Версия 0.9pre-release от 24.12.2015. Возможны незначительные изменения.

Лекция 1. Основные понятия теории автоматов

Главные вопросы, которые мы обсуждаем, представлены на СЛАЙДЕ 1. Все примеры этого, а также большинства последующих разделов, если это не оговаривается особо, мы демонстрируем в системе JFLAP, работающей под управлением ОС GNU Linux.

1.1 Введение.

Теория автоматов занимается изучением абстрактных «вычислительных» устройств (или абстрактных машин). Еще в 1930 гг. Алан Тьюринг исследовал одну такую абстрактную машину, которая обладала (с некоторой долей условности, конечно) всеми возможностями современных вычислительных машин. Целью А.Тьюринга была точность описания границы между тем, что вычислительная машина «может», и тем, чего она «не может». Как показала практика, полученные им результаты применимы не только к абстрактным *машинам Тьюринга*, но к реальным современным компьютерам.

В 1940-1950 гг. многие исследователи занимались изучением простейших машин, которые в наши дни мы называем *конечными автоматами*. Такие автоматы вначале предлагались как модели функционирования человеческого мозга. Однако, как оказалось, они - весьма полезный инструмент для множества других целей, многие из которых будут освещены позднее.

Примерно в то же время известный американский лингвист Наум Хомский занялся изучением *формальных грамматик*. Они (грамматики) не являются машинами в строгом смысле, но, тем не менее, они тесно связаны с абстрактными автоматами и служат основой некоторых важнейших компонентов программного обеспечения, в частности компиляторов.

В конце 1960 – начале 1970 гг. Стивен Кук и Леонид Левин независимо друг от друга смогли, в частности, разделить задачи на те, которые могут быть эффективно решены вычислительной машиной, и те, которые могут быть решены в принципе, но требуют для этого столько времени, что компьютер оказывается бесполезным для решения почти всех экземпляров задачи, за исключением небольших. Задачи последнего класса называют труднорешаемыми (или NP-сложными). Даже при экспоненциальном росте быстродействия вычислительных машин (известный «Закон Мура») маловероятно, что удастся достигать значительных успехов в решении задач этого класса.

Все эти кажущиеся ненужными теоретические построения на самом деле достаточно широко применяются на практике. Например, автоматы и некоторые типы грамматик, как указывалось выше, используются при создании средств трансляции языков программирования. Другие понятия, в том числе упомянутая машина Тьюринга, позволяют уяснить теоретические возможности программного обеспечения. В частности, теория сложности вычислений позволяет понять, сможем ли мы с налету решить ту или иную задачу, или же нам придется добираться до финиша окольными путями (искать эвристический, приближенный или иной метод, позволяющий решить задачу за разумное время).

Теория автоматов и формальных языков лежит на стыке математики (дискретной) и прикладной информатики, поэтому не лишне уяснить, а зачем собственно мы взялись за изучение данной учебной дисциплины.

Конечные автоматы (далее – КА) являются моделью многих компонентов аппаратного и программного обеспечения. Далее будут подробно примеры их использования, а сейчас перечислим наиболее важные (СЛАЙД 2):

- ПО для верификации цифровых схем.

Версия 0.9pre-release от 24.12.2015. Возможны незначительные изменения.

- Лексический анализатор типичного транслятора.
- ПО для сканирования Web-страниц для обеспечения поиска шаблонов.
- ПО для верификации сетевых протоколов и прочих систем, которые могут находиться в конечном числе различных состояний.

Теперь попробуем дать неформальное описание КА. Существует множество систем и их подсистем, которые можно рассматривать, как находящихся в любой момент времени в одном из конечного числа *состояний*. Назначение каждого из них – запоминать моменты истории системы. Поскольку состояний может быть много, то запомнить все моменты невозможно, а значит, следует хранить действительно важную информацию и забывать о несущественной. Преимущество *конечности* заключается в том, что систему можно реализовать с ограниченным количеством ресурсов (в «железе» или в виде небольшой программы).

Пример 1.

Простейшим нетривиальным КА является переключатель «включено-выключено». Такое устройство помнит свое текущее состояние (*state*), и от этого состояния зависит результат нажатия кнопки. Из состояния «выключено» нажатие кнопки переводит переключатель в состояние «включено», и наоборот.

