случае ответом будет "да" (язык пуст). Заметим, что если автомат имеет n состояний, то вычисление множества достижимых состояний занимает время не более $O(n^2)$ (практически это время пропорционально числу дуг на диаграмме переходов автомата, которое может быть и меньше n^2).

Алгоритм проверки слайд 2/2

- Если язык L представлен регулярным выражением, а не автоматом, то можно преобразовать это выражение в ε-НКА, а далее продолжить так, как описано выше. Поскольку автомат, полученный в результате преобразования регулярного выражения длины *n,* содержит не более $O(n)$ состояний и переходов, для выполнения алгоритма потребуется время $O(n)$**.**
- Можно проверить само выражение — пустое оно, или нет. Сначала заметим, что если в данном выражении ни разу не встречается ∅, то его язык гарантированно не пуст. Если же в выражении встречается ∅, то язык такого выражения не обязательно пустой. Используя следующие рекурсивные правила, можно определить, представляет ли заданное регулярное выражение пустой язык.
- Базис. ∅ обозначает пустой язык, но ε и **a** для любого входного символа *a* обозначают не пустой язык.
- Индукция. Пусть *R* — регулярное выражение. Нужно рассмотреть четыре варианта, соответствующие возможным способам построения этого выражения.
- $R = R_1 + R_2$. $L(R)$ пуст тогда и только тогда, когда оба языка $L(R_1)$ и $L(R_2)$ пусты.
- $R = R_1 R_2$. $L(R)$ пуст тогда и только тогда, когда хотя бы один из языков $L(R_1)$ или $L(R_2)$ пуст.
- $R = R_1^* R_2$. $L(R)$ не пуст: он содержит цепочку ε.
- $R = R_1$. $L(R)$ пуст тогда и только тогда, кода $L(R_1)$ пуст, так как эти языки равны.

Проверка принадлежности регулярному языку слайд 1/2

- Следующий важный вопрос состоит в том, принадлежит ли данная цепочка *w* данному регулярному языку *L.* В то время, как цепочка *w* задается явно, язык *L* представляется с помощью автомата или регулярного выражения.
- Если язык *L* задан с помощью ДКА, то алгоритм решения данной задачи очень прост. Имитируем ДКА, обрабатывающий цепочку входных символов w, начиная со стартового состояния. Если ДКА заканчивает в допускающем состоянии, то цепочка w принадлежит этому языку, в противном случае— нет. Этот алгоритм является предельно быстрым. Если $|w| = n$ и ДКА представлен с помощью подходящей структуры данных, например, двухмерного массива (таблицы переходов), то каждый переход требует постоянного времени, а вся проверка занимает время ***0(n).***

Проверка принадлежности регулярному языку слайд 2/2

- Если *L* представлен способом, отличным от ДКА, то преобразуем это представление в ДКА и применим описанную выше проверку. Такой подход может занять время, экспоненциально зависящее от размера данного представления и линейное относительно $|w|$. Однако, если язык задан с помощью НКА или ε-НКА, то намного проще и эффективнее непосредственно проимитировать этот НКА. Символы цепочки *w* обрабатываются по одному, и запоминается множество состояний, в которые НКА может попасть после прохождения любого пути, помеченного префиксом цепочки *w.*
- Если длина цепочки *w* равна *n,* а количество состояний НКА равно *s,* то время работы этого алгоритма равно $O(ns^2)$. Чтобы обработать очередной входной символ, необходимо взять предыдущее множество состояний, число которых не больше *s,* и для каждого из них найти следующее состояние. Затем объединяем не более *s* множеств, состоящих из не более, чем *s* состояний, для чего нужно время $O(s^2)$.
- Если заданный НКА содержит ε-переходы, то перед тем, как начать имитацию, необходимо вычислить ε-замыкание. Такая обработка очередного входного символа *a* состоит из двух стадий, каждая из которых занимает время $O(s^2)$. Сначала для предыдущего множества состояний находим последующие состояния при входе *a.* Далее вычисляем ε-замыкание полученного множества состояний. Начальным множеством состояний для такого моделирования будет ε-замыкание начального состояния НКА.
- И наконец, если язык *L* представлен регулярным выражением, длина которого *s,* то за время *0(s)* можно преобразовать это выражение в ε-НКА с числом состояний не больше *2s.* Выполняем описанную выше имитацию, что требует $O(ns^2)$ времени для входной цепочки *w* длиной *n.*

Эквивалентность и минимизация автоматов

- В отличие от предыдущих вопросов — пустоты и принадлежности, алгоритмы решения которых были достаточно простыми, вопрос о том, определяют ли два представления двух регулярных языков один и тот же язык, требует значительно больших интеллектуальных усилий. В текущем разделе рассмотрим, как проверить, являются ли два описания регулярных языков *эквивалентными* в том смысле, что они задают один и тот же язык. Важным следствием этой проверки является возможность минимизации ДКА, т.е. для любого ДКА можно найти эквивалентный ему ДКА с минимальным количеством состояний. По существу, такой ДКА один: если даны два эквивалентных ДКА с минимальным числом состояний, то всегда можно переименовать состояния так, что эти ДКА станут одинаковыми. Определения, алгоритмы и примеры приведены в соответствующей презентации.

Контекстно-свободные грамматики

- Выше в презентации были рассмотрены регулярные языки. В этом разделе рассмотрим более широкий класс языков, которые называются контекстно-свободными (КС-грамматиками). Они имеют естественное рекурсивное описание в виде контекстно-свободных грамматик. Эти грамматики играют главную роль в технологии компиляции с начала 1960-х годов; они превратили непростую задачу реализации синтаксических анализаторов, распознающих структуру программы, из неформальной в рутинную, которую можно решить за один вечер. Позже контекстно-свободные грамматики ста ли использоваться для описания форматов документов в виде так называемых определений типа документов (document-type definition — DTD), которые применяются в языке XML (extensible markup language) для обмена информацией в Internet.

Неформальный пример КС-грамматики слайд 1/2

- Рассмотрим язык палиндромов. *Палиндром* — это цепочка, читаемая одинаково слева направо и наоборот. Другими словами, цепочка w является палиндромом тогда и только тогда, когда $w = w^R$. Для упрощения рассмотрим описание палиндромов только в алфавите {0, 1}. Этот язык включает цепочки вроде 0110, 11011, ε, но не включает 011 или0101.
- При помощи леммы о накачке нетрудно проверить, что язык $\mathrm{L}_{\mathrm{pal}}$ палиндромов из символов 0 и 1 не является регулярным.
- Существует следующее естественное рекурсивное определение того, что цепочка из символов 0 и 1 принадлежит языку $\mathrm{L}_{\mathrm{pal}}$. Оно начинается с базиса, утверждающего, что несколько очевидных цепочек принадлежат $\mathrm{L}_{\mathrm{pal}}$ а затем использует идею того, что если цепочка является палиндромом, то она должна начинаться и заканчиваться одним и тем же символом. Кроме того, после удаления первого и последнего символа остаток цепочки также должен быть палиндромом.
- Базис: ε, 0 и 1 являются палиндромами.
- Индукция: Если w — палиндром, то $0w0$ и $1w1$ — также палиндромы. Ни одна цепочка не является палиндромом, если не определяется этими базисом и индукцией.

Неформальный пример КС-грамматики слайд 2/2

- Контекстно-свободная грамматика представляет собой формальную запись подобных рекурсивных определений языков. Грамматика состоит из одной или нескольких переменных, которые представляют классы цепочек, или языки. В данном примере нужна только одна переменная, представляющая множество палиндромов, т.е. класс цепочек, образующих язык L_{pal}. Имеются правила построения цепочек каждого класса. При построении используются символы алфавита и уже построенные цепочки из различных классов.

Пример. Правила определения палиндромов, выраженные в виде контекстно-свободной грамматики, представлены ниже.

$$1.\ P \to \varepsilon$$
$$2.\ P \to 0$$
$$3.\ P \to 1$$
$$4.\ P \to 0P0$$
$$5.\ P \to 1P1$$

Определение КС-грамматики

- Описание языка с помощью грамматики состоит из следующих четырех важных компонентов.
  - Есть конечное множество символов, из которых состоят цепочки определяемого языка. В примере о палиндромах это было множество {0, 1}. Его символы называются *терминальными,* или *терминалами.*
  - Существует конечное множество переменных, называемых иногда также *нетерминалами,* или *синтаксическими категориями.* Каждая переменная представляет язык, т.е. множество цепочек. В рассмотренном примере была только одна переменная, P, которая использовалась для представления класса палиндромов в алфавите {0, 1}.
  - Одна из переменных представляет определяемый язык; она называется *стартовым символом.* Другие переменные представляют дополнительные классы цепочек, которые помогают определить язык, заданный стартовым символом.
  - Имеется конечное множество *продукций,* или *правил вывода,* которые представляют рекурсивное определение языка. Каждая продукция состоит из следующих частей:
    - а) переменная, (частично) определяемая продукцией. Эта переменная часто называется *головой* продукции;
    - б) символ продукции →
    - в) конечная цепочка, состоящая из терминалов и переменных, возможно, пустая. Она называется *телом* продукции и представляет способ образования цепочек языка, обозначаемого переменной в голове. По этому способу мы оставляем терминалы неизменными и вместо каждой переменной в теле подставляем любую цепочку, про которую известно, что она принадлежит языку этой переменной.
- Пример множества продукций был приведён на предыдущем слайде.
- Описанные четыре компонента образуют *контекстно-свободную грамматику,* или *КС-грамматику,* или просто *грамматику,* или *КСГ.* Мы будем представлять КС- грамматику *G* ее четырьмя компонентами в виде $\mathrm{G} = (\mathrm{V}, \mathrm{T}, \mathrm{P}, \mathrm{S})$, где *V*— множество переменных, *T*— терминалов, *P* — множество продукций, *S* — стартовый символ.
- Пример. Грамматика $\mathrm{G}_{\mathrm{pal}}$ для палиндромов имеет вид $\mathrm{G}_{\mathrm{pal}} = (\{\mathrm{P}\}, \{0,1\}, \mathrm{A}, \mathrm{P})$, *где A* обозначает множество из пяти продукций (см. слайд 126).

КС-грамматика для выражений, слайд 1/2

Рассмотрим более сложную КС-грамматику, которая представляет выражения упрощенного языка программирования (в несколько упрощенном виде).

Во-первых, ограничимся операторами + и *, представляющими сложение и умножение.

Во-вторых, допустим, что аргументы могут быть идентификаторами, однако не произвольными, т.е. буквами, за которыми следует любая последовательность букв и цифр, в том числе пустая. Допустим только буквы *a u b u* цифры 0 и 1. Каждый идентификатор должен начинаться с буквы *a* или *b,* за которой следует цепочка из $\{\mathrm{a}, \mathrm{b}, 0,1\}^*$.

В рассматриваемой грамматике нужны две переменные. Одна обозначается E и представляет выражения определяемого языка. Она является стартовым символом. Другая переменная, I, представляет идентификаторы. Ее язык в действительности регулярен и задается регулярным выражением $(a + b)(a + b + 0 + 1)^*$.

В грамматиках, однако, регулярные выражения непосредственно не используются. Вместо этого будем обращаться к множеству продукций, задающих в точности то же самое, что и соответствующее регулярное выражение.

$$1.\ E \to I \qquad 2.\ E \to E + E \qquad 3.\ E \to E * E$$
$$4.\ E \to (E) \qquad 5.\ I \to a \qquad 6.\ I \to b$$
$$7.\ I \to Ia \qquad 8.\ I \to Ib \qquad 9.\ I \to I0$$
$$10.\ I \to I1$$

Грамматика для выражений определяется формально как $\mathrm{G} = (\{\mathrm{E}, \mathrm{I}\}, \mathrm{T}, \mathrm{P}, \mathrm{E})$, где $T = \{+, *, (,), a, b, 0,1\}$, а P представляет собой множество продукций, показанное на этом слайде.

Сокращённая запись продукций

- Продукцию удобно рассматривать как "принадлежащую" переменной в ее голове. Мы будем часто пользоваться словами "продукции для A" или "A-продукции" для обозначения продукций, головой которых является переменная *A*. Продукции грамматики можно записать, указав каждую переменную один раз и затем перечислив все тела продукций для этой переменной, разделяя их вертикальной чертой. Таким образом, продукции $A \to \alpha_1, \cdots, A \to \alpha_n$ можно заменить записью $\mathrm{A} \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$. Например, грамматику для палиндромов можно записать в виде $\mathrm{P} \to \varepsilon | 0 | 1 | 0P0 | 1P1$.

Порождения с использованием грамматики

- Для того чтобы убедиться, что данные цепочки принадлежат языку некоторой переменной, применяются продукции КС-грамматики. Есть два способа действий. Простейший подход состоит в применении правил "от тела к голове". Берутся цепочки, про которые известно, что они принадлежат языкам каждой из переменных в теле правила, они записываются в соответствующем порядке вместе с терминалами этого тела и делается вывод о том, что полученная цепочка принадлежит языку переменной в голове. Такая процедура называется *рекурсивным выводом* (recursive inference).
- Есть еще один способ определения языка грамматики, по которому продукции применяются "от головы к телу". Стартовый символ разворачивается, используя одну из его продукций, т.е. продукцию, головой которой является этот символ. Затем разворачивается полученная цепочка, заменяя одну из переменных телом одной из ее продукций, и так далее, пока не будет получена цепочка, состоящую из одних терминалов. Язык грамматики— это все цепочки из терминалов, получаемые таким способом. Такое использование грамматики называется *выведением,* или *порождением* (derivation).

Пример рекурсивного вывода

- Результаты этих выводов показаны ниже. Например, строчка (i) говорит, что в соответствии с продукцией 5 цепочка *a* принадлежит языку переменной I. Строчки (ii)—(iv) свидетельствуют, что цепочка b00 является идентификатором, полученным с помощью одного применения продукции 6 и затем двух применений продукции 9.

	Выведенная цепочка	Для языка переменной	Использованная продукция	Использованные цепочки
1	a	I	5	
2	b	I	6	
3	b0	I	9	(ii)
4	b00	I	9	(iii)
5	a	I	1	(i)
6	b00	E	1	(iv)
7	a+b00	E	2	(v), (vi)
8	(a+b00)	E	4	(vii)
9	a*(a+b00)	E	3	(v), (vii)

Отношение порождения

- Процесс порождения цепочек путем применения продукций "от головы к телу" требует определения нового символа отношения ⇒. Пусть *G*= (*V,T,P,S*)— КС-грамматика. Пусть $\alpha A\beta$— цепочка из терминалов и переменных, где *A* — переменная. Таким образом, α и β— цепочки из $(V \cup T)^*$, $A \in V$. Пусть $A \to \gamma$ — продукция из *G.* Тогда мы говорим, что $\alpha A\beta \underset{G}{\Rightarrow} \alpha\gamma\beta$. Если грамматика *G* подразумевается, то это записывается просто как $\alpha A\beta \Rightarrow \alpha\gamma\beta$. Заметим, что один шаг порождения заменяет любую переменную в цепочке телом одной из ее продукций.
- Для представления нуля, одного или нескольких шагов порождения можно расширить отношение ⇒ подобно тому, как функция переходов δ расширялась до $\hat{\delta}$. Для обозначения нуля или нескольких шагов порождения используется символ $*$, а расширенное отношение обозначим символом $\overset{*}{\Rightarrow}$.

Пример порождения

Пример. Вывод о том, что *a*(a +* **b00)** принадлежит языку переменной E, можно отразить в порождении этой цепочки, начиная с E. Запишем одно из таких порождений $\mathrm{E} \Rightarrow \mathrm{E} * \mathrm{E} \Rightarrow \mathrm{I} * \mathrm{E} \Rightarrow \mathrm{a} * \mathrm{E} \Rightarrow \mathrm{A} * (\mathrm{E}) \Rightarrow \mathrm{a} * (\mathrm{E} + \mathrm{E}) \Rightarrow \mathrm{a} * (\mathrm{I} + \mathrm{E}) \Rightarrow \mathrm{a} * (\mathrm{a} + \mathrm{E}) \Rightarrow \mathrm{a} * (\mathrm{a} + \mathrm{I}) \Rightarrow \mathrm{a} * (\mathrm{a} + \mathrm{I0}) \Rightarrow \mathrm{a} * (\mathrm{a} + \mathrm{I00}) \Rightarrow \mathrm{a} * (\mathrm{a} + \mathrm{b00})$. На первом шаге E заменяется телом продукции 3 (см. слайд 8). На втором шаге применяется продукция 1 для изменения E на I и т.д. Заметим, что в процессе систематически выдерживалась тактика замены крайней слева переменной в цепочке. На каждом шаге, однако, можно произвольно выбирать переменную для замены и использовать любую из продукций для этой переменной. Например, на втором шаге можно изменить второе E на (E), используя продукцию 4. В этом случае $E * E \Rightarrow E * (E)$. Также можно было бы выбрать замену, не приводящую к той же самой цепочке терминалов. Простым примером было бы использование продукции 2 на первом же шаге, в результате чего $E \Rightarrow E + E$, но теперь никакая замена переменных E не превратит цепочку $+E$ *в* $\mathrm{a} * (\mathrm{a} + \mathrm{b00})$**.**

Можно использовать отношение $\overset{*}{\Rightarrow}$ для сокращения порождения. Известно, что $E \overset{*}{\Rightarrow} E$. Несколько использований продукций дают утверждения $E \overset{*}{\Rightarrow} E * E$, $E \overset{*}{\Rightarrow} I * E$ и так далее до заключительного $E \overset{*}{\Rightarrow} a * (a + b00)$.

Две точки зрения — рекурсивный вывод и порождение — являются эквивалентными. Таким образом, можно заключить, что цепочка терминалов w принадлежит языку некоторой переменной *A* тогда и только тогда, когда $A \overset{*}{\Rightarrow} w$.

Левые и правые порождения слайд 1/2

- Для ограничения числа выборов в процессе порождения цепочки полезно потребовать, чтобы на каждом шаге мы заменяли крайнюю слева переменную одним из тел ее продукций. Такое порождение называется *левым* (leftmost), и для его указания используются отношения $\overset{lm}{\Rightarrow}$ и $\overset{lm*}{\Longrightarrow}$. Если используемая грамматика G не очевидна из контекста, то ее имя G также добавляется внизу.
- Аналогично можно потребовать, чтобы на каждом шаге заменялась крайняя справа переменная на одно из тел. В таком случае порождение называется *правым* (rightmost), и для его обозначения используются символы $\overset{rm}{\Rightarrow}$ и $\overset{rm*}{\Longrightarrow}$. Имя используемой грамматики также при необходимости записывается внизу.
- **Пример.** Порождение из примера на слайде 13 в действительности было левым, и его можно представить следующим образом. $E \overset{lm}{\Rightarrow} E * E \overset{lm}{\Rightarrow} I * E \overset{lm}{\Rightarrow} a * E \overset{lm}{\Rightarrow} A * (E) \overset{lm}{\Rightarrow} a * (E + E) \overset{lm}{\Rightarrow} a * (I + E) \overset{lm}{\Rightarrow} a * (a + E) \overset{lm}{\Rightarrow} a * (a + I) \overset{lm}{\Rightarrow} a * (a + I0) \overset{lm}{\Rightarrow} a * (a + I00) \overset{lm}{\Rightarrow} a * (a + b00)$
- Можно резюмировать левое порождение в виде $E \overset{lm*}{\Longrightarrow} a * (a + b00)$ или записать несколько его шагов.

Левые и правые порождения слайд 2/2

- Следующее правое порождение использует те же самые замены для каждой переменной, хотя и в другом порядке. $E \overset{rm}{\Rightarrow} E * E \overset{rm}{\Rightarrow} E * (E) \overset{rm}{\Rightarrow} E * (E + E) \overset{rm}{\Rightarrow} E * (E + I) \overset{rm}{\Rightarrow} E * (E + I0) \overset{rm}{\Rightarrow} E * (E + I00) \overset{rm}{\Rightarrow} E * (E + b00) \overset{rm}{\Rightarrow} E * (I + b00) \overset{rm}{\Rightarrow} E * (a + b00) \overset{rm}{\Rightarrow} a * (a + b00)$. Данное порождение также позволяет заключить, что $E \overset{rm\,*}{\Longrightarrow} a * (a + b00)$.
- Любое порождение имеет эквивалентные левое и правое порождения. Это означает, что если w — терминальная цепочка, а A — переменная, то $A \overset{*}{\Rightarrow} w$ тогда и только тогда, когда $A \overset{lm\,*}{\Longrightarrow} w$, а также $A \overset{*}{\Rightarrow} w$ тогда и только тогда, когда $A \overset{rm\,*}{\Longrightarrow} w$.

Обозначения для порождений

- Строчные буквы из начала алфавита (a, b, и т.д.) являются терминальными символами. Будем предполагать, что цифры н другие символы типа знака + или круглых скобок — также терминалы.
- Прописные начальные буквы (A, B и т.д.) являются переменными.
- Строчные буквы из конца алфавита, такие как w или z, обозначают цепочки терминалов. Это соглашение напоминает, что терминалы аналогичны входным символам конечного автомата.
- Прописные буквы из конца алфавита, вроде X или Y, могут обозначать как терминалы, так и переменные.
- Строчные греческие буквы (α, β и т.д.) обозначают цепочки, состоящие из терминалов и или переменных.
- Для цепочек, состоящих только из переменных, специального обозначения нет, поскольку это понятие не имеет большого значения. Вместе с тем, цепочка, обозначенная α или другой греческой буквой, возможно, состоит только из переменных

Язык грамматики и выводимые цепочки

Если *G = (V, T,P,S*)— КС-грамматика, то язык, *задаваемый грамматикой G,* или *язык грамматики G,* обозначается *L(G)* и представляет собой множество терминальных цепочек, порождаемых из стартового символа. Таким образом,

$$L(G) = \left\{ w \in T^* | S \overset{*G}{\Rightarrow} w \right\}$$

Если язык *L* задается некоторой КС-грамматикой, то он называется *контекстно-свободным,* или *КС-языком.* Например, мы утверждали, что грамматика, представленная выше, определяет множество палиндромов в алфавите {0, 1}. Таким образом, множество палиндромов является КС-языком.

Порождения из стартового символа грамматики приводят к цепочкам, имеющим особое значение. Они называются "выводимыми цепочками" ("sentential form"). Итак, если

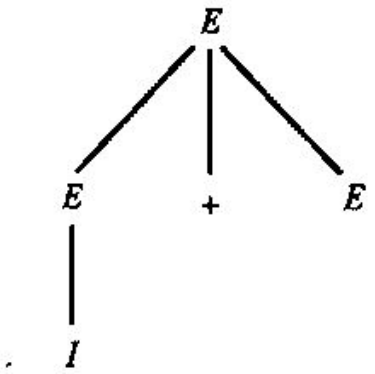
G = (V, T, P, S) — КС-грамматика, то любая цепочка α из $(V \cup T)^*$, для которой $S \underset{lm}{\Rightarrow} \alpha$, называется *выводимой цепочкой.* Если $S \underset{lm}{\Rightarrow} \alpha$, то а является *левовыводимой цепочкой,* а если $S \overset{rm}{\Rightarrow} \alpha$, то — *правовыводимой.* Заметим, что язык *L(G)* образуют выводимые цепочки из T*, состоящие исключительно из терминалов

Деревья разбора

- Для порождений существует чрезвычайно полезное представление в виде дерева. Это дерево наглядно показывает, каким образом символы цепочки группируются в подцепочки, каждая из которых принадлежит языку одной из переменных грамматики. Возможно, более важно то, что дерево, известное в компиляции как "дерево разбора", явля ется основной структурой данных для представления исходной программы. В компиляторе древовидная структура исходной программы облегчает ее трансляцию в исполняемый код за счет того, что допускает естественные рекурсивные функции для выполнения этой трансляции.

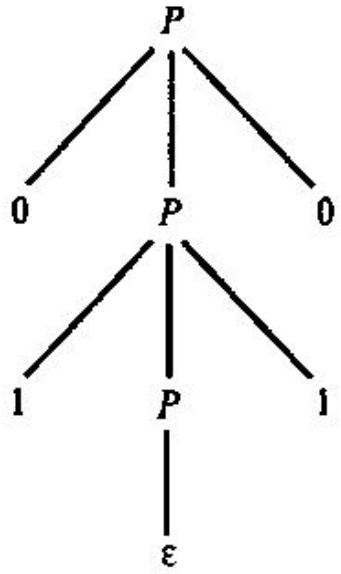
Построение дерева разбора

- Будем рассматривать грамматику ***G = (V, T, P, S). Деревья разбора*** для ***G*** — это деревья со следующими свойствами:
- Каждый внутренний узел отмечен переменной из ***V.***
- Каждый лист отмечен либо переменной, либо терминалом, либо ε. При этом, если лист отмечен ε, он должен быть единственным сыном своего родителя.
- Если внутренний узел отмечен ***A,*** и его сыновья отмечены слева направо $\boldsymbol{X_1, X_2, \cdots, X_k}$, соответственно, то $\boldsymbol{A \to X_1, X_2, \cdots, X_k}$ является продукцией в ***P.*** Отметим, что X может быть $\boldsymbol{\varepsilon}$ лишь в одном случае— если он отмечает единственного сына, и $\boldsymbol{A \to \varepsilon}$ — продукция грамматики ***G.***
- **Пример.** На рисунке показано дерево разбора, которое использует грамматику выражений (см. слайд 8). Корень отменен переменной E. В корне применена продукция $E \to E + E$, поскольку три сына корня отмечены слева направо как E, +, E. В левом сыне корня применена продукция $E \to I$, так как у этого узла один сын, отмеченный переменной I.



Пример дерева разбора

- На рисунке показано дерево разбора для грамматики палиндромов (см. слайд 126). В корне применена продукция $P \to 0P0$, а в среднем сыне корня — $P \to 1P1$. Отметим, что внизу использована продукция $P \to \varepsilon$. Это использование, при котором у узла есть сын с отметкой ε, является единственным случаем, когда в дереве может быть узел, отмеченный ε.



Крона дерева разбора

Если посмотреть на листья любого дерева разбора и выписать их отметки слева направо, то можно получить цепочку, которая называется *кроной дерева* и всегда является цепочкой, выводимой из переменной, отмечающей корень.

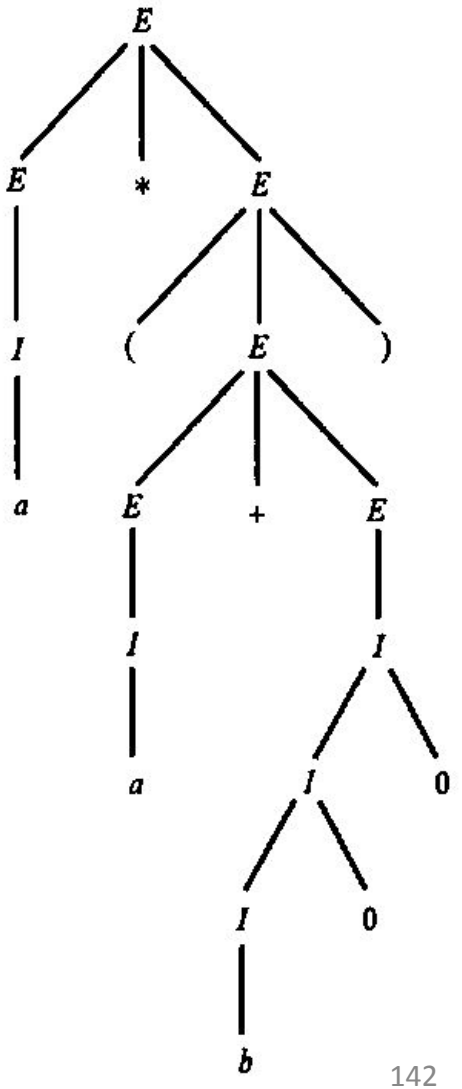
Особый интерес представляют деревья разбора со следующими свойствами:

- Крона является терминальной цепочкой, т.е. все листья отмечены терминалами или ε.
- Корень отмечен стартовым символом.

Кроны таких деревьев разбора представляют собой цепочки языка рассматриваемой грамматики. Также можно доказать, что еще один способ описания языка грамматики состоит в определении его как множества крон тех деревьев разбора, у которых корень отмечен стартовым символом, а крона является терминальной цепочкой.

Пример кроны

- На рисунке представлен пример дерева с терминальной цепочкой в качестве кроны и стартовым символом в корне. Оно основано на грамматике для выражений (см. слайд 128). Крона этого дерева образует цепочку *a * (a +* b00), выведенную в примере выше. В действительности, как будет показано далее, это дерево разбора представляет по рождение данной цепочки.

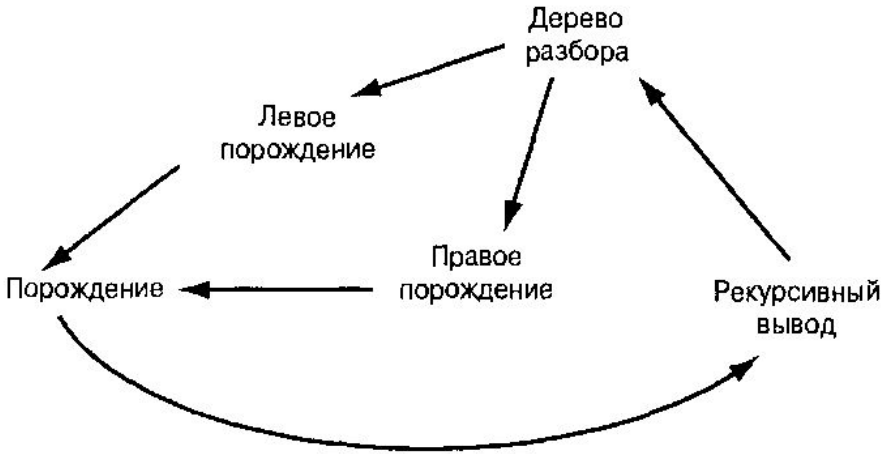


Вывод, порождения и деревья разбора

При любой грамматике *G = (V, T,P,S)* следующие утверждения равносильны:

- Процедура рекурсивного вывода определяет, что цепочка w принадлежит языку переменной *A.*
- $A \overset{*}{\Rightarrow} w.$
- $A \overset{lm}{\Rightarrow} w.$
- $A \overset{rm}{\Rightarrow} w.$
- Существует дерево разбора с корнем *A* и кроной *w.*

В действительности, за исключением использования рекурсивного вывода, определенного только для терминальных цепочек, все остальные условия (существование порождений, левых и правых порождений или деревьев разбора) также равносильны, если w имеет переменные.



Переход от выводов к деревьям разбора

- **Теорема.** Пусть *G = (V, T, P,S)* — КС-грамматика. Если рекурсивный вывод ут верждает, что терминальная цепочка *w* принадлежит языку переменной *A,* то существует дерево разбора с корнем *A* и кроной *w.*
- Доказательство производится индукцией по числу шагов, используемых в выводе того, что *w* принадлежит языку *A.*

Переход от деревьев к порождениям слайд 1/2

Пример. Рассмотрим еще раз грамматику выражений (см. слайд 128). Нетрудно убедиться, что существует следующее порождение: $E \Rightarrow I \Rightarrow Ib \Rightarrow ab$. Отсюда для произвольных цепочек α и β возможно следующее порождение:

$\alpha E \beta \Rightarrow \alpha I \beta \Rightarrow \alpha I b \beta \Rightarrow \alpha a b \beta$

Доказательством служит то, что головы продукций можно заменять их телами в контексте α и β точно так же, как и вне его.

Например, если порождение начинается заменами $E \Rightarrow \mathrm{E} + \mathrm{E} \Rightarrow \mathrm{E} + (\mathrm{E})$, то можно было бы применить порождение цепочки *ab* из второго α, рассматривая " $\mathrm{E} + ($" в качестве а и ")" — β Указанное порождение затем продолжалось бы

следующим образом. $\mathrm{E} + (\mathrm{E}) \Rightarrow \mathrm{E} + (\mathrm{I}) \Rightarrow \mathrm{E} + (\mathrm{Ib}) \Rightarrow \mathrm{E} + (\mathrm{ab})$.

Теперь можно сформулировать теорему, позволяющую преобразовывать дерево разбора в левое порождение. Доказательство проводится индукцией по *высоте* дерева, которая представляет собой максимальную из длин путей, ведущих от корня через его потомков к листьям.

Например, высота дерева, изображенного на слайде 142, равна 7. Самый длинный из путей от корня к листьям ведет к листу, отмеченному *b*. Заметим, что длины путей обычно учитывают ребра, а не узлы, поэтому путь, состоящий из единственного узла, имеет длину 0.

Переход от деревьев к порождениям слайд 2/2

Теорема. Пусть *G = (V, T, P,S)* — КС-грамматика. Предположим, что существует дерево разбора с корнем, отмеченным *A*, и кроной w, где $w \in T^*$. Тогда в грамматике *G* существует левое порождение $A \overset{lm*}{\Longrightarrow} w$

Пример. Рассмотрим левое порождение для дерева, изображенного на слайде 22. Продемонстрируем лишь заключительный шаг, на котором строится порождение по целому дереву из порождений, соответствующих поддеревьям корня. Итак, предположим, что с помощью рекурсивного применения техники из теоремы, показанной выше, мы убедились, что поддерево, корнем которого является левый сын корня дерева, имеет левое порождение $E \overset{lm}{\Rightarrow} I \overset{lm}{\Rightarrow} a$, а поддерево с корнем в третьем сыне корня имеет следующее левое порождение:

$$E \overset{lm}{\Rightarrow} (E) \overset{lm}{\Rightarrow} (E + E) \overset{lm}{\Rightarrow} (I + E) \overset{lm}{\Rightarrow} (a + E) \overset{lm}{\Rightarrow} (a + I) \overset{lm}{\Rightarrow} (a * I0) \overset{lm}{\Rightarrow} (a + I00) \overset{lm}{\Rightarrow} (a + b00)$$

Чтобы построить левое порождение для целого дерева, начинаем с шага в корне: $A \overset{lm}{\Rightarrow} E * E$. Затем заменяем первую переменную E и в соответствии с ее порождением

приписываем на каждом шаге **E*, чтобы учесть контекст, в котором это порождение используется. Левое порождение на текущий момент представляет собой следующее.

$$E \overset{lm}{\Rightarrow} E * E \overset{lm}{\Rightarrow} I * E \overset{lm}{\Rightarrow} a * E$$

Символ * в продукции, использованной в корне, не требует порождения, поэтому указанное левое порождение также учитывает первые два сына корня. Дополним левое порождение, используя порождение $E \overset{lm*}{\Longrightarrow} (a + b00)$, левый контекст которого образован цепочкой *a**, а правый пуст. Это порождение в действительности появлялось в примере выше и имело следующий вид:

$$E \overset{lm}{\Rightarrow} E * E \overset{lm}{\Rightarrow} I * E \overset{lm}{\Rightarrow} a * E \overset{lm}{\Rightarrow} A * (E) \overset{lm}{\Rightarrow} a * (E + E) \overset{lm}{\Rightarrow} a * (I + E) \overset{lm}{\Rightarrow} a * (a + E) \overset{lm}{\Rightarrow} a * (a + I) \overset{lm}{\Rightarrow} a * (a + I0) \overset{lm}{\Rightarrow} a * (a + I00) \overset{lm}{\Rightarrow} a * (a + b00)$$

Порождения и рекурсивные выводы слайд 1/2

- Покажем, что если существует порождение $A \Rightarrow w$ для некоторой КС-грамматики, то факт принадлежности *w* языку *A* доказывается путем процедуры рекурсивного вывода. Перед тем как приводить теорему, сделаем важные замечания о порождениях.

Предположим, у нас есть порождение $A \Rightarrow X_1 X_2 \cdots X_k \Rightarrow w$. Тогда *w* можно разбить на подцепочки w = $w_1 w_2 \cdots w_k$, где $X_i \overset{*}{\Rightarrow} w_i$,. Заметим, что если X_i является терминалом, то $X_i = w_i$ и порождение имеет 0 шагов.

Если X_i является переменной, то можно получить порождение $X_i \overset{*}{\Rightarrow} w_i$ начав с порождения $A \overset{*}{\Rightarrow} w$ и отбрасывая следующее:

а) все шаги, не относящиеся к порождению w_i из X_i;

б) все позиции выводимой цепочки, которые находятся либо справа, либо слева от позиций, порождаемых из X_i.

Порождения и рекурсивные выводы слайд 2/2

Пример. Используем грамматику выражений и рассмотрим следующее порождение.

$$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow I * E + E \Rightarrow I * I + E + I * I + I \Rightarrow a * i + I \Rightarrow a * b + I \Rightarrow a * b + a$$

Рассмотрим третью выводимую цепочку, E*E + E, и среднее E в ней.

Начав с E*E + E, можно пройти по шагам указанного выше порождения, выбрасывая позиции, порожденные из E* слева от центрального E и из +E справа от него. Шагами порождения тогда становятся E, E, I, I, I, *b, b.* Таким образом, следующий шаг не меняет центральное E, следующий за ним меняет его на I, два шага за ними сохраняют I, следующий меняет его на b, и заключительный шаг не изменяет того, что порождено из центрального E.

