Лекция 4. Регулярные выражения

4.1 Введение 1
4.2 Операторы регулярных выражений.....1
4.3 Построение регулярных выражений.....2
4.4 Приоритеты операторов регулярных выражений.....3
4.5 Конечные автоматы и регулярные выражения.....4
Литература к лекции 4.....8

Главные вопросы, которые мы обсуждаем, представлены на СЛАЙДЕ 1. Рассмотрим еще один способ определения языков – регулярные выражения. Они тесно связаны с НКА и могут стать альтернативным способом описания программных компонентов. Также мы покажем, что регулярные выражения могут описывать только регулярные языки.

4.1 Введение

Мы переходим от машинного задания языков с помощью ДКА и НКА к алгебраическому описанию языков с помощью регулярных выражений (РВ). Установим, что РВ определяют те же языки, что и различные типы КА, а именно регулярные языки (РЯ). В отличие от автоматов РВ позволяют определять допустимые строки декларативным способом, поэтому РВ используются в качестве входного языка во многих системах, обрабатывающих цепочки (СЛАЙД 2).

В UNIX-подобных системах имеются команды *grep*/*egrep* или аналогичные им для поиска строк. В них РВ предназначены для описания шаблонов, которые пользователь ищет, например, в файле. Различные поисковые системы преобразуют РВ в ДКА или НКА и применяют этот автомат к содержимому файла.

Существуют специальные генераторы лексических анализаторов, например, *flex* в *Linux*. Как известно, лексический анализатор – это такой компонент транслятора, который разбивает текст программы на лексические единицы (**лексемы**), состоящие из одного или нескольких символов и имеют определенный смысл. Таковыми, например, являются ключевые слова, идентификаторы, знаки операций и т.д. Генератор лексических анализатор получает формальные описания лексем (по сути – РВ) и создает КА, который распознает, какая из лексем появляется на его входе.

Если V – алфавит, то под РВ понимается:
- любой элемент из V;
- пустая строка (ε);
- комбинация РВ, полученная посредством операторов.

4.2 Операторы регулярных выражений

С помощью РВ обозначаются языки. Рассмотрим в качестве примера простое РВ 01*+10*. Оно определяет язык всех строк, состоящих из одного нуля, за которым следует произвольное количество 1, либо из одной единицы, за которой следует произвольное количество 0. Чтобы понять, почему мы правильно интерпретируем заданное РВ, необходимо определить все использованные в этом РВ символы, а для этого покажем три операции над РЯ, которые соответствуют операторам РВ (СЛАЙД 3).

1. **Объединение** языков L и M, которое обозначается через $L \cup M$ – это множество строк, которые содержатся либо в L, либо в M, либо в обоих языках. Например, если $L = \{001, 0, 11\}$ и $M = \{\varepsilon, 11\}$, то их объединение дает $\{\varepsilon, 11, 001, 0\}$.

2. **Конкатенация** языков L и M – это множество строк, образуемых дописыванием к любой строке из L любой строки из M, подобно тому, как мы выполняем конкатенацию двух строк. Конкатенацию языков обозначают либо точкой, либо никак специальным образом не обозначают. Например, если $L = \{001, 0, 11\}$ и $M = \{\varepsilon, 11\}$, то их конкатенация $LM = \{001, 00111, 0, 011, 11, 1111\}$. Первые три строки в LM – это строки из L,

соединенные с пустой строкой. Она, как мы отмечали ранее, является единицей для операции конкатенации. Значит, результатом будут такие же, как и строки из L. Последние три строки в LM образованы конкатенацией каждой строки из L с второй строкой из M.

3. **Итерация** (звездочка, замыкание Клини) языка L обозначается как L^* и представляет собой множество всех тех строк, которые можно образовать конкатенацией любого количества строк из L. При этом допустимы повторения, т.е. одна и та же строка из L может быть выбрана для конкатенации более одного раза. Например, если $L = \{0, 1\}$ то L^* образуют все строки из 0 и 1. Если $L = \{0, 11\}$ то в L^* входят строки из 0 и 1, содержащие четное количество единиц (011, 11110 или ε). Не входят в этот язык такие строки, как 01011 или 101. Более точно язык L^* можно представить как бесконечное объединение $\bigcup_{i\geq 0} L^i$. Получается, что $L^0=\{\varepsilon\}$, $L^1=L$, $L^2=LL$, ..., $L^i=LL...L$ (конкатенация i копий языка L).