На СЛАЙДЕ 3 представлена КА-модель переключателя. Здесь, как и во многих последующих автоматах, состояния мы обозначаем кружками, имеющими те или иные метки (у нас «Вкл.» и «Выкл.»). Дуги (*transition*) между состояниями также имеют метки (у нас – «Нажатие», что означает нажатие на кнопку переключателя). Стрелки на дугах указывают, что всякий раз при «Нажатии» система переходит из одного состояния в другое.

Одно из состояний является «начальным» (*initial state*). Это состояние (у нас – «Выкл.»), в котором система находится изначально. На рисунке оно отмечено треугольником, одним из углов указывающим на это состояние.

Часто необходимо выделять одно или несколько «заключительных» или, что то же самое, «допускающих» состояний (*final state*). Попав в одно из них в результате реализации некоторой последовательности входных воздействий, можно считать такую последовательность в известном смысле «правильной». Например, мы можем считать состояние «вкл.» допускающим, т.к. если переключатель находится в этом состоянии, то устройство, управляемое им, находится в рабочем режиме. Допускающие состояния принято обозначать двойным кружком, но мы не использовали это обозначение из соображений очевидности.

Пример 2.

Понятно, что состояние КА может запоминать гораздо более сложную информацию, нежели выбор в примере 1. На СЛАЙДЕ 3 представлен КА, который может служить частью лексического анализатора. Он предназначен для распознавания ключевого слова *else*. Этот автомат должен иметь 5 различных состояний, каждое из которых представляет позицию в слове *else*, достигнутую на данный момент. Эти позиции соответствуют *префиксам* слова, начиная от пустой последовательности (т.е. еще ни одна позиция не достигнута) и заканчивая целым словом.

Мы обозначили каждое из пяти состояний частью слова, прочитанной на конкретном шаге. Входным воздействиям соответствуют буквы. Мы можем считать, что данный лексический анализатор каждый раз просматривает по одном слову транслируемой программы. Каждый последующий символ рассматривается как воздействие для данного автомата. Начальное состояние соответствует пустой цепочке символов, и каждое состояние имеет переход по очередной букве слова *else* в состояние, соответствующее следующему префиксу. Состояние, обозначенное меткой *else*,

достигается тогда, когда введено по буквам все данное слово. Так как назначение нашего КА заключается в распознавании слова *else*, то последнее состояние мы и будем считать допускающим. В нашем случае оно единственное.

Структурные представления

Следующие системы записи не являются автоматными, однако играют важную роль в теории автоматов и ее приложениях (СЛАЙД 5).

1. **Грамматики** являются полезными моделями при проектировании программного обеспечения, обрабатывающего данные рекурсивной природы. Чаще всего их используют «синтаксические анализаторы», являющиеся компонентами трансляторов, работающими с такими конструкциями языков программирования, как выражения. К примеру, грамматическое правило (или «продукция») вроде *E -> E * E* означает, что выражение может быть получено соединением любых двух подвыражений с помощью знака умножения. Это типичное правило построения выражений в современных языках программирования. Далее будут определены «контекстно-свободные грамматики».

2. **Регулярные выражения** также задают структуры данных, в частности, текстовых цепочек. Как будет показано далее, шаблоны описываемых ими цепочек представляют собой то же самое, что и КА. Стиль регулярных выражений в значительной степени отличается от стиля, который используется в грамматиках. Например, регулярное выражение в Unix-стиле – [A-Z][a-z]*[ ][A-Z][A-Z][A-Z] – представляет собой множество слов, начинающихся с прописной английской литеры, за которыми следуют пробел и три прописные литеры. В тексте такое выражение задает шаблоны, которые могут быть названиями городов и государств, например, Moscow RUS (Москва, Россия) или Prague CZE (Прага, Чехия). Очевидно, что мы не учли случай, когда название города состоит из нескольких слов, скажем, Nizhny Novgorod RUS (т.е. Нижний Новгород, Россия). Измененное регулярное выражение окажется более сложным ([A-Z][a-z]*[ ])*[ ][A-Z][A-Z][A-Z]. Для правильной интерпретации подобных выражений достаточно иметь сведения о том, что [A-Z] означает любую прописную литеру английского алфавита, а [ ] означает единственный пробел. Кроме того, символ * трактуется, как «любое число вхождений предшествующего элемента выражения». Круглые скобки имеют тот же смысл, что и в арифметических выражениях и используются для группировки элементов.

Автоматы и сложность

Автоматы являются неплохим инструментом исследования пределов вычислимости. Выше мы упоминали две связанные с этим проблемы (СЛАЙД 6).