Если рассмотреть только шаги, которые изменяют то, что порождается из центрального E, то последовательность E, E, I, I, I, *b, b* превращается в порождение

$E \Rightarrow I \Rightarrow b$. Оно корректно описывает, как центральное E эволюционирует в полном порождении.

Теорема. Пусть *G = (V, T, P, S)* — КС-грамматика, и пусть существует порождение $A \overset{*}{\Rightarrow} w$, где $w \in T^*$. Тогда процедура рекурсивного вывода, примененная к *G,* определяет, что *w* принадлежит языку переменной *A.*

Приложения КС-грамматик

- Контекстно-свободные грамматики были придуманы Н. Хомским (N. Chomsky) как способ описания естественных языков, но их оказалось недостаточно. Однако по мере того, как множились примеры использования рекурсивно определяемых понятий, возрастала и потребность в КС-грамматиках как в способе описания примеров таких понятий. Рассмотрим кратко два применения КС-грамматик, одно старое и одно новое.
- Грамматики используются для описания языков программирования. Более важно здесь то, что существует механический способ превращения описания языка, вроде КС-грамматики, в синтаксический анализатор — часть компилятора, которая изучает структуру исходной программы и представляет ее с помощью дерева разбора. Это приложение является одним из самых ранних использований КС-грамматик; в дей ствительности, это один из первых путей, по которым теоретические идеи компьютерной науки пришли в практику.
- Развитие XML (Extensible Markup Language) призвано облегчить электронную коммерцию тем, что ее участникам доступны соглашения о форматах ордеров, описаний товаров, и многих других видов документов. Существенной частью XML является ***определение типа документа*** (DTD — Document Type Definition), представляющее собой КС-грамматику, которая описывает допустимые дескрипторы (tags) и способы их вложения друг в друга. Например, можно было бы заключить в скобки <PHONE> и </PHONE> последовательности символов, интерпретируемые как телефонные номера.

Синтаксические анализаторы

- Каждое из этих условий можно представить как некоторое состояние.
- Условию (3) соответствует начальное состояние q_0. Конечно, находясь в самом начале процесса, нужно последовательно прочитать 0 и 1. Но если в состоянии q_0 читается 1, то это нисколько не приближает к ситуации, когда прочитана последовательность 01, поэтому нужно оставаться в состоянии q_0 . Таким образом, $\delta(q_0, 1) = q_0$. Однако если в состоянии q_0 читается 0, то мы попадаем в условие (2), т.е. 01 еще не прочитаны, но уже прочитан 0. Пусть q_2 обозначает ситуацию, описываемую условием (2). Переход из q_0 по символу 0 имеет вид $\delta(q_0, 0) = q_2$.
- Рассмотрим теперь переходы из состояния q_2. При чтении 0 мы попадаем в ситуацию, которая не лучше предыдущей, но и не хуже. 01 еще не прочитаны, но уже прочитан 0, и теперь ожидается 1. Эта ситуация описывается состоянием q_2, поэтому определим $\delta(q_2, 0) = q_2$. Если же в состоянии q_2 читается 1, то становится ясно, что во входной последовательности непосредственно за 0 следует 1. Таким образом, можно перейти в допускающее состояние, которое обозначается q_1 и соответствует приведенному выше условию (1), т.е. $\delta(q_2, 1) = q_1$
- Наконец, нужно построить переходы в состоянии q_1. В этом состоянии уже прочитана последовательность 01, и, независимо от дальнейших событий, мы будем находиться в этом же состоянии, т.е. $\delta(q_1, 0) = \delta(q_1, 1) = q_1$.
- Таким образом, $Q = \{q_0, q_1 q_2\}$. Ранее упоминалось, что q_0 — начальное, а q_1 — единственное допускающее состояние автомата, т.е. $F = \{q_1\}$. Итак, полное описание автомата Л, допускающего язык/, цепочек, содержащих 01 в качестве подцепочки, имеет вид $A = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\})$, где δ — функция, описанная выше.

Оператор if - then – else слайд 1/2

- Многие объекты типичного языка программирования ведут себя подобно сбалансированным скобкам. Обычно это сами скобки в выражениях всех типов, а также начала и окончания блоков кода, например, слова **begin** и **end** в языке Pascal, или фигурные скобки { и } в C. Таким образом, любое появление фигурных скобок в С-программе должно образовывать сбалансированную последовательность с { в качестве левой скобки и } — правой.
- Есть еще один способ балансирования "скобок", отличающийся тем, что левые скобки могут быть несбалансированны, т.е. не иметь соответствующих правых. Примером является обработка **if** и **else** в **C.** Произвольная **if**-часть может быть как сбалансирована, так и не сбалансирована некоторой **else**-частью. Грамматика, порождающая возможные последовательности слов **if** и **else,** представленных i и e соответственно, имеет следующие продукции: $S \rightarrow \varepsilon|SS|iS|iSeS$
- Например, *ieie, iie* и *iei* являются возможными последовательностями слов **if** и **else,** и каждая из этих цепочек порождается данной грамматикой. Примерами неправильных последовательностей, не порождаемых грамматикой, являются ei и ieeii.
- Простая проверка (доказательство ее корректности оставляется в качестве упражнения) того, что последовательность символов i и e порождается грамматикой, состоит в рассмотрении каждого e по очереди слева направо. Найдем первое i слева от рассматриваемого e. Если его нет, цепочка не проходит проверку и не принадлежит языку. Если такое i есть, вычеркнем его и рассматриваемое e. Затем, если больше нет символов e, цепочка проходит проверку и принадлежит языку. Если символы e еще есть, то проверка продолжается.

Оператор if - then – else слайд 2/2

Пример. Рассмотрим цепочку iee. Первое e соответствует i слева от него. Оба удаляются. Оставшееся e не имеет i слева, и проверка не пройдена; слово iee не принадлежит языку. Отметим, что это заключение правильно, поскольку в C-программе слов **else** не может быть больше, чем **if.**

В качестве еще одного примера рассмотрим *iieie.* Соответствие первого e и i слева от него оставляет цепочку *iie.* Соответствие оставшегося e и i слева оставляет i. Символов e больше нет, и проверка пройдена. Это заключение также очевидно, поскольку последовательность *iieie* соответствует C-программе, структура которой подобна приведенной ниже. В действительности, алгоритм проверки соответствия (и компилятор C) говорит нам также, какое именно **if** совпадает с каждым данным **else.** Это знание сущест венно, если компилятор должен создавать логику потока управления, подразумеваемую программистом.

```
if (Условие) {
if (Условие) Инструкция; else Инструкция;
if (Условие) Инструкция; else Инструкция;
}
```

Генератор синтаксических анализаторов YACC

- Генерация синтаксического анализатора (функция, создающая деревья разбора по ис ходным программам) воплощена в программе YACC, реализованной во всех системах UNIX. На вход YACC подается КС-грамматика, запись которой отличается от используемой здесь только некоторыми деталями. С каждой продукцией связывается *действие* (action), представляющее собой фрагмент С-кода, который выполняется всякий раз, когда создается узел дерева разбора, соответствующий (вместе со своими сыновьями) этой продукции. Обычно действием является код для построения этого узла, хотя в некоторых приложениях YACC дерево разбора не создается, и действие задает что-то другое, например, выдачу порции объектного кода.
- Пример. На рисунке показан пример КС-грамматики в нотации YACC, совпадающей с грамматикой выражений со слайда 8. Грамматика совпадает с приведенной выше.

```
Exp     : Id                    {...}
        | Exp '+' Exp           {...}
        | Exp '*' Exp           {...}
        | '(' Exp ')'           {...}
        ;
Id      : 'a'                   {...}
        'b'                     {...}
        | Id 'a'                {...}
        | Id 'b'                {...}
        | Id '0'                {...}
        | Id '1'                {...}
        ;
```

Особенности нотации YACC

Теперь дадим строгое определение языка ДКА. С этой целью определим *расширенную функцию переходов,* которая описывает ситуацию, при которой отслеживается произвольную последовательность входных символов, начиная с произвольного состояния.

Если $\boldsymbol{\delta}$ — функция переходов, то расширенную функцию, построенную по $\boldsymbol{\delta}$, обозначим $\hat{\delta}$. Расширенная функция переходов ставит в соответствие состоянию q и цепочке w состояние p, в которое автомат попадает из состояния q, обработав входную последовательность w. Определим $\hat{\delta}$ индукцией по длине входной цепочки следующим образом:

Базис. $\hat{\delta}(q, \varepsilon) = q$, т.е., находясь в состоянии q и не читая вход, мы остаемся в состоянии q.

Индукция. Пусть w — цепочка вида xa, т.е. a — последний символ в цепочке, ax — цепочка, состоящая из всех символов цепочки w, за исключением последнего. Например, w = 1101 разбивается на x: = 110 и a = 1. Тогда

$$\hat{\delta}(q, \omega) = \delta\big(\hat{\delta}(q, x), a\big) \quad (1)$$

Выражение (1) может показаться довольно громоздким, но его идея проста. Для того чтобы найти $\hat{\delta}(q, \omega)$, вначале находим $\hat{\delta}(q, x)$ — состояние, в которое автомат попадает, обработав все символы цепочки w, кроме последнего. Предположим, что это состояние p, т.е. $\hat{\delta}(q, x) = p$. Тогда $\hat{\delta}(q, \omega)$ - это состояние, в которое автомат переходит из p при чтении a — последнего символа w. Таким образом, $\hat{\delta}(q, \omega) = \delta(p, a)$.

Языки описания документов: HTML, XML

- Рассмотрим семейство "языков", которые называются языками *описания документов* (markup languages). "Цепочками" этих языков являются документы с определенными метками, которые называются *дескрипторами* (tags). Дескрипторы говорят о семантике различных цепочек внутри документа.
- Основным языком сети Internet является язык описания документов, как HTML (Hypertext Markup Language). Этот язык имеет две основные функции: создание связей между документами и описание формата ("вида") документа. Мы дадим лишь упрощенный взгляд на структуру HTML, но следующие примеры должны показать его структуру и способ использования КС-грамматики как для описания правильных HTML-документов, так и для управления обработкой документа, т.е. его отображением на мо ниторе или принтере.
- **Пример.** На рисунке показан текст, содержащий список пунктов и его выражение в HTML. Показано что HTML состоит из обычного текста, перемежаемого дескрипторами. Соответствующие друг другу, т.е. парные дескрипторы имеют вид <x> и </x> для некоторой цепочки *x*.

Вещи, которые я *ненавижу*.

1. Заплесневелый хлеб.
2. Людей, которые ведут машину по узкой дороге слишком медленно.

а) видимый текст

```
<P>Вещи, которые я <EM>ненавижу</EM>:
<OL>
<LI>Заплесневелый хлеб.
<LI>Людей, которые ведут машину по узкой дороге слишком медленно.
</OL>
```

Фрагмент грамматики HTML

1. ***Text*** (текст) — это произвольная цепочка символов, которая может быть проинтер претирована буквально, т.е. не имеющая дескрипторов.
2. ***Char*** (символ) — цепочка, состоящая из одиночного символа, допустимого в HTML-тексте. Заметим, что пробелы рассматриваются как символы.
3. ***Doc*** (документ) представляет документы, которые являются последовательностями "элементов". Мы определим элементы следующими, и это определение будет вза имно рекурсивным с определением класса ***Doc.***
4. ***Element*** (элемент) — это или цепочка типа ***Text,*** или пара соответствующих друг другу дескрипторов и документ между ними, или непарный дескриптор, за которым следует документ.
5. ***ListItem*** (элемент списка) есть дескриптор <LI> со следующим за ним документом, который представляет собой одиночный элемент списка. .
6. *List* (список) есть последовательность из нуля или нескольких элементов списка.

1.	*Char*	→	*a* | *A* | ...
2.	*Text*	→	ε | *Char Text*
3.	*Doc*	→	ε | *Element Doc*
4.	*Element*	→	*Text* | <EM> *Doc* </EM> | <P> *Doc* | <OL> *List* </OL> | ...
5.	*ListItem*	→	<LI> *Doc*
6.	*List*	→	ε | *ListItem List*

Язык XML и определения типа документа

Цель XML состоит не в описании форматирования документа; это работа для HTML. Вместо этого XML стремится описать "семантику" текста. Например, текст наподобие "Кленовая ул., 12" выглядит как адрес, но является ли им? В XML дескрипторы окружали бы фразу, представляющую адрес, например: <ADDR>Кленовая ул., 12</ADDR>

Однако сразу не очевидно, что <ADDR> означает адрес дома. Например, если бы документ говорил о распределении памяти, мы могли бы предполагать, что дескриптор <ADDR> ссылается на адрес в памяти. Ожидается, что стандарты описания различных типов дескрипторов и структур, которые могут находиться между парами таких дескрипторов, будут развиваться в различных сферах деятельности в виде определений типа документа (DTD — Document-Type Definition).

DTD, по существу, является КС-грамматикой с собственной нотацией для описания переменных и продукций. Приведем простое DTD и представим некоторые средства, используемые в языке описания DTD. Язык DTD сам по себе имеет КС-грамматику, но не она интересует нас. Мы хотим увидеть, как КС-грамматики выражаются в этом языке.

DTD имеет следующий вид.

<!DOCTYPE имя-DTD [

список определений элементов

]>

Определение элемента DTD

Определение элемента, в свою очередь, имеет вид

<! ELEMENT имя-элемента (описание элемента)>

Описания элементов являются, по существу, регулярными выражениями. Их базис обра зуется следующими выражениями.

- Имена других элементов, отражающие тот факт, что элементы одного типа могут появляться внутри элементов другого типа, как в HTML мы могли бы найти выде ленный текст в списке.
- Специальное выражение \#PCDATA, обозначающее любой текст, который не включает дескрипторы XML. Это выражение играет роль переменной *Text* в примере выше.

Допустимы следующие знаки операций.

- |, обозначающий объединение, как в записи регулярных выражений.
- Запятая для обозначения конкатенации.
- Три варианта знаков операции замыкания. Знак * означает "нуль или несколько появлений", + — "не менее одного появления", ? — "нуль или одно появление".

Скобки могут группировать операторы и их аргументы; в их отсутствие действуют обычные приоритеты регулярных операций.

DTD для компьютера, упрощенное

```
<!DOCTYPE PcSpec [
    <!ELEMENT PCS (PC*)>
    <!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
    <!ELEMENT MODEL (\#PCDATA)>
    <!ELEMENT PRICE (\#PCDATA)>
    <!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
    <!ELEMENT MANF (\#PCDATA)>
    <!ELEMENT MODEL (\#PCDATA)>
    <!ELEMENT SPEED (\#PCDATA)>
    <!ELEMENT RAM (\#PCDATA)>
    <!ELEMENT DISK (HARDDISK | CD | DVD)>
    <!ELEMENT HARDDISK (MANF, MODEL, SIZE)>
    <!ELEMENT SIZE (\#PCDATA)>
    <!ELEMENT CD (SPEED)>
    <!ELEMENT DVD (SPEED)>
]>
```

XML-документ, соответствующий DTD

```
<PCS>
<PC>
<MODEL>4560</MODEL>
<PRICE>$2295</PRICE>
<PROCESSOR>
<MANF>Intel</MANF>
<MODEL>Pentium</MODEL>
<SPEED>800mhZ</SPEED>
</PROCESSOR>
<RAM>256</RAM>
<DISK>
<HARDDISK>
<MANF>Maxtor</MANF>
<MODEL>Diamond</MODEL>
<SIZE>30.5Gb</SIZE>
</HARDDISK>
</DISK>
<DISK>
```

Соответствие между КС и DTD

- Можно заметить, что правила для элементов в DTD (см. выше) не полностью совпадают с продукциями в КС-грамматиках.

Многие правила имеют корректный вид, например, правило

<!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>

аналогично продукции

$\text{Processor} \rightarrow \text{Manf Model Speed}$

Однако в правиле

<!ELEMENT DISK (HARDDISK | CD | DVD)>

определение для DISK не похоже на тело продукции. Расширение в этом случае является простым: это правило можно интерпретировать как три продукции, у которых вертикальная черта играет ту же роль, что и в продукциях обычного вида. Таким образом, это правило эквивалентно следующим трем продукциям.

$Disk \rightarrow Harddisk \mid Cd \mid Dvd$

Труднее всего следующее правило. <! ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)> Здесь "тело" содержит оператор замыкания. Решение состоит в замене DISK+ новой переменной, скажем, *Disks,* которая порождает с помощью пары продукций один или несколько экземпляров переменной *Disk.* Итак, получаются следующие эквивалентные продукции.

$$\text{Pc} \rightarrow Model\ Price\ Processor\ Ram\ Disks$$
$$Disks \rightarrow Disk \mid Disk\ Disks$$

Переход от КС-грамматик с рег. выражениями к обычным слайд 1/2

- **Язык ДКА**
- Теперь можно определить *язык* ДКА вида $A = (Q, \Sigma, \delta, q_0, F)$. Этот язык обозначается *L(A)* и определяется как
- $L(A) = \{\omega | \hat{\delta}(q_0, \omega) \text{ принадлежит } F\}$.
- Таким образом, язык — множество цепочек, приводящих автомат из состояния q_0 в одно из допускающих состояний. Если язык *L* есть *L(A)* для некоторого ДКА *A*, то говорят, что *L* является *регулярным языком*.

Переход от КС-грамматик с рег. выражениями к обычным слайд 2/2

- На слайде изображен недетерминированный конечный автомат, допускающий те и только те цепочки из 0 и 1, которые оканчиваются на 01. Начальным является состояние q_0 и можно считать, что автомат находится в этом состоянии (а также, возможно, и в других состояниях) до тех пор, пока не "догадается", что на входе началась замыкающая подцепочка 01. Всегда существует вероятность того, что следующий символ не является начальным для замыкающей подцепочки 01, даже если это символ 0. Поэтому состояние q_0 может иметь переходы в себя как по 1, так и по 0.

Неоднозначности в грамматиках и языках

- Как было показано, в приложениях КС-грамматики часто служат основой для обеспечения структуры различного рода файлов. Например, в разделе выше грамматики использовались для придания структуры программам и документам. Там действовало неявное предположение, что грамматика однозначно определяет структуру каждой цепочки своего языка. Однако можно показать, что не каждая грамматика обеспечивает уникальность структуры.
- Иногда, когда грамматика не может обеспечить уникальность структуры, ее можно преобразовать, чтобы структура была единственной для каждой цепочки. К сожалению, это возможно не всегда, т.е. существуют "существенно неоднозначные" языки; каждая грамматика для такого языка налагает несколько структур на некоторые его цепочки.

Неоднозначные грамматики

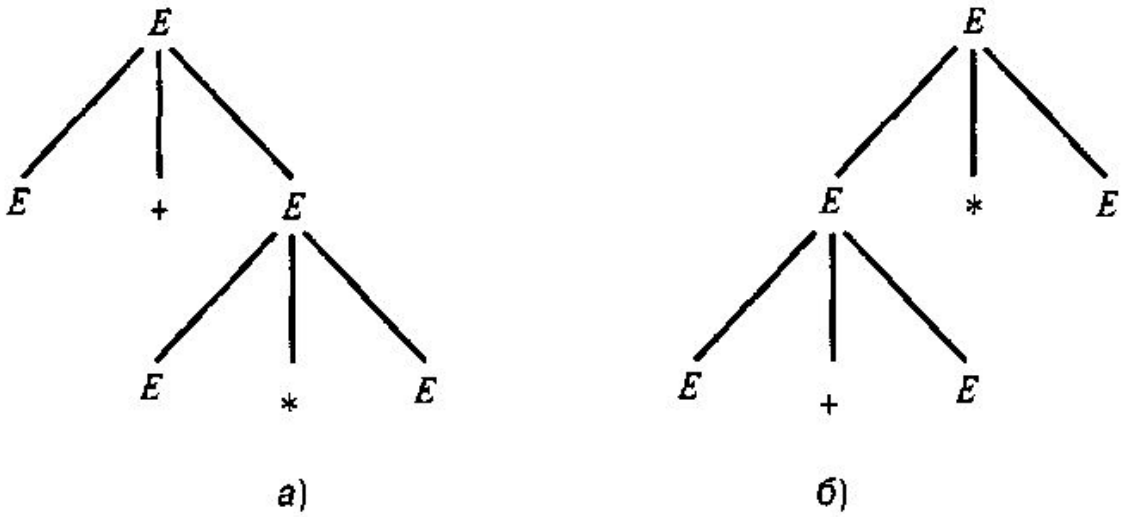
Вернемся к грамматике выражений (см. слайд 128). Эта грамматика дает возможность порождать выражения с любой последовательностью операторов + и *, а продукции $E \to E + E \mid E * E$ позволяют порождать эти выражения в произвольно выбранном порядке.

Пример. Рассмотрим выводимую цепочку E + E* E. Она имеет два порождения из E:

1. $E \Rightarrow E + E \Rightarrow E + E * E$

2. $E \Rightarrow E * E \Rightarrow E + E * E$

Заметим, что в порождении 1 второе E заменяется на E*E, тогда как в порождении 2 — первое E на E+E. На рисунке показаны два действительно различных дерева разбора.

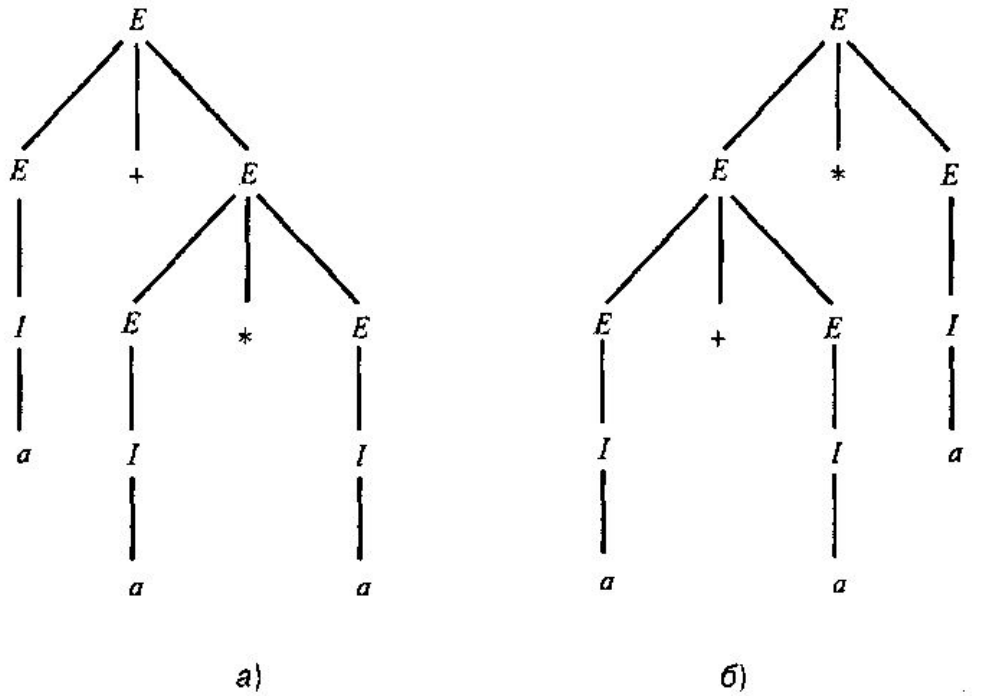


Различие между деревьями разбора

- Разница между этими двумя порождениями значительна. Когда рассматривается структура выражений, порождение 1 говорит, что второе и третье выражения перемножаются, и результат складывается с первым. Вместе с тем, порождение 2 задает сложение первых двух выражений и умножение результата на третье. Более конкретно, первое порождение задает, что 1+2*3 группируется как 1 + (2 * 3) = 7, а второе — что группирование имеет вид (I + 2) * 3 = 9. Очевидно, что первое из них (но не второе) соответствует нашему понятию о правильном группировании арифметических выражений.
- Поскольку рассматриваемая грамматика выражений дает две различные структуры любой цепочке терминалов, порождаемой заменой трех выражений в E + E* E идентификаторами, для обеспечения уникальности структуры она не подходит. В частности, хотя она может давать цепочкам как арифметическим выражениям правильное группирование, она также дает им и неправильное. Для того чтобы использовать грамматику вы ражений в компиляторе, мы должны изменить ее, обеспечив только правильное группирование.
- С другой стороны, само по себе существование различных порождений цепочки (что не равносильно различным деревьям разбора) еще не означает порочности грамматики.

Неоднозначность грамматики выражений

- Теперь определим формально понятия, связанные с недетерминированными конечными автоматами, выделив по ходу различия мех ряет структуру ДК
  Эти обозначе
  Q - конечное
  Σ - конечное
  δ – функция п вляются состояние из рое подмножеств ичие между НКА и ДКА со множество состояний, а
  q_0 – начальн
  F – множеств – подмножество Q



Исключение неоднозначности

- В идеальном мире можно было бы дать алгоритм исключения неоднозначности из КС-грамматик, почти как в теории автоматов, где был приведен алгоритм удаления несущественных состояний конечного автомата. Однако, как будет показано ниже, не существует даже алгоритма, способного различить, является ли КС-грамматика неоднозначной. Более того, будет показано, что существуют КС-языки, имеющие только неоднозначные КС-грамматики; исключение неоднозначности для них вообще невозможно.
- К счастью, положение на практике не настолько мрачное. Для многих конструкций, возникающих в обычных языках программирования, существует техника устранения не однозначности. Проблема с грамматикой выражений типична, и продемонстрируем устранение ее неоднозначности в качестве важной иллюстрации.

Причины неоднозначности грамматики слайда 128

- Не учитываются приоритеты операторов. В то время как на рис. слайда 167 а) оператор * правильно группируется перед оператором +, на рис. слайда 167 б) показано также допустимое дерево разбора, группирующее + перед *. Необходимо обеспечить, чтобы в однозначной грамматике была допустимой только структура, показанная на рис. слайда 167 а).
- Последовательность одинаковых операторов может группироваться как слева, так и справа. Например, если бы операторы * были заменены операторами +, то существовало бы два разных дерева разбора для цепочки E + E + E. Поскольку оба оператора ассоциативны, не имеет значения, происходит ли группировка слева или справа, но для исключения неоднозначности нужно выбрать что-то одно. Обычный подход состоит в группировании слева, поэтому только структура, изображенная на рис. слайда 167 б), представляет правильное группирование двух операторов +.

Установление приоритетов

Решение проблемы установления приоритетов состоит в том, что вводится несколько разных переменных, каждая из которых представляет выражения, имеющие один и тот же уровень "связывающей мощности". В частности, для грамматики выражений это решение имеет следующий вид.

1. *Сомножитель,* или *фактор* (factor), — это выражение, которое не может быть раз делено на части никаким примыкающим оператором, ни *, ни +. Сомножителями в рассматриваемом языке выражений являются только следующие выражения:

   а) идентификаторы. Буквы идентификатора невозможно разделить путем при соединения оператора;

   б) выражения в скобках, независимо от того, что находится между ними. Имен но для предохранения операндов в скобках от действия внешних операторов и предназначены скобки.

2. *Терм* (term), или *слагаемое,* — это выражение, которое не может быть разорвано оператором +. В рассматриваемом примере, где операторами являются только + и *, терм представляет собой произведение одного или несколько сомножителей. Например, терм a* b может быть "разорван", если мы используем левую ассоциативность * и поместим *a1** слева, поскольку *a1 * a * b* группируется слева как (al * *a)* b,* разрывая a* b. Однако помещение аддитивного выражения слева, типа a1+, или справа, типа *+a1,* не может разорвать *a** b. Правильным группированием выражения *a1 + a* b* является *a1 + (a* Ь),* а выражения *a* b + a* 1 — *(a* b) + a1.*

3. Выражение (expression) будет обозначать любое возможное выражение, включая те, которые могут быть разорваны примыкающими + и *. Таким образом, выражение для рассматриваемого примера представляет собой сумму одного или нескольких термов.

Однозначная грамматика выражений

я НКА, так же, как и для ДКА, нам потребуется расширить функцию δ до функции $\hat{\delta}$, аргументами которой ляются состояние q и цепочка входных символов w, а значением — множество состояний, в которые НКА падает из состояния q, обработав цепочку w.

сути, $\hat{\delta}(q,w)$ есть столбец состояний, которые получаются при чтении цепочки w, при условии, что q — инственное состояние в первом столбце. Так, выше показано, что $\hat{\delta}(q_0,001)=\{q_0,q_2\}$. Формально $\hat{\delta}$ для А определяется следующим образом:

зис $\hat{\delta}(q,\varepsilon)=\{q\}$:, т.е., не прочитав никаких входных символов, НКА находится только в том состоянии, в гором начинал.

дукция. Предположим, цепочка w имеет вид $w=xa$, где a — последний символ цепочки w, ax — ее оставшаяся сть. Кроме того, предположим, что $\hat{\delta}(q,x)=\{p_1,p_2,\cdots,p_k\}$.

сть $\bigcup_{i=1}^{k}\delta(p_i,a)=\{r_1,r_2,\cdots,r_m\}$

да $\delta(q,w)=\{r_1,r_2,\cdots,r_m\}$.

воря менее формально, для того, чтобы найти $\hat{\delta}(q,w)$, нужно найти $\delta(q,x)$, а затем совершить из всех лученных состояний все переходы по символу a.

имер. Используем $\hat{\delta}$ для описания того, как НКА обрабатывает цепочку 00101.

$$\hat{\delta}(q_0,\varepsilon)=\{q_0\}$$

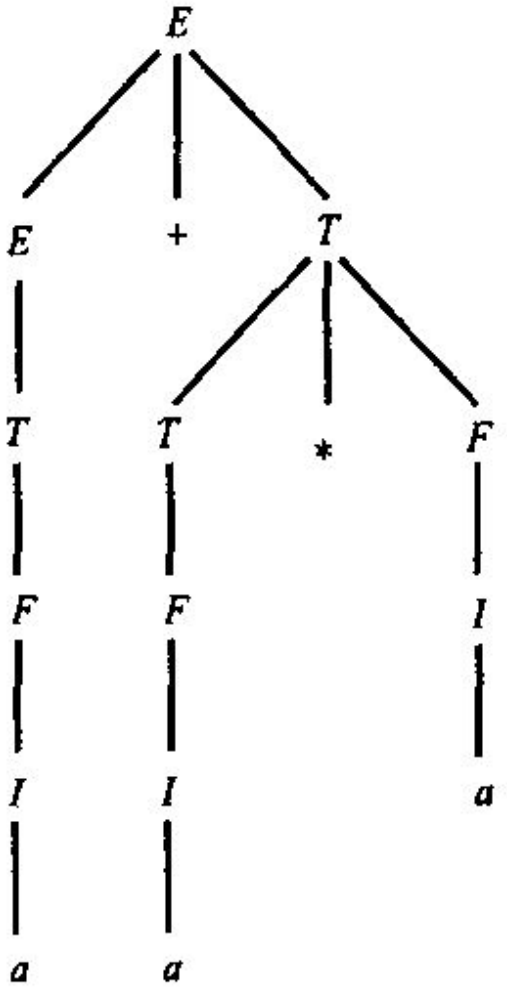
$$\hat{\delta}(q_0,0)=\delta(q_0,0)=\{q_0,q_1\}$$

$$\hat{\delta}(q_0,00)=\delta(q_0,0)\bigcup\delta(q_1,0)=\{q_0,q_1\}\bigcup\emptyset=\{q_0,q_1\}$$

$$\hat{\delta}(q_0,001)=\delta(q_0,1)\bigcup\delta(q_1,1)=\{q_0\}\bigcup\{q_2\}=\{q_0,q_2\}$$

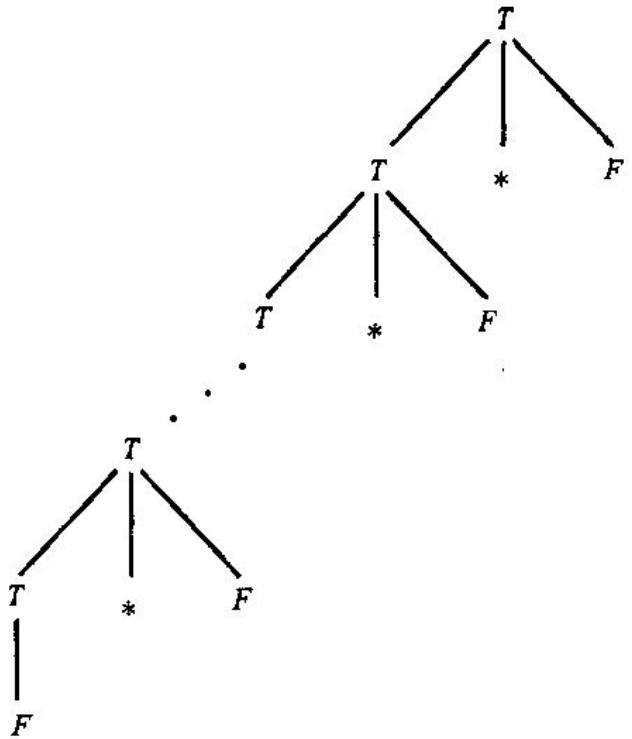
$$\hat{\delta}(q_0,0010)=\delta(q_0,0)\bigcup\delta(q_2,0)=\{q_0,q_1\}\bigcup\emptyset=\{q_0,q_1\}$$

$$\hat{\delta}(q_0,00101)=\delta(q_0,1)\bigcup\delta(q_1,1)=\{q_0\}\bigcup\{q_2\}=\{q_0,q_2\}$$



Сложности однозначной грамматики

- То, что грамматика слайда 171 однозначна, может быть далеко не очевидно.
- Нельзя породить выражение вида $f_1 * f_2 * \cdots * f_n$ без введения скобок вокруг него. Таким образом, при использовании продукции $T \to T * F$ из *F* невозможно породить ничего, кроме последнего из сомножителей, т.е. дерево разбора для терма может выглядеть только так, как на рисунке.
- Аналогично, выражение есть последовательность термов, связанных знаками +. Когда используется продукция $\mathrm{E} \to \mathrm{E} + \mathrm{T}$ для порождения $t_1 + t_2 + \cdots + t_n$ из *T* должно порождаться только t_n, а из E в теле — $t_1 + t_2 + \cdots + t_{n-1}$. Причина этого опять-таки в том, что из T невозможно породить сумму двух и более термов без заключения их в скобки.



Выражение неоднозначности через левые порождения

Хотя порождения не обязательно уникальны, даже если грамматика однозначна, оказывается, что в однозначной грамматике и левые, и правые порождения уникальны. Рассмотрим только левые.

Пример. Заметим, что оба дерева разбора, представленные на слайде 167, имеют крону E + E * E. По ним получаются следующие соответствующие им левые порождения.

а)

$$E \overset{lm}{\Rightarrow} E + E \overset{lm}{\Rightarrow} I + E \overset{lm}{\Rightarrow} a + E \overset{lm}{\Rightarrow} a + E * E \overset{lm}{\Rightarrow} a + I * E \overset{lm}{\Rightarrow} a + a * E \overset{lm}{\Rightarrow} a + a * I \overset{lm}{\Rightarrow} a + a * a$$

б)

$$E \overset{lm}{\Rightarrow} E * E \overset{lm}{\Rightarrow} E + E * E \overset{lm}{\Rightarrow} I + E * E \overset{lm}{\Rightarrow} a + E * E \overset{lm}{\Rightarrow} a + I * E \overset{lm}{\Rightarrow} a + a * E \overset{lm}{\Rightarrow} a + a * I \overset{lm}{\Rightarrow} a + a * a$$

Заметим, что эти левые порождения различаются. Данный пример не доказывает теорему, но демонстрирует, как различия в деревьях разбора влияют на выбор шагов в левом порождении.