Пример 25.

Рассмотрим три конкретных языка. Возьмем для начала язык $L = \{1, 00\}$.

$L^0 = \{\varepsilon\}$ независимо от языка. Дело в том, что нулевая степень означает выборку нулевого количества строк из языка. $L^1 = L$, это означает выборку одной строки из языка. Получаем первые два элемента в разложении множества $L^* = \{\varepsilon, 1, 00\}$.

Далее рассматриваем L^2. Выбираем две строки из L, но т.к. мы условились, что можно выбирать с повторениями, то получаем четыре варианта. В итоге $L^2 = \{11, 100, 001, 0000\}$. По аналогии получаем третью степень, $L^3 = \{111, 1100, 1001, 0011, 1000000100, 00001, 000000\}$.

Для вычисления L^* вычисляем L^i для всех i и объединяем эти языки. Язык L^i содержит 2^i элементов. Хотя наше множество L^i конечно, однако объединение бесконечного числа таких множеств дает, как правило, бесконечный язык. Это справедливо для нашего примера.

Продолжая наши исследования, мы подошли к языку L, который представляет собой множество строк, состоящих из нулей. Этот язык бесконечен. Тем не менее, достаточно легко увидеть, что представляет собой итерация L^*. $L^0 = \{\varepsilon\}$, а $L^1 = L$, и L^2 – это набор строк, которые можно образовать путем объединения одной строки из 0 с другой строкой из 0. Проще говоря, любую строку можно записать как конкатенацию двух строк из 0. Значит, $L^2 = L$, и аналогично $L^3 = L$ и так далее. В итоге, бесконечное объединение $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$ совпадает с L в том случае, когда сам язык L является множеством всех строк из 0.

Наконец, рассмотрим еще один простой пример $\varnothing^* = \{\varepsilon\}$. Очевидно, что $\varnothing^0 = \{\varepsilon\}$, тогда как $\forall i \geq 1 | \varnothing^i = \varnothing$. Мы получили один из двух языков, итерация которых **не является бесконечным множеством**.

4.3 Построение регулярных выражений

Все алгебры начинаются с некоторых элементарных выражений. Как правило, это переменные и/или константы. Применяя определенный набор операций к этим элементарным выражениям и уже построенным их комбинациям, можно конструировать более сложные выражения. Необходимо также иметь некоторые методы группирования операция и операндов, обычно с помощью скобок. Например, алгебра арифметических выражений начинается с целых и вещественных констант и переменных, а также позволяет конструировать более сложные выражения с помощью таких операций как «сложение» или «умножение».

Похожим образом строится алгебра РВ: используются константы и переменные для обозначения языков и знаки для обозначения трех операторов – объединения, конкатенации и итерации. РВ можно определить рекурсивно, не только характеризуя правильные РВ, но и для каждого РВ E описывая представленный ими язык $L(E)$.

СЛАЙД 4.

Базис состоит из трех частей:

1. Константы ε и $\varnothing$ являются РВ, определяющими языки $\{\varepsilon\}$ и $\varnothing$ соответственно. Строго говоря, $L(\varepsilon)=\varepsilon$ и $L(\varnothing)=\varnothing$.

2. Если a – произвольный символ, то a – РВ, определяющее язык $\{a\}$, т.е. $L(a)=a$. Так мы записываем выражения, соответствующие символу.

3. Переменная, записываемая заглавным курсивным символом, представляет собой произвольный язык.

Индуктивный шаг состоит из 4 частей, по одной из трех операторов и для введения скобок.

1. Если E и F – РВ, то $E + F$ – РВ, определяющее объединение языков $L(E+F)=L(E)\cup L(F)$.

2. Если E и F – РВ, то EF – РВ, определяющее конкатенацию языков $L(EF)=L(E)L(F)$. Например, РВ 01 представляет язык $\{01\}$.