1. Что может вычислительная машина? Это **проблема разрешимости**, а задачи, которые могут быть решены на такой машине, называются **разрешимыми**.

2. Что вычислитель может делать эффективно? Это **проблема труднорешаемости** задач. Если на решение какой-то из них компьютеру требуется время, зависящее от размера входных данных как некая медленно растущая функция, то задача называется **легкоразрешимой**. Медленно растущими функциями чаще всего являются полиномиальные, а функции, растущие быстрее, чем полином, считаются растущими слишком быстро.

Оба вопроса будут обсуждаться в дальнейшем.

1.2. Основные понятия теории автоматов

Алфавитом называется конечное непустое множество символов. Мы будем обозначать алфавиты символом **Σ**. Среди алфавитов известными являются двоичные и десятичные числа, прописные и строчные английские литеры, множества ASCII-символов и печатаемых ASCII-символов. См. также СЛАЙД 7.

Цепочки

Цепочка (**слово**, **строка**) – это конечная последовательность символов некоторого алфавита. В частности 01011 – это цепочка в двоичном алфавите. Цепочка 00 также является цепочкой в этом алфавите. Произвольную цепочку мы будем обозначать символом *w*.

Пустая цепочка – это цепочка, не содержащая ни одного символа. Ее принято обозначать как ε и можно рассматривать как цепочку в любом алфавите.

Цепочки можно классифицировать по **длине**, т.е. по числу позиций символов в строке. Например, цепочка 01011 имеет длину 5. Нередко дают ошибочное определение длины как число символов в строке, но в нашей строке (01011) всего два символа, тогда как число **позиций** в ней – 5, поэтому-то длина цепочка и равна 5. Если в специальной литературе мы видим термин «число символов», то следует иметь в виду, что подразумевается «число позиций».

Длину некоторой цепочки *w* принято обозначать $|w|$. К примеру, $|1011| = 4$, а $|\varepsilon| = 0$. СЛАЙД 8.

Если $\mathbf{\Sigma}$ – некоторый алфавит, то множество всех строк определенной длины, состоящих из символов алфавита, выражается с использованием знака **степени**. Определим $\mathbf{\Sigma}^k$ как множество всех строк длины *k*, состоящих из символов алфавита $\mathbf{\Sigma}$, при $k \geq 0$.

Независимо от алфавита, $\mathbf{\Sigma}^0 = \{\varepsilon\}$. Иными словами, ε – это единственная цепочка длины 0.

Если $\mathbf{\Sigma} = \{0, 1\}$, то $\mathbf{\Sigma}^1 = \{0,1\}$, $\mathbf{\Sigma}^2 = \{00,01,10,11\}$ и т.д. СЛАЙД 9.

Для большей ясности отметим различие между множествами $\mathbf{\Sigma}$ и $\mathbf{\Sigma}^1$. Причина различия в том, что первое множество – это алфавит, и его элементы 0 и 1 являются символами, а второе множество – это набор строк, и его элементы – это цепочки 0 и 1, длина каждой из которых равна 1. Далее мы будем полагать, что из контекста понятно, является ли {0,1} или подобное ему множество алфавитом или множеством цепочек.

С тем чтобы понять, элементы какого типа рассматриваются в конкретном случае, примем соглашение о том, что строчными буквами из начальной части английского алфавита обозначаются символы, а из конца алфавита – цепочки.

Множество всех цепочек над алфавитом $\mathbf{\Sigma}$ принято обозначать $\mathbf{\Sigma}^*$. Например, $\{0, 1\}^* = \{\varepsilon,0,1,00,01,10,11, 000, 001, ...\}$, или то же самое, но другим способом:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$$

Иногда пустую строку нужно исключить. Через $\mathbf{\Sigma}^+$ принято обозначать множество всех непустых строк в алфавите $\mathbf{\Sigma}$. В таком случае справедливы следующие равенства:

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$
- $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. См. СЛАЙД 10.

Теперь мы можем определить операцию соединения цепочек. Пусть *x* и *y* – строки в некотором алфавите, тогда *xy* обозначает их **конкатенацию** (соединение). Результатом является строка, в которой последовательно записаны цепочки *x* и *y*. Или более строго: если *x* – строка из *i* символов ($x = a_1a_2...a_i$), а *y* – строка из *j* символов ($y = b_1b_2...b_j$), то *xy* – это строка длины $i+j$ ($xy = a_1a_2...a_ib_1b_2...b_j$). СЛАЙД 11.