Теорема. Для каждой грамматики *G = (V, T, P, S)* и *w* из T^* цепочка *w* имеет два разных дерева разбора тогда и только тогда, когда w имеет два разных левых порождения из *S.*

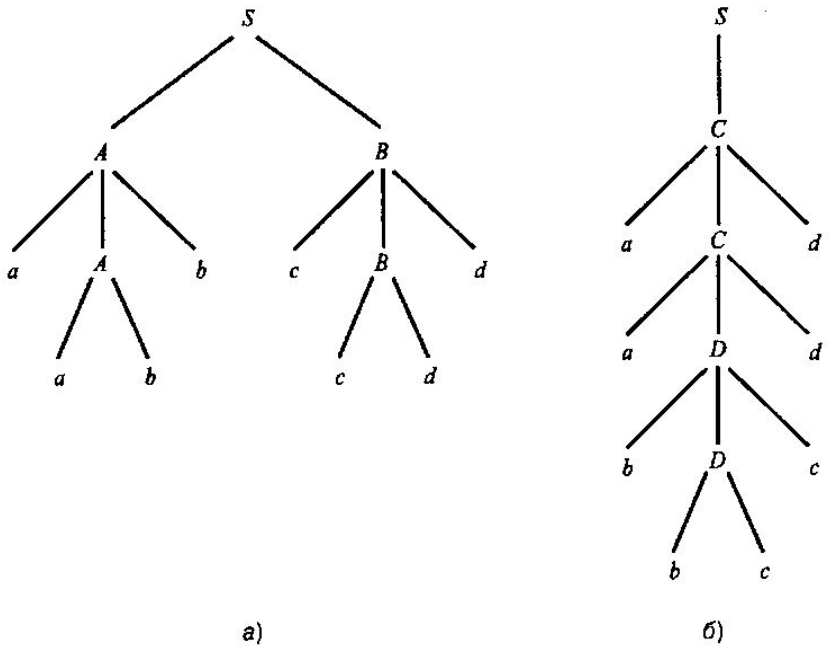
Существенная неоднозначность

- Контекстно-свободный язык *L* называется *существенно неоднозначным,* если все его грамматики неоднозначны. Если хотя бы одна грамматика языка *L* однозначна, то *L* является однозначным языком. Было показано, например, что язык выражений, порождаемый грамматикой выражений, приведенной выше, в действительности однозначен. Хотя данная грамматика и неоднозначна, этот же язык задается еще одной грамматикой, которая однозначна — она показана выше.
- В рамках курса не будем доказывать, что существуют неоднозначные языки. Вместо этого рассмотрим пример языка, неоднозначность которого можно доказать, и объясним неформально, почему любая грамматика для этого языка должна быть неоднозначной:
- $L = \{a^n b^n c^m d^m | n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n | n \geq 1, m \geq 1\}.$
- Из определения видно, что *L* состоит из цепочек вида $a^+b^+c^+d^+$, в которых поровну символов *a* и *b,* а также *c u d,* либо поровну символов *a* и *d,* а также *b u c.*

КС-грамматика языка L

L является КС-языком. Очевидная грамматика для него показана ниже. Для порождения двух видов цепочек в ней используются два разных множества продукций.

$$S \to AB|C$$
$$A \to aAb|ab$$
$$\boldsymbol{B \to cBd|cd}$$
$$C \to aCd|aDd$$
$$\boldsymbol{D \to bDC|bc}$$



Эта грамматика неоднозначна. Например, у цепочки ***aabbccdd*** есть два следующих левых порождения.

$$S \overset{lm}{\Rightarrow} AB \overset{lm}{\Rightarrow} aAbB \overset{lm}{\Rightarrow} aabbB \overset{lm}{\Rightarrow} aabbccBd \overset{lm}{\Rightarrow}$$

$$\mathbf{S \overset{lm}{\Rightarrow} C \overset{lm}{\Rightarrow} aCd \overset{lm}{\Rightarrow} aaDdd \overset{lm}{\Rightarrow} aabDcdd \overset{lm}{\Rightarrow} aab}$$

Доказательство неоднозначности L

Доказательство того, что все грамматики для языка ***L*** неоднозначны, весьма непросто, однако сущность его такова. Нужно обосновать, что все цепочки (за исключением конечного их числа), у которых поровну всех символов, должны порождаться двумя различными путями. Первый путь — порождение их как цепочек, у которых поровну символов ***a*** и ***b,*** а также ***c*** и ***d,*** второй путь — как цепочек, у которых поровну символов ***a*** и ***d,*** как ибис.

Например, единственный способ породить цепочки, у которых поровну *a* и b, состоит в использовании переменной, подобной *A* в грамматике, показанной выше. Конечно же, возможны варианты, но они не меняют картины в целом, как это видно из следующих примеров.

- Некоторые короткие цепочки могут не порождаться после изменения, например, базисной продукции $\mathrm{A} \rightarrow \mathrm{ab}$ на $\mathrm{A} \rightarrow \mathrm{aaabbb}$.
- Можно организовать продукции так, чтобы переменная *A* делила свою работу с другими, например, используя переменные A_1 и A_2, из которых A_1 порождает нечетные количества символов *a,* а A_2 — четные: $\mathrm{A}_1 \rightarrow \mathrm{aA}_2\mathrm{b}|\mathrm{ab};\ \mathrm{A}_2 \rightarrow \mathrm{aA}_1\mathrm{b}|\mathrm{ab}$.
- Можно организовать продукции так, чтобы количества символов *a* и b, порождаемые переменной *A,* были не равны, но отличались на некоторое конечное число. Например, можно начать с продукции $\mathrm{S} \rightarrow \mathrm{AaB}$ и затем использовать $\mathrm{A} \rightarrow \mathrm{aAb}|\mathrm{a}$ для порождения символов *a* на один больше, чем b.

В любом случае, не избежать некоторого способа порождения символов *a,* при котором соблюдается их соответствие с символами b.

Аналогично можно обосновать, что должна использоваться переменная, подобная *B,* которая порождает соответствующие друг другу символы си *d.* Кроме того, в грамматике должны быть переменные, играющие роль переменных **Си D** порождающих соответственно парные *a* и *d* и парные *b u c.* Приведенные аргументы, если их формализовать, доказывают, что независимо от изменений, которые можно внести в исходную грамматику, она будет порождать *хотя бы одну* цепочку вида $a^n b^n c^n d^n$ двумя способами, как и грамматика, показанная выше.

Автоматы с магазинной памятью

Нам известно, что $\{q_0\}$ есть одно из состояний ДКА *D*. Находим, что $\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ и $\delta_D(\{q_0\}, 1) = \{q_0\}$. Оба эти факта следуют из диаграммы переходов для автомата; как видно, по символу 0 есть переходы из q_0 в q_0 и q_1 а по символу 1 — только в q_0. Таким образом, получена вторая строка таблицы переходов ДКА.

Одно из найденных множеств, $\{q_0\}$, уже рассматривалось. Но второе, $\{q_0, q_1\}$, — новое, и переходы для него нужно найти: $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ и $\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_2\}$. Проследить последние вычисления можно, например, так:

$$\delta_D(\{q_0, q_1\}, 1) = \delta_N(q_0, 1) \bigcup \delta_N(q_1, 1) = \{q_0\} \bigcup \{q_2\} = \{q_0, q_2\}$$

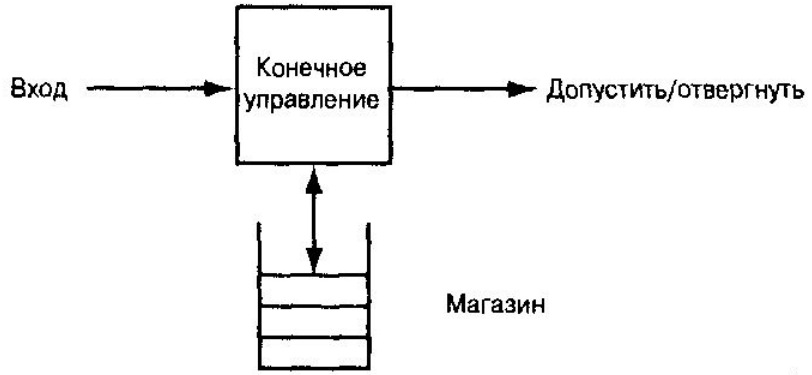
Теперь получена пятая строка таблицы и одно новое состояние $\{q_0, q_2\}$. Аналогичные вычисления показывают, что

$$\delta_D(\{q_0, q_2\}, 0) = \delta_N(q_0, 0) \bigcup \delta_N(q_2, 0) = \{q_0, q_1\} \bigcup \{\emptyset\} = \{q_0, q_1\}$$

$$\delta_D(\{q_0, q_2\}, 1) = \delta_N(q_0, 1) \bigcup \delta_N(q_{21}, 1) = \{q_0\} \bigcup \{\emptyset\} = \{q_0\}$$

Эти вычисления дают шестую строку таблицы, но при этом не получено ни одного нового множества состояний.

Неформальное определение автомата с магазинной памятью



Магазинный автомат — это, по существу, недетерминированный конечный автомат с ε-переходами и одним дополнением — магазином, в котором хранится цепочка "магазинных символов". Присутствие магазина означает, что в отличие от конечного автомата магазинный автомат может "помнить" бесконечное количество информации. Однако в отличие от универсального компьютера, который также способен запоминать неограниченные объемы информации, магазинный автомат имеет доступ к информации в магазине только с одного его конца в соответствии с принципом "последним пришел — первым ушел" ("last-in-first-out").

Вследствие этого существуют языки, распознаваемые некоторой программой компьютера и нераспознаваемые ни одним магазинным автоматом. В действительности, магазинные автоматы распознают в точности КС-языки. Многие языки контекстно-свободны, включая, как было показано, некоторые нерегулярные, однако существуют языки, которые просто описываются, но не являются контекстно-свободными. Примеры таких языков буду приведены ниже. Примером такого языка является $\{0^n 1^n 2^n | n \geq 1\}$, т.е. множество цепочек, состоящих из одинаковых групп символов 0, 1 и 2.

Магазинный автомат можно рассматривать неформально как устройство, представленное на слайде.

Конечное управление

- В рассмотренном выше примере число состояний ДКА и число состояний НКА одинаково. Как уже было сказано, ситуация, когда количества состояний НКА и построенного по нему ДКА примерно одинаковы, на практике встречается довольно часто. Однако при переходе от НКА к ДКА возможен и экспоненциальный рост числа состояний, т.е. все 2^n состояний, которые могут быть построены по НКА, имеющему n состояний, оказываются достижимыми. В следующем примере мы немного не дойдем до этого предела, но будет ясно, каким образом наименьший ДКА, построенный по НКА с n + 1 состояниями, может иметь 2^n состояний.

Пример

- Рассмотрим язык $L_{wwr} = \{ww^R | w \in (0+1)^*\}$. Этот язык образован палиндромами четной длины над алфавитом {0, 1} и порождается КС-грамматикой (см. выше) с исключенными продукциями $P \to 0$ и $P \to 1$.

Дадим следующие неформальное описание магазинного автомата, допускающего L_{wwr}

- Работа начинается в состоянии q_0, представляющем "догадку", что не достигнута середина входного слова, т.е. конец слова w, за которым должно следовать его отражение. В состоянии q_0 символы читаются и их копии по очереди записываются в магазин.
- В любой момент можно предположить, что достигнута середина входа, т.е. конец слова *w.* В этот момент слово *w* находится в магазине: левый конец слова на дне магазина, а правый — на вершине. Этот выбор отмечается спонтанным переходом в состояние q_1. Поскольку автомат недетерминирован, в действительности предполагаются обе возможности, т.е. что достигнут конец слова *w,* но можно оставаться в состоянии q_0 и продолжать читать входные символы и записывать их в магазин.
- В состоянии q_1 входные символы сравниваются с символами на вершине магазина. Если они совпадают, то входной символ пропускается, магазинный удаляется, и работа продолжается. Если же они не совпадают, то предположение о середине слова неверно, т.е. за предполагаемым w не следует w^R. Эта ветвь вычислений отбрасывается, хотя другие могут продолжаться и вести к тому, что цепочка допускается.

Если магазин опустошается, то действительно обнаружен вход w, за которым следует w^R. Прочитанный к этому моменту вход допускается.

Формальное определение автомата с магазинной памятью

Формальная запись *магазинного автомата* (МП-автомата) содержит семь компонентов и выглядит следующим образом.

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Компоненты имеют такой смысл:

Q: конечное множество состояний, как и у конечного автомата.

Σ: конечное множество входных символов, такое же, как у конечного автомата.

Γ: конечный магазинный алфавит. Он не имеет аналога в конечных автоматах и является множеством символов, которые можно помещать в магазин.

δ: функция переходов. Как и у конечных автоматов, δ управляет поведением автомата. Формально, аргументами δ являются тройки $\delta(q, a, X)$, в которых *q* — состояние из множества *Q*, *a* — либо входной символ, либо пустая цепочка ε, которая, по предположению, не принадлежит входному алфавиту, X— магазинный символ из Г. Выход δ образуют пары (p, γ), где *p* — новое состояние, а γ — цепочка магазинных символов, замещающая *X* на вершине магазина. Например, если $\gamma = \varepsilon$, то магазинный символ снимается, если *y=X,* то магазин не изменяется, а если *y= YZ,* то *X* заменяется на Z, и *Y* помещается в магазин.

q_0: начальное состояние. МП-автомат находится в нем перед началом работы.

Z_0*:* начальный магазинный символ ("маркер дна"). Вначале магазин содержит только этот символ и ничего более.

F*:* множество допускающих, или заключительных, состояний.

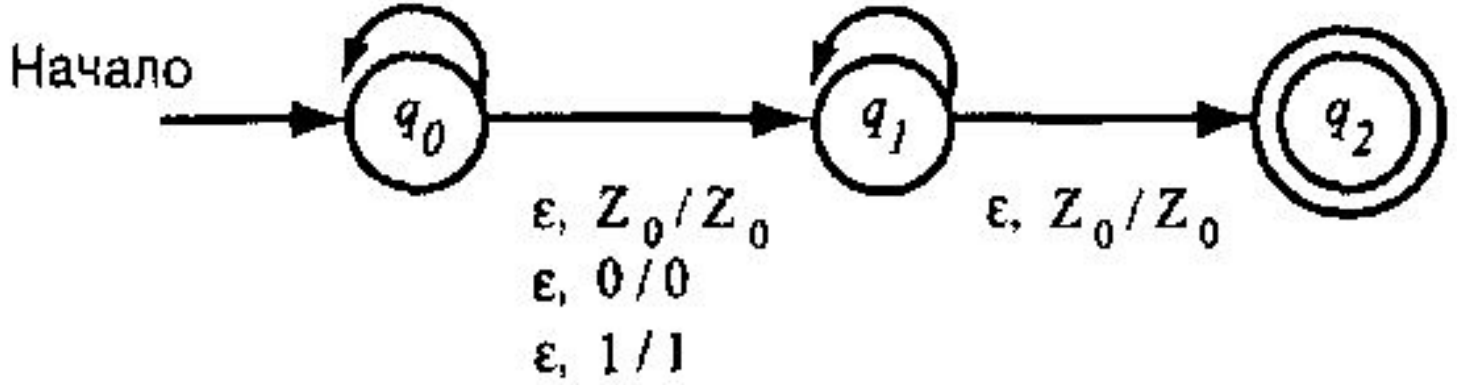
Следует отметить следующее: В некоторых случаях МП-автомат может иметь несколько пар на выбор. Например, пусть $\delta(q, a, X) = \{(p, YZ), (r, \varepsilon)\}$. Совершая переход, автомат должен выбрать пару целиком, т.е. нельзя взять состояние из одной пары, а цепочку для замещения в магазине — из другой. Таким образом, имея состояние *q,* символ X на вершине магазина и *a* на входе, автомат может либо перейти в состояние p и изменить X на *YZ,* перейти либо в r и вытолкнуть X. Однако перейти в состояние p и вытолкнуть X или перейти в *r* и изменить X на *YZ* нельзя.

Пример создания МП-автомата

- Теперь рассмотрим, как работает НКА *N*, показанный выше. Существует состояние q_0, в котором этот НКА находится всегда, независимо от входных символов. Если следующий символ — 1, то *N* может "догадаться", что эта 1 есть *n*-й символ с конца. Поэтому одновременно с переходом в q_0 НКА *N* переходит в состояние q_1. Из состояния q_x по любому символу *N* переходит в состояние q_2. Следующий символ переводит *N* в состояние q_3 и так далее, пока *n* - 1 последующий символ не переведет *N* в допускающее состояние q_n. Формальные утверждения о работе состояний N выглядят следующим образом:
  - *N* находится в состоянии q_0 по прочтении любой входной последовательности ω.
  - *N* находится в состоянии q_i (i = 1,2, ..., *n)* по прочтении входной последовательности ω тогда и только тогда, когда i-й символ с конца ω есть 1, т.е. ω имеет вид $x1a_1a_2 \cdots a_{i-1}$, где a_j — входные символы.

Графическое представление МП-автомата

- П
- в
- э
- н
- д
- Т
- р
- з



- Д
- в
- д
- о

Конфигурации МП-автомата

Сейчас есть лишь неформальное понятие того, как МП-автомат "вычисляет". Интуитивно МП-автомат переходит от конфигурации к конфигурации в соответствии с входными символами (или в), но в отличие от конечного автомата, о котором известно только его состояние, конфигурация МП-автомата включает как состояние, так и содержимое магазина. Поскольку магазин может быть очень большим, он часто является наиболее важной частью конфигурации. Полезно также представлять в качестве части конфигурации непрочитанную часть входа.

Таким образом, конфигурация МП-автомата представляется тройкой (q, w, γ), где q — состояние, w— оставшаяся часть входа, γ — содержимое магазина. По соглашению вершина магазина изображается слева, а дно — справа. Такая тройка называется *конфигурацией* МП-автомата, или его *мгновенным описанием*, сокращенно МО (instantaneous description — ID).

Поскольку МО конечного автомата— это просто его состояние, для представления последовательностей конфигураций, через которые он проходил, было достаточно использовать $\hat{\delta}$. Однако для МП-автоматов нужна нотация, описывающая изменения состояния, входа и магазина. Таким образом, используются пары конфигураций, связи между которыми представляют переходы МП-автомата.

Определение перехода

Пусть *P* = *(Q,* Г, δ, q_0, Z_0, *F)* — МП-автомат. Определим отношение $\vdash_{\mathrm{P}}$, или просто $\vdash$, когда *P* подразумевается, следующим образом. Предположим, что $\delta(q, a, X)$ содержит (p, α). Тогда для всех цепочек *w* из Σ^* и β из Γ^* полагаем $\delta(q, aw, X\beta) \vdash (\mathrm{p}, \mathrm{w}, \alpha\beta)$.

Этот переход отражает идею того, что, прочитывая на входе символ *a,* который может быть ε, и заменяя X на вершине магазина цепочкой α, можно перейти из состояния *q* в состояние *p.* Заметим, что оставшаяся часть входа (w) и содержимое магазина под его вершиной *(*β*)* не влияют на действие МП-автомата; они просто сохраняются, возможно, для того, чтобы влиять на события в дальнейшем.

Используем также символ $\vdash_{\mathrm{P}}^*$, или просто $\vdash^*$, когда МП-автомат P подразумевается, для представления нуля или нескольких переходов МП-автомата. Итак, имеем следующее индуктивное определение:

Базис. $\mathrm{I} \vdash^* \mathrm{I}$ для любого МО I.

Индукция. $I \vdash^* \mathrm{J}$, если существует некоторое МО *K,* удовлетворяющее условиям $\mathrm{I} \vdash \mathrm{K}$ и $\mathrm{K} \vdash^* \mathrm{J}$.

Таким образом, $\mathrm{I} \vdash^* \mathrm{J}$, если существует такая последовательность МО $K_1, K_2, \cdots, K_n$, у которой $\mathrm{I} = K_1, J = K_n$ и $K_i \vdash K_{i+1}$ для всех $\mathrm{i} = 1,2, \cdots, \mathrm{n} - 1$.

Соглашения по записи МП-автоматов

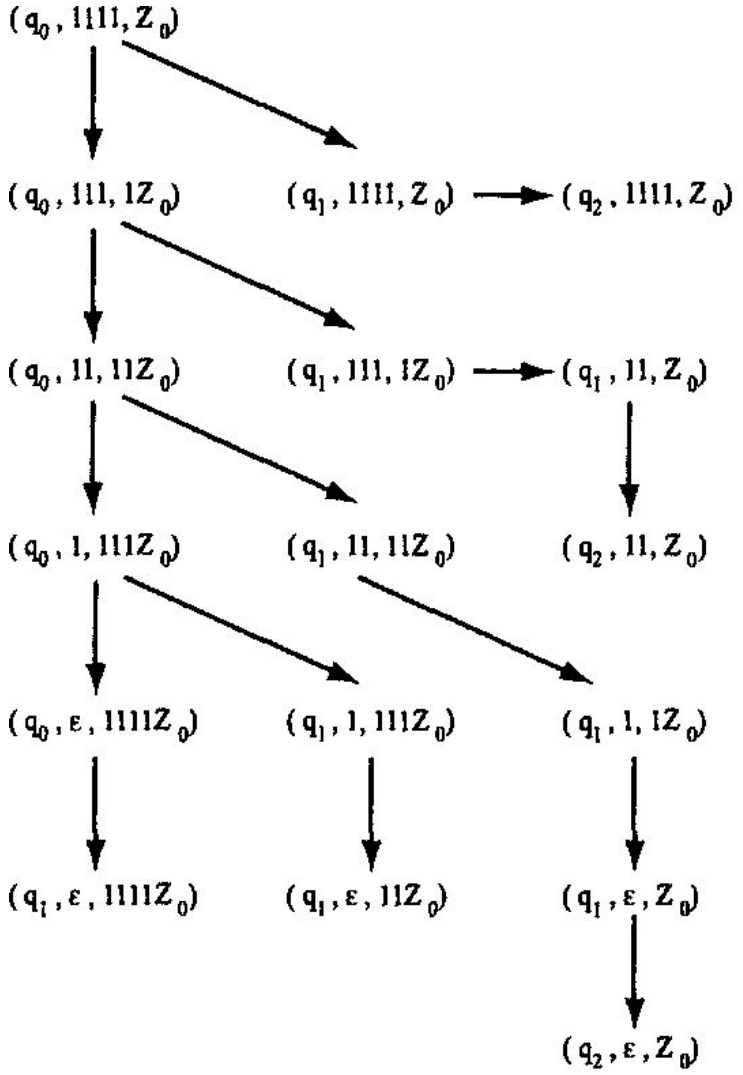
Продолжим соглашения об использовании символов, введенные для конечных автоматов и грамматик. Придерживаясь системы записи, полезно представлять себе, что магазинные символы играют роль, аналогичную объединению терминалов и переменных в КС-грамматиках.

1. Символы входного алфавита представлены строчными буквами из начала алфавита, например, *a* или *b*.
2. Состояния обычно представляются буквами *p* и *q* или другими, близкими к ним в алфавитном порядке.
3. Цепочки входных символов обозначаются строчными буквами из конца алфавита, например, w или z.
4. Магазинные символы представлены прописными буквами из конца алфавита, например, *X* или *Y*.

Цепочки магазинных символов обозначаются греческими буквами, например, α или β.

Пример работы МП-автомата

- Рассмотрим работу МП-автомата из примера выше со входом 1111. Поскольку q_0 — начальное состояние, а Z_0 — стартовый символ, то начальным МО будет $(q_0, 1111, Z_0)$. На этом входе МП-автомат имеет возможность несколько раз делать ошибочные предположения. Вся последовательность МО, достижимых из начальной конфигурации, показана на слайде. Стрелки представляют отношение $\vdash$.



Важные принципы МП-автоматов

- ε -НКА можно представлять точно так же, как и НКА, с той лишь разницей, что функция переходов должна содержать информацию о переходах по ε. Формально, ε-НКА A можно представить в виде $A = (Q, \Sigma, \delta, q_0, F)$, где все компоненты имеют тот же смысл, что и для НКА, за исключением δ, аргументами которой теперь являются состояние из Q и элемент множества $\Sigma \cup \{\varepsilon\}$, т.е. либо некоторый входной символ, либо ε. Никаких недоразумений при этом не возникает, поскольку мы оговариваем, что e, символ пустой цепочки, не является элементом алфавита Σ.
- Пример: e-НКА на слайде 96 можно формально представить как
  $E = (\{q_1, q_2, \cdots, q_5\}, \{., +, -, 0,1, \cdots, 9\}, \delta, q_0, \{q_5\})$

Языки МП-автоматов

- Выше предполагалось, что МП-автомат допускает свой вход, прочитывая его и достигая заключительного состояния. Такой подход называется "допуск по заключительному состоянию". Существует другой способ определения языка МП-автомата, имеющий важные приложения. Для любого МП-автомата можно определить язык, "допускаемый по пустому магазину", т.е. множество цепочек, приводящих МП-автомат в начальной конфигурации к опустошению магазина.
- Эти два метода эквивалентны в том смысле, что для языка L найдется МП-автомат, допускающий его по заключительному состоянию тогда и только тогда, когда для *L* найдется МП-автомат, допускающий его по пустому магазину. Однако для конкретных МП- автоматов языки, допускаемые по заключительному состоянию и по пустому магазину, обычно различны. В этом разделе показывается, как преобразовать МП-автомат, допускающий *L* по заключительному состоянию, в другой МП-автомат, который допускает *L* по пустому магазину, и наоборот.

Допустимость по заключительному состоянию

- Пусть $P = \{Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\}$ — МП-автомат. Тогда *L(P),* языком, допускаемым *P* по заключительному состоянию, является $\{w | (q_0, w, Z_0) \vdash_{\mathrm{P}}^{*} (q, \varepsilon, \alpha)\}$ для некоторого состояния *q* из *F* и произвольной магазинной цепочки α. Таким образом, начиная со стартовой конфигурации с *w* на входе, *P* прочитывает *w* и достигает допускающего состояния. Содержимое магазина в этот момент не имеет значения.

Допустимость по пустому магазину

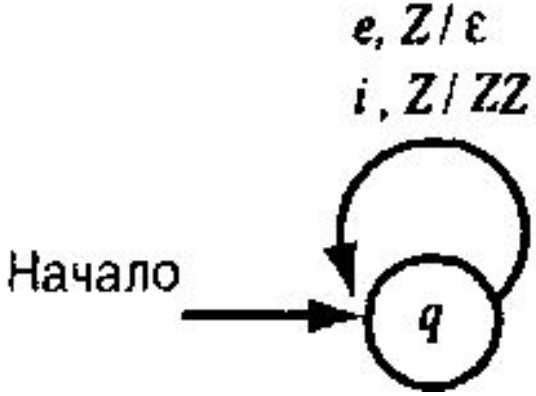
- Дадим формальное определение расширенной функции переходов для ε -НКА, которое приведет к определению допустимости цепочек и языков для данного типа автоматов и в конце концов поможет понять, почему ДКА могут имитировать работу ε -НКА. Однако прежде нужно определить одно из центральных понятий, так называемое ε -*замыкание* состояния. Говоря нестрого, мы получаем ε -замыкание состояния *q,* совершая все возможные переходы из этого состояния, отмеченные ε. Но после совершения этих переходов и получения новых состояний снова выполняются ε -переходы, уже из новых состояний, и т.д. В конце концов, мы находим все состояния, в которые можно попасть из *q* по любому пути, каждый переход в котором отмечен символом ε. Формально определяем ε -замыкание, ECLOSE, рекурсивно следующим образом:
- **Базис.** ECLOSE(q) содержит состояние *q.*
- **Индукция.** Если ECLOSE(q) содержит состояние *p,* и существует переход, отмеченный ε, из состояния p в состояние r, то ECLOSE(q) содержит r. Точнее, если δ есть функция переходов рассматриваемого e-НКА и ECLOSE(q) содержит *p,* то ECLOSE(q) содержит также все состояния из $\delta(\mathrm{p}, \varepsilon)$.

Переход от пустого магазина к заключительному состоянию

- Покажем, что классы языков, допускаемых МП-автоматами по заключительному состоянию и по пустому магазину, совпадают. Ниже будет доказано, что МП-автоматы определяют КС-языки. Наша первая конструкция показывает, как, исходя из МП-автомата P_N, допускающего язык L по пустому магазину, построить МП-автомат P_F, допускающий L по заключительному состоянию.
- Теорема. Если $L = N(\mathrm{P}_N)$ **для** некоторого МП-автомата $P_N = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, то существует такой МП-автомат P_F, у которого $L = L(P_F)$.

Пример перехода слайд 1/3

- **Пример.** Построим МП-автомат, который обрабатывает последовательности слов if и else в С-программе, где ***i*** обозначает if а e — else. Как было показано выше, в любом префиксе программы количество else не может превышать числа if, поскольку в противном случае слову else нельзя сопоставить предшествующее ему if. Итак, магазинный символ Z используется для подсчета разницы между текущими количествами просмотренных ***i*** и ***e.*** Этот простой МП-автомат с единственным состоянием представлен диаграммой переходов на слайде



Пример перехода слайд 2/3

Вычислим $\delta(q_0, 5.6)$ для ε -НКА на слайде 96. Для этого выполним следующие шаги.

$\hat{\delta}(q_0, \varepsilon) = ECLOSE(q_0) = \{q_0, q_1\}$.

Вычисляем $\hat{\delta}(q_0, 5)$ следующим образом:

Находим переходы по символу 5 из состояний q_0 и q_1 полученных при вычислении $\hat{\delta}(q_0, \varepsilon)$: $\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$.

Находим ε -замыкание элементов, вычисленных на шаге (1). Получаем: $ECLOSE(q_1) \cup ECLOSE(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$.

Эта двушаговая схема применяется к следующим двум символам.

Вычисляем $\hat{\delta}(q_0, 5)$.

Сначала $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$

Затем $\hat{\delta}(q_0, 5.) = ECLOSE(q_2) \cup ECLOSE(q_4) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$.

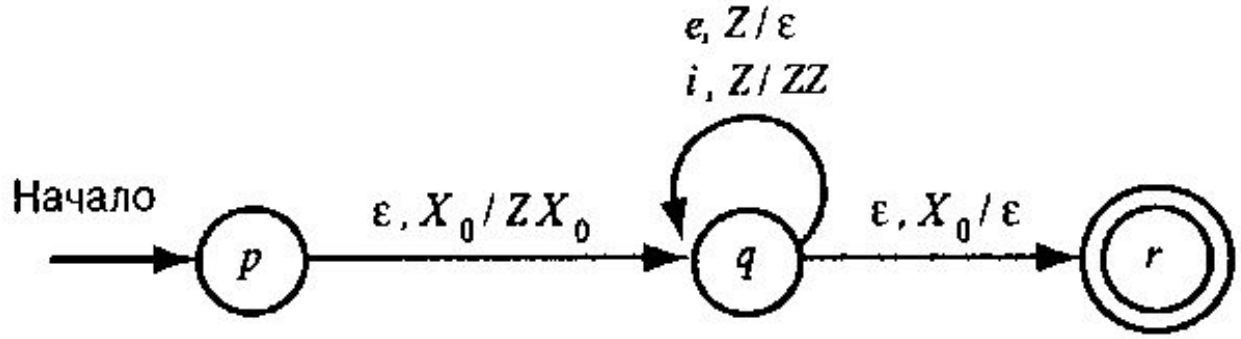
Наконец, вычисляем $\hat{\delta}(q_0, 5.6)$.

Сначала $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \emptyset = \{q_3\}$.

Затем $\hat{\delta}(q_0, 5.6) = ECLOSE(q_3) = \{q_3, q_5\}$

Теперь можно определить язык ε -НКА $E = (Q, \Sigma, \delta, q_0, F)$ так, как и было задумано ранее: $L(E) = \{\Omega | \hat{\delta}(q, w) \cap F \neq \emptyset\}$. Таким образом, язык E— это множество цепочек *w*, которые переводят автомат из начального состояния хотя бы в одно из допускающих.

$\hat{\delta}(q_0, 5.6)$ содержит допускающее состояние q_5, поэтому цепочка 5 . 6 принадлежит языку ε -НКА.

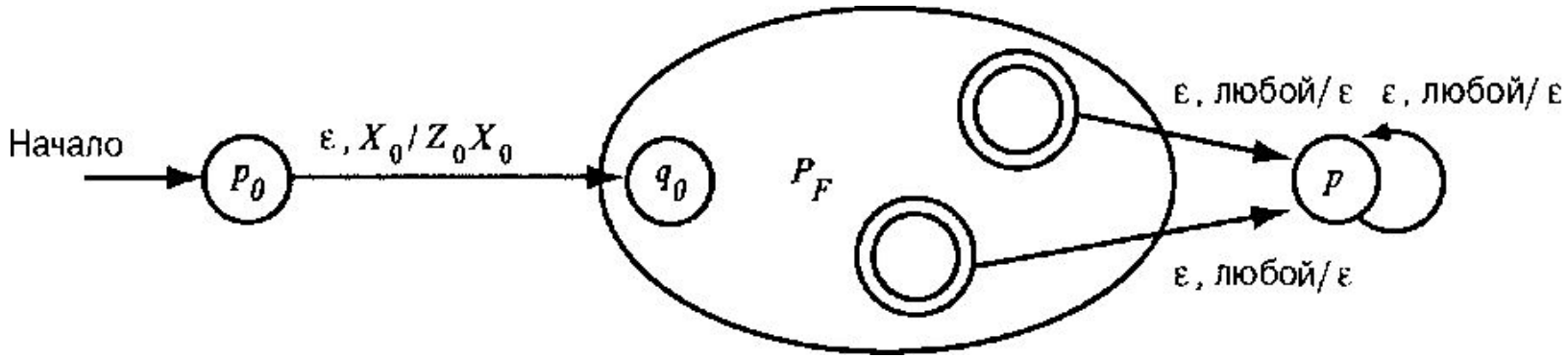


Пример перехода слайд 3/3

- Вводим новые начальное и заключительное состояния ***p*** и ***r,*** а также используем $\mathbf{X_0}$ в качестве маркера дна магазина. Формально автомат P_F определяется следующим образом: $P_F = (\{\mathbf{p}, \mathbf{q}, \mathbf{r}\}, \{\mathbf{i}, \mathbf{e}\}, \{\mathbf{Z}, \mathbf{X_0}\}, \boldsymbol{\delta}_\mathbf{F}, \mathbf{q}, \mathbf{Z})$
- ***Функция $\boldsymbol{\delta}_\mathbf{F}$ задается таким образом.***
  - $\boldsymbol{\delta}_\mathbf{F}(p, \varepsilon, X_0) = \{(q, ZX_0)\}$**.** По этому правилу ***P_F*** начинает имитировать ***P_N*** с маркером дна магазина.
  - $\boldsymbol{\delta}_\mathbf{F}(q, i, Z) = \{(q, ZZ)\}$. Z заталкивается при входе ***i,*** как у ***P_N*.**
  - $\boldsymbol{\delta_F}(\boldsymbol{q}, \boldsymbol{e}, \boldsymbol{Z}) = \{(\boldsymbol{q}, \boldsymbol{\varepsilon})\}$. Z выталкивается при входе ***e,*** как у ***P_N*.**
  - $\boldsymbol{\delta_F}(\boldsymbol{q}, \boldsymbol{e}, \boldsymbol{X_0}) = \{(\boldsymbol{r}, \boldsymbol{\varepsilon})\}$**.** ***P_F*** допускает, когда имитируемый ***P_N*** опустошает свой магазин.

Переход от заключительного состояния к пустому магазину слайд 1/2

- Теперь пойдем в обратном направлении: исходя из МП-автомата P_F, допускающего язык L по заключительному состоянию, построим другой МП-автомат P_N, который допускает L по пустому магазину. Конструкция проста и представлена на слайде.
- Добавляется переход по ε из каждого допускающего состояния автомата P_F в новое состояние p. Находясь в состоянии p, P_N опустошает магазин и ничего не прочитывает на входе.
- Таким образом, как только P_F приходит в допускающее состояние после прочтения w, P_N опустошает свой магазин, также прочитав w.



Переход от заключительного состояния к пустому магазину слайд 2/2

- Во избежание имитации случая, когда $\boldsymbol{P_F}$ случайно опустошает свой магазин без допуска, $\boldsymbol{P_N}$ должен также использовать маркер $\boldsymbol{X_0}$ на дне магазина. Маркер является его стартовым символом и аналогично конструкции теоремы слайда 192 $\boldsymbol{P_N}$ должен начинать работу в новом состоянии $\boldsymbol{p_0}$, единственная функция которого — затолкнуть стартовый символ автомата $\boldsymbol{P_F}$ в магазин и перейти в начальное состояние $\boldsymbol{P_F}$**.** В сжатом виде конструкция представлена на слайде 196, а ее формальное описание приводится в следующей теореме.
- Теорема. Пусть $\boldsymbol{L = L(P_F)}$ для некоторого МП-автомата $P_F = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$**.** Тогда существует такой МП-автомат $\boldsymbol{P_N}$**,** у которого $\boldsymbol{L = N(P_F)}$**.**

Эквивалентность МП-автоматов и КС-грамматик

Ниже будет показано, что МП-автоматы определяют КС-языки. План доказательства изображен на слайде. Цель состоит в том, чтобы доказать равенство следующих классов языков.

- Класс КС-языков, определяемых КС-грамматиками.
- Класс языков, допускаемых МП-автоматами по заключительному состоянию.
- Класс языков, допускаемых МП-автоматами по пустому магазину.

Уже было показано, что классы 2 и 3 равны. После этого достаточно доказать, что совпадают классы 1 и 2. Алгоритмы перехода показаны в [соответствующей презентации]

Детерминированные автоматы с магазинной памятью

- Хотя МП-автоматы по определению недетерминированы, их детерминированный случай чрезвычайно важен. В частности, синтаксические анализаторы в целом ведут себя как детерминированные МП-автоматы, поэтому класс языков, допускаемых этими авто матами, углубляет понимание конструкций, пригодных для языков программирования. В этом разделе определяются детерминированные МП-автоматы и частично исследуются работы, которые им под силу и на которые они не способны.