3. Если E – РВ, то E^* – РВ, определяющее итерацию языка $L(E)$, или, то же самое, $L(E^*)=L(L(E))^*$.

4. Если E – РВ, то (E) – РВ, определяющее тот же язык $L(E)$, что и выражение E. Более строго, $L((E))=L(E)$.

Пример 26.

Сконструируем РВ для множества строк из чередующихся символов 0 и 1. Сначала строим язык из единственной строки 01. Затем, используя звездочку Клини, построим РВ из строк вида 010101..01.

Согласно 2 части базисного правила, 0 и 1 – это РВ, обозначающие языки {0} и {1}. Соединив эти РВ, мы получим РВ 01 для языка {01}. Иначе говоря, если мы хотим написать РВ для языка из одной строки w, то саму ее и используем как РВ.

Далее, для получения всех строк из 0 или более вхождений 01, используем РВ $(01)^*$. Это выражение по понятным причинам отличается от РВ 01^*. Однако язык $L((01)^*)$, очевидно, еще не тот язык, который нам нужен. Он включает те строки, которые начинаются с 0 и заканчиваются 1. Нам хотелось бы получить возможность указывать вначале 1 и/или в конце 0. Возможно несколько решений. Опишем два из них.

В первом случае строятся еще три РВ, описывающие три других возможности. Получим РВ $(01)^* + (10)^* + 0(10)^* + 1(01)^*$. С помощью «+» мы обеспечили объединение четырех языков, которые совокупно дают все строки из чередующихся 0 и 1.

Во втором случае мы тоже начинаем с РВ 01^*. Можно добавить необязательную единицу в начале, если слева к этому выражения мы допишем выражение ε+1. Аналогично, добавим необязательный 0 в конце с помощью конкатенации с РВ ε+0. Используя свойства операции альтернативы получим язык $L(\varepsilon+1)=L(\varepsilon)\cup L(1)=\{\varepsilon\}\{1\}=\{\varepsilon,1\}$.

Если мы допишем к этому языку любой другой язык L, то выбор пустой строки даст нам строки из L, а выбрав символ 1, мы получим $1w$ для каждой строки w из L. Тогда набор строк из чередующихся 0 и 1 можно представить следующим выражением $(\varepsilon+1)(01)^*(\varepsilon+0)$. Мы заключаем подвыражения в скобки, чтобы обеспечить правильную группировку операторов.

4.4 Приоритеты операторов регулярных выражений

Приоритетность операторов означает, что они связываются с операндами в определенном порядке. Нам хорошо знакомо это понятие для арифметических выражений. Для операторов РВ определен следующий понижающий порядок приоритетов.

1. **Итерация**. Этот оператор применяется только к наименьшей последовательности

символов, находящихся слева от него и являющихся правильным РВ.

2. **Конкатенация**. Связав все итерации с их операндами, дальше со своими операндами связываются операторы конкатенации. Иначе говоря, все **смежные** РВ группируются вместе. Конкатенация является ассоциативным оператором, поэтому не важно, в каком порядке группируются последовательные конкатенации. Если потребуется сделать выбор, то нужно группировать их слева направо. Так 210 группируется как (21)0.

3. Дальше с операндами связывается **объединение** (или альтернатива). Оно тоже ассоциативно, то и здесь порядок не имеет значения. Дальше мы, тем не менее, будем группировать выражения от левого края.

Когда нежелательно, чтобы группирование РВ определялось только приоритетами. На помощь приходят скобки, которые можно использовать даже в том случае, когда правильный порядок обеспечивается правилами приоритетности.

Пример 27.

РВ 01^*+1 группируется как $(0(1^*))+1$. Сначала выполняется итерация. Поскольку символ 1, находящийся слева от операнда является допустимым РВ, то он будет операндом итерации. Дальше группируется конкатенация 0 и (1^*). Получаем выражение $(0(1^*))$. В последнюю очередь объединяем данное РВ с РВ, находящимся справа от оператора объединения (у нас – 1).