Например, если $x = 01011$ и $y = 101$, то $xy = 01011101$, а $yx = 10101011$.

Для любой строки *w* справедливы равенства $\varepsilon w = w\varepsilon = w$, поэтому говорят, что ε является **единицей (или нейтральным элементом) относительно операции конкатенации**, поскольку результат ее конкатенации с любой цепочкой дает ту же самую цепочку (аналогично арифметический 0 является нейтральным элементом относительно операции сложения).

Языки

Множество строк, каждая из которых принадлежит $\mathbf{\Sigma}^*$, где $\mathbf{\Sigma}$ – некоторый алфавит,

называется **формальным языком** (далее просто **языком**). Если **Σ** – алфавит, и $L \subseteq \Sigma^*$, то L – это **язык над Σ** (или **язык в Σ**). Язык в алфавите не обязан содержать строки, в которые входят все символы алфавита, поэтому, если известно, что L является языком в **Σ**, то L – это язык над любым алфавитом, содержащим **Σ**. СЛАЙД 12.

Термин язык применительно к формальным системам на первый взгляд кажется странным, однако и естественные языки содержат множества цепочек. Например, русский язык мы можем рассматривать как набор всех литературных русских слов, составленных из символов кириллицы. Аналогично, язык программирования С, как и любой другой ЯП, также состоит из программ, представляющих собой подмножества множества всех возможных цепочек, которые, в свою очередь, состоят из символов алфавита конкретного языка. Этот алфавит может быть подмножеством всех ASCII-символов. Алфавиты разных ЯП могут различаться, хотя есть и общие элементы: прописные и строчные литеры, цифры, знаки пунктуации и символы операций.

В теории автоматов, однако, изучаются и другие языки. Примеры (СЛАЙД 13):

1. $\{\varepsilon, 01, 0011, 000111, ...\}$.
2. $\{\varepsilon, 01, 10, 0011, 1100, 1001, ...\}$.
3. $\{01, 11, 101, 111, 1011, ...\}$.
4. $\mathbf{\Sigma}^*$.
5. $\varnothing$.
6. $\{\varepsilon\}$.

Единственное существенное для множеств-языков ограничение состоит в том, что алфавиты должны быть конечны, хотя сами языки могут содержать бесконечное число цепочек.

Проблемы

В теории автоматов под **проблемой** понимается вопрос о том, является ли некоторая строка элементом некоторого конкретного языка. Далее мы покажем, что все, называемое проблемой в более широком смысле, может быть выражено в виде проблемы принадлежности некоторому языку. Более формально: если **Σ** – алфавит, и L язык в **Σ**, то проблема L выглядит следующим образом (СЛАЙД 14).

- Дана строка w из $\mathbf{\Sigma}^*$, требуется выяснить, принадлежит ли w языку L или нет.

Пример 3.

Задачу проверки заданного числа на простоту можно выразить в терминах принадлежности языку L_p, состоящего из двоичных строк, выражающих простые числа. Тогда, ответ «принадлежит» соответствует ситуации, когда строка из 0 и 1 является двоичным представлением простого числа. В противном случае – ответ «не принадлежит». Для некоторых строк принять решение несложно. Скажем, цепочка 001101 не может представлять простое число. К сожалению, решение данной проблемы для 1101 не так очевидно. Более того, оно может потребовать значительных затрат вычислительных ресурсов.

Способ определения языков с помощью множеств

Языки часто задаются с помощью конструкций, принятых для описания множеств.

$\{w \mid$ сведения о $w\}$

Это выражение «расшифровывается» как «множество слов w, соответствующих тому, что сказано справа от вертикальной черты». Примеры (СЛАЙД 15):

1. $\{w \mid w$ содержит одинаковое число 0 и 1$\}$.
2. $\{w \mid w$ является двоичным представлением простого числа$\}$.
3. $\{w \mid w$ является синтаксически правильной программой на языке программирования высокого уровня$\}$.

Часто вместо w записывается параметризованное выражение, а также описывают строки языка с помощью ограничений на параметры. Примеры:

1. $\{0^n1^n \mid n \geq 1\}$. Этот язык содержит строки 01, 0011, 000111 и т.д., и не содержит строк 00, 111, 10, 011 и т.д. Как видно, здесь, так же как и для алфавита, используется обозначение кратности (степени). Мы можем определить n-ую степень одиночного символа, как строку из n экземпляров данного символа.