Определение ДМП-автомата слайд 1/2

Интуитивно МП-автомат является детерминированным, если в любой ситуации у него нет возможности выборов перехода. Эти выборы имеют два вида. Если $\delta(q, a, X)$ содержит более одной пары, то МП-автомат безусловно не является детерминированным, поскольку можно выбирать из этих двух пар. Однако если $\delta(q, a, X)$ всегда одноэлементно, все равно остается возможность выбора между чтением входного символа и совершением -перехода. Таким образом, МП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ определяется как *детерминированный* (ДМП-автомат), если выполнены следующие условия.

1. $\delta(q, a, X)$ имеет не более одного элемента для каждого *q* из *Q*, *a* из Σ или *a* = ε и *X из* Г.

2. Если $\delta(q, a, X)$ непусто для некоторого *a* из X, то $\delta(q, \varepsilon, X)$ должно быть пустым.

Пример. Оказывается, КС-язык L_{wwr} из примера выше не имеет ДМП-автомата.

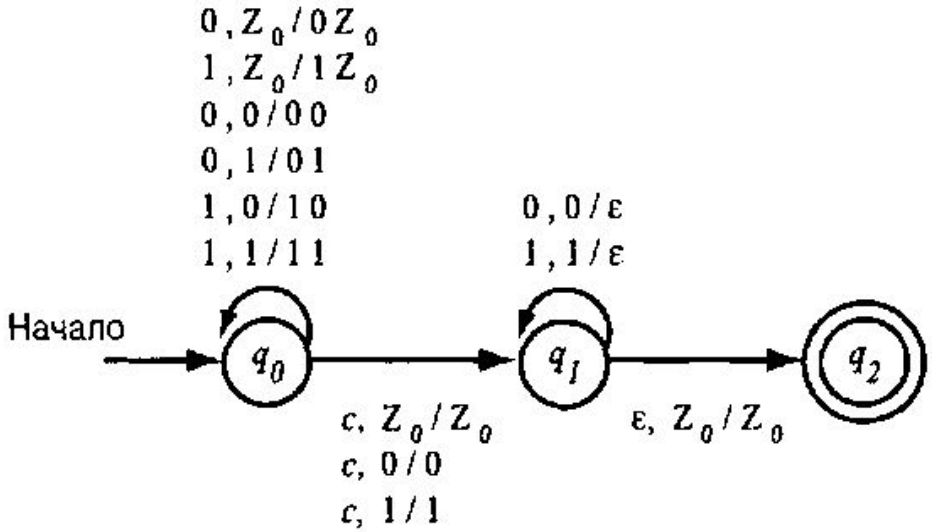
Однако путем помещения "центрального маркера" *c* в середину слов получается язык $L_{wcwr} = \{wcw^R | w \in (0+1)^*\}$, распознаваемый некоторым ДМП-автоматом.

Стратегией этого ДМП-автомата является заталкивание символов 0 и 1 в магазин до появления на входе маркера с. Затем автомат переходит в другое состояние, в котором сравнивает входные и магазинные символы и выталкивает магазинные в случае их совпадения.

Находя несовпадение, он останавливается без допускания ("умирает"); его вход не может иметь вид wcw^R. Если путем удаления магазинных символов он достигает стартового символа, отмечающего дно магазина, то он допускает свой вход.

Определение ДМП-автомата слайд 2/2

- По своей идее этот автомат очень похож на МП-автомат, изображенный на слайде 183. Однако тот МП-автомат был недетерминированным, поскольку в состоянии q_0 всегда имел возможность выбора между заталкиванием очередного входного символа в магазин и переходом в состояние q_1 без чтения входа, т.е. он должен был угадывать, достигнута ли середина. ДМП-автомат для L_{wcw} изображен в виде диаграммы переходов на слайде.



Регулярные языки и ДМП

ДМП-автоматы допускают класс языков, который находится между регулярными и КС- языками. Вначале докажем, что языки ДМП-автоматов включают в себя все регулярные.

Теорема. Если *L* — регулярный язык, то *L = L(P)* для некоторого ДМП-автомата *P.*

Если необходимо, чтобы ДМП-автомат допускал по пустому магазину, то обнаруживаем, что возможности по распознаванию языков существенно ограничены. Говорят, что язык *L* имеет *префиксное свойство,* или *свойство префиксности,* если в *L* нет двух различных цепочек *x и y,* где *x* является префиксом *y.*

Пример**.** Язык L_{wcwr} из примера выше имеет префиксное свойство, т.е. в нем не может быть двух разных цепочек wcw^R и xcx^R, одна из которых является префиксом другой. Чтобы убедиться в этом, предположим, что wcw^R — префикс xcx^R, и $w \neq x$. Тогда *w* должна быть короче, чем *x.* Следовательно, *c* в *w* приходится на позицию, в которой x имеет 0 или 1, а это противоречит предположению, что wcw^R — префикс xcx^R.

С другой стороны, есть очень простые языки, не имеющие префиксного свойства. Рассмотрим $\{0\}^*$, т.е. множество всех цепочек из символов 0. Очевидно, в этом языке есть пары цепочек, одна из которых является префиксом другой, так что этот язык не обладает префиксным свойством. В действительности, из *любых* двух цепочек одна является префиксом другой, хотя это условие и сильнее, чем то, которое нужно для отрицания префиксного свойства. Заметим, что язык $\{0\}^*$ регулярен. Таким образом, неверно, что каждый регулярный язык есть *N(P)* для некоторого ДМП-автомата *P.* Оставляем в качестве упражнения следующее утверждение.

Теорема. Язык *L* есть *N(P)* для некоторого ДМП-автомата *P* тогда и только тогда, когда *L* имеет префиксное свойство и *L* есть $L(P')$ для некоторого ДМП-автомата P'.

ДМП и КС-языки

- Выше было показано, что ДМП-автоматы могут допускать языки вроде L_{wcwr} которые не являются регулярными. Для того чтобы убедиться в его нерегулярности, предположим, что это не так, и используем лемму о накачке.

С другой стороны, существуют КС-языки, вроде L_{wwr} которые не могут допускаться по заключительному состоянию никаким ДМП-автоматом. Формальное доказательство весьма сложно, но интуитивно прозрачно. Если *P* — ДМП-автомат, допускающий L_{wwr}, то при чтении последовательности символов 0 он должен записать их в магазин или сделать что-нибудь равносильное для подсчета их количества. Например, записывать один X для каждых 00 на входе и использовать состояние для запоминания четности или нечетности числа символов 0.

Предположим, P прочитал n символов 0 и затем видит на входе 110^n. Он должен проверить, что после 11 находятся n символов 0, и для этого он должен опустошить свой магазин. Теперь он прочитал 0^n110^n. Если далее он видит идентичную цепочку, он должен допускать, поскольку весь вход имеет вид ww^R, где *w* = 0^n110^n. Однако если *P* видит 0^m110^n, где $m \neq n$, он должен *не допускать.* Поскольку его магазин пуст, он не может запомнить, каким было произвольное целое n, и не способен допустить L_{wwr}. Подведем итог.

- Языки, допускаемые ДМП-автоматами по заключительному состоянию, включают регулярные языки как собственное подмножество, но сами образуют собственное подмножество КС-языков.

ДМП и неоднозначные грамматики

Мощность ДМП-автоматов можно уточнить, заметив, что все языки, допускаемые ими, имеют однозначные грамматики. К сожалению, класс языков ДМП-автоматов не совпадает с подмножеством КС-языков, не являющихся существенно неоднозначными. Например, L_{wwr} имеет однозначную грамматику $S \rightarrow 0S0 \mid 1S1 \mid \varepsilon$, хотя и не является ДМП-автоматным языком. Следующая теорема уточняет заключительное утверждение из раздела выше.

Теорема. Если *L = N(P)* для некоторого ДМП-автомата *P,* то *L* имеет однозначную КС-грамматику.

Однако можно доказать больше: даже языки, допускаемые ДМП-автоматами по заключительному состоянию, имеют однозначные грамматики. Поскольку известно лишь, как строятся грамматики, исходя из МП-автоматов, допускающих по пустому магазину, то придется изменять язык, чтобы он обладал префиксным свойством, а затем модифицировать грамматику, чтобы она порождала исходный язык. Обеспечим это с помощью "концевого маркера".

Теорема. Если *L = L(P)* для некоторого ДМП-автомата *P,* то *L* имеет однозначную КС-грамматику.

Свойства контекстно-свободных языков

- Вначале определяются ограничения на структуру продукций КС-грамматик и доказывается, что всякий КС-язык имеет грамматику специального вида. Этот факт об легчает доказательство утверждений о КС-языках.
- Можно доказать "лемму о накачке" для КС-языков. Эта теорема аналогична теореме для регулярных языков, но может быть использована для доказательства того, что некоторые языки не являются контекстно-свободными. Далее рассматриваются свойства, изученные для регулярных языков, — свойства замкнутости и разре шимости. Показывается, что некоторые, но не все, свойства замкнутости регулярных языков сохраняются и у КС-языков. Часть задач, связанных с КС-языками, разрешается с помощью обобщения проверок, построенных для регулярных языков, но есть и ряд во просов о КС-языках, на которые нельзя дать ответ.

Нормальные формы КС-грамматик

Каждый КС-язык (без ε) порождается грамматикой, все продукции которой имеют форму $A \to BC$ или $A \to \alpha$, где A, B и C — переменные, α — терминал. Эта форма называется *нормальной формой Хомского.* Для ее получения нужно несколько предварительных преобразований, имеющих самостоятельное значение.

- Удалить *бесполезные символы,* т.е. переменные или терминалы, которые не встречаются в порождениях терминальных цепочек из стартового символа.
- Удалить *ε -продукции,* т.е. продукции вида $A \to \varepsilon$ для некоторой переменной A.
- Удалить *цепные продукции* вида $A \to B$ с переменными A и B.

Алгоритмы преобразований показаны в соответствующей презентации

Свойства замкнутости КС-языков

- Операции, порождающие контекстно-свободные языки, представлены в соответствующей презентации

Свойства разрешимости КС-языков

- Теперь рассмотрим, на какие вопросы о контекстно-свободных языках можно дать ответ. По аналогии с конечными автоматами, где речь шла о свойствах разрешимости регулярных языков, все начинается с представления КС-языка— с помощью грамматики или МП- автомата. Поскольку выше были показаны взаимные преобразования грамматик и МП-автоматов, можно предполагать, что доступны оба представления, и в каждом конкретном случае будем использовать более удобное.
- Разрешимых вопросов, связанных с КС-языками, совсем не много. Основное, что можно сделать, — это проверить, пуст ли язык, и принадлежит ли данная цепочка языку. Этот раздел завершается кратким обсуждением проблем, которые являются, как будет показано ниже, "неразрешимыми", т.е. не имеющими алгоритма разрешения. Начнем этот раздел с некоторых замечаний о сложности преобразований между грамматиками и МП-автоматами, задающими язык. Эти расчеты важны в любом вопросе об эффективности разрешения свойств КС-языков по данному их представлению.

Преобразование КС-грамматики и МП-автоматов слайд 1/4

- Прежде чем приступать к алгоритмам разрешения вопросов о КС-языках, рассмотрим сложность преобразования одного представления в другое. Время выполнения преобразования является составной частью стоимости алгоритма разрешения в тех случаях, когда алгоритм построен для одной формы представления, а язык дан в другой.
- В дальнейшем n будет обозначать длину представления МП-автомата или КС - грамматики. Использование этого параметра в качестве представления размера грамматики или автомата является "грубым", в том смысле, что некоторые алгоритмы имеют время выполнения, которое описывается точнее в терминах других параметров, например, число переменных в грамматике или сумма длин магазинных цепочек, встречаю щихся в функции переходов МП-автомата.
- Однако мера общей длины достаточна для решения наиболее важных вопросов: является ли алгоритм линейным относительно длины входа (т. е. требует ли он времени, чуть большего, чем нужно для чтения входа), экспоненциальным (т.е. преобразование выполнимо только для примеров малого размера) или нелинейным полиномиальным (т.е. алгоритм можно выполнить даже для больших примеров, но время будет значительным).

Преобразование КС-грамматики и МП-автоматов слайд 2/4

- Следующие преобразования линейны, будет показано ниже, относительно размера входных данных.
  - Преобразование КС-грамматики в МП-автомат.
  - Преобразование МП-автомата, допускающего по заключительному состоянию, в МП-автомат, допускающий по пустому магазину.
  - Преобразование МП-автомата, допускающего по пустому магазину, в МП-автомат, допускающий по заключительному состоянию.
- С другой стороны, время преобразования МП-автомата в грамматику существенно больше. Заметим, что *n,* общая длина входа, гарантированно является верхней границей числа состояний или магазинных символов, поэтому переменных вида $[pXq]$, построенных для грамматики, может быть не более n^3. Однако время выполнения преобразования может быть экспоненциальным, если у МП-автомата есть переход, помещающий большое число символов в магазин. Отметим, что одно правило может поместить в магазин почти *n* символов.

Преобразование КС-грамматики и МП-автоматов слайд 3/4

Если мы вспомним построение продукций грамматики по правилу вида "$\delta(q,a,X)$ содержит $(r_0,Y_1,Y_2,\cdots,Y_k)$" то заметим, что оно порождает набор продукций вида $[qXr_k]\to a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$ для всех последовательностей состояний $r_1,r_2\cdots r_k$. Поскольку к может быть близко к n, общее число продукций возрастает как n^n. Такое построение невозможно довести до конца для МП-автомата разумного размера, даже если он имеет всего одну цепочку, записываемую в магазин.

К счастью, этого наихудшего случая всегда можно избежать. Как предлагалось в примере выше, помещение длинной цепочки в магазин можно разбить на последовательность из не более, чем n шагов, на каждом из которых в магазин помещается всего один символ. Таким образом, если $\delta(q,a,X)$ содержит $(r_0,Y_1,Y_2,\cdots,Y_k)$, можно ввести новые состояния $p_2,p_3,\cdots,p_{k-1}$. Затем изменим $(r_0,Y_1,Y_2,\cdots,Y_k)$ в $\delta(q,a,X)$ на (p_{k-1},Y_{k-1},Y_k) и введем новые переходы $\delta(p_{k-1},\varepsilon,Y_{k-1})=\{(p_{k-2},Y_{k-2},Y_{k-1})\},\delta(p_{k-2},\varepsilon,Y_{k-2})=\{(p_{k-3},Y_{k-3},Y_{k-2})\}$

и так далее до $\delta(p_2,\varepsilon,Y_2)=\{(r_0,Y_1,Y_2)\}$.

Преобразование КС-грамматики и МП-автоматов слайд 4/4

Пример. Докажем нерегулярность языка L_{pr}, образованного всеми цепочками из единиц, длины которых — простые числа. Предположим, что язык L_{pr} регулярен. Тогда должна существовать константа n, удовлетворяющая условиям леммы о накачке. Рассмотрим некоторое простое число $p \geq n + 2$. Такое *p* должно существовать, поскольку множество простых чисел бесконечно. Пусть $w = 1^p$.

Согласно лемме о накачке можно разбить цепочку *w = xyz* так, что $y \neq \varepsilon$ и $|xy| \leq n$. Пусть $y = |m|$. Тогда $|xy| = p - m$. Рассмотрим цепочку $xy^{p-m}z$, которая по лемме о накачке должна принадлежать языку L_{pr}, если он действительно регулярен. Однако

$$|xy^{p-m}z| = |xy| + (\mathrm{p} - \mathrm{m})|\mathrm{y}| = \mathrm{p} - \mathrm{m} + (\mathrm{p} - \mathrm{m})\mathrm{m} = (\mathrm{m} + 1)(\mathrm{p} - \mathrm{m})$$

Очевидно, что число $|xy^{p-m}z|$ не простое, так как имеет два множителя m+1 и p-m. Однако нужно еще убедиться, что ни один из этих множителей не равен 1, потому что тогда число *(m+1)(p-m)* будет простым. Из неравенства $y \neq \varepsilon$ следует *m* > 1 и *m* + 1 > 1. Кроме того, $m = |y| \leq |xy| \leq \mathrm{n}$, а *p* > *п* + 2, поэтому *p-m >2.*

Начав с предположения, что предлагаемый язык регулярен, пришли к противоречию, доказав, что существует некоторая цепочка, которая не принадлежит этому языку, тогда как по лемме о накачке она должна ему принадлежать. Таким образом, язык L_{pr} нерегулярен.

Преобразование к НФХ слайд 1/2

- Алгоритмы могут зависеть от первичного преобразования в нормальную форму Хомского, поэтому посмотрим на время выполнения различных алгоритмов, использованных для приведения произвольной грамматики к НФХ. Большинство шагов сохраняют, с точностью до константного сомножителя, длину описания грамматики, т.е. по грамматике длиной *n* они строят другую длиной *O(n).* "Хорошие" (с точки зрения затрат времени) преобразования перечислены в следующем списке.

- С использованием подходящего алгоритма определение достижимых и порождающих символов грамматики может быть выполнено за линейное время, *O(n).* Удаление получившихся бесполезных символов требует *O(n)* времени и не увеличивает размер грамматики.
- Построение цепных пар и удаление цепных продукций требует времени $O(n^2)$, и получаемая грамматика имеет размер $O(n^2)$.
- Замена терминалов переменными в телах продукций(нормальная форма Хомского) требует времени $\boldsymbol{O(n)}$ и приводит к грамматике длиной $\boldsymbol{O(n)}$.
- Разделение тел продукций длины 3 и более на тела длины 2 также требует времени $\boldsymbol{O(n)}$ *и* приводит к грамматике длиной $\boldsymbol{O(n)}$.

"Плохой" является конструкция, в которой удаляются ε-продукции. По телу продукции длиной $\boldsymbol{k}$ можно построить $2^k - 1$ продукций новой грамматики. Поскольку $\boldsymbol{k}$ может быть пропорционально n, эта часть построения может занимать $O(2^n)$ времени и приводить к грамматике длиной $O(2^n)$.

Преобразование к НФХ слайд 2/2

Во избежание этого экспоненциального взрыва достаточно ограничить длины тел продукций. К каждому телу продукции можно применить прием, показанный выше, но только если в теле нет терминалов. Таким образом, в качестве предварительного шага перед удалением ε - продукций рекомендуется разделить все продукции с длинными телами на последовательность продукций с телами длины 2. Этот шаг требует времени $\boldsymbol{O(n)}$ и увеличивает грамматику только линейно. Конструкция для удаления ε-продукций будет работать с телами длиной не более 2 так, что время выполнения будет $\boldsymbol{O(n)}$ и полученная грамматика будет длиной $\boldsymbol{O(n)}$.

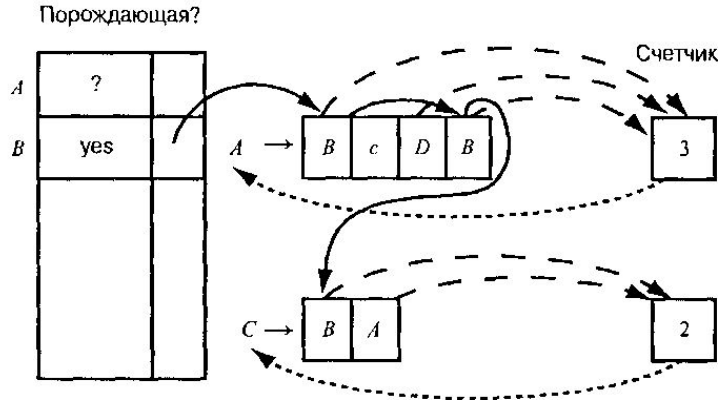
С такой модификацией общего построения НФХ единственным нелинейным шагом будет удаление цепных продукций. Поскольку этот шаг требует $O(n^2)$ времени, можно заключить следующее.

Теорема. По грамматике G длиной n можно найти грамматику в нормальной форме Хомского, эквивалентную G, за время $O(n^2)$; полученная грамматика будет иметь длину $O(n^2)$.

Проверка пустоты КС-языков слайд 1\4

- Алгоритм проверки пустоты КС-языка *L* нам уже знаком. Чтобы определить, является ли стартовый символ *S* данной грамматики *G* для языка *L* порождающим, можно использовать алгоритм, показанный выше. *L* пуст тогда и только тогда, когда *S* не является порождающим.
- Поскольку эта проверка весьма важна, рассмотрим детальнее, сколько времени требуется для поиска всех порождающих символов грамматики *G.* Пусть *G* имеет длину *n.* Тогда у нее может быть порядка *n* переменных, и каждый проход индуктивного обнаружения порождающих переменных может занимать $O(n)$ времени для проверки всех продукций G. Если на каждом проходе обнаруживается только одна новая порождающая переменная, то может понадобиться $O(n)$ проходов. Таким образом, простая реализация проверки на порождающие символы требует $O(n^2)$ времени, т.е. является квадратичной.

Проверка пустоты КС-языков слайд 2\4



Структура данных (см. рисунок) начинает с массива, индексированного переменными, как показано слева, который говорит, установлено ли, что переменная является порождающей. Массив на рисунке показывает, что переменная *B* уже обнаружена как порождающая, но о переменной A это еще неизвестно. В конце алгоритма каждая отметка "?" превращается в "нет", поскольку каждая переменная, не обнаруженная алгоритмом как порождающая, на самом деле является непорождающей.

Для продукций предварительно устанавливается несколько видов полезных ссылок. Во-первых, для каждой переменной заводится список всех возможных позиций, в которых эта переменная встречается. Например, список для переменной B представлен сплошными линиями. Во-вторых, для каждой продукции ведется счетчик числа позиций, содержащих переменные, способность которых породить терминальную цепочку еще не учтена. Пунктирные линии представляют связи, ведущие от продукций к их счетчикам. Счетчики, показанные на рис. 54, предполагают, что ни одна из переменных в телах продукций еще не учитывалась, хотя уже и установлено, что *B* — порождающая.

Предположим, мы уже обнаружили, что *B* — порождающая. Мы спускаемся по списку позиций в телах, содержащих B. Для каждой такой позиции счетчик ее продукции уменьшаем на 1 позицию, которые нужны для заключения, что переменная в голове продукции тоже порождающая, остается на одну меньше.

Если счетчик достигает 0, то понятно, что переменная в голове продукции является порождающей. Связь, представленная точечными линиями, приводит к переменной, и эту переменную можно поместить в очередь переменных, о которых еще неизвестно, порождают ли они (переменная *B* уже исследована). Эта очередь не показана.

Проверка пустоты КС-языков слайд 3\4

Обоснуем, что этот алгоритм требует $O(n)$ времени. Важными являются следующие утверждения.

Поскольку грамматика размера n имеет не более n переменных, создание и инициализация массива требует времени $O(n)$.

Есть не более n продукций, и их общая длина не превосходит $O(n)$, поэтому инициализация связей и счетчиков, представленных на слайде 127, может быть выполнена за время $O(n)$.

Когда обнаруживается, что счетчик продукции получил значение 0, т.е. все позиции в ее теле являются порождающими, вся проделанная работа может быть разделена на следующие два вида.

- Работа, выполненная для продукции: обнаружение, что счетчик обнулен, поиск переменной, скажем, *A*, в голове продукции, проверка, установлено ли, что эта переменная является порождающей, и помещение ее в очередь, если это не так. Все эти шаги требуют $O(1)$ времени для каждой продукции, поэтому вся такая работа в целом требует $O(n)$ времени.
- Работа, выполненная при посещении позиций в телах продукций, имеющих переменную *A* в голове. Эта работа пропорциональна числу позиций с переменной *A*. Следовательно, совокупная работа, выполненная со всеми порождающими переменными, пропорциональна сумме длин тел продукций, а это $O(n)$.

Отсюда делаем вывод, что общая работа, выполненная этим алгоритмом, есть $O(n)$.

Проверка пустоты КС-языков слайд 4\4

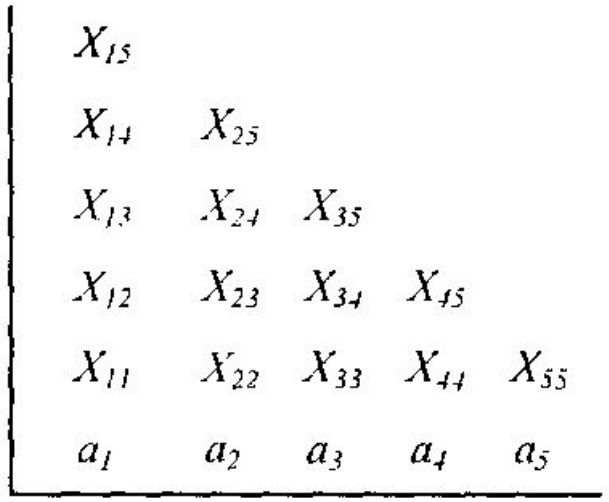
- Структура данных и счетчики, показанные выше для проверки, является ли переменная порождающей, могут использоваться для обеспечения линейности времени некоторых других проверок. Назовем два важных примера.
  - Какие символы достижимы?
  - Какие символы являются ε-порождающими?

Проверка принадлежности КС-языку слайд 1/4

- Проблема принадлежности цепочки *w* КС-языку *L* также разрешима. Есть несколько неэффективных способов такой проверки; они требуют времени, экспоненциального относительно $|w|$, в предположении, что язык *L* представлен заданной грамматикой или МП-автоматом, и размер представления считается константой, не зависящей от w. Например, начнем с преобразования какого-либо данного нам представления в НФХ-грамматику. Поскольку дерево разбора в такой грамматике является бинарным, при длине n слова w в дереве будет ровно $2^n - 1$ узлов, отмеченных переменными. Количество возможных деревьев и разметок их узлов, таким образом, "всего лишь" экспоненциально относительно n, поэтому в принципе их можно перечислить, и проверить, имеет ли какое-нибудь из деревьев крону *w.*
- Существует гораздо более эффективный метод, основанный на идее "динамического программирования" и известный также, как "алгоритм заполнения таблицы" или "табуляция". Данный алгоритм известен как *CYK-алгоритм (алгоритм Кока-Янгера-Касами).* Он начинает с НФХ-грамматики $G = (V, T, P, S)$ для языка *L.* На вход алгоритма подается цепочка $w = a_1 a_2 \cdots a_n$ из *T**. За время $O(n^3)$ алгоритм строит таблицу, которая говорит, принадлежит ли *w* языку *L.* Отметим, что при вычислении этого времени сама по себе грамматика рассматривается фиксированной, и ее размер вносит лишь константный сомножитель в оценку времени, измеряемого в терминах длины цепочки, проверяемой на принадлежность *L.*

Проверка принадлежности КС-языку слайд 2/4

- В CYK-алгоритме строится треугольная таблица (см. рис). Горизонтальная ось соответствует позициям цепочки $w = a_1 a_2 \cdots a_m$ имеющей в нашем примере длину 5. Содержимое клетки, или вход таблицы X_{ij}, есть множество таких переменных *A*, для которых $A \overset{*}{\Rightarrow} a_i a_{i+1} \cdots a_j$. Заметим, в частности, что нас интересует, принадлежит ли *S* множеству X_{1n}, поскольку это то же самое, что $S \Rightarrow w$, т.е. $w \in L$.
- Таблица заполняется построчно снизу вверх. Отметим, что каждая строка соответствует определенной длине подцепочек; нижняя— подцепочкам длины 1, вторая снизу — подцепочкам длины 2 и так далее до верхней строки, соответствующей одной подцепочке длиной п, т.е. w. Ниже обсуждается метод, с помощью которого вычисление одного входа требует времени *0(п)*. Поскольку всего входов *п(п* + 1)/2, весь процесс построения таблицы занимает $O(n^3)$ времени.

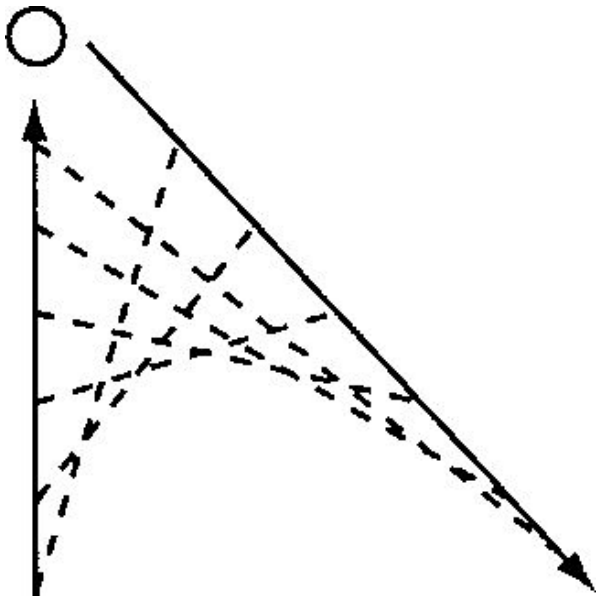


Проверка принадлежности КС-языку слайд 3/4

- Таким способом для каждого состояния *q,* можно вычислить множество состояний, достижимых из q_i, вдоль пути с меткой *a* (возможно, включая дуги, отмеченные ε). Поскольку $k \leq n$, то существует не более *n* таких состояний q_i и для каждого из них вычисление достижимых состояний занимает время $O(n^2)$. Таким образом, общее время вычисления достижимых состояний равно $O(n^3)$. Для объединения множеств достижимых состояний потребуется только $O(n^2)$ дополнительного времени, следовательно, вычисление одного перехода ДКА занимает время $O(n^3)$.
- Заметим, что количество входных символов считается постоянным и не зависит от n. Таким образом, как в этой, так и в других оценках времени работы количество входных символов не рассматривается. Размер входного алфавита влияет только на постоянный коэффициент, скрытый в обозначении "О большого".
- Итак, время преобразования НКА в ДКА, включая ситуацию, когда НКА содержит е- переходы, равно $O(n^3 2^n)$. Конечно, на практике обычно число состояний, которые строятся, намного меньше 2^n. Иногда их всего лишь *n.* Поэтому можно установить оценку времени работы равной $O(n^3 s)$, где s — это число состояний, которые в действительности есть у ДКА.

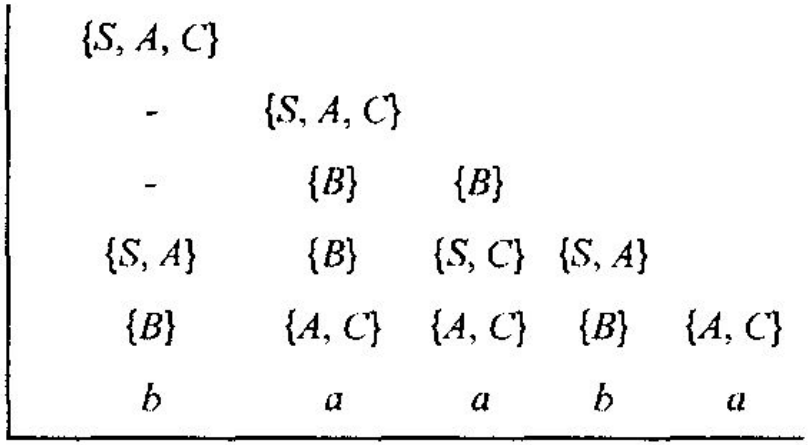
Проверка принадлежности КС-языку слайд 4/4

- Таким образом, в процессе работы происходит подъём по колонке, расположенной под X_{ij}, и одновременно спуск по диагонали
- Теорема. Вышеописанный алгоритм корректно вычисляет X_{ij} для всех i и j. Таким образом, $w \in L(G)$ тогда и только тогда, когда $S \in X_{1n}$. Кроме того, время выполнения алгоритма есть $O(n^3)$.



Пример проверки принадлежности слайд 1/2

- Для преобразования регулярного выражения в ε-НКА потребуется линейное время. Необходимо эффективно проанализировать регулярное выражение, используя метод, занимающий время $O(n)$ для регулярного выражения длины n. В результате получим дерево с одним узлом для каждого символа регулярного выражения (хотя скобки в этом дереве не встречаются, поскольку они только управляют разбором выражения).
- Полученное дерево заданного регулярного выражения можно обработать, конструируя ε-НКА для каждого узла. Правила преобразования регулярного выражения, представленные выше, никогда не добавляют более двух состояний и четырех дуг для каждого узла дерева выражения. Следовательно как число состояний, так и число дуг результирующего ε-НКА равны $O(n)$. Кроме того, работа по созданию этих элементов в каждом узле дерева анализа является постоянной при условии, что функция, обрабатывающая каждое поддерево, возвращает указатели в начальное и допускающие состояния этого автомата.
- Приходим к выводу, что построение ε-НКА по регулярному выражению занимает время, линейно зависящее от размера выражения. Можно исключить ε-переходы из ε-НКА с ***n*** состояниями, преобразовав его в обычный НКА за время $O(n^3)$ и не увеличив числа состояний. Однако преобразование в ДКА может занять экспоненциальное время.



Пример проверки принадлежности слайд 2/2

- Если язык L представлен регулярным выражением, а не автоматом, то можно преобразовать это выражение в ε-НКА, а далее продолжить так, как описано выше. Поскольку автомат, полученный в результате преобразования регулярного выражения длины *n,* содержит не более $O(n)$ состояний и переходов, для выполнения алгоритма потребуется время $O(n)$.
- Можно проверить само выражение — пустое оно, или нет. Сначала заметим, что если в данном выражении ни разу не встречается Ø, то его язык гарантированно не пуст. Если же в выражении встречается Ø, то язык такого выражения не обязательно пустой. Используя следующие рекурсивные правила, можно определить, представляет ли заданное регулярное выражение пустой язык.
- Базис. Ø обозначает пустой язык, но ε и **a** для любого входного символа *a* обозначают не пустой язык.
- Индукция. Пусть *R* — регулярное выражение. Нужно рассмотреть четыре варианта, соответствующие возможным способам построения этого выражения.
- $R = R_1 + R_2$. $L(R)$ пуст тогда и только тогда, когда оба языка $L(R_1)$ и $L(R_2)$ пусты.
- $R = R_1 R_2$. $L(R)$ пуст тогда и только тогда, когда хотя бы один из языков $L(R_1)$ или $L(R_2)$ пуст.
- $R = R_1^* R_2$. $L(R)$ не пуст: он содержит цепочку ε.
- $R = R_1$. $L(R)$ пуст тогда и только тогда, кода $L(R_1)$ пуст, так как эти языки равны.

Неразрешимые проблемы КС-языков

- Следующий важный вопрос состоит в том, принадлежит ли данная цепочка *w* данному регулярному языку *L.* В то время, как цепочка *w* задается явно, язык *L* представляется с помощью автомата или регулярного выражения.
- Если язык *L* задан с помощью ДКА, то алгоритм решения данной задачи очень прост. Имитируем ДКА, обрабатывающий цепочку входных символов w, начиная со стартового состояния. Если ДКА заканчивает в допускающем состоянии, то цепочка w принадлежит этому языку, в противном случае— нет. Этот алгоритм является предельно быстрым. Если $|w| = n$ и ДКА представлен с помощью подходящей структуры данных, например, двухмерного массива (таблицы переходов), то каждый переход требует постоянного времени, а вся проверка занимает время ***0(п).***

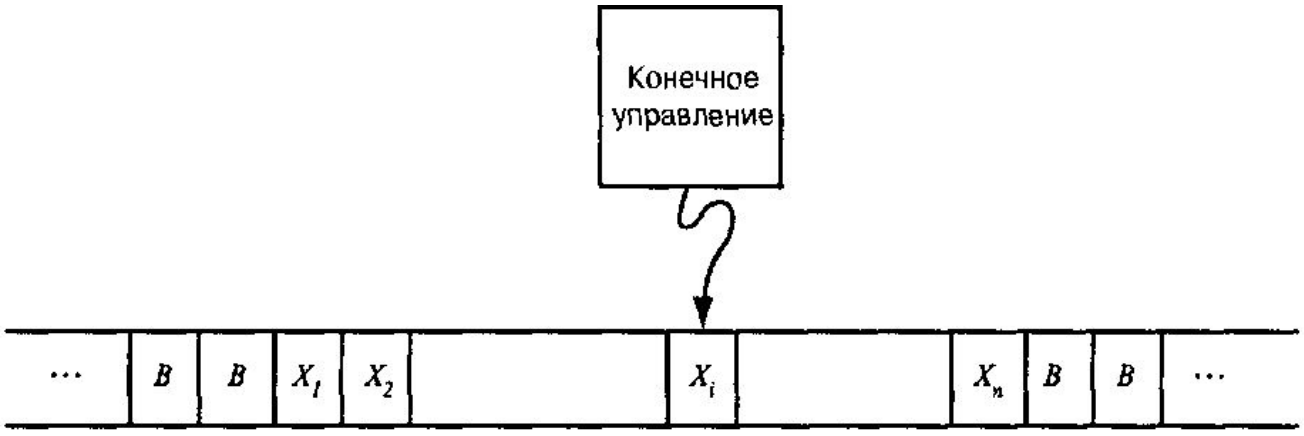
Машина Тьюринга и теория неразрешимых проблем

- Цель теории неразрешимых проблем состоит не только в том, чтобы установить существование таких проблем (что само по себе не просто), но также в том, чтобы обеспечить программистов информацией, какие задачи программирования можно решить, а какие нельзя. Теория также имеет огромное практическое значение, когда рассматриваются проблемы, которые хотя и разрешимы, но требуют слишком большого времени для их решения. Эти проблемы, называемые "трудно разрешимыми", или "труднорешаемыми", подчас доставляют программистам и разработчикам систем больше хлопот, чем неразрешимые проблемы. Причина в том, что неразрешимые проблемы редко возникают на практике, тогда как трудно разрешимые встречаются каждый день. Кроме того, они часто допускают небольшие модификации условий или эвристические решения. Таким образом, разработчику постоянно приходится решать, относится ли данная проблема к классу труднорешаемых и что с ней делать, если это так.
- Нужен инструмент, позволяющий доказывать неразрешимость или труднорешаемость повседневных вопросов. Методика, применимая для вопросов, касающихся программ, очень сложно переносится на проблемы, не связанные с программами. Например, было бы нелегко свести проблему "hello, world" к проблеме неоднозначности грамматики.
- Таким образом, нужно перестроить теорию неразрешимости, основанную не на программах на каком-либо языке, а на очень простой модели компьютера, которая называется машиной Тьюринга. Это устройство по существу представляет собой конечный автомат с бесконечной лентой, на которой он может как читать, так и записывать данные. Одно из преимуществ машин Тьюринга по сравнению с программами как представлением вычислений состоит в том, что машина Тьюринга достаточно проста и ее конфигурацию можно точно описать, используя нотацию, весьма похожую на МО МП-автоматов. Для сравнения, состояние С-программы включает все переменные, возникающие при выполнении любой цепочки вызовов функций, и нотация для описания этих состояний настолько сложна, что практически не позволяет проводить понятные формальные доказательства.
- С использованием нотации машин Тьюринга можно доказать неразрешимость некоторых проблем, не связанных с программированием. Например, ниже будет показано, что "проблема соответствий Поста", простой вопрос о двух списках цепочек, неразрешим, и что эта проблема облегчает доказательство неразрешимости вопросов о грамматиках, вроде неоднозначности. Аналогично, когда будут введены трудно разрешимые проблемы, мы увидим, что к их числу принадлежат и определенные вопросы, казалось бы, не связанные с вычислениями, например, выполнимость булевских формул.