Полученным РВ мы описываем язык, который содержит строку из одного символа 1, плюс все строки, начинающиеся с 0, за которыми идет любое количество 1.

4.5 Конечные автоматы и регулярные выражения

Как видно, описание языков с помощью РВ принципиально отличается от конечно-автоматного. Однако обе нотации представляют одно и то же множество языков, которые мы ранее назвали регулярными. Выше мы показывали, что ДКА и оба вида НКА допускают один и тот же класс языков. Для доказательства того, что этот же класс можно описывать с помощью РВ, мы должны доказать следующее.

1. Любой язык, задаваемый одним из этих автоматов, может быть также определен РВ. Далее мы предполагаем, что язык допускается некоторым НКА.

2. Любой язык, определяемый РВ, можно задать с помощью КА. Здесь для простоты мы предположим, что существует ε-НКА, допускающий тот же язык.

На СЛАЙДЕ 5 показаны все классы эквивалентности, которые уже доказаны или будут доказаны. Дуга от класса *K1* к классу *K2* означает, что каждый язык определяемый классом *K1*, определяется также классом *K2*. Этот граф является сильно связным. Значит, все 4 класса на самом деле эквивалентны.

Преобразование РВ в КА

Поскольку РВ по мощности эквивалентны КА, то можем преобразовывать их друг в друга. Покажем такое преобразование на примере конвертации РВ в НКА. Сначала создадим РВ $(q+a)+b^*+cd$ в системе JFLAP, а затем начнем конвертацию путем выбора пункт *Convert*: *Convert to NFA*. Мы получим начальный обобщенный граф переходов (*Generalized transition graph*, далее – GTG).

GTG представляет собой расширение НКА, которое позволяет помечать переходы выражениями, а не только одиночными символами (назовем их **РВ-переходами**). В GTG состояние может меняться при переходе по РВ *R*, если непрочитанная часть его входа начинается со строки *s*, принадлежащей *R*. Это состояние, когда строка *s* уже прочитана. У нас получился GTG с двумя состояниями и одним РВ-переходом от начального к заключительному состоянию. Основная идея конвертации заключается в том, что мы заменяем один переход новыми состояниями, соединенными другими переходами по операндам соответствующего РВ-оператора верхнего уровня.

Оператор верхнего уровня – это оператор, который в заданном выражении выполняется в последнюю очередь. Например, в РВ $ab+c$ оператором верхнего уровня

будет +, т.к. оператор конкатенации имеет более высокий приоритет, следовательно, будет выполняться раньше сложения. Затем операнды мы соединим ε-переходами, чтобы продублировать функциональность утраченного оператора. Таким образом, на каждом шаге мы имеем GTG, эквивалентный оригинальному РВ. В конце концов мы удалим все операторы и останемся с односимвольными и ε-переходами. В этой точке наш GTG можно будет рассматривать как правильный НКА.

Далее мы продолжаем преобразование путем нажатия на пиктограмму *De-expressionify Transition tool* и последующего щелчка на переход, помеченный исходным РВ. Мы видим, что этот переход расщепляется согласно логике оператора верхнего уровня (у нас – объединение), а операнды, которые были объединены, получили теперь собственные переходы. Использованный инструмент определяет оператор верхнего уровня для РВ, и затем размещает его операнды в новые РВ-переходы.

Также видно, что JFLAP сообщает «De-oring (q+a)+b*+cd» и «6 more ε-transitions needed». Эти сообщения подают идею о возможных дальнейших шагах.

Мы можем добавить ε-переходы вручную, по одному, и каждый раз будет изменяться количество оставшихся переходов. Результат очевиден. Таким способом мы сможем приблизиться к функциональности оператора объединения. При ручном добавлении возможны только ε-переходы, поэтому JFLAP не спрашивает нас о символе перехода, а сам автоматически его ставит. Мы можем расставить все ε-переходы автоматически, нажав на пиктограмму *Do Step*.

После того, как мы заменили один (начальный) РВ-переход на три РВ-перехода, мы можем продолжать выполнение конвертации. В нашем случае это действие, обратное конкатенации (или «деконкатенация»