2. $\{0^n1^m \mid 0 \leq n \leq m\}$.

Теперь мы можем вернуться к определению проблемы, данному выше. На практике под проблемами понимаются не вопросы разрешения («принадлежит или не принадлежит»), а запросы на обработку или преобразование некоторого набора входных данных (как правило, наилучшим способом). К примеру, задача синтаксического анализатора в компиляторе языка С – определить, принадлежит ли данная строка ASCII-символов множеству L_C всех корректных программ на этом языке, нашему определению соответствует в полной мере. Однако в задачи анализатора входит еще формирование древовидного представления программы, формирование таблицы имен, обработка ошибок и другие возможные действия. Сам компилятор в целом решает задачу **перевода** программы в объектный код для целевой вычислительной машины, а ответ на вопрос о правильности такой программы весьма далек от простого «принадлежит» или «не принадлежит». СЛАЙД 16.

Тем не менее, определение проблем как языков со временем значительных изменений не претерпело и позволяет решать ряд задач теории вычислительной сложности. В рамках теории сложности разыскиваются нижние границы сложности тех или иных задач. Особенно важными являются методы доказательства того, что определенные задачи не могут быть решены за экспоненциальное время (т.е. количественно меньшее, чем экспонента от размера водных данных). Как показала практика, задача в терминах теории языков (с ответом на вопрос принадлежности) так же сложна, как и задача, требующая «найти решение».

Итак, если мы докажем трудность определения принадлежности заданной строки множеству L_X всех корректных программ на языке X, то перевод программы с этого языка в целевой код, по крайней мере, не легче. Если бы код легко генерировался, то мы попросту запустили транслятор и после успешного завершения его работы могли бы утверждать, что входная строка является правильной программой на языке X. Последний этап определения, создан или нет объектный код, не является сложным. Значит, с помощью быстрого алгоритма кодогенерации мы могли бы решать задачу о принадлежности строки множеству L_X. Однако в таком случае мы придем к противоречию с предположением о том, что ответить на вопрос принадлежности строки языку трудно. Используя известную методику доказательства от противного, мы добились справедливости утверждения «если проверка принадлежности языку трудна, то компиляция программ на этом языке, также трудна».

Попутно мы продемонстрировали еще одну полезную методику. Мы показали трудность (сложность) одной задачи с использованием гипотетически эффективного алгоритма ее решения для решения другой заведомо сложной задачи. Этот метод называется **сведением одной задачи к другой**. Это довольно мощный инструмент, который можно использовать при исследованиях сложности проблем. Его применение на практике упрощается еще и в связи с нашим замечанием относительно того, что проблемами являются только вопросы принадлежности некоторому языку.

Более того, язык и проблема – это одно и то же, и употреблять термины можно в зависимости от нашего угла зрения. Если мы интересуемся строками как таковыми, то в множестве цепочек мы будем видеть некоторый язык. В завершающих разделах лекционного курса нас будет больше интересовать смысл строк, т.е. не цепочки символов, а те объекты, которые они представляют. В этом случае мы будем рассматривать строки как некоторую проблему.

Литература к лекции 1

1. Машина Тьюринга// Лекция Александра Шеня в проекте ПостНаука (06.04.2013) – Видео - http://postnauka.ru/video/10777
2. Кук, Д. Компьютерная математика / Д. Кук, Г.Бейз. – М.: Наука, 1990. – 384 с.
3. Молчанов, А. Ю. Системное программное обеспечение. 3-е изд. / А.Ю. Молчанов. – СПб.: Питер, 2010. – 400 с.
4. Stephen A. Cook: The Complexity of Theorem-Proving Procedures - http://4mhz.de/cook.html
5. Л. А. Левин Универсальные задачи перебора // Проблемы передачи информации. — 1973. — Т. 9. — № 3. — С. 115—116.
6. Теорема Кука-Левина – http://ru.wikipedia.org/wiki/Теорема_Кука
7. Кормен, Т.Х. Алгоритмы. Построение и анализ / Т.Х. Кормен, Ч.И. Лейзерсон, Р.Л. Ривест, К.Штайн. – М:. Вильямс, 2012. – 1296 с.
8. Aho, A.V. Foundations of Computer Science: C Edition (Principles of Computer Science Series) / A.V. Aho, J.D. Ullman. - New York: Computer Science Press, 1994. – 786 p.
9. Закон Мура – http://cs.usu.edu.ru/study/moore/