Решение математических вопросов

- В начале XX столетия математик Д. Гильберт поставил вопрос о поисках алгоритма, который позволял бы определить истинность или ложность любого математического утверждения. В частности, он спрашивал, есть ли способ определить, истинна или ложна произвольная формула в исчислении предикатов первого порядка с целыми числами. Исчисление предикатов первого порядка с целыми (арифметика) достаточно мощно, чтобы выразить утверждения типа "эта грамматика неоднозначна" или "эта программа печатает hello, world". Поэтому, окажись предположение Гильберта правильным, данные проблемы были бы разрешимыми.
- Однако в 1931 г. К. Гедель опубликовал свою знаменитую теорему о неполноте. Он доказал, что существует истинная формула первого порядка с целыми, которую нельзя ни доказать, ни опровергнуть в исчислении предикатов первого порядка над целыми.
- Исчисление предикатов было не единственным понятием, применяемым для форма лизации "любого возможного вычисления". В действительности, исчисление предикатов, будучи декларативным, а не вычислительным, конкурировало с различными нотациями, включая "частично рекурсивные функции". В 1936 г. А. М. Тьюринг в качестве модели "любого возможного вычисления" предложил машину Тьюринга. Эта модель была не декларативной, а "машиноподобной", хотя настоящие электронные и даже электромеханические машины появились лишь несколько лет спустя (и сам Тьюринг занимался их разработкой во время второй мировой войны).
- Интересно, что все когда-либо предложенные серьезные вычислительные модели имеют одинаковую мощность, т.е. все они вычисляют одни и те же функции или распо знают одни и те же множества. Недоказуемое предположение, что любой общий метод вычислений позволяет вычислять лишь частично рекурсивные функции (и их же способны вычислять машины Тьюринга или современные компьютеры), известно как *гипотеза Черча* (следуя логике А. Черча), или *тезис Черча-Тьюринга.*

Описание машины Тьюринга



- Машина Тьюринга изображена на рисунке. Машина состоит из *конечного управления,* которое может находиться в любом из конечного множества состояний. Есть *лента,* разбитая на *клетки*; каждая клетка может хранить любой символ из конечного их множества.

Элементы машины Тьюринга

- Изначально на ленте записан *вход,* представляющий собой цепочку символов конечной длины. Символы выбраны из *входного алфавита.* Все остальные клетки, до бесконечности, слева и справа от входа содержат специальный символ, называемый *пустым символом,* или *пробелом.* Он является не входным, а *ленточным символом.* Кроме вход ных символов и пробела возможны другие ленточные символы.
- *Ленточная головка* (далее просто *головка*) всегда устанавливается на какую-то из клеток ленты, которая называется *сканируемой,* или *обозреваемой.* Вначале обозревается крайняя слева клетка входа.
- *Переход* машины Тьюринга — это функция, зависящая от состояния конечного управления и обозреваемого символа. За один переход машина Тьюринга должна выполнить следующие действия.

1. Изменить состояние. Следующее состояние может совпадать с текущим.
2. Записать ленточный символ в обозреваемую клетку. Этот символ замещает любой символ в этой клетке, в частности, символы могут совпадать.
3. Сдвинуть головку влево или вправо. В нашей формализации не допускается, чтобы головка оставалась на месте. Это ограничение не изменяет того, что могут вычислить машины Тьюринга, поскольку любая последовательность переходов с остающейся на месте головкой и последующим сдвигом может быть сжата до одного перехода с из менением состояния и ленточного символа и сдвигом головки влево или

Формальное описание машины Тьюринга

- Формальная запись, используемая для *машин Тьюринга* (МТ), похожа на запись конечных автоматов или МП-автоматов. МТ описывается семеркой $\boldsymbol{M} = (\boldsymbol{Q}, \boldsymbol{\Sigma}, \Gamma, \boldsymbol{\delta}, \boldsymbol{q_0}, \boldsymbol{B}, \boldsymbol{F})$, компоненты которой имеют следующий смысл:
- Q — конечное множество *состояний* конечного управления.
- $\boldsymbol{\Sigma}$ — конечное множество *входных символов.*
- Γ — множество *ленточных символов,* $\boldsymbol{\Sigma}$ всегда есть подмножество Г.
- $\boldsymbol{\delta}$ — *функция переходов.* Аргументами $\boldsymbol{\delta}(\boldsymbol{q}, \boldsymbol{X})$ являются состояние q и ленточный символ X, а значением, если оно определено, — тройка $(\boldsymbol{p}, \boldsymbol{Y}, \boldsymbol{D})$. В этой тройке p есть следующее состояние из Q, Y— символ из Г, который записывается вместо символа в обозреваемой клетке, а D — *направление* сдвига головки "влево" или "вправо", обозначаемое, соответственно, как L или R.
- $\boldsymbol{q_0}$ — *начальное состояние* из Q, в котором управление находится в начале.
- B — *пустой символ,* или *пробел.* Этот символ принадлежит Г, но не $\boldsymbol{\Sigma}$, т.е. не является входным. Вначале он записан во всех клетках, кроме конечного их числа, где хранятся входные символы. Остальные символы называются значащими.
- F — множество *заключительных,* или *допускающих,* состояний; является подмножеством Q.

Конфигурации машины Тьюринга слайд 1/2

- Для формального описания работы машины Тьюринга нужно построить систему записи конфигураций, или *мгновенных описаний (МО),* подобную нотации, определенной для МП-автоматов. Поскольку МТ имеет неограниченно длинную ленту, можно подумать, что конечное описание конфигураций МТ невозможно. Однако после любого конечного числа шагов МТ может обозреть лишь конечное число клеток, хотя и ничем не ограниченное. Таким образом, в любом МО есть бесконечный префикс и бесконечный суффикс из клеток, которые еще не обозревались. Все эти клетки должны содержать или пробелы, или входные символы из конечного их множества. Таким образом, в МО включаются только клетки между крайними слева и справа значащими символами. В отдельных случаях, когда головка обозревает один из пробелов перед или за участком значащих символов, конечное число пробелов также включается в МО.
- Кроме ленты, нужно представить конечное управление и позицию головки. Для этого состояние помещается непосредственно слева от обозреваемой клетки. Во избежание не-однозначности получаемой цепочки, состояния обозначаются символами, отличными от ленточных. Таким образом, для представления МО используется цепочка $X_1X_2 \cdots X_{i-1}qX_iX_{i+1} \cdots X_n$. Здесь q — состояние МТ, головка обозревает i-ю слева клетку, а $X_1X_2 \cdots X_n$ представляет собой часть ленты между крайними слева и справа значащими символами. Как исключение, если головка находится слева или справа от значащих символов, некоторое начало или окончание $X_1X_2 \cdots X_n$ пусто, а i имеет значение, соответственно, 1 или n.

Конфигурации машины Тьюринга слайд 2/2

Переходы МТ $\boldsymbol{M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)}$ описываются с помощью отношения $\vdash_M$, использованного для МП-автоматов. Подразумевая МТ М, для отображения переходов используем $\vdash$. Как обычно, для указания нуля или нескольких переходов МТ М используется отношение $\vdash_M^*$ или $\vdash^*$.

Пусть $\boldsymbol{\delta(q, X) = (p, Y, L)}$, т.е. головка сдвигается влево. Тогда

$\boldsymbol{X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \vdash X_1 X_2 \cdots X_{i-2} p X_{i-1} Y X_{i+1} \cdots X_n}$

Заметим, как этот переход отображает изменение состояния на р и сдвиг головки на клетку i-1. Здесь есть два важных исключения.

- Если i= 1, то М переходит к пробелу слева от $\boldsymbol{X_1}$. В этом случае $\boldsymbol{q X_1 X_2 \cdots X_n \vdash pBY X_2 \cdots X_n}$.
- Если $\boldsymbol{i = n}$ и $\boldsymbol{Y=}$ B, то символ $\boldsymbol{B}$, заменяющий $\boldsymbol{X_n}$, присоединяется к бесконечной последовательности пробелов справа и не записывается в следующем МО. Таким образом,$\boldsymbol{X_1 X_2 \cdots X_{n-1} q X_n \vdash X_1 X_2 \cdots X_{n-2} p X_{n-1}}$

Пусть теперь $\boldsymbol{\delta(q, X) = (p, Y, R)}$, т.е. головка сдвигается вправо. Тогда

$\boldsymbol{X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \vdash X_1 X_2 \cdots X_{i-1} Y p X_{i+1} \cdots X_n}$.

Этот переход отражает сдвиг головки в клетку i + 1. Здесь также есть два важных исключения.

- Если $\boldsymbol{i = n}$, то (i+ 1)-я клетка содержит пробел и не является частью предыдущего МО. Таким образом, $\boldsymbol{X_1 X_2 \cdots X_{n-1} q X_n \vdash X_1 X_2 \cdots X_{n-1} Y p B}$
- Если i= 1 и $\boldsymbol{Y= B}$, то символ $\boldsymbol{B}$, записываемый вместо X_1 присоединяется к бесконечной последовательности пробелов слева и опускается в следующем МО. Таким образом, $\mathrm{qX_1X_2 \cdots X_n \vdash pX_2 \cdots X_n}$.

Пример машины Тьюринга слайд 1/4

- Построим машину Тьюринга и посмотрим, как она ведет себя на типичном входе. Данная машина Тьюринга будет допускать язык $\{0^n 1^n | n \geq 1\}$. Изначально на ее ленте записана конечная последовательность символов 0 и 1, перед и за которыми находятся бесконечные последовательности пробелов. МТ попеременно будет изменять 0 на ***X*** *и* 1 на ***Y*** до тех пор, пока все символы 0 и 1 не будут сопоставлены друг другу.
- Более детально, начиная с левого конца входной последовательности, МТ циклично меняет 0 на X и движется вправо через все символы 0 и Y, пока не достигнет 1. Она меняет 1 на Y и движется вправо через все символы Y и 0, пока не найдет X. В этот момент она ищет 0 непосредственно справа и, если находит, меняет его на X и продолжает процесс, меняя соответствующую 1 на *Y.*
- Если непустой вход не принадлежит 0*1*, то МТ рано или поздно не сможет совершить следующий переход и остановится без допускания. Однако если она заканчивает работу, изменив все символы 0 на X в том же цикле, в котором она изменила последнюю 1 на *Y,* то ее вход имеет вид $0^n 1^n$, и она его допускает.

Пример машины Тьюринга слайд 2/4

Состояние	Символ 0	1	X	Y	B
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—

- Формальным описанием данной МТ является $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0,1\}, \{0,1, X, Y, B\}, \boldsymbol{\delta}, \boldsymbol{q_0}, \boldsymbol{B}, \{q_4\})$, для которой $\boldsymbol{\delta}$ представлена в таблице

Пример машины Тьюринга слайд 3/4

- В процессе вычислений M часть ленты, на которой побывала ее головка, всегда содержит последовательность символов, описываемую регулярным выражением $X^*0^*1^*Y^*$. Таким образом, там есть последовательность символов X, заменивших 0, за которыми идут символы 0, еще не измененные на X. Затем идут символы *Y,* заменившие 1, и символы 1, еще не замененные Y. За этой последовательностью еще могут находиться символы 0 и 1.
- Состояние q_0 является начальным, и в него же переходит *M,* возвращаясь к крайнему слева из оставшихся символов 0. Если *M* находится в состоянии q_0 и обозревает 0, то в соответствии с правилами она переходит в состояние q_1, меняет 0 *на X и* сдвигается вправо. Попав в состояние q_1 M продолжает движение вправо через все символы 0 и *Y.* Если *M* видит *X* или *B,* она останавливается ("умирает"). Однако, если *M* в состоянии q_1 видит 1, она меняет ее на *Y,* переходит в состояние q_2 и начинает движение влево.
- Находясь в состоянии q_2, *M* движется влево через все символы 0 и *Y.* Достигая крайнего справа X, который отмечает правый конец блока нулей, измененных на X, M возвращается в состояние q_0 и сдвигается вправо. Возможны два случая.
- Если M видит 0, то она повторяет описанный только что цикл.
- Если же *M* обозревает *Y,* то она уже изменила все нули на *X.* Если все символы 1 заменены *Y,* то вход имел вид $\{\mathbf{0^n1^n}\}$, и *M* должна допускать. Таким образом, *M* переходит в состояние q_3 и начинает движение вправо по символам *Y.* Если первым после *Y* появляется пробел, то символов 0 и 1 было поровну, поэтому M переходит в состояние q_4 и допускает. Если же *M* обнаруживает еще одну 1, то символов 1 слишком много, и M останавливается, не допуская. Если M находит 0, то вход имеет ошибочный вид, и M также "умирает".

Пример машины Тьюринга слайд 4/4

Приведем пример допускающего вычисления M на входе 0011. Вначале M находится в состоянии q_0, обозревая 0, т.е. начальное МО имеет вид $q_0 0011$. Полная последовательность переходов образована следующими МО.

$$q_0 0011 \vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0 Y 1 \vdash \mathrm{q}_2 X 0 Y 1 \vdash \mathbf{X q_0} 0 Y 1 \vdash \mathbf{XX q_1} Y 1$$
$$\vdash \mathbf{XXY q_1} 1 \vdash \mathbf{XX q_2} YY \vdash \mathbf{X q_2} XYY \vdash \mathbf{XX q_0} YY \vdash \mathbf{XXY q_3} Y$$
$$\vdash \mathbf{XXYY q_3} B \vdash \mathbf{XXYYB q_4} B$$

Рассмотрим поведение M на входе 0010, который не принадлежит допускаемому языку.

$$q_0 0010 \vdash X q_1 010 \vdash X 0 q_1 10 \vdash X q_2 0 Y 0 \vdash \mathrm{q}_2 X 0 Y 0 \vdash \mathbf{X q_0} 0 Y 0 \vdash \mathbf{XX q_1} Y 0$$
$$\vdash \mathbf{XXY q_1} 0 \vdash \mathbf{XXY0 q_1} B$$

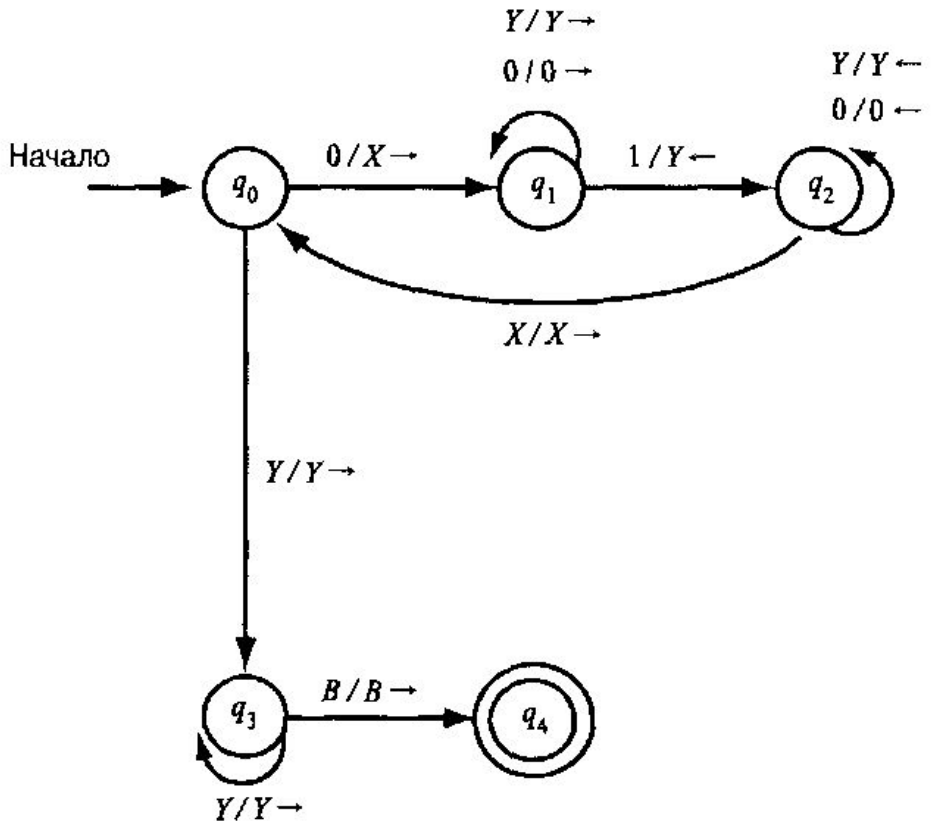
Это поведение похоже на обработку входа 0011 до МО $\mathbf{XXY q_1} 0$, где *M* впервые обозревает последний 0. *M* должна двигаться вправо, оставаясь в состоянии q_1, что приводит к $\mathbf{XXY0 q_1} B$. Однако у *M в* состоянии $\mathbf{q_1}$ по символу *B* перехода нет, поэтому она останавливается, не допуская.

Диаграмма переходов для машины Тьюринга

- Переходы машин Тьюринга можно представить графически, как и переходы МП-автоматов. *Диаграмма переходов* состоит из множества узлов, соответствующих состояниям МТ. Дуга из состояния *q* в состояние *p* отмечена одним или несколькими элементами вида $\mathrm{X/Y\ D}$, где *X* и *Y*— ленточные символы, а *D* — направление *(L* или *R).* Таким образом, если $\delta(\mathrm{q, X}) = (\mathrm{p, Y, D})$, то отметка *X/ YD* находится на дуге из *q* в *p.* Направление на диаграммах представляется не буквами LnR,а стрелками ← и →, соответственно.
- Как и в других видах диаграмм переходов, начальное состояние представлено словом "начало" и стрелкой, входящей в это состояние. Допускающие состояния выделены двойными кружками. Таким образом, непосредственно из диаграммы о МТ известно все, кроме того, какой символ обозначает пробел. В дальнейшем считается, что это *B,* если не оговорено иное.

Пример диаграммы переходов

- Ниже представлена диаграмма переходов для машины Тьюринга из рассмотренного выше примера. Ее функция переходов изображена на слайде 234



Машина Тьюринга для усечённой разности слайд 1/4

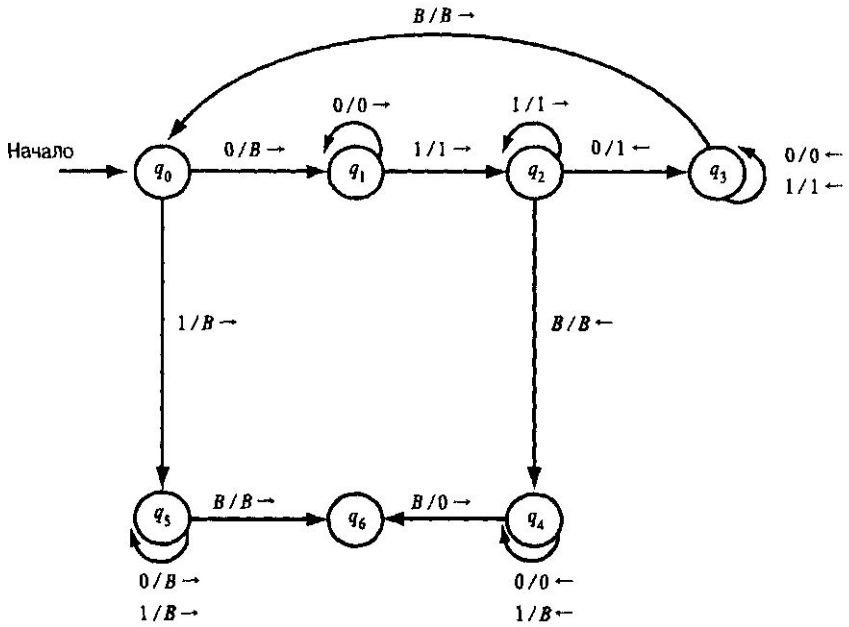
- Сегодня машины Тьюринга рассматриваются чаще всего в качестве "распознавателей языков" или, что равносильно, "решателей проблем". Однако сам Тьюринг рассматривал свою машину как вычислитель натуральнозначных функций. В его схеме натуральные числа представлялись в единичной системе счисления, как блоки из одного и того же символа, и машина вычисляла, изменяя длину блоков или строя новые блоки где-нибудь на ленте. В данном простом примере будет показано, как машина Тьюринга может вычислить функцию $\dot{-}$, которая называется ***монусом*** (monus) или ***усеченной разностью*** (proper subtraction) и определяется соотношением $\mathrm{m}\dot{-}n = max(m - n, 0)$**.**
- МТ, выполняющая эту операцию, определяется в виде $M = \left(\{q_0, q_1, \cdots, q_6\}, \{0,1\}, \{0,1, B\}, \delta, q_0, B\right)$**.** Отметим, что, поскольку МТ не используется для допускания входа, множество допускающих состояний не рассматривается. М начинает с ленты, состоящей из 0^m10^n и пробелов вокруг, и заканчивает лентой ***с*** $\mathrm{m}\dot{-}n$ символами 0, окруженными пробелами.

Машина Тьюринга для усечённой разности слайд 2/4

- МТ, выполняющая эту операцию, определяется в виде $M = \left(\{q_0, q_1, \cdots, q_6\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B\right)$. Отметим, что, поскольку МТ не используется для допускания входа, множество допускающих состояний не рассматривается. М начинает с ленты, состоящей из 0^m10^n и пробелов вокруг, и заканчивает лентой **с** $\mathrm{m}\dot{-}n$ символами 0, окруженными пробелами.
- М циклически находит крайний слева из оставшихся 0 и заменяет его пробелом. Затем движется вправо до 1. Найдя 1, М продолжает движение вправо до появления 0, который меняется на 1. Затем М возвращается влево в поисках крайнего слева 0, который идентифицируется после того, как М выходит на пробел и сдвигается вправо на одну клетку. Повторения заканчиваются в одной из следующих ситуаций.
- В поисках 0 справа М встречает пробел. Это значит, что все n нулей в 0^n изменены на 1, и n+1 нулей в 0^m заменены пробелами. Тогда М изменяет n+1 единицу на пробелы и добавляет 0, оставляя m-n нулей на ленте. Поскольку в этом случае ***m > n,*** то $\mathrm{m}\dot{-}n = m - n$**.**
- Начиная цикл, М не может найти 0, чтобы заменить его пробелом, поскольку первые *m* нулей уже изменены на *B*. Это значит, что *n>m,*$\mathrm{m}\dot{-}n = 0$. *M* заменяет все оставшиеся символы 1 и 0 пробелами и заканчивает работу с пустой лентой.

Машина Тьюринга для усечённой разности слайд 3/4

Состояние	Символ 0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	—
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	—
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	—	—	—



Машина Тьюринга для усечённой разности слайд 4/4

- q_0 — данное состояние начинает цикл и прерывает его, когда нужно. Если ***M*** обозревает О, цикл должен повториться. О меняется на ***B,*** головка сдвигается вправо, и ***M*** переходит в состояние q_1. Если же М обозревает 1, то все возможные соответствия между двумя группами нулей на ленте установлены, и ***M*** переходит в состояние q_j для опустошения ленты.
- q_1 — в этом состоянии ***M*** пропускает начальный блок из 0 в поисках 1. Найдя ее, ***M*** переходит в состояние q_2.
- q_2 — ***M*** движется вправо, пропуская группу из 1 до появления 0. 0 меняется на 1, головка сдвигается влево, и ***M*** переходит в состояние q_3. Однако возможно также, что после блока из единиц символов 0 уже не осталось. Тогда ***M*** в состоянии q_2 обнаруживает В. Возникает ситуация 1, описанная выше, где п нулей второго блока использованы для удаления п из т нулей первого блока, и вычитание почти закончено. М переходит в состояние q_4, предназначенное для преобразования всех 1 в пробелы, а одного пробела — в 0.
- q_3 — ***M*** движется влево, пропуская 0 и 1 до появления пробела. Тогда ***M*** сдвигается вправо и возвращается в состояние q_0, начиная новый цикл.
- q_4 — вычитание закончено, но один лишний 0 из первого блока был ошибочно заменен пробелом. М движется влево, заменяя все 1 пробелами, до появления В. Последний меняется на 0, и ***M*** переходит в состояние q_6, где и останавливается.
- q_5 — это состояние достигается из q_0, если обнаруживается, что все 0 из первого блока заменены пробелами. В случае, описанном выше в 2, усеченная разность равна 0. М заменяет все оставшиеся 0 и 1 пробелами и переходит в состояние q_6.
- q_6 — единственной целью этого состояния является разрешить ***M*** остановиться после выполнения работы. Если бы вычисление разности было подпрограммой другой, более сложной функции, то q_6 начинало бы следующий шаг этого более объемного вычисления.

Язык машины Тьюринга

- Способ допускания языка машиной Тьюринга уже описан интуитивно. Входная цепочка помещается на ленту, и головка машины начинает работу на крайнем слева символе. Если МТ в конце концов достигает допускающего состояния, то вход допускается, в противном случае — нет.
- Языки, допустимые с помощью машин Тьюринга, часто называются *рекурсивно пе речислимыми,* или РП-языками. Термин "рекурсивно перечислимые" происходит от вы числительных формализмов, предшествовавших машинам Тьюринга, но определявших тот же класс языков или арифметических функций.

Допустимость по останову

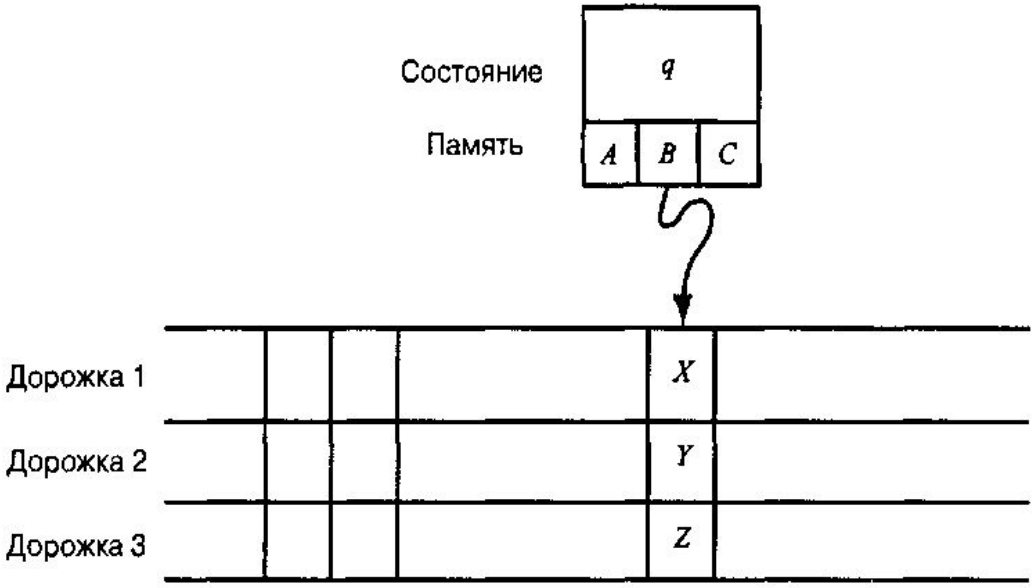
- Существует еще одно понятие "допустимости" для машин Тьюринга — допустимость по останову. Говорят, что машина Тьюринга *останавливается,* если попадает в состояние *q,* обозревая ленточный символ *X,* и в этом положении нет переходов, т.е. $\delta(q, X)$ не определено.
- Машина Тьюринга M из примера на слайде 234 была построена для того, чтобы допускать язык; она не рассматривалась с точки зрения вычисления функции. Заметим, однако, что M останавливается на всех цепочках из символов 0 и 1, поскольку независимо от входной цепочки машина M в конце концов удаляет вторую группу нулей, если может ее найти, достигает состояния q_6 и останавливается.
- Всегда можно предполагать, что МТ останавливается, если допускает. Таким образом, без изменения допускаемого языка можно сделать $\delta(q, X)$ неопределенной, если q — допускающее состояние. Итак, предполагается, что МТ всегда останавливается в допускающем состоянии.
- К сожалению, не всегда можно потребовать, чтобы МТ останавливалась, если не допускает. Язык машины Тьюринга, которая в конце концов останавливается независимо от того, допускает она или нет, называется *рекурсивным,* и его важные свойства будут рассматриваться ниже. Машина Тьюринга, которая всегда останавливается, представляет собой хорошую модель "алгоритма". Если алгоритм решения данной проблемы существует, то проблема называется "разрешимой", поэтому машины Тьюринга, которые всегда останавливаются, имеют большое значение во введении в теорию разрешимости

Программирование машины Тьюринга

- Цель данного раздела состоит в обосновании того, что машину Тьюринга можно использовать для вычислений так же, как и обычный компьютер. В конечном счете, можно показать, что МТ равна по своей мощи обычному компьютеру. В частности, она может выполнять некоторые вычисления, имея на входе другие машины Тьюринга, подобно тому, как программа проверяет другие программы. Именно это свойство "интроспективности" как машин Тьюринга, так и компьютерных программ позволяет доказывать неразрешимость проблем.
- Для иллюстрации возможностей МТ представим многочисленные приемы интерпретации ленты и конечного управления машины Тьюринга. Ни один из этих приемов не расширяет базовую модель МТ; они лишь делают запись более удобной. В дальнейшем они используются для имитации расширенных моделей машин Тьюринга с дополнительными свойствами, например, с несколькими лентами, на базовой модели МТ.

Память в состоянии слайд 1/3

- Конечное управление можно использовать не только для представления позиции в "программе" машины Тьюринга, но и для хранения конечного объема данных. рисунок иллюстрирует этот прием, а также идею многодорожечной ленты.
- При этом конечное управление содержит не только "управляющее" состояние q но и три элемента данных A, B и C. Данная техника не требует никакого расширения модели МТ; состояние просто рассматривается состояние в виде кортежа. На рисунке состояние имеет вид $[q, A, B, C]$. Такой подход позволяет описывать переходы более систематично, что зачастую проясняет программу МТ.



Память в состоянии слайд 2/3

- Построим МТ $M = (Q, \{0,1\}, \{0,1,B,\delta, [q_0, B]\}, \{[q_1, B]\})$, которая запоминает в своем конечном управлении первый увиденный символ (0 или 1) и проверяет, не встречается ли он еще где-нибудь во входной цепочке. Таким образом, М допускает язык $01^* + 10^*$. Допускание регулярных языков (вроде данного) не сужает возможностей машин Тьюринга, а служит лишь простым примером.
- Множество состояний **Q** есть $\{q_0, q_1\} \times \{0,1,B\}$, т.е. состояния рассматриваются как пары из двух следующих компонентов.
- Управляющая часть (q_0 или q_1) запоминает, что делает МТ. Управляющее состояние q_0 сигнализирует о том, что **M** еще не прочитала свой первый символ, а q_1 — что уже прочитала, и проверяет, не встречается ли он где-нибудь еще, продвигаясь вправо до достижения пустой клетки.
- В части данных хранится первый увиденный символ (0 или 1). Пробел B в этом компоненте означает, что никакой символ еще не прочитан.

Память в состоянии слайд 3/3

- Функция переходов δ определена следующим образом.
  - $\delta([q_0, B], a) = ([q_1, a], a, R)$ для ***a =*** 0 или ***a=*** 1. Вначале управляющим состоянием является q_0***,*** а частью данных — ***B.*** Обозреваемый символ копируется во второй компонент состояния, и М сдвигается вправо, переходя при этом в управляющее состояние q_1.
  - $\delta([q_1, a], \overline{a}) = ([q_1, a], \overline{a}, R)$***,*** где $\overline{a}$ — "дополнение" ***a,*** т.е. 0 при ***a*** = 1 и 1 при ***a*** = 0. В состоянии q_1 машина М пропускает каждый символ 0 или 1, который отличается от хранимого в состоянии, и продолжает движение вправо.
  - $\boldsymbol{\delta([q_1, a], B) = ([q_1, B], B, R)}$ для *a* = 0 или *a* = 1. Достигая первого пробела, *M* переходит в допускающее состояние $\boldsymbol{[q_1, B]}$.
- Заметим, что *M* не имеет определения переходов $\boldsymbol{\delta([q_1, a], B)}$ для *a* = 0 или *a* = 1. Таким образом, если М обнаруживает второе появление символа, который был записан в ее память конечного управления, она останавливается, не достигнув допускающего состояния.

Многодорожечные ленты

- Еще один полезный прием состоит в том, чтобы рассматривать ленту МТ как образованную несколькими дорожками. Каждая дорожка может хранить один символ (в одной клетке), и алфавит МТ состоит из кортежей, с одним компонентом для каждой "дорожки". Например, клетка, обозреваемая ленточной головкой на слайде 246, содержит символ $[X, Y, \mathrm{Z}]$. Как и память в конечном управлении, множественные дорожки не расширяют возможностей машин Тьюринга. Это просто описание полезной структуры ленточных символов.

Пример многодорожечной ленты слайд 1/4

- Типичное использование многодорожечных лент состоит в том, что одна дорожка хранит данные, а другая — отметку. Можно отмечать каждый символ, использованный определенным образом, или небольшое число позиций в данных. Данный прием фактически применялся в примерах выше, хотя лента там и не рассматривалась явно как многодорожечная. В данном примере вторая дорожка используется явно для распознавания следующего языка, который не является контекстно-свободным.
- $L_{wcw} = \{wcw | w \in (0+1)^*\}$
- Построим машину Тьюринга
- $M = (Q, \Sigma, \Gamma, \delta, [q_0, B], [B, B], \{[q_9, B]\})$, компоненты которой имеют следующий смысл.
- *Q*— множество состояний, которое представляет собой $\{q_1, q_2, \cdots, \boldsymbol{q_9}\} \times \{0,1, B\}$, т.е. множество пар, состоящих из управляющего состояния q_i и компонента данных 0, 1 или B. Вновь используется разрешенное выше запоминание символа в конечном управлении.
- Г— множество ленточных символов $\{B,*\} \times \{0,1,c,B\}$. Первый компонент, т.е. дорожка, может быть пустым или "отмеченным", что представлено, соответственно, пробелом или *. Символ * используется для отметки символов первой и второй групп из 0 и 1, в итоге подтверждающей, что цепочки слева и справа от центрального маркера *c* совпадают. Второй компонент ленточного символа представляет то, чем как бы является сам по себе ленточный символ, т.е. $[B, X]$ рассматривается как ленточный символ *X* для *X*= **0, 1, c,** *B.*

Пример многодорожечной ленты слайд 2/4

- Σ — входными символами являются $[B, 0]$ и $[B, 1]$, которые, как только что указано, обозначают, соответственно, 0 и 1.
- δ — функция переходов определена следующими правилами, в которых a и b могут обозначать как 0, так и 1.

1. $\delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R)$. В начальном состоянии *M* берет символ *a* (которым может быть 0 или 1), запоминает его в конечном управлении и переходит в управляющее состояние q_2. Затем "отмечает" обозреваемый символ, изменив его первый компонент с *B* на *, и сдвигается вправо.
2. $\delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R)$. *M* движется вправо в поисках символа *c*. Напомним, что *a* и *b* независимо друг от друга могут быть 0 или 1, но не могут быть c.
3. $\delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R)$. Обнаружив *c, M* продолжает двигаться вправо, но меняет управляющее состояние на q_3.
4. $\delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R)$. В состоянии q_3 продолжается пропуск всех отмеченных символов.
5. $\delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L)$. Если первый неотмеченный символ, найденный M, совпадает с символом в ее управлении, она отмечает его, поскольку он является парным к соответствующему символу из первого блока нулей и единиц. M переходит в управляющее состояние q_4, выбрасывая символ из управления, и начинает движение влево.

Пример многодорожечной ленты слайд 3/4

6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$. На отмеченных символах *M* движется влево.
7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$. Обнаружив символ с, *M* переходит в состояние q_5 и продолжает движение влево. В состоянии q_5 она должна принять решение, зависящее от того, отмечен или нет символ непосредственно слева от с. Если отмечен, то первый блок из нулей и единиц уже полностью рассмотрен. Если же символ слева от *c* не отмечен, то *M* ищет крайний слева неотмеченный символ, запоминает его, и после этого в состоянии q_1 начинается новый цикл.
8. $\delta([q_5, B], [B, a]) = ([q_6, B], [B, a], L)$. Если символ слева от *c* не отмечен, начинается соответствующая ветка вычислений. *M* переходит в состояние q_6 и продолжает движение влево в поисках отмеченного символа.
9. $\delta([q_6, B], [B, a]) = ([q_6, B], [B, a], L)$. Пока символы не отмечены, *M* остается в состоянии q_6 и движется влево.

Пример многодорожечной ленты слайд 4/4

10. $\delta([q_6, B], [*, a]) = ([q_1, B], [B, a], R)$. Обнаружив отмеченный символ, *M* переходит в состояние q_1 и движется вправо, чтобы взять первый неотмеченный символ.
11. $\delta([q_5, B], [*, a]) = ([q_7, B], [*, a], R)$. Теперь рассмотрим ветку вычислений, когда в состоянии q_5 непосредственно слева от *c* обнаружен отмеченный символ. Начинается движение вправо в состоянии q_7.
12. $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$. В состоянии q_7 обозревается с. Происходит переход в состояние q_8 и продолжается движение вправо.
13. $\delta([q_8, B], [*, a]) = ([q_8, B], [*, a], R)$. В состоянии q_8 машина движется вправо, пропуская все отмеченные символы.
14. $\delta([q_8, B], [B, B]) = ([q_9, B], [B, B], R)$**.** Если ***M*** достигает пробела в состоянии q_8**,** не обнаружив ни одного неотмеченного символа, то она допускает. Если же она сначала находит неотмеченный символ 0 или 1, то блоки слева и справа от ***c*** не совпадают, и M останавливается без допускания.

Подпрограммы

- Машины Тьюринга — это программы, и их полезно рассматривать как построенные из набора взаимодействующих компонентов, или "подпрограмм". Подпрограмма машины Тью ринга представляет собой множество состояний, выполняющее некоторый полезный процесс. Это множество включает в себя стартовое состояние и еще одно состояние, которое не имеет переходов и служит состоянием "возврата" для передачи управления какому-либо множеству состояний, вызвавшему данную подпрограмму. "Вызов" подпрограммы возникает везде, где есть переход в ее начальное состояние. Машины Тьюринга не имеют механизма для запоминания "адреса возврата", т.е. состояния, в которое нужно перейти после завершения подпро граммы. Поэтому для реализации вызовов одной и той же МТ из нескольких состояний можно создавать копии подпрограммы, используя новое множество состояний для каждой копии. "Вызовы" ведут в начальные состояния разных копий подпрограммы, и каждая копия "возвращает" в соответствующее ей состояние.

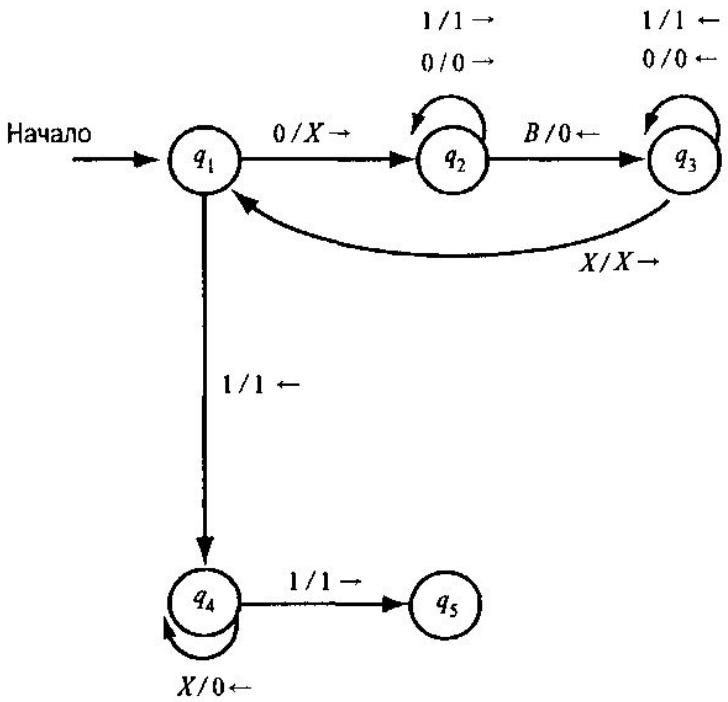
Подпрограмма Copy слайд 1/2

- Построим МТ для реализации функции "умножение". Наша МТ начинает с 0^m10^n1 на ленте и заканчивает с 0^{mn}. Опишем вкратце ее работу.

1. В начале каждого из m циклов работы лента содержит одну непустую цепочку вида $0^i10^n10^{kn}$ для некоторого к. Перед первым циклом i = m *и* к = 0.
2. За один цикл 0 из первой группы меняется на B, и n нулей добавляются к последней группе, приводя к цепочке вида $0^{i-1}10^n10^{(k+1)n}$.
3. В результате группа из n нулей копируется m раз с изменением каждый раз одного 0 в первой группе на B. Когда первая группа нулей целиком превратится в пробелы, в последней группе будет ***mn*** нулей.
4. Заключительный шаг — заменить пробелами 10^n1 в начале, и работа закончена. "Сердцем" этого алгоритма является подпрограмма, которая называется Copy. Она реализует шаг 2, копируя блок из n нулей в конец. Точнее, Copy преобразует МО вида $0^{m-k}1q_10^n10^{(k-1)n}$ в МО $0^{m-k}1q_50^n10^{kn}$.

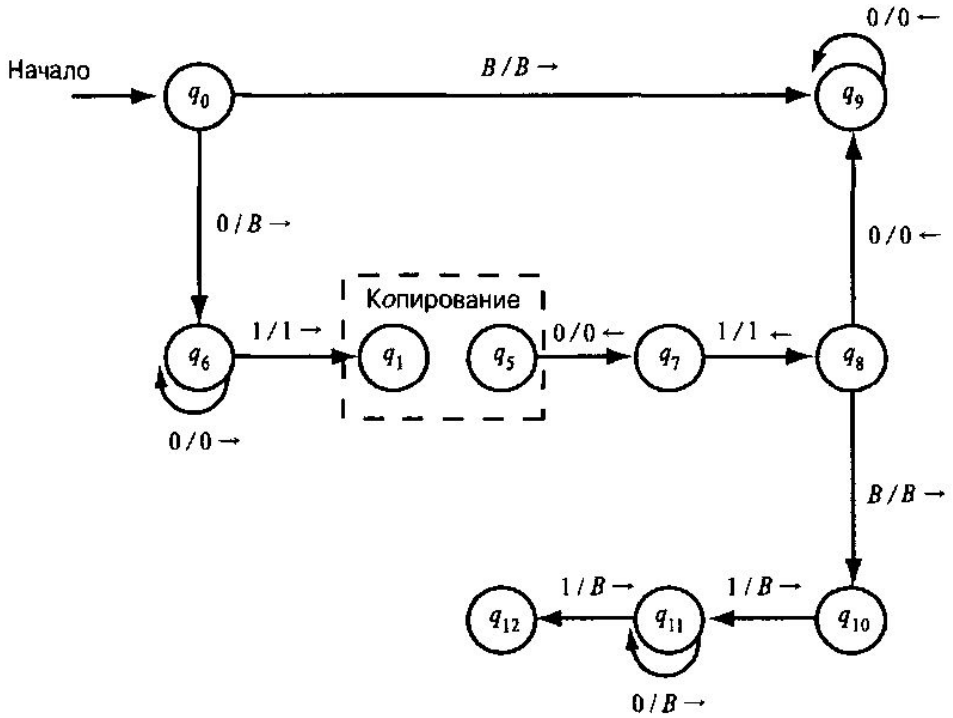
Подпрограмма Copy слайд 2/2

- Она заменяет первый 0 маркером X, в состоянии q_2 движется вправо до появления пробела, копирует в эту клетку 0, и в состоянии q_3 движется влево, пока не появится маркер ***X.*** На маркере она переходит вправо и возвращается в q_1. Она повторяет данный цикл до тех пор, пока в состоянии q_1 не встретит 1 вместо 0. Тогда она использует состояние q_4 для обратной замены маркеров ***X*** нулями и заканчивает в состоянии q_5.



Машина Тьюринга и Copy слайд 1/2

- Вся машина Тьюринга для умножения начинает в состоянии q_0. Вначале она за несколько шагов переходит от МО $q_0 0^m 1 0^n 1$ к МО $0^{m-1} 1 q_1 0^n 1$. Необходимые переходы показаны на рисунке слева от вызова подпрограммы; в них участвуют только состояния q_0 и q_6.



Машина Тьюринга и Copy слайд 2/2

- Справа от вызова подпрограммы (см. слайд 257) представлены состояния $q_7 - \ q_{12}$. Состояния q_7, q_8 и q_9 предназначены для получения управления после того, как Copy скопировала блок из n нулей и находится в МО $0^{m-k}1q_5 0^n 10^{kn}$. Эти состояния приводят к конфигурации $q_0 0^{m-k} 10^n 10^{kn}$. В этот момент опять начинается цикл, удаляется краиний слева 0 и вызывается Copy для нового копирования блока из n нулей.
- Как исключение, в состоянии q_8 МТ может обнаружить, что все n нулей заменены пробелами, т.е. ***k=m***. В данном случае производится переход в состояние q_{10}. Это состояние с помощью состояния q_{11} заменяет символы $10^n 1$ в начале ленты пробелами, после чего достигается состояние останова q_{12}. В этот момент МТ имеет МО $q_{12}0^{mn}$, и ее работа завершена.

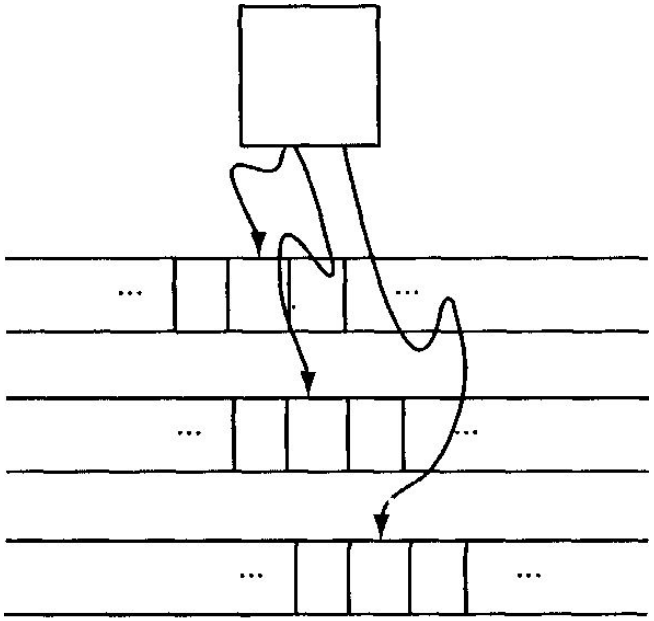
Расширения машины Тьюринга

- Ниже представлены некоторые вычислительные модели, связанные с машинами Тьюринга и имеющие такую же мощность (в смысле распознавания языков), что и базовая модель МТ, с которой мы работали до сих пор. Одна из них, многоленточная МТ, позволяет легко имитировать работу компьютера или других видов машин Тьюринга. И хотя многоленточные машины не мощнее базовой модели, речь также пойдет об их способности допускать языки.
- Затем рассматриваются недетерминированные машины Тьюринга — расширение основной модели, в котором разрешается совершать любой из конечного множества переходов в данной ситуации. Это расширение в некоторых случаях также облегчает "программирование" машин Тьюринга, не добавляя ничего к распознавательной мощности базовой модели.

Многоленточные машины Тьюринга

Многоленточная машина Тьюринга представлена на рисунке

Устройство имеет конечное управление (состояния) и некоторое конечное число лент. Каждая лента разделена на клетки, и каждая клетка может содержать любой символ из конечного ленточного алфавита. Как и у одноленточных МТ, множество ленточных символов включает пробел, а также имеет подмножество входных символов, к числу которых пробел не принадлежит. Среди состояний есть начальное и допускающие. Начальная конфигурация такова:



- Вход (конечная последовательность символов) размещен на первой ленте.
- Все остальные клетки всех лент содержат пробелы.
- Конечное управление находится в начальном состоянии.
- Головка первой ленты находится в левом конце входа.
- Головки всех других лент занимают произвольное положение. Поскольку все ленты, кроме первой, пусты, начальное положение головок на них не имеет значения (все клетки "выглядят" одинаково).

Переход многоленточной МТ

- Переход многоленточной МТ зависит от состояния и символа, обозреваемого каждой головкой. За один переход многоленточная МТ совершает следующие действия.

1. Управление переходит в новое состояние, которое может совпадать с предыдущим.
2. На каждой ленте в обозреваемую клетку записывается новый символ. Любой из них может совпадать с символом, бывшим там раньше.
3. Каждая из ленточных головок сдвигается влево или вправо или остается на месте. Головки сдвигаются независимо друг от друга, поэтому разные головки могут двигаться в разных направлениях, а некоторые вообще оставаться на месте.

- Формальная запись переходов не приводится — ее вид является непосредственным обобщением записи для одноленточной МТ, за исключением того, что сдвиги теперь обозначаются буквами *L, R* или *S.* Возможность оставлять головку на месте не была предусмотрена для одноленточных МТ, поэтому в их записи не было *S.* Многоленточные МТ, как и одноленточные, допускают, попадая в допускающее состояние.

Эквивалентность одноленточных и многоленточных МТ

- Напомним, что рекурсивно перечислимые языки определяются как языки, допускаемые одноленточными МТ. Очевидно, что многоленточные МТ допускают все рекурсивно перечислимые языки, поскольку одноленточная МТ является частным случаем многоленточной. Но существуют ли не рекурсивно перечислимые языки, допускаемые мно голенточными МТ? Ответом является "нет", и мы докажем это, показав, как многоленточная МТ имитируется с помощью одноленточной.
- Теорема. Каждый язык, допускаемый многоленточной МТ, рекурсивно перечислим.

Время работы и «много лент к одной»

- Здесь представлено понятие, которое в дальнейшем окажется чрезвычайно важным, а именно: "временная сложность", или "время работы" машины Тьюринга. *Временем работы* машины Тьюринга на входе *w* называется число шагов, которые *M* совершает до останова. Если *M* не останавливается на *w,* то время работы бесконечно. *Временная сложность* МТ *M* есть функция T(n), которая представляет собой максимум времени работы M на *w* по всем входам *w* длины *n.* Для МТ, не останавливающихся на всех своих входах, *T(n)* может быть бесконечным для некоторых или даже для всех *n.* Однако особое внимание будет уделено машинам Тьюринга, которые обязательно останавливаются на всех своих входах, и в частности, тем машинам, которые имеют полиномиальную временную сложность..
- Конструкция теоремы выше кажется неуклюжей. Действительно, построенной одноленточной МТ может понадобится гораздо больше времени для работы, чем многоленточной. Однако время работы этих двух машин соразмерно в том смысле, что время работы одноленточной МТ есть не более чем квадрат времени работы многоленточной. Хотя "возведение в квадрат" не является хорошей соразмерностью, оно сохраняет свойство времени работы быть полиномиальным. Ниже будут показаны следующие факты.
- Разница между полиномиальным временем и более высокими степенями его возрастания — это в действительности граница между тем, что можно решить с помощью компьютера, и тем, что практически нерешаемо.
- Несмотря на обширные исследования, время, необходимое для решения многих проблем, не удается определить точнее, чем "некоторое полиномиальное". Таким образом, когда изучается время, необходимое для решения некоторой проблемы, использование многоленточной или одноленточной МТ не является критичным.
- Докажем, что время работы многоленточной МТ является не более чем квадратом времени работы одноленточной МТ.
- Теорема. Время, необходимое одноленточной МТ *N* для имитации *n* переходов k-ленточной МТ M, есть $\boldsymbol{O}(\mathbf{n}^{\mathbf{2}})$.

Недетерминированные МТ

- *Недетерминированная* машина Тьюринга (*НМТ)* отличается от изученных нами детерминированных тем, что ее $\delta(\mathrm{q}, \mathrm{X})$ для каждого состояния *q* и ленточного символа *X* представляет собой множество троек $\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \cdots, (q_k, Y_k, D_k)\}$, где k — натуральное число. НМТ может выбирать на каждом шаге любую из троек для следующего перехода. Она не может, однако, выбрать состояние из одной тройки, ленточный символ из другой, а направление из какой-нибудь еще.
- Язык, допускаемый НМТ M, определяется по аналогии с другими недетерминированными устройствами, вроде НКА или МП-автоматов. Таким образом, M допускает вход w, если существует некоторая последовательность выборов переходов, ведущая из начального МО с w на входе в МО с допускающим состоянием. Существование других выборов, которые *не ведут* в допускающее состояние, не имеет значения, как и для НКА или МП-автоматов.

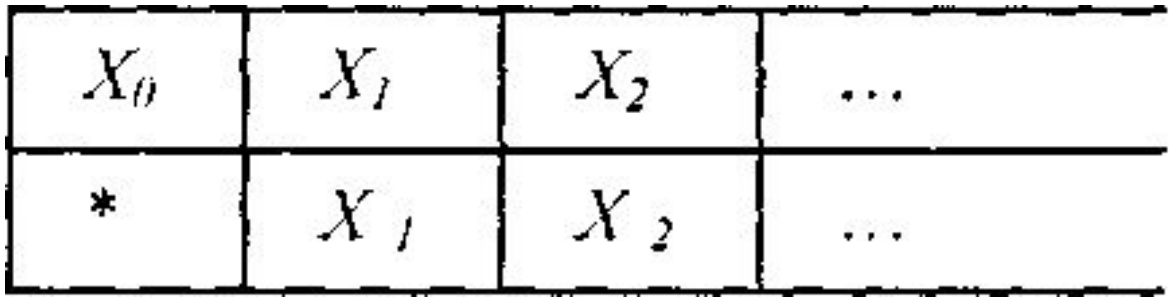
Языки НМТ

- Недетерминированные МТ допускают те же языки, что и детерминированные (или *ДМТ,* если нам нужно подчеркнуть их детерминированность). Доказательство основано на том, что для любой НМТ M_N можно построить ДМТ M_D, которая исследует конфигурации, достигаемые M_N с помощью любой последовательности переходов. Если M_D находит хотя бы одно МО с допускающим состоянием, то сама переходит в допускающее состояние. M_D должна помещать новые МО в очередь, а не в магазин, чтобы после некоторого конечного времени были проимитированы все последовательности длиной до *к (к = 1,2, ...).*
- Теорема. Если M_N — недетерминированная машина Тьюринга, то существует детерминированная машина Тьюринга M_D, у которой $\mathrm{L(M_D)} = \mathrm{L(M_N)}$.
- Отметим, что построенная детерминированная МТ может потребовать времени, экспоненциально большего, чем время работы недетерминированной МТ. Неизвестно, является ли это экспоненциальное соотношение необходимым.

Ограниченные МТ

- Выше были показаны обобщения машин Тьюринга, которые в действительности не добавляют никакой мощности в смысле распознаваемых языков. Теперь рассмотрим несколько примеров ограничений МТ, которые также не изменяют их распознавательной мощности. Первое ограничение невелико, но полезно во многих конструкциях, которые мы увидим позже: бесконечная в обе стороны лента заменяется бесконечной только вправо. Ограниченным МТ запрещается также записывать пробел вместо других ленточных символов. Это ограничение позволяет считать, что МО состоят только из значащих символов и всегда начинаются левым концом ленты.
- Затем исследуются определенные виды многоленточных МТ, которые обобщают МП-автоматы. Во-первых, ленты МТ ограничиваются и ведут себя, как магазины. Во- вторых, ленты ограничиваются еще больше, становясь "счетчиками", т.е. они могут представлять лишь одно целое число, а МТ имеет возможность только отличать 0 от любого ненулевого числа. Значение этих ограничений в том, что существует несколько очень простых разновидностей автоматов, обладающих всеми возможностями компьютеров. Более того, неразрешимые проблемы, связанные с машинами Тьюринга и описанные в главе 9, в равной мере относятся и к этим простым машинам.

Односторонняя лента



- Ранее было предположено, что ленточная головка машины Тьюринга может находиться как слева, так и справа от начальной позиции, однако достаточно того, что головка может находиться на ней или только справа от нее. В действительности, можно считать, что лента является *бесконечной в одну сторону,* или *односторонней* (semi-infinite), т.е. слева от начальной позиции головки вообще нет клеток. В следующей теореме приводится конструкция, показывающая, что МТ с односторонней лентой может имитировать обычную МТ с бесконечной в обе стороны лентой.
- В основе этой конструкции лежит использование двух дорожек на односторонней ленте. Верхняя дорожка представляет клетки исходной МТ, находящиеся справа от начальной позиции, и ее саму. Нижняя дорожка представляет позиции слева от начальной, но в обратном порядке. Точное упорядочение показано на рис. 67. Верхняя дорожка представляет клетки $X_0, X_1, \cdots$, где X_0— начальная позиция головки, а $X_1, \cdots$ — клетки справа от нее. Клетки $X_{-1}, X_{-2}, \cdots$ и последующие представляют клетки слева от начальной позиции. Обратите внимание на * в левой клетке на нижней дорожке. Этот символ служит концевым маркером и предохраняет головку односторонней ленты от случайного выхода за левый конец ленты.

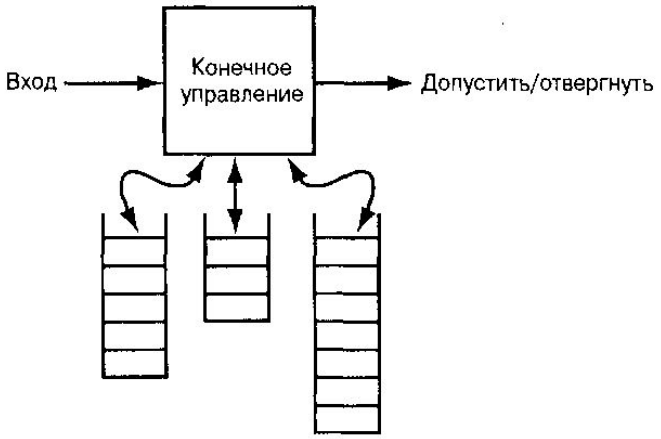
Усиленное ограничение

- Можно усилить ограничение нашей машины Тьюринга, чтобы она никогда не записывала пробелов. Это простое ограничение, вместе с лентой, бесконечной только в одну сторону, означает, что лента всегда представляет собой префикс из непустых символов, за которым следует бесконечная последовательность пробелов. Кроме того, крайний слева непустой символ всегда находится в начальной позиции ленты. В теоремах ниже будет видно, насколько полезно предполагать, что МО имеют именно такой вид.
- Теорема. Каждый язык, допускаемый МТ M_2, допускается также МТ M_1 со следующими ограничениями:

1. Головка M_1 никогда не смещается влево от своей начальной позиции.
2. M_1 никогда не записывает пробелы.

Мультистековые МТ слайд 1/2

- Теперь рассмотрим несколько вычислительных моделей, основанных на обобщениях магазинных автоматов. Вначале рассмотрим, что происходит, если МП-автомат имеет несколько магазинов. Из примера выше уже известно, что МТ может допускать языки, не допускаемые никаким МП-автоматом с одним магазином. Однако оказывается, что если снабдить МП-автомат двумя магазинами, то он может допустить любой язык, допускаемый МТ.
- Рассмотрим также класс машин, называемых "счетчиковыми машинами". Они обладают возможностью запоминать конечное число целых чисел ("счетчиков") и совершать различные переходы в зависимости от того, какие из счетчиков равны 0 (если таковые вообще есть). Счетчиковая машина может только прибавить 1 к счетчику или вычесть 1 из него, но отличить значения двух различных ненулевых счетчиков она не способна. Следовательно, счетчик похож на магазин с двухсимвольным алфавитом, состоящим из маркера дна (он появляется только на дне) и еще одного символа, который можно заносить в магазин и выталкивать из него.
- Формальное описание работы *мультистековой,* или *многомагазинной* машины здесь не приводится, но идея иллюстрируется рисунке; *k*-магазинная машина представляет собой детерминированный МП-автомат с *k* магазинами.



Мультистековые МТ слайд 2/2

- Он получает свои входные данные, как и МП-автомат, из некоторого их источника, а не с ленты или из магазина, как МТ. Мультистековая машина имеет конечное управление, т.е. конечное множество состояний, и конечный магазинный алфавит, используемый для всех магазинов. Переход мультистековой машины основывается на состоянии, входном символе и верхних символах всех магазинов.
- Мультистековая машина может совершить ε-переход, не используя входной символ, но для того, чтобы машина была детерминированной, в любой ситуации не должно быть возможности выбора ε -перехода или перехода по символу.
- За один переход мультистековая машина может изменить состояние и заменить верхний символ в каждом из магазинов цепочкой из нуля или нескольких магазинных символов. Обычно для каждого магазина бывает своя замещающая цепочка.
- Итак, типичное правило перехода для k-магазинной машины имеет вид $\delta(q, a, X_1, X_2, \cdots, X_k) = (p, \gamma_1, \gamma_2, \cdots, \gamma_k)$. Его смысл состоит в том, что, находясь в состоянии *q,* с X_i на вершине i-го магазина, $i = 1,2, \cdots, k$, машина может прочитать на входе *a* (либо символ алфавита, либо ε), перейти в состояние *p* и заменить X_i на вершине i-го магазина цепочкой $\gamma_i, i = 1,2, \cdots, k$. Мультистековая машина допускает, попав в заключительное состояние.
- Добавим одно свойство, которое упрощает обработку входа описанной детерминированной машиной. Будем предполагать, что есть специальный символ $, называемый *концевым маркером (маркером конца входа),* который встречается только в конце входа и не является его частью. Присутствие маркера позволяет узнать, когда прочитан весь доступный вход. Заметим, что обычная МТ не нуждается в специальном маркере конца, поскольку в этой роли выступает первый пробел.
- Теорема. Если язык *L* допускается машиной Тьюринга, то *L* допускается двухмагазинной машиной.

Счётчиковые машины

Счетчиковую машину можно представить одним из двух способов.

1. Счетчиковая машина имеет такую же структуру, как и мультистековая (см. слайд 179), но вместо магазинов у нее счетчики. Счетчики содержат произвольные неотрицательные целые числа, но отличить можно только ненулевое от нулевого. Таким образом, переход счетчиковой машины зависит от ее состояния, входного символа и того, какие из счетчиков являются нулевыми. За один переход машина может изменить состояние и добавить или отнять 1 от любого из счетчиков. Однако счетчик не может быть отрицательным, поэтому отнимать 1 от счетчика со значением 0 нельзя.

2. Счетчиковая машина также может рассматриваться, как мультистековая машина со следующими ограничениями:

а) есть только два магазинных символа, Z_0 *(маркер дна)* и *X;*

б) вначале Z_0 находится в каждом магазине;

в) Z_0 можно заменить только цепочкой вида XZ_0 для некоторого $i \geq 0$;

г) ***X*** можно заменить только цепочкой вида X^i для некоторого $i \geq 0$. Таким образом, Z_0 встречается только на дне каждого магазина, а все остальные символы (если есть) — это символы X.

Для счетчиковых машин будем использовать определение 1, хотя оба они, очевидно, задают машины одинаковой мощности. Причина в том, что магазин X^iZ_0 может быть идентифицирован значением i. В определении 2 значение 0 можно отличить от остальных, поскольку значению 0 соответствует Z_0 на вершине магазина, в противном случае там помещается X. Однако отличить два положительных числа невозможно, поскольку обоим соответствует X на вершине магазина.

Мощность счётчиковых машин

О языках счетчиковых машин стоит сделать несколько очевидных замечаний.

Каждый язык, допускаемый счетчиковой машиной, рекурсивно перечислим. Причина в том, что счетчиковые машины являются частным случаем магазинных, а магазинные — частным случаем многоленточных машин Тьюринга, которые по теореме выше допускают только рекурсивно перечислимые языки.

Каждый язык, допускаемый односчетчиковой машиной, является КС-языком. Заметим, что счетчик, с точки зрения определения 2, является магазином, поэтому односчетчиковая машина представляет собой частный случай одномагазинной, т.е. МП-автомата. Языки односчетчиковых машин допускаются детерминированными МП-автоматами, хотя доказать это на удивление сложно. Трудность вызывает тот факт, что мультистековые и счетчиковые машины имеют маркер $ в конце входа. Недетерминированный МП-автомат может "догадаться", что он видит последний входной символ, и следующим будет маркер. Таким образом, ясно, что недетерминированный МП-автомат без концевого маркера может имитировать ДМП-автомат с маркером. Однако доказать, что ДМП-автомат без концевого маркера может имитировать ДМП-автомат *с маркером,* весьма трудно.

Удивительно, но для имитации машины Тьюринга и, следовательно, для допускания любого рекурсивно перечислимого языка, достаточно двух счетчиков. Для обоснования этого утверждения вначале доказывается, что достаточно трех счетчиков, а затем три счетчика имитируются с помощью двух.

Теорема. Каждый рекурсивно перечислимый язык допускается трехсчетчиковой машиной.

Теорема. Каждый рекурсивно перечислимый язык допускается двухсчетчиковой машиной.

Машина Тьюринга и компьютеры

- Связь между компьютерами и машинами Тьюринга, а также структура универсальной машины Тьюринга показана в соответствующей презентации

Неразрешимость и МТ

В разделе выше были приведены рассуждения, которые неформально обосновывали существование проблем, не разрешимых с помощью компьютера. Недостатком тех "правдоподобных рассуждений" было вынужденное пренебрежение реальными ограничениями, возникающими при реализации любого языка программирования на любом реальном компьютере. Однако эти ограничения, вроде конечности адресного пространства, не являются фундаментальными. Наоборот, с течением времени мы вправе ожидать от компьютеров неограниченного роста таких количественных характеристик, как размеры адресного пространства, оперативной памяти и т.п.

Сосредоточившись на машинах Тьюринга, для которых ограничения такого рода от сутствуют, можно лучше понять главное — принципиальные возможности вычислитель ных устройств, если не сегодня, то в перспективе. Текущем разделе дается формальное доказательство того, что никакая машина Тьюринга не может решить следующую задачу:

- Допускает ли данная машина Тьюринга сама себя (свой код) в качестве входа?

Выше было показано, что на машинах Тьюринга можно имитировать работу реальных компьютеров, даже не имеющих сегодняшних ограничений. Поэтому независимо от того, насколько ослаблены эти практические ограничения, у нас будет строгое доказательство, что указанную задачу нельзя решить с помощью компьютера.

Далее разделим проблемы, разрешимые с помощью машин Тьюринга, на два класса. В первый класс войдут те из них, которые имеют *алгоритм* решения (т.е. машину Тьюринга, которая останавливается независимо от того, допускает она свой вход или нет). Во второй класс войдут проблемы, разрешимые лишь с помощью таких машин Тьюринга, которые с недопустимыми входами могут работать бесконечно. В последнем случае проверить допустимость входа весьма проблематично, поскольку независимо от того, сколь долго работает МТ, неизвестно, допускается вход или нет. Поэтому мы сосредоточимся на методах доказательства "неразрешимости" проблем, т.е. что для них не существует алгоритма решения независимо от того, допускаются ли они машиной Тьюринга, которая не останавливается на некоторых входах, или нет.

Будет доказано, что неразрешима следующая проблема.

- Допускает ли данная машина Тьюринга данный вход?

Неперечислимый язык

Напомним, что язык *L* является *рекурсивно-перечислимым* (РП-языком), если *L = L(M)* для некоторой МТ M. Ниже будут введены так называемые "рекурсивные", или "разрешимые", языки, которые не только рекурсивно перечислимы, но и допускаются некоторой МТ, останавливающейся на всех своих входах независимо от их допустимости.

Основная цель цель — доказать неразрешимость языка, состоящего из пар (M, w), которые удовлетворяют следующим условиям.

- M — машина Тьюринга (в виде соответствующего двоичного кода) с входным алфавитом {0, 1}.
- *w* — цепочка из символов 0 и 1.
- M допускает вход w.

Если эта проблема неразрешима при условии, что алфавит — двоичный, то она, безусловно, будет неразрешимой и в более общем виде, при произвольном входном алфавите.

Прежде всего, необходимо сформулировать эту проблему как вопрос о принадлежности некоторому конкретному языку. Поэтому нужно определить такое кодирование машин Тьюринга, в котором использовались бы только символы 0 и 1 независимо от числа состояний МТ. Имея такие коды, можно рассматривать любую двоичную цепочку как машину Тьюринга. Если цепочка не является правильным представлением какой-либо МТ, то ее можно считать представлением МТ без переходов.

Для достижения промежуточной цели, представленной в данном разделе, используется "язык диагонализации" L_d. Он состоит из всех цепочек *w,* каждая из которых не допускается машиной Тьюринга, представленной этой цепочкой. Мы покажем, что такой машины Тьюринга, которая бы допускала L_d, вообще не существует. Напомним: доказать, что язык не допускается никакой машиной Тьюринга, значит доказать нечто большее, чем то, что данный язык неразрешим (т.е. что для него не существует алгоритма, или МТ, которая останавливается на всех своих входах).

Двоичное представление МТ слайд 1/3

В дальнейшем нам понадобится приписать всем двоичным цепочкам целые числа так, чтобы каждой цепочке соответствовало одно целое число и каждому числу — одна цепочка. Если w — двоичная цепочка, то iw рассматривается как двоичное представление целого числа i. Тогда **w** называется i-й цепочкой. Таким образом, $\boldsymbol{\varepsilon}$ есть первая цепочка, **0** — вторая, 1 — третья, 00 — четвертая, 01 — пятая, и так далее. Это равносильно тому, что цепочки упорядочены по длине, а цепочки равной длины упорядочены лексикографически. В дальнейшем **i**-я цепочка обозначается через $\boldsymbol{w_i}$.

Наша следующая цель — разработать для машин Тьюринга такой код, чтобы всякую МТ с входным алфавитом {0, 1} можно было рассматривать как двоичную цепочку. Мы только что показали, как перечислить двоичные цепочки. Поэтому у нас есть идентификация машин Тьюринга целыми числами, и можно говорить об "i-ой машине Тьюринга M_i". Для того чтобы представить МТ $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$ как двоичную цепочку, нужно приписать целые числа состояниям, ленточным символам и направлениям *L* и *R*.

- Будем предполагать, что состояниями являются $q_1 q_2 \cdots q_r$ при некотором *r*. Состояние q_1 всегда будет начальным, а q_r — единственным допускающим. Заметим, что поскольку можно считать, что МТ останавливается, попадая в допускающее состояние, то одного такого состояния всегда достаточно.
- Предположим, что ленточными символами являются $X_1 X_2 \cdots X_s$ при некотором *s*. X_1 всегда будет символом 0, X_2 — 1, *а* X_3 — *B,* пробелом (пустым символом). Остальным же ленточным символам целые числа могут быть приписаны произвольным образом.
- Направление *L* обозначается как D_1, а направление *R* — как D_2.

Двоичное представление МТ слайд 2/3

Поскольку состояниям и ленточным символам любой МТ М можно приписать целые числа не единственным образом, то и кодировок МТ будет, как правило, более одной. Но в дальнейшем это не будет играть роли, так как мы покажем, что МТ *M* с $L(M) = L_d$ вообще непредставима никаким кодом.

Установив, какие целые числа соответствуют каждому из состояний, символов и направлений, можно записать код функции переходов δ. Пусть $\delta(q_i, X_j) = (q_k, X_l, D_m)$ есть одно из правил перехода, где *i, j, к, l, m* — некоторые целые числа. Это правило кодируется цепочкой $0^i 10^j 10^k 10^l 10^m$. Заметим, что каждое из чисел *i, j, к, l, m* не меньше единицы, так что в коде каждого отдельного перехода нет двух 1 подряд.

Код МТ М состоит из кодов всех ее переходов, расположенных в некотором порядке и разделенных парами единиц: $C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$, где каждое C_i означает код одного перехода М.

Двоичное представление МТ слайд 3/3

Рассмотрим МТ $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_2\})$, *где функция* δ *состоит из следующих правил.*

- $\delta(q_1, 1) = (q_3, 0, R)$
- $\delta(q_3, 0) = (q_1, 1, R)$
- $\delta(q_3, 1) = (q_2, 0, R)$
- $\delta(q_3, B) = (q_3, 1, L)$

Переходы кодируются соответственно следующим образом.

0100100010100

0001010100100

00010010010100

0001000100010010

Первое правило, например, можно записать как $\delta(q_1, X_2) = (q_3, X_1, D_2)$, поскольку 1 = X_2, 0 — X_1, *R* = D_2.

- Поэтому его кодом будет цепочка $0^1 10^2 10^3 10^1 10^2$, как и записано выше. Кодом всей М является следующая цепочка:
- 0100100010100110001010100100110001001001010011000100010001001 0 Заметим, что существуют другие возможные коды машины М. В частности, коды четырех ее переходов можно переставить 4! способами, что дает 24 кода для М. □
- Ниже возникнет необходимость кодировать пары вида (M, w), состоящие из МТ и цепочки. В качестве такого кода последовательно записываются код М, 111 и цепочка w. Поскольку правильный код МТ не может содержать три единицы подряд, первое вхождение 111 гарантированно отделяет код М от w. Например, если М — это МТ из примера выше, а w— цепочка 1011, то кодом (M, w) будет цепочка, представленная в конце примера выше, с дописанной к ней последовательностью 1111011.

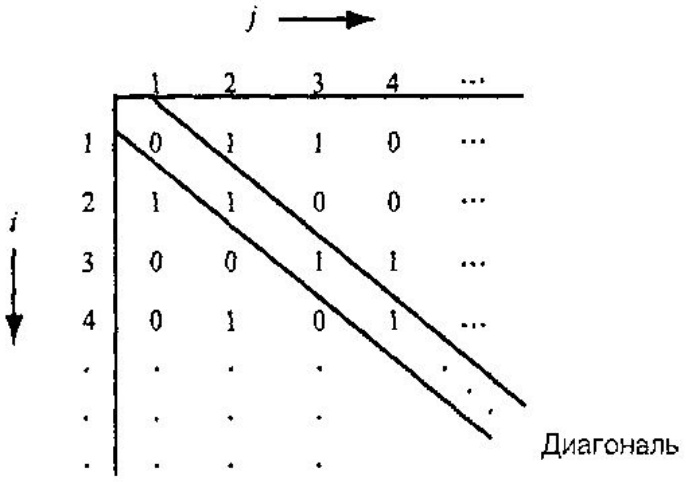
Язык диагонализации слайд 1/2

Выше были определены коды машин Тьюринга, и теперь есть вполне конкретное понятие M_i, 'i-й машины Тьюринга". Это МТ М, кодом которой является i-я двоичная цепочка w_i Многие целые числа вообще не соответствуют машинам Тьюринга. Например, 11001 не начинается с 0, а цепочка 0010111010010100 содержит три 1 подряд. Если w_i не является правильным кодом МТ, то будем считать, что M_i — МТ, у которой одно состояние и нет переходов, т.е. М, для этих значений i есть МТ, которая сразу останавливается на любом входе. Итак, если w_i не является правильным кодом МТ, то $L(M_i) = \emptyset$.

Теперь можно дать весьма существенное определение.

- ***Язык диагонализации*** $\boldsymbol{L_d}$ — это множество всех цепочек w_i, не принадлежащих $L(M_i)$**.**

Таким образом, $\boldsymbol{L_d}$ состоит из всех цепочек w, каждая из которых не допускается соответствующей машиной Тьюринга с кодом w.



Язык диагонализации слайд 2/2

- Из рис. на слайде 279 видно, почему L_d называется языком "диагонализации". Для всех *i* и *j* эта таблица показывает, допускает ли МТ M_i входную цепочку w_j; 1 означает "да, допускает", а 0 — "нет, не допускает". Можно считать i-ую строку таблицы *характеристическим вектором* языка $\mathrm{L(M_i)}$; единицы в этой строке указывают на цепочки, которые принадлежат данному языку.

Числа на диагонали показывают, допускает ли $\mathrm{M_i}$ цепочку $\mathrm{w_i}$. Чтобы построить язык нужно взять дополнение диагонали. Например, если бы таблица на слайду 279 была корректной, то дополнение диагонали имело бы начало 1, 0, 0, 0, ... Таким образом, L_d должен был бы содержать $\mathrm{w_1} = \varepsilon$ и не содержать цепочки с $\mathrm{w_2}$ по $\mathrm{w_4}$, т.е. 0, 1 и 00 и т.д.

Операция с дополнением диагонали для построения характеристического вектора языка, которому не может соответствовать никакая строка, называется *диагонализацией.* Она приводит к желаемому результату, поскольку дополнение диагонали само по себе является характеристическим вектором, описывающим принадлежность некоторому языку, а именно— L_d. Действительно, этот характеристический вектор отличается от каждой из строк таблицы на слайде 279 хотя бы в одном столбце. Поэтому дополнение диагонали не может быть характеристическим вектором никакой машины Тьюринга.

Развивая рассуждения о характеристических векторах и диагонали, можно доказать основной результат, касающийся машин Тьюринга, — МТ, допускающей язык L_d, не существует.

Теорема. Язык L_d не является рекурсивно-перечислимым, т.е. не существует машины Тьюринга, которая допускала бы L_d.

Неразрешимая РП-проблема

- Итак, было показано, что существует проблема (язык диагонализации L_d), которая не допускается никакой машиной Тьюринга. Следующая цель — уточнить структуру рекурсивно-перечислимых языков (РП-языков), т.е. допустимых машинами Тьюринга, разбив их на два класса. В первый класс войдут языки, которым соответствует то, что обычно понимается как алгоритм, т.е. МТ, которая не только распознает язык, но и сообщает, что входная цепочка не принадлежит этому языку. Такая машина Тьюринга в конце концов всегда останавливается независимо от того, достигнуто ли допускающее состояние.
- Второй класс языков состоит из тех РП-языков, которые не допускаются никакой машиной Тьюринга, останавливающейся на всех входах. Эти языки допускаются несколько неудобным образом: если входная цепочка принадлежит языку, то мы в конце концов об этом узнаем, но если нет, то машина Тьюринга будет работать вечно. Поэтому никогда нельзя быть уверенным, что данная входная цепочка не будет когда-нибудь допущена. Примером языка данного типа, как будет показано ниже, является множество таких закодированных пар (M, w), в которых МТ M допускает вход w.

Рекурсивные языки слайд 1/2

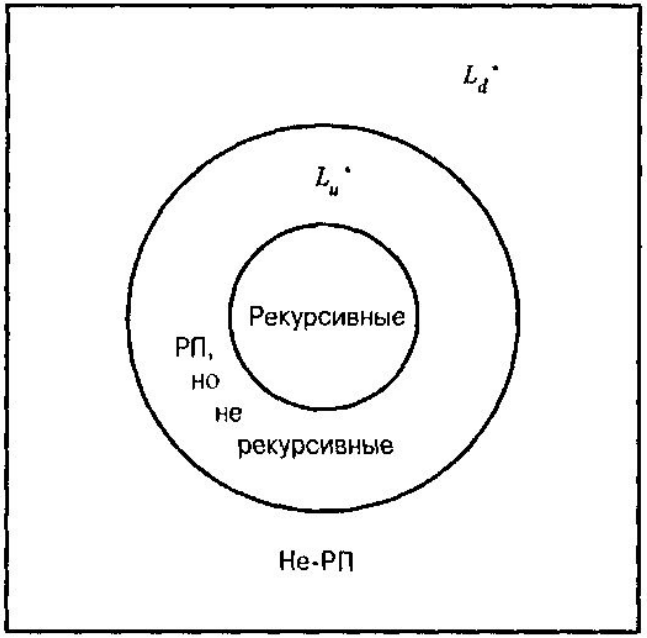
- Язык *L* называется *рекурсивным,* если *L = L(M)* для некоторой машины Тьюринга *M,* удовлетворяющей следующим условиям.
  - Если w принадлежит *L,* то *M* попадает в допускающее состояние (и, следовательно, останавливается).
  - Если w не принадлежит *L,* то *M* в конце концов останавливается, хотя и не попадает в допускающее состояние.
- МТ этого типа соответствует интуитивному понятию "алгоритма" — правильно определенной последовательности шагов, которая всегда заканчивается и приводит к некоторому ответу. Если мы рассматриваем язык *L* как "проблему", то проблема *L* называется *разрешимой,* если она является рекурсивным языком. В противном случае проблема называется *неразрешимой.*

Рекурсивные языки слайд 2/2

Существование алгоритма зачастую важнее, чем наличие некоторой МТ, решающей данную проблему. Как упоминалось ранее, машины Тьюринга, которые могут не останавливаться, не дают информации, необходимой для утверждения, что цепочка не принадлежит языку, т.е. они "не решают проблему". Таким образом, разделение проблем и языков на разрешимые (для них существует алгоритм решения) и неразрешимые часто важнее, чем разделение на рекурсивно перечислимые (для них существует МТ какого- либо типа) и неперечислимые (для них вообще не существует МТ). На рисунке представлено соотношение трех классов языков.

- Рекурсивные языки.
- Языки, которые рекурсивно перечислимы (РП), но не рекурсивны.
- Неперечислимые (*не-РП)* языки.

На рисунке указаны правильные положения не РП-языка L_d и "универсального языка" L_u, который, будет показано, является РП, но не рекурсивным.



Дополнение рекурсивных и РП-языков

Для доказательства того, что некоторый язык принадлежит второму кольцу на слайде 283 (т.е. является РП, но не рекурсивным), часто используется дополнение этого языка. Покажем, что рекурсивные языки замкнуты относительно дополнения. Поэтому, если язык L является РП, а его дополнение $\overline{L}$ — нет, то L не может быть рекурсивным. Если бы *L* был рекурсивным, то $\overline{L}$ также был бы рекурсивным, а следовательно, и РП.

Теорема. Если *L* — рекурсивный язык, то язык $\overline{L}$ также рекурсивен.

С дополнениями языков связан еще один важный факт. Он еще больше ограничивает область на диаграмме (см. слайд 283), в которую могут попасть язык и его дополнение. Это ограничение утверждается следующей теоремой.

Теорема. Если язык *L* и его дополнение являются РП, то *L* рекурсивен. Отметим, что тогда по теореме выше язык $\overline{L}$ также рекурсивен.

Эти две теоремы можно объединить следующим образом. Из девяти способов расположения языка *L* и его дополнения *L* возможны лишь следующие четыре.

- Оба языка $\overline{L}$ и L рекурсивны, т.е. оба они находятся во внутреннем кольце.
- Ни *L,* ни $\overline{L}$ не являются РП, т.е. оба они находятся во внешнем кольце.
- L является РП, но не рекурсивным, и $\overline{L}$ не является РП, т.е. один находится в среднем кольце, а другой — во внешнем.
- $\overline{L}$ является РП, но не рекурсивным, и *L* не является РП, т.е. *L* и $\overline{L}$ меняются местами по отношению к случаю 3.

В качестве доказательства отметим, что вышеприведённые теоремы исключают возможность того, что один из языков ($\overline{L}$ или L) является рекурсивным, а второй принадлежит какому-либо из оставшихся классов. При этом исключается возможность того, что оба языка являются РП, но не рекурсивными.

Пример дополнения

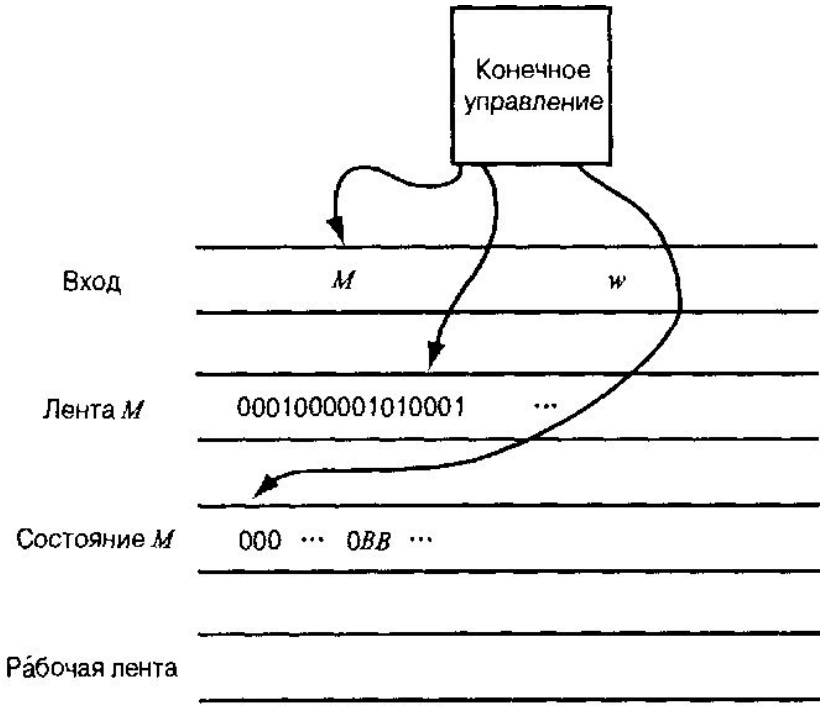
- **Пример**. Рассмотрим язык L_d, который, как было показано, не является РП. Поэтому язык $\overline{L_d}$ не может быть рекурсивным. Однако $\overline{L_d}$ может быть либо не-РП, либо РП, но не рекурсивным. На самом деле верно последнее.
- $\overline{L_d}$ есть множество цепочек w_i допускаемых соответствующими M_i. Этот язык похож на универсальный язык L_w, состоящий из всех пар (M, w), где M допускает w, который, как будет показано ниже, является РП. Точно так же можно показать, что $\overline{L_d}$ является РП.

Универсальный язык слайд 1/3

- Выше уже неформально обсуждалось, как можно использовать машину Тьюринга в качестве модели компьютера с загруженной в него произвольной программой. Иными словами, отдельно взятая МТ может использоваться как "компьютер с записанной программой", который считывает свою программу и данные с одной или нескольких лент, содержащих входную информацию. В данном разделе будет формализована идея того, что машина Тьюринга является представлением программы, записанной в память компьютера.
- *Универсальный язык* L_u определяется как множество двоичных цепочек, которые являются кодами пар (M, w), где M— МТ с двоичным входным алфавитом, а w — цепочка из $(0 + 1)^*$, принадлежащая L(M). Таким образом, L_u — это множество цепочек, представляющих некоторую МТ и допускаемый ею вход. Покажем, что существует МТ *U,* которую часто называют *универсальной машиной Тьюринга,* для которой $L_u = L(U)$. Поскольку входом *U* является двоичная цепочка, то в действительности U — это некоторая M_j в списке машин Тьюринга с двоичным входом.

Универсальный язык слайд 2/3

- Проще всего *U* описывается как многоленточная машина Тьюринга. В машине *U* переходы *M* вначале хранятся на первой ленте вместе с цепочкой w. Вторая лента используется для моделирования ленты машины M, в том же формате, что и у кода M. Таким образом, ленточный символ X, машины M кодируется как 0^i, а коды разделяются одиночными 1. На третью ленту *U* записывается состояние *M,* причем состояние *q,* представляется в виде *i* нулей. Схема *U* представлена на рисунке.



Универсальный язык слайд 3/3

Машина U производит следующие операции.

1. Исследует вход для того, чтобы убедиться, что код M является правильным кодом МТ. Если это не так, *U* останавливается, не допуская. Это действие корректно, поскольку неправильные коды по договоренности представляют МТ без переходов, а такие МТ не допускают никакого входа.
2. Записывает на вторую ленту код входной цепочки w. Таким образом, каждому символу 0 из w на второй ленте ставится в соответствие цепочка 10, а каждой 1 — цепочка 100. Отметим, что пустые символы, которые находятся на ленте самой M и представляются цепочками вида 1 OOO, на этой ленте появляться не будут. Все клетки, кроме занятых символами цепочки w, будут заполняться пустыми символами машины *U.* Но при этом *U* знает, что если она ищет имитируемый символ *M* и находит свой собственный пустой символ, то она должна заменить его последовательностью 1000, имитирующей пустой символ M.
3. На третьей ленте записывает 0, т.е. начальное состояние M, и перемещает головку второй ленты *U* в первую имитируемую клетку.
4. Для того чтобы отобразить переход *M, U* отыскивает на своей первой ленте переход $0^i10^j10^k10^l10^m$, у которого 0^i есть состояние на ленте 3, а 0^j — ленточный символ M, начинающийся с позиции на ленте 2, обозреваемой *U.* Это переход, который *M* совершает на следующем шаге. Машина *U* должна выполнить следующие действия:

   а) изменить содержимое ленты 3 на 0^k, т.е. отобразить изменение состояния M. Для этого U вначале заменяет все символы 0 на ленте 3 пустыми символами, а затем копирует 0^k с ленты 1 на ленту 3;

   б) заменить 0^j на ленте 2 символами 0^l, т.е. поменять ленточный символ M. Если потребуется больше или меньше места (т.е. *j<>l),* то для управления пространством используются рабочая лента и техника переноса;

   в) переместить головку на ленте 2 в ту позицию слева или справа, в которой находится ближайший символ 1. При m = 1 совершается сдвиг влево, а при m = 2 — вправо. Таким образом, U имитирует движение M влево или вправо.

5. Если M не имеет перехода, соответствующего имитируемым состоянию и ленточному символу, то в п. 4 переход не будет найден. Таким образом, в данной конфигурации *M* останавливается, и то же самое должна делать *U.*
6. Если *M* попадает в свое допускающее состояние, то *U* допускает.

Неразрешимость универсального языка

Представим проблему, которая является РП, но не рекурсивной. Это язык L_u. Неразрешимость L_u (т.е. его нерекурсивность) во многих отношениях важнее, чем предыдущий вывод о том, что язык L_d не является РП. Причина состоит в следующем. Сведением L_u к другой проблеме *P* можно показать, что алгоритма, решающего *P,* не существует, независимо от того, является ли P РП. Однако сведение L_d к P возможно лишь тогда, когда P не является РП, поэтому L_d не может использоваться при доказательстве неразрешимости проблем, которые являются РП, но не рекурсивными. Вместе с тем, если нужно показать, что проблема не является РП, то можно использовать только L_d, в то время как язык L_u оказывается бесполезным, поскольку он *является* РП.

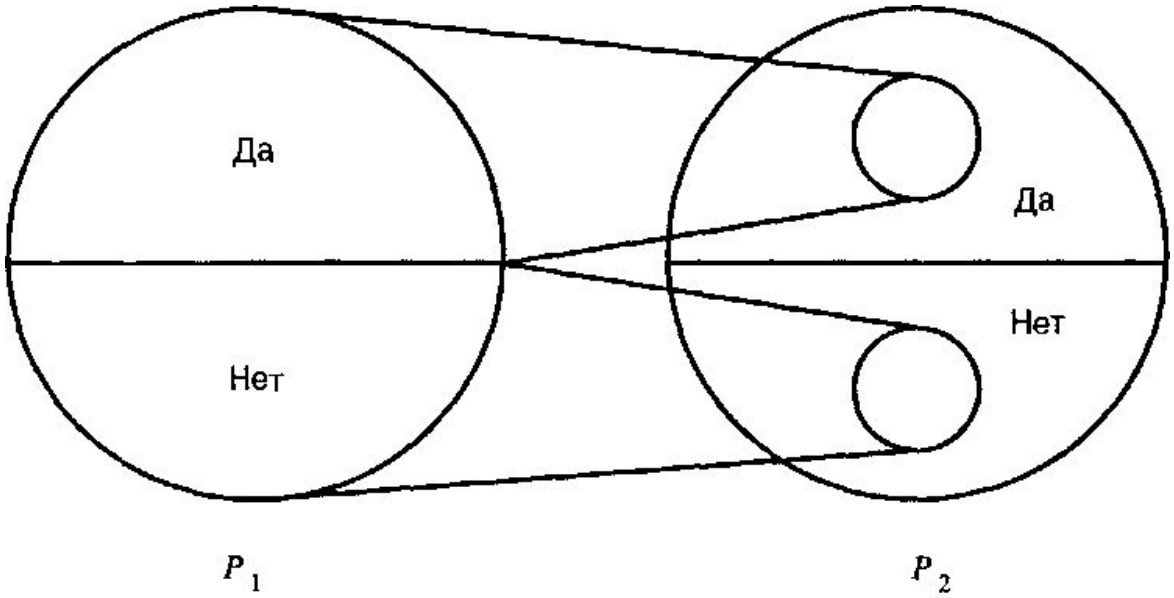
Теорема. L_u является **РП,** но не рекурсивным.

Неразрешимые проблемы и машины Тьюринга

- Статус языков L_u и L_d относительно неразрешимости и рекурсивной перечислимости был показан выше. Используем их для демонстрации других неразрешимых или не РП-языков. В доказательствах используется техника сведения. Все неразрешимые проблемы, которые рассматриваются вначале, связаны с машинами Тьюринга. Завершением данного раздела является "теорема Райса". В ней утверждается, что любое нетривиальное свойство МТ, зависящее лишь от языка, допускаемого машиной Тьюринга, должно быть неразрешимым. Материал этого раздела позволяет исследовать некоторые неразрешимые проблемы, не связанные ни с машинами Тьюринга, ни с их языками.

Сведение проблем

- Понятие сведения было введено выше. В общем случае, если у нас есть алгоритм, преобразующий экземпляры проблемы P_1 в экземпляры проблемы P_2, которые имеют тот же ответ (да/нет), то говорят, что P_1 *сводится к* P_2. Доказательство такой сводимости используется, чтобы показать, что P_2 не менее трудна, чем P_1. Таким образом, если P_1 не является рекурсивной, то и P_2 не может быть рекурсивной. Если P_1 не является РП, то и P_2 не может быть РП. Как уже упоминалось выше, чтобы доказать, что проблема P_2 не менее трудна, чем некоторая известная P_1 нужно свести P_1 к P_2, а не наоборот.
- Как показано на рисунке, сведение должно переводить всякий экземпляр P_1 с ответом "да" (позитивный) в экземпляр P_2 с ответом "да", и всякий экземпляр P_1, с ответом "нет" (негативный) — в экземпляр P_2 с ответом "нет". Отметим, что неважно, является ли каждый экземпляр P_2 образом одного или нескольких экземпляров P_1. В действительности, обычно лишь небольшая часть экземпляров P_2 образуется в результате сведения.



Теорема о сведении

Формально сведение P_1 к P_2 является машиной Тьюринга, которая начинает работу с экземпляром проблемы P_1 на ленте и останавливается, имея на ленте экземпляр проблемы P_2. Как правило, сведения описываются так, как если бы они были компьютерными программами, которые на вход получают экземпляры P_1 и выдают экземпляры P_2. Эквивалентность машин Тьюринга и компьютерных программ позволяет описывать сведение в терминах как тех, так и других. Значимость процедуры сведения подчеркивается следующей широко используемой теоремой.

Теорема. Если P_1 можно свести к P_2, то:

а) если P_1 неразрешима, то и P_2 неразрешима;

б) если P_1 не-**РП,** то P_2 также не-**РП.**

Машина Тьюринга и пустой язык

В качестве примера сведения, связанного с машинами Тьюринга, исследуем языки, известные как $\mathrm{L_e}$ и $\mathrm{L_{ne}}$. Оба они состоят из двоичных цепочек. Если w — двоичная цепочка, то она представляет некоторую МТ M_i из перечисления, описанного выше.

Если $\mathrm{M}_i = \emptyset$, т.е. M_i не допускает никакого входа, то M_i принадлежит L_e. Таким образом, L_e состоит из кодов всех МТ, языки которых пусты. С другой стороны, если $\mathrm{L(M_i)}$ не пуст, то w принадлежит $\mathrm{L_{ne}}$. Таким образом, $\mathrm{L_{ne}}$ состоит из кодов всех машин Тьюринга, которые допускают хотя бы одну цепочку.

В дальнейшем будет удобно рассматривать цепочки как машины Тьюринга, представленные этими цепочками. Итак, упомянутые языки определяются следующим образом.

$$\mathrm{L_e} = \{\mathbf{M}|\mathbf{L}(\mathbf{M}) = \emptyset\}$$
$$\mathrm{L_{ne}} = \{\mathbf{M}|\mathbf{L}(\mathbf{M}) \neq \emptyset\}$$

Отметим, что $\mathrm{L_e}$ и $\mathrm{L_{ne}}$ — языки над алфавитом {0, 1}, дополняющие друг друга. Как будет далее, $\mathrm{L_{ne}}$ является "более легким". Он РП, но не рекурсивный, в то время как $\mathrm{L_e}$ — не-РП.

Теорема. Язык $\mathrm{L_{ne}}$ рекурсивно перечислим.

Теорема. Язык $\mathrm{L_{ne}}$ не является рекурсивным.

Теорема. Язык $\mathrm{L_e}$ не является РП.

Теорема Райса и свойства РП-языков слайд 1/2

- Неразрешимость языков, подобных L_e и L_{ne}, в действительности представляет собой частный случай гораздо более общей теоремы: любое нетривиальное свойство РП-языков неразрешимо в том смысле, что с помощью машин Тьюринга невозможно распознать цепочки, которые являются кодами МТ, обладающих этим свойством. Примером свойства РП-языков служит выражение "язык является контекстно-свободным". Вопрос о том, допускает ли данная МТ контекстно-свободный язык, является неразрешимым частным случаем общего закона, согласно которому все нетривиальные свойства РП- языков неразрешимы.
- *Свойство* РП-языков представляет собой некоторое множество РП-языков. Таким образом, формально свойство языка быть контекстно-свободным — это множество всех КС-языков. Свойство быть пустым есть множество $\{\emptyset\}$, содержащее только пустой язык.

Теорема Райса и свойства РП-языков слайд 2/2

Свойство называется *тривиальным,* если оно либо пустое (т.е. никакой язык вообще ему не удовлетворяет), либо содержит все РП-языки. В противном случае свойство называется *нетривиальным.*

• Отметим, что пустое свойство 0 и свойство быть пустым языком — не одно и то же.

Мы не можем распознать множество языков так, как сами эти языки. Причина в том, что обычный бесконечный язык нельзя записать в виде цепочки конечной длины, которая может быть входом некоторой МТ. Вместо этого мы должны распознавать машины Тьюринга, которые допускают эти языки. Код самой МТ конечен, даже если язык, который она допускает, бесконечен. Таким образом, если P — это свойство РП-языков, то $\mathrm{L_P}$— множество кодов машин Тьюринга M, языки $L(M_i)$ которых принадлежат *P.* Говоря о разрешимости свойства P, мы имеем в виду разрешимость языка $\mathrm{L_P}$.

Теорема (теорема Райса). Всякое нетривиальное свойство РП-языков неразрешимо.

Проблемы описания языка машиной Тьюринга

Согласно теореме, приведённой выше, все проблемы, связанные только с языками машин Тьюринга, неразрешимы. Некоторые из них интересны сами по себе. Например, неразрешимы следующие вопросы.

- Пуст ли язык, допускаемый МТ?
- Конечен ли язык МТ?
- Регулярен ли язык, допускаемый МТ?
- Является ли язык МТ контекстно-свободным?

Вместе с тем, теорема Райса не означает, что любая проблема, связанная с МТ, неразрешима. Например, вопросы о состояниях МТ, в отличие от вопросов о языке, могут быть разрешимыми.

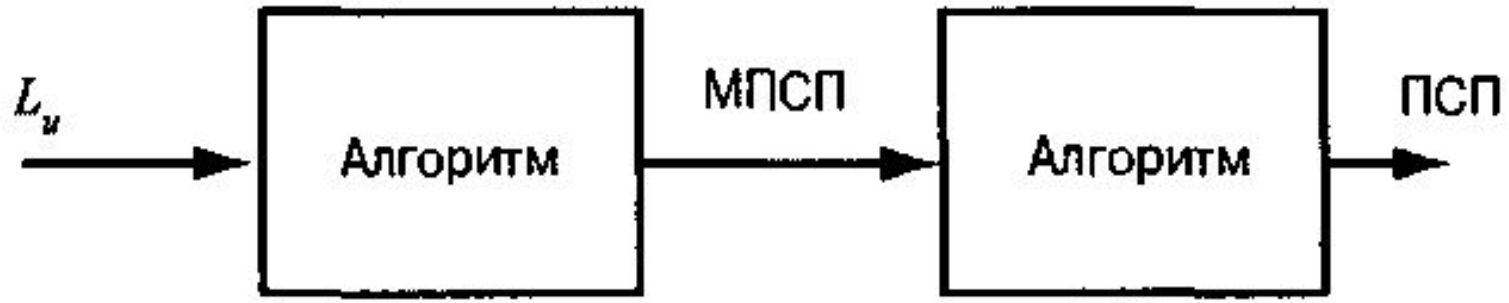
Пример. Вопрос о том, имеет ли МТ пять состояний, разрешим. Алгоритм, ре шающий его, просматривает код МТ и подсчитывает число состояний, встреченных в его переходах.

Еще один пример разрешимого вопроса — существует ли вход, при обработке которого МТ совершает более пяти переходов?

Алгоритм решения становится очевидным, если заметить, что, когда МТ делает пять переходов, она обозревает не более девяти клеток вокруг начальной позиции головки. Поэтому можно проимитировать пять переходов МТ на любом из конечного числа входов, длина которых не более девяти. Если все эти имитации не достигают останова, то делается вывод, что на любом входе данная МТ совершает более пяти переходов.

Проблема соответствий Поста

- начнём сведение неразрешимых вопросов о машинах Тьюринга к неразрешимым вопросам о "реальных вещах", т.е. обычных предметах, не имеющих отношения к абстрактным машинам Тьюринга. Начнем с проблемы, которая называется "проблемой соответствий Поста" (ПСП). Эта проблема по-прежнему довольно абстрактна, но она связана уже не с машинами Тьюринга, а с цепочками. Основная цель — доказать неразрешимость этой проблемы, чтобы затем доказывать неразрешимость других проблем путем сведения ПСП к ним.
- Докажем неразрешимость ПСП сведением L_u к ней. Чтобы облегчить доказательство, рассмотрим вначале "модифицированную" версию ПСП, а потом сведем ее к исходной ПСП. Затем сведем L_u к модифицированной ПСП. Цепь этих сведений представлена на рисунке. Поскольку известно, что исходная проблема L_u неразрешима, можно сделать вывод, что ПСП также неразрешима.



Определение ПСП

- Экземпляр *проблемы соответствий Поста* (ПСП) состоит из двух списков равной длины в некотором алфавите Σ. Как правило, они называются списками *A* и *B,* и пишутся $A = w_1, w_2, \cdots, w_k$ и $B = x_1, x_2, \cdots, x_k$ при некотором целом *к.* Для каждого i пара (w_i, x_i) называется парой *соответствующих цепочек.*
- Говорят, что экземпляр ПСП *имеет решение,* если существует последовательность из одного или нескольких целых чисел $i_1, i_2, \cdots, i_m$, которая, если считать эти числа индексами цепочек и выбрать соответствующие цепочки из списков A и *B,* дает одну и ту же цепочку, т.е. $w_{i1} w_{i2} \cdots w_{im} = x_{i1} x_{i2} \cdots x_{im}$. В таком случае последовательность $i_1, i_2, \cdots, i_m$ называется *решающей последовательностью,* или просто *решением,* данного экземпляра ПСП. Проблема соответствий Поста состоит в следующем: выяснить, имеет ли решение данный экземпляр ПСП.

Пример ПСП 1

Пусть $\Sigma = \{0, 1\}$, A и B — списки (см. рисунок). Данный экземпляр ПСП имеет решение. Пусть, например, $\mathrm{m} = 4, \mathrm{i}_1 = 2, \mathrm{i}_2 = 1, \mathrm{i}_3 = 1, \mathrm{i}_4 = 3$, т.е. решающая последовательность имеет вид 2, 1, 1,3. Для проверки записываются конкатенации соответствующих цепочек каждого из списков в порядке, задаваемом данной последовательностью, т.е. $\mathrm{w}_2\mathrm{w}_1\mathrm{w}_1\mathrm{w}_3 = \mathrm{x}_2\mathrm{x}_1\mathrm{x}_1\mathrm{x}_3 = 101111110$. Отметим, что это решение — не единственное. Например, 2, 1, 1, 3, 2, 1, 1, 3 также является решением.

	Список A	Список B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Пример ПСП 2

- Пусть, как и раньше, Σ = {0, 1}, но теперь экземпляр проблемы представлен списками, как на рисунке.
- В этом примере решения нет. Допустим, что экземпляр ПСП, представленный на слайде, имеет решение, скажем, $i_1, i_2, \cdots, i_m$ при некотором $m > 1$. Утверждаем, что $i_1 = 1$. При $i_1 = 2$ цепочка, начинающаяся с $w_2 = 011$, должна равняться цепочке, которая начинается с $x_2 = 11$. Но это равенство невозможно, поскольку их первые символы — 0 и 1, соответственно. Точно так же невозможно, чтобы $i_1 = 2$, поскольку тогда цепочка, начинающаяся с $w_3 = 101$, равнялась бы цепочке, которая начинается с $x_3 = 011$.

	Список A	Список B
i	w_i	x_i
1	10	101
2	011	11
3	101	011

Неразрешимость ПСП 2

Если $i_1 = 1$, то две соответствующие цепочки из списков *A* и *B* должны начинаться так:

A: 10...

B: 101...

Рассмотрим теперь, каким может быть i_2.

- Вариант $i_2 = 1$ невозможен, поскольку никакая цепочка, начинающаяся с $w_1 w_1 = 1010$, не может соответствовать цепочке, которая начинается с $x_1 x_1 = 101101$; эти цепочки различаются в четвертой позиции.
- Вариант $i_2 = 2$ также невозможен, поскольку никакая цепочка, начинающаяся с $w_1 w_2 = 10011$, не может соответствовать цепочке, которая начинается с $x_1 x_2 = 1011$; они различаются в третьей позиции.
- Возможен лишь вариант $i_2 = 3$.

При $i_2 = 3$ цепочки, соответствующие списку чисел 1, 3, имеют следующий вид.

A: 10101...

B: 101011...

Пока не видно, что последовательность 1, 3 невозможно продолжить до решения. Однако обосновать это несложно. Действительно, задача находится в тех же условиях, в которых была после выбора $i_1 = 1$. Цепочка из списка *B* отличается от цепочки из списка *A* лишним символом 1 на конце. Чтобы избежать несовпадения, необходимо выбирать $i_3 = 3, i_4 = 3$ и так далее. Таким образом, цепочка из списка *A* никогда не догонит цепочку из списка *B,* и решение никогда не будет получено.

Модифицированная ПСП

- Свести L_u к ПСП легче, если рассмотреть вначале промежуточную версию ПСП, которая называется *модифицированной проблемой соответствий Поста,* или МПСП. В модифицированной ПСП на решение накладывается дополнительное требование, чтобы первой парой в решении была пара первых элементов списков *A* и *B.* Более формально, экземпляр МПСП состоит из двух списков $A = w_1, w_2, \cdots, w_k$ и $B = x_1, x_2, \cdots, x_k$, и решением является последовательность из 0 или нескольких целых чисел $i_1, i_2, \cdots, i_m$, при которой $w_1 w_{i1} \cdots w_{im} = x_1 x_{i1} \cdots x_{im}$.
- Отметим, что цепочки обязательно начинаются парой (w_1, x_1), хотя индекс 1 даже не указан в качестве начального элемента решения. Кроме того, в отличие от ПСП, решение которой содержит хотя бы один элемент, решением МПСП может быть и пустая последовательность (когда $w_1 = x_1$). Однако такие экземпляры не представляют никакого интереса и далее не рассматриваются.

Пример МПСП

Списки, представленные на слайде 219, можно рассматривать как экземпляр МПСП. Однако этот экземпляр МПСП не имеет решения. Для доказательства заметим, что всякое его частичное решение начинается с индекса 1, поэтому цепочки, образующие решение, должны начинаться следующим образом.

A: 1...

B: 111...

Следующим целым в решении не может быть 2 или 3, поскольку и w_2, и w_3 начинаются с 10, и поэтому при таком выборе возникнет несоответствие в третьей позиции. Таким образом, следующий индекс должен быть 1, и получаются такие цепочки.

A: 11...

B: 111111...

Эти рассуждения можно продолжать бесконечно. Только еще одно число 1 позволяет избежать несоответствия в решении. Но если все время выбирается индекс 1, то цепочка *B* остается втрое длиннее *A,* и цепочки никогда не сравняются.

Важным шагом в доказательстве неразрешимости ПСП является сведение МПСП к ПСП. Далее будет показано, что МПСП неразрешима, путем сведения $\mathrm{L_u}$ к МПСП. Тем самым будет доказана неразрешимость ПСП. Действительно, если бы она была разрешима, то можно было бы решить МПСП, а следовательно, и $\mathrm{L_u}$.

По данному экземпляру МПСП с алфавитом Σ соответствующий экземпляр ПСП строится следующим образом. Вначале вводится новый символ *, который помещается между символами цепочек экземпляра МПСП. При этом в цепочках списка *A* он следует за символами алфавита Σ, а в цепочках списка *B* — предшествует им. Исключением является новая пара, которая строится по первой паре экземпляра МПСП; в этой паре есть дополнительный символ * в начале w_1 поэтому она может служить началом решения ПСП. К экземпляру ПСП добавляется также заключительная пара ($, *$). Она служит последней парой в решении ПСП, имитирующем решение экземпляра МПСП.

Формализация МПСП

- Формализуем описанную конструкцию. Пусть дан экземпляр МПСП со списками $A = w_1, w_2, \cdots, w_k$ и $B = x_1, x_2, \cdots, x_k$. Предполагается, что символы * и $ отсутствуют в алфавите Σ данного экземпляра МПСП. Строится экземпляр ПСП со списками $C = y_0, y_1, \cdots, y_{k+1}$ и $D = z_0, z_1, \cdots, z_{k+1}$ следующим образом:
  - Для i=1,2,…,k положим y_i равной w_i с символом * после каждого ее символа, а z_i — равной x_i с символом * перед каждым ее символом.
  - $y_0 =* y_1$ и $z_0 = z_1$, т.е. нулевая пара выглядит так же, как первая, с той лишь разницей, что в начале цепочки из первого списка есть еще символ *. Отметим, что нулевая пара будет единственной парой экземпляра ПСП, в которой обе цепочки начинаются одним и тем же символом, поэтому всякое решение данного экземпляра ПСП должно начинаться с индекса 0.
  - $y_{k+1} = \$$ и $z_{k+1} = * \$$.
- **Теорема.** МПСП сводится к ПСП.

Сведение к МПСП слайд 1/4

- Завершим цепь сведений (см. слайд 297), сведя L_u к МПСП. По заданной паре (M, *w)* строится экземпляр МПСП *(A, B),* имеющий решение тогда и только тогда, когда МТ *M* допускает вход *w.*
- Основная идея состоит в том, что частичные решения экземпляра МПСП *(A, B)* имитируют обработку входа *w* машиной M. Частичные решения будут состоять из цепочек, которые являются префиксами последовательности МО M $\#\alpha_1\#\alpha_2\#\alpha_3\#\cdots$, где α_1 — начальное МО M при входной цепочке *w* и $\alpha_i \vdash \alpha_{i+1}$ для всех i. Цепочка из списка *B* всегда на одно МО впереди цепочки из списка *A,* за исключением ситуации, когда *M* попадает в допускающее состояние. В этом случае используются специальные пары, позволяющие списку *A* "догнать" список B и в конце концов выработать решение. Однако если машина не попадает в допускающее состояние, то эти пары не могут быть использованы, и проблема не имеет решения.
- Для того чтобы упростить построение экземпляра МПСП, воспользуемся теоремой, согласно которой можно считать, что МТ никогда не печатает пробел и ее головка не сдвигается левее исходного положения. Тогда МО машины Тьюринга всегда будет цепочкой вида $\alpha q \beta$, где α и β — цепочки непустых символов на ленте, а q — состояние. Однако тогда, когда головка обозревает пробел непосредственно справа от α, позволим цепочке β быть пустой вместо того, чтобы помещать пробел справа от состояния. Таким образом, символы цепочек α *и* β будут в точности соответствовать содержимому клеток, в которых записан вход, а также всех тех клеток справа, в которых головка уже побывала.

Сведение к МПСП слайд 2/4

Пусть $M = (Q, \Sigma, Г, \delta, q_0, B, F)$ — машина Тьюринга, удовлетворяющая условиям теоремы и w — входная цепочка из Σ^*. По ним строится экземпляр МПСП следующим образом. Чтобы понять, почему пары выбираются пары именно так, а не иначе, нужно помнить, что нам нужно, чтобы первый список всегда на одно МО отставал от второго, если только М не попадает в допускающее состояние.

1. Первая пара имеет следующий вид.

Список *A* Список *B*

\# \#q_0w#

В соответствии с правилами МПСП эта пара — первая в любом решении. С нее начинается имитация *M* на входе *w.* Заметим, что в начальный момент список *B* опережает список *A* на одно полное МО.

2. Ленточные символы и разделитель # могут быть добавлены в оба списка. Пары

Список *A* Список *B*

X X для каждого X из Г

\# #

позволяют "копировать" символы, не обозначающие состояния. Выбирая такие пары, можно одновременно и удлинить цепочку списка *A* до соответствующей цепочки списка *B,* и скопировать часть предыдущего МО в конец цепочки списка *B.* Это поможет нам записать в конец цепочки списка *B* следующее МО в последовательности переходов *M.*

Сведение к МПСП слайд 3/4

3. Для имитации перехода *M* применяются специальные пары. Для всех *q* из *Q* - *F* (т.е. состояние *q* не является допускающим), *p* из *Q* и *X, Y,Z* из Г есть следующие пары.

Список *A*	Список *B*	
qX	Yp	*если* $\delta(q, X) = (p, Y, R)$
ZqX	pZY	*если* $\delta(i, X) = (p, Y, L)$; Z— любой ленточный символ
q#	Yp#	*если* $\delta(q, B) = (p, Y, R)$
Zq#	pZY#	*если* $\delta(q, B) = (p, Y, L)$; Z — любой ленточный символ

Как и в пункте 2, эти пары позволяют добавить к цепочке *B* следующее МО, дописывая цепочку *A* таким образом, чтобы она соответствовала цепочке *B*. Однако эти пары используют состояние для определения, как нужно изменить текущее МО, чтобы получить следующее. Эти изменения — новое состояние, ленточный символ и сдвиг головки — отображаются в МО, которое строится в конце цепочки B.

4. Если МО, которое находится в конце цепочки *B*, содержит допускающее состояние, нужно сделать частичное решение полным. Для этого добавляются "МО", которые в действительности не являются МО машины М, а отображают ситуацию, при которой в допускающем состоянии разрешается поглощать все ленточные символы по обе стороны от головки. Таким образом, если *q* — допускающее состояние, то для всех ленточных символов X и Г существуют следующие пары.

Список *A*	Список *B*
XqY	q
Xq	q
qY	q

Сведение к МПСП слайд 4/4

5. Наконец, если допускающее состояние поглотило все ленточные символы, то оно ос тается в одиночестве как последнее МО в цепочке *B.* Таким образом, *разность* цепочек (суффикс цепочки *B*, который нужно дописать к цепочке *A* для того, чтобы она соответствовала *B)* есть q#, и для завершения решения используется последняя пара.

Список *A* Список *B*

q## #

Теорема. Проблема соответствий Поста неразрешима.

Проблемы, связанные с программами

- Прежде всего, отметим, что на любом привычном языке можно написать программу, которая на вход получает экземпляр ПСП и ищет решение по определенной системе, например, в порядке возрастания ***длины*** (числа пар) потенциальных решений. Поскольку ПСП может иметь произвольный алфавит, нужно закодировать символы ее алфавита с помощью двоичного или другого фиксированного алфавита.
- Программа может выполнять любое конкретное действие, например, останавливаться или печатать фразу hello, world, если она нашла решение. В противном случае программа никогда не выполняет это конкретное действие. Таким образом, вопрос о том, печатает ли программа hello, world или выполняет любое другое не тривиальное действие, неразрешим. В действительности для программ справедлив аналог теоремы Райса: всякое нетривиальное свойство программы, связанное с ее действиями (но не лексическое или синтаксическое свойство самой программы), является нераз*решимым.*

Неразрешимость неоднозначности КС-грамматик слайд 1/3

Программы и машины Тьюринга — это в общем-то одно и то же. Теперь покажем, как ПСП можно свести к проблеме, которая, на первый взгляд, совсем не похожа на вопрос о компьютерах. Это проблема выяснения, является ли данная КС-грамматика неоднозначной.

Основная идея заключается в рассмотрении цепочек, представляющих обращение последовательности индексов, вместе с соответствующими им цепочками одного из списков экземпляра ПСП. Такие цепочки могут порождаться некоторой грамматикой. Аналогичное множество цепочек для второго списка экземпляра ПСП также может порождаться некоторой грамматикой. Если объединить эти грамматики очевидным способом, то найдется цепочка, которая порождается правилами каждой исходной грамматики тогда и только тогда, когда данный экземпляр ПСП имеет решение. Таким образом, решение существует тогда и только тогда, когда в грамматике для объединения присутствует неоднозначность.

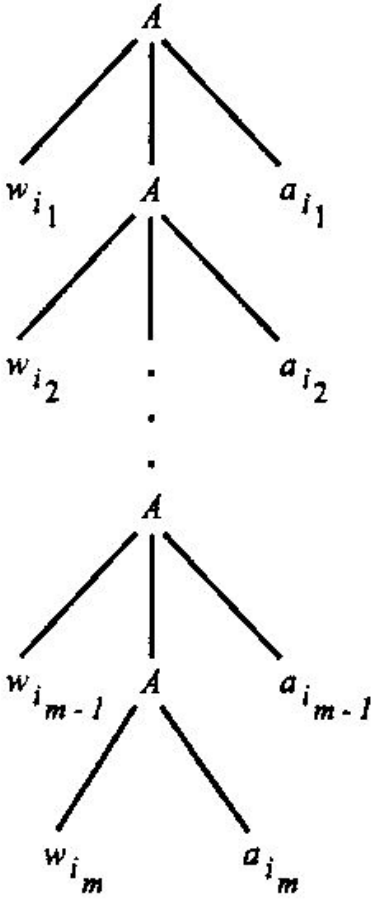
Уточним эту идею. Пусть экземпляр ПСП состоит из списков $A = w_1, w_2 \cdots w_k$ и $B = x_1, x_2 \cdots x_k$. Для списка *A* построим КС-грамматику с одной переменной *A*. Терминалами будут все символы алфавита Σ, используемые в данном экземпляре ПСП, плюс не пересекающееся с Σ множество *индексных символов* $a_1, a_2 \cdots a_k$, которые представляют выборы пар цепочек в решении ПСП. Таким образом, индексный символ a_i представляет выбор w_i из списка *A* или x_i из списка *B*. Продукции КС-грамматики для списка *A* имеют следующий вид.

$$A \to w_1 A a_1 | w_2 A a_2 | \cdots | w_k A a_k | w_1 a_1 | w_2 a_2 | \cdots | w_k a_k$$

Обозначим эту грамматику через G_A, а ее язык — через L_A. Язык типа L_A будет называться в дальнейшем *языком для списка A.*

Неразрешимость неоднозначности КС-грамматик слайд 2/3

- Заметим, что все терминальные цепочки, порожденные переменной *A* в G_A, имеют вид $w_{i1} w_{i2} \cdots w_{im} a_{i1} a_{i2} \cdots a_{im}$ при некотором $m > 1$, где $i_1, i_2 \cdots i_m$ — последовательность целых чисел в пределах от 1 до к. Все выводимые цепочки содержат одиночный символ *A*, отделяющий цепочки *w* от индексных символов *a*, до тех пор, пока не используется одно из *к* правил последней группы, в которых *A* отсутствует. Поэтому деревья разбора имеют структуру, представленную на рисунке.
- Заметим также, что любая терминальная цепочка порождается в G_A одним-единственным способом. Индексные символы в конце цепочки однозначно определяют, какое именно правило должно использоваться на каждом шаге, поскольку тела лишь двух продукций оканчиваются данным индексным символом a_1: $A \to w_i A a_i$, и $A \to w_i a_i$. Первую продукцию нужно использовать, когда данный шаг порождения не последний, в противном же случае используется вторая продукция.



Неразрешимость неоднозначности КС-грамматик слайд 3/3

Рассмотрим теперь вторую часть экземпляра ПСП — список $\mathrm{B} = \mathrm{x}_1, \mathrm{x}_2 \cdots \mathrm{x}_\mathrm{k}$. Этому списку мы поставим в соответствие еще одну грамматику $\mathrm{G_B}$ с такими продукциями.

$$B \to x_1 B \mathrm{a}_1 | x_2 B \mathrm{a}_2 | \cdots | x_\mathrm{k} B \mathrm{a}_\mathrm{k} | x_1 \mathrm{a}_1 | x_2 \mathrm{a}_2 | \cdots | x_\mathrm{k} \mathrm{a}_\mathrm{k}$$

Язык этой грамматики обозначим через L_B. Все замечания, касающиеся $\mathrm{G_A}$, справедливы и для $\mathrm{G_B}$. В частности, терминальная цепочка из L_B имеет одно-единственное порождение, определяемое индексными символами в конце этой цепочки.

Наконец, мы объединяем языки и грамматики двух списков, формируя грамматику $\mathrm{G_{AB}}$ для всего данного экземпляра ПСП. $\mathrm{G_{AB}}$ имеет следующие компоненты.

- Переменные *A, B* и *S,* последняя из которых — стартовый символ.
- *Продукции* $\mathrm{S} \to \mathrm{A}|\mathrm{B}$.
- Все продукции грамматики $\mathrm{G_A}$.
- Все продукции грамматики $\mathrm{G_B}$.

Утверждаем, что грамматика $\mathrm{G_{AB}}$ неоднозначна тогда и только тогда, когда экземпляр *(A, B)* ПСП имеет решение. Это утверждение является основным в следующей теореме.

Теорема. Вопрос о неоднозначности КС-грамматики неразрешим.

Дополнение списка языка

- Наличие контекстно-свободных языков, подобных L_A для списка *A,* позволяет показать неразрешимость ряда проблем, связанных с КС-языками. Еще больше фактов о неразрешимости свойств КС-языков можно получить, рассмотрев дополнение этого языка $\overline{L_A}$. Отметим, что язык $\overline{L_A}$ состоит из всех цепочек в алфавите $\Sigma \cup \{a_1, a_2, \cdots, a_k\}$, не содержащихся в L_A, где Σ — алфавит некоторого экземпляра ПСП, а a_i — символы, которые не принадлежат Σ и представляют индексы пар в этом экземпляре ПСП.

Интересными элементами языка $\overline{L_A}$ являются цепочки, префикс которых принадлежит Σ^* , т.е. является конкатенацией цепочек из списка *A,* а суффикс состоит из индексных символов, *не соответствующих* цепочкам из префикса. Однако помимо этих элементов в $\overline{L_A}$ содержатся цепочки неправильного вида, не принадлежащие языку регулярного выражения $\Sigma^*(a_1 + a_2 + \cdots + a_k)^*$.

Можно показать, что $\overline{L_A}$ является КС-языком. В отличие от L_A, грамматику для $\overline{L_A}$ построить нелегко, но можно построить МП-автомат, точнее — детерминированный МП-автомат.

Теорема. Если L_A — язык для списка *A,* то $\overline{L_A}$ является контекстно-свободным языком.

Неразрешимость КС-грамматики и регулярного выражения

Для получения результатов о неразрешимости КС-языков можно использовать L_A, L_B и их дополнения различными способами. Часть этих фактов собрана в следующей теореме.

Теорема. Пусть G_1 и G_2 — КС-грамматики, а R — регулярное выражение. Тогда неразрешимы следующие проблемы:

а) $\mathrm{L}(G_1) \cap \mathrm{L}(G_2) = 0$?

б) $\mathrm{L}(G_1) = \mathrm{L}(G_2)$?

в) $\mathrm{L}(G_1) = \mathrm{L}(\mathrm{R})$?

г) верно ли, что $\mathrm{L}(G_1) — \mathrm{T}^*$ для некоторого алфавита *T?*

д) $\mathrm{L}(G_1) \subseteq \mathrm{L}(G_2)$?

е) $\mathrm{L}(R) \subseteq \mathrm{L}(G_1)$?

Трудноразрешимые задачи

- Обсуждение вычислимости переносится на уровень ее эффективности или неэффективности. Предметом изучения становятся разрешимые проблемы, и в данном разделе рассматривается вопрос о том, какие из них можно решить на машинах Тьюринга за время, полиномиально зависящее от размера входных данных.
- Напомним два важных факта:
  - Проблемы, разрешимые за полиномиальное время на обычном компьютере, — это именно те проблемы, которые разрешимы за такое же время с помощью машины Тьюринга.
  - Практика показывает, что разделение проблем на разрешимые за полиномиальное время и требующие экспоненциального или большего времени, является фундаментальным. Полиномиальное время решения реальных задач, как правило, является приемлемым, тогда как задачи, требующие экспоненциального времени, практически (за приемлемое время) неразрешимы, за исключением небольших экземпляров.

Состав раздела

- В этом разделе кратко рассматривается теория "труднорешаемости", т.е. методы доказательства неразрешимости проблем за полиномиальное время. Вначале рассматривается конкретный вопрос *выполнимости* булевой формулы, т.е. является ли она истинной для некоторого набора значений **ИСТИНА** и **ЛОЖЬ** ее переменных. Эта проблема играет в теории сложности такую же роль, как L_u и ПСП для неразрешимых проблем. Вначале будет показана "теорема Кука", которая означает, что проблему выполнимости булевых формул нельзя разрешить за полиномиальное время. Затем будет показано, как свести эту проблему ко многим другим, доказывая тем самым их труднорешаемость.
- При изучении полиномиальной разрешимости проблем изменяется понятие сведения. Теперь уже недостаточно просто наличия алгоритма, который переводит экземпляры одной проблемы в экземпляры другой. Алгоритм перевода сам по себе должен занимать не больше времени, чем полиномиальное, иначе сведение не позволит сделать вывод, что доказываемая проблема труднорешаема, даже если исходная проблема была таковой. Таким образом, вначале вводится понятие "полиномиальной сводимости" (сводимости за полиномиальное время).
- Между выводами, которые дают теории неразрешимости и труднорешаемости, существует еще одно принципиальное различие. Теоремы неразрешимости, приведенные выше, неопровержимы. Они основаны только на определении машины Тьюринга и общих математических принципах. В отличие от них, все приводимые здесь результаты о труднорешаемости проблем основаны на недоказанном, но безоговорочно принимаемом на веру предположении, которое обычно называется предположением $P \neq NP$.

Краткое введение в трудноразрешимость

- Итак, предполагается, что класс проблем, разрешимых недетерминированными МТ за полиномиальное время, содержит, по крайней мере, несколько проблем, которые нельзя решить за полиномиальное время детерминированными МТ (даже если для последних допускается более высокая степень полинома). Существуют буквально тысячи проблем, при надлежность которых данной категории *практически не вызывает сомнений,* поскольку они легко разрешимы с помощью НМТ с полиномиальным временем, но для их решения не известно ни одной ДМТ (или, что то же самое, компьютерной программы) с полиномиальным временем. Более того, важным следствием теории трудноразрешаемости является то, что либо все эти проблемы имеют детерминированные решения с полиномиальным временем — решения, ускользавшие от нас в течение целых столетий, либо таких решений не имеет ни одна из них, и они действительно требуют экспоненциального времени.

Определение класса P

- Говорят, что машина Тьюринга *M* имеет *временную сложность* $T(\mathrm{n})$ (или "время работы $T(\mathrm{n})$"), если, обрабатывая вход *w* длины *n*, *M* делает не более $T(\mathrm{n})$ переходов и останавливается независимо от того, допущен вход или нет. Это определение применимо к любой функции $T(\mathrm{n})$, например, $T(\mathrm{n}) = 50\mathrm{n}^2$ или $T(\mathrm{n}) = 5\mathrm{n}^2 + 2^{\mathrm{n}}$. В текущем разделе интересен, главным образом, случай, когда $T(\mathrm{n})$ является полиномом относительно *n*. Говорят, что язык *L* принадлежит классу *P*, если существует полином $T(\mathrm{n})$, при котором *L=L(M)* для некоторой детерминированной МТ *M* с временной сложностью $T(\mathrm{n})$.
- *Пример: алгоритм Краскала (см. курс «Дискретная математика»)*

Алгоритм Краскала и МТ: проблемы

- Выход алгоритмов разрешения различных проблем может иметь много разных форм, например, список ребер экономического дерева. Проблемы же, решаемые машинами Тьюринга, можно интерпретировать только как языки, а их выходом может быть только **ДА** или **НЕТ** (допустить или отвергнуть). Например, проблему поиска экономического дерева можно перефразировать так: "Для данного графа *G* и предельного числа *W* выяснить, имеет ли *G* остовное дерево с весом не более *W?".* Может показаться, что эту проблему решить легче, чем проблему экономического дерева в знакомой формулировке, поскольку не нужно даже искать остовное дерево. Однако в рамках теории труднорешаемости, как правило, нужно доказать, что проблема трудна (не легка). А из того, что "да/нет"-версия проблемы трудна, следует, что трудна и ее версия, предполагающая полный ответ.
- Хотя "размер" графа можно неформально представить себе как число его узлов или ребер, входом МТ является цепочка в некотором конечном алфавите. Поэтому такие элементы, фигурирующие в проблеме, как узлы и ребра, должны быть подходящим образом закодированы. Выполняя это требование, получаем в результате цепочки, длина которых несколько превышает предполагаемый "размер" входа. Однако есть две причины проигнорировать эту разницу.
  - Размеры входной цепочки МТ и входа неформальной проблемы всегда отличаются не более, чем малым сомножителем, обычно — логарифмом размера входных данных. Таким образом, все, что делается за полиномиальное время с использованием одной меры, можно сделать за полиномиальное время, используя другую меру.
  - Длина цепочки, представляющей вход, — в действительности более точная мера числа байтов, которые должен прочитать реальный компьютер, обрабатывая свой вход. Например, если узел задается целым числом, то количество байтов, необходимых для представления этого числа, пропорционально его логарифму, а это не "1 байт на каждый узел", как можно было бы предположить при неформальном подсчете размера входа.

Пример кодирования графа

- Рассмотрим код для графов и предельных весов, который может быть входом для проблемы ОДМВ. В коде используются пять символов: 0, 1, левая и правая скобки, а также запятая.

1. Припишем всем узлам целые числа от 1 до ***m.***
2. В начало кода поместим значения m и предельного веса *W* в двоичной системе счисления, разделенные запятой.
3. Если существует ребро, соединяющее узлы ***i*** и j и имеющее вес ***w,*** то в код записывается тройка ***(i,j, w),*** где целые числа i, ***j*** и ***w*** кодируются в двоичном виде. Порядок записи узлов ребра и порядок ребер в графе не играют роли.

- Таким образом, один из возможных кодов графа с предельным весом W= 40 имеет вид 100, 101000(1, 10, 1111)(1, 11, 1010)(10, 11, 1100)(10, 100, 10100)(11, 100, 10010).

Оценка кодирования

- Если кодировать входы проблемы экономического дерева так, как показано выше, то вход длины n может представлять максимум *O(*$\mathrm{n}/log\ \mathrm{n}$*)* ребер. Если число ребер очень мало, то число узлов m может быть экспоненциальным относительно n. Однако если число ребер e меньше m- 1, то граф не может быть связным, и независимо от того, каковы эти ребра, не имеет экономического дерева. Следовательно, если количество узлов превосходит число $\mathrm{n}/log\ \mathrm{n}$, то нет никакой необходимости запускать алгоритм Краскала — достаточно сказать: "нет, остовного дерева с таким весом не существует".
- Итак, пусть имеется верхняя граница времени работы алгоритма Краскала, выражаемая в виде функции от *m* и *e,* например, как найденная выше верхняя граница *0(e(e + m)).* Можно изменить *m* и *e* на *п* и сказать, что время работы выражается функцией от длины входа *п,* имеющей вид *0(п(п* + я)) или *0(*n^2*).* В действительности, более удачная реализация алгоритма Краскала требует времени *0(п* log *п),* но сейчас это не важно.
- Представленный алгоритм Краскала предназначен для реализации на языке программирования с такими удобными структурами данных, как массивы и указатели, но в качестве модели вычислений мы используем машины Тьюринга. Тем не менее, описанную выше версию алгоритма можно реализовать за *0(*n^2*)* шагов на многоленточной машине Тьюринга.

Дополнительные ленты МТ

- Дополнительные ленты используются для выполнения нескольких вспомогательных задач.

1. На одной из лент можно хранить информацию об узлах и их текущих номерах компонентов. Длина такой таблицы составит *O(n).*
2. Еще одна лента может применяться в процессе просмотра входной ленты для хранения информации о ребре, имеющем на данный момент наименьший вес среди ребер, не помеченных как "использованные" (выбранные). С помощью второй дорожки входной ленты можно отмечать те ребра, которые были выбраны в качестве ребер наименьшего веса на одном из предыдущих этапов работы алгоритма. Поиск непомеченного ребра наименьшего веса занимает время ***0(n),*** поскольку каждое ребро рассматривается лишь один раз, и сравнить вес можно, просматривая двоичные числа линейно, справа налево.
3. Если ребро на некотором этапе выбрано, то соответствующие два узла помещаются на ленту. Чтобы установить компоненты этих двух узлов, нужно просмотреть таблицу узлов и компонентов. Это займет *0(п*) времени.
4. Еще одна лента может хранить информацию об объединяемых компонентах *i* и *j,* когда найденное ребро соединяет различающиеся на данный момент компоненты. В этом случае просматривается таблица узлов и компонентов, и для каждого узла из компонента *i* номер компонента меняется на *j.* Эта процедура также занимает *O(п)* времени.

Оценка сложности алгоритма Краскала

- Теперь нетрудно завершить доказательство, что на многоленточной МТ каждый этап работы алгоритма может быть выполнен за время *O(n).* Поскольку чис ло этапов *e* не превышает n, делаем вывод, что времени $0(n^2)$ достаточно для многоленточной МТ. Теперь вспомним теорему, утверждавшую, что работу, которую многоленточная МТ выполняет за *s* шагов, можно выполнить на одноленточной МТ за $0(s^2)$ шагов. Таким образом, если многоленточной МТ требуется сделать $0(n^2)$ шагов, то можно построить одноленточную МТ, которая делает то же самое за $0((n^2)^2) = 0(n^4)$ шагов. Следовательно, "да/нет"-версия проблемы экономического дерева ("имеет ли граф *G* ОДМВ с общим весом не более *W?)* принадлежит *P.*

Недетерминированное полиномиальное время

- Фундаментальный класс проблем в изучении труднорешаемости образован проблемами, разрешимыми с помощью недетерминированной МТ за полиномиальное время. Формально, можно сказать, что язык L принадлежит классу NP (недетерминированный полиномиальный), если существуют недетерминированная МТ M и полиномиальная временная сложность *T(n),* для которых *L = L(M),* и у *M* нет последовательностей переходов длиной более T(n) при обработке входа длины n.
- Поскольку всякая детерминированная МТ представляет собой недетерминированную МТ, у которой нет возможности выбора переходов, то *V* с *MV.* Однако оказывается, что в *MV* содержится множество проблем, не принадлежащих *V.* Интуиция подсказывает: причина в том, что НМТ с полиномиальным временем работы имеет возможность угадывать экспоненциальное число решений проблемы и проверять "параллельно" каждое из них за полиномиальное время. И все-таки, одним из самых серьезных нерешенных вопросов математики является вопрос о том, верно ли, что *P = NP,* т.е. все, что с помощью НМТ делается за полиноми альное время, ДМТ в действительности также может выполнить за полиномиальное время (которое, возможно, выражается полиномом более высокой степени).

Проблема (задача) коммивояжера

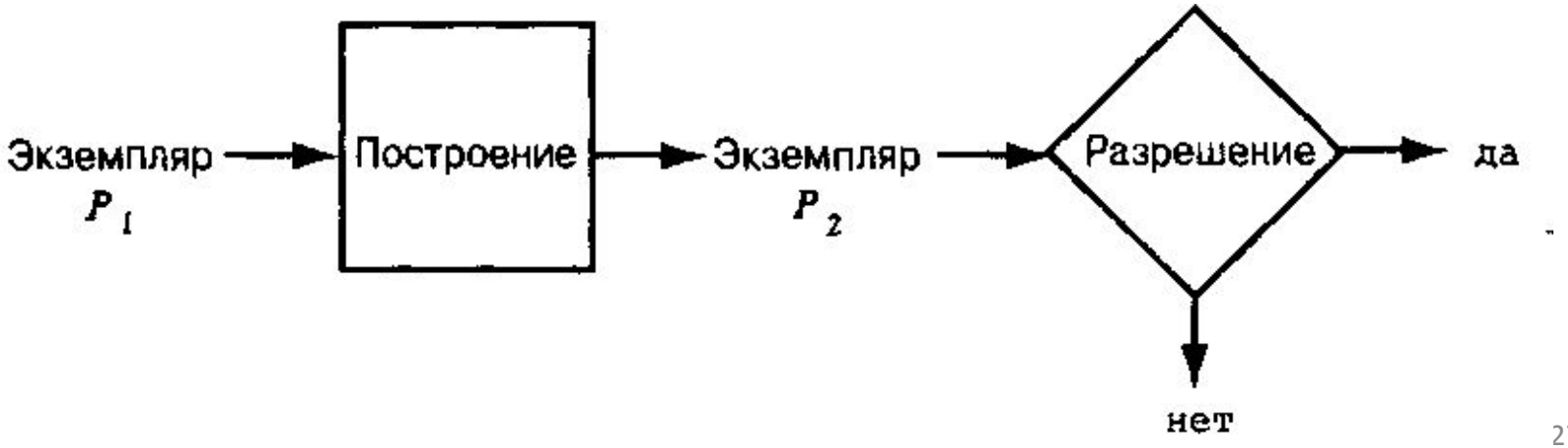
- Для того чтобы осознать, насколько обширен класс *NP,* рассмотрим пример проблемы, которая принадлежит классу *NP,* но, предположительно, не принадлежит *P, — проблему коммивояжера* (ПКОМ). Вход ПКОМ (как и у ОДМВ) — это граф, каждое ребро которого имеет целочисленный вес (рис. 10.1), а также предельный вес *W.* Вопрос состоит в том, есть ли в данном графе "гамильтонов цикл" с общим весом, не превышающим *W. Гамильтонов цикл* — это множество ребер, соединяющих узлы в один цикл, причем каждый узел встречается в нем только один раз. Заметим, что число ребер в гамильтоновом цикле должно равняться числу узлов графа.
- **Пример.** Граф, закодированный на слайде 239**,** имеет в действительности лишь один гамильтонов цикл — (1, 2, 4, 3, 1). Его общий вес составляет 15 + 20 + 18 + 10 = 63. Таким образом, если *W* имеет значение 63 или больше, то ответ— "да", а если *W* < 63, то ответ — "нет".
- Однако простота ПКОМ для графа с четырьмя узлами обманчива. В данном графе попросту не может быть более двух различных гамильтоновых циклов, учитывая, что один и тот же цикл может иметь начало в разных узлах и два направления обхода. Но в графе с *m* узлами число различных циклов достигает *0(m!),* факториала числа m, что превышает 2^{cm} для любой константы *c.*

Сложность решения НМТ

- Оказывается, что любой способ решения ПКОМ включает перебор, по сути, всех циклов и подсчет общего веса каждого из них. Можно поступить умнее, отбросив некоторые, очевидно неподходящие, варианты. Однако, по всей видимости, при любых усилиях все равно придется рассматривать экспоненциальное число циклов перед тем, как убедиться в том, что ни один из них не имеет вес меньше предельного W, или отыскать такой цикл.
- С другой стороны, если бы был доступен недетерминированный компьютер, то можно было бы "угадать" перестановку узлов и вычислить общий вес цикла, образованного узлами в этом порядке. Если бы существовал реальный недетерминированный компьютер, то при обработке входа длины п ни на одной из ветвей ему не пришлось бы сделать более *0(n)* шагов. На многоленточной НМТ перестановку можно угадать за $O(n^2)$ шагов, и столько же времени понадобится для проверки ее общего веса. Таким образом, одноленточная НМТ может решить ПКОМ за время, не превышающее $O(n^4)$. Делаем вывод, что ПКОМ принадлежит классу *NP.*

Полиномиальные сведения слайд 1/3

- Основной метод доказательства того, что проблему P_2 нельзя решить за полиномиальное время (т.е. P_2 не принадлежит P), состоит в сведении к ней проблемы P_1 относительно которой известно, что она не принадлежит P Данный подход представлен на рисунке.



Полиномиальные сведения слайд 2/3

- Допустим, что необходимо доказать утверждение: «если P_2 принадлежит P, то **и** P_1 принадлежит P». Поскольку мы утверждаем, что P_1 *не принадлежит* P, можно будет утверждать, что и P_2 не принадлежит P. Однако одного лишь существования алгоритма, обозначенного как "Построение", не достаточно для доказательства нужного утверждения.
- В самом деле, пусть по входному экземпляру проблемы P_1 длиной m алгоритм вырабатывает выходную цепочку длины 2^m, которая подается на вход гипотетическому алгоритму, решающему P_2 за полиномиальное время. Если решающий P_2 алгоритм делает это, скажем, за время $O(n^k)$, то вход длиной 2^m он обработает за время $O(2^{km})$, экспоненциальное по m. Таким образом, алгоритм, решающий P_1 обрабатывает вход длиной m за время, экспоненциальное по m. Эти факты целиком согласуются с ситуацией, когда P_2 принадлежит P, а P_1 — нет.
- Даже если алгоритм, переводящий экземпляр P_1 в экземпляр P_2, всегда вырабатывает экземпляр, длина которого полиномиальна относительно длины входа, то все равно можно не добиться желаемого результата. Предположим, что построенный экземпляр P_2 имеет ту же длину m, что и P_1 но сам алгоритм преобразования занимает время, экспоненциальное по m, скажем, $O(2^m)$. Тогда из того, что алгоритм решения P_2 тратит на обработку входа длиной n время $O(n^k)$, следует лишь то, что существует алгоритм решения P_1, которому для обработки входа длиной m нужно время $O(2^m + m^k)$. В этой оценке времени работы учитывается тот факт, что необходимо не только решить итоговый экземпляр P_2, но и получить его. И вновь возможно, что P_2 принадлежит P,а P_1 — нет.

Полиномиальные сведения слайд 3/3

- Правильное ограничение, которое необходимо наложить на преобразование P_1 в P_2, состоит в том, что время его выполнения должно полиномиально зависеть от размера входных данных. Отметим, что если при входе длины m преобразование занимает время $0(m^j)$, то длина выходного экземпляра P_2 не может превышать числа совершенных шагов, т.е. она не больше cm^j, где c — некоторая константа. Теперь можно доказать, что если P_2 принадлежит P, то и P_1 принадлежит P.
- Для доказательства предположим, что принадлежность цепочки длины n к P_2 можно выяснить за время $0(n)$. Тогда вопрос о принадлежности P_1 цепочки длины m можно решить за время $O\left(m^j + \left(cm^j\right)^k\right)$ слагаемое m^j учитывает время преобразования, а $\left(cm^j\right)^k$ — время выяснения, является ли результат экземпляром P_2. Упрощая выражение, видим, что P_1 может быть решена за время $O\left(m^j + cm^{jk}\right)$. Поскольку c, j и k — константы, это время зависит от m полиномиально, и, следовательно, P_1 принадлежит P.
- Итак, в теории труднорешаемости используются только *полиномиальные сведения ((сведения за полиномиальное время)*. Сведение P_1 к P_2 является полиномиальным, если оно занимает время, полиномиальное по длине входного экземпляра P_1. Как следствие, длина выходного экземпляра P_2 будет полиномиальной по длине экземпляра P_1.

NP-полные проблемы

Теперь познакомимся с группой проблем, которые являются наиболее известными кандидатами на то, чтобы принадлежать *NP,* но не принадлежать *P.* Пусть *L* — язык (проблема) из класса *NP.* Говорят, что *L* является *NP-полным,* если для него справедливы следующие утверждения.

- *L* принадлежит *NP.*
- Для всякого языка *L'* из NP существует полиномиальное сведение *L'к L.*

Как было показано, примером NP-полной проблемы является проблема коммивояжера. Предполагается, что $P \neq NP$, и, в частности, что все NP-полные проблемы содержатся в NP-P, поэтому доказательство NP-полноты проблемы можно рассматривать как свидетельство того, что она не принадлежит *P.*

Вначале докажем NP-полноту так называемой проблемы выполнимости булевой формулы (ВЫП), показав, что язык всякой НМТ с полиномиальным временем полиномиально сводится к ВЫП. Имея в распоряжении некоторые NP-полные проблемы, можно доказать NP-полноту еще одной, новой проблемы посредством полиномиального сведения к ней одной из известных проблем. Следующая теорема объясняет, почему такое сведение доказывает, что новая проблема является NP-полной.

Теорема. Если проблема P_1 является NP-полной, и существует полиномиальное сведение P_1 к P_2, то проблема P_2 также NP-полна.

Теорема Кука

- Для доказательства, что некоторая проблема является NP-полной, сначала нужно показать, что она принадлежит классу *NP,* а затем — что к ней сводится любой язык из *NP.*
- **Теорема** (теорема Кука). Проблема выполнимости булевой формулы NP-полна.
- Доказательство: [1], раздел 10.2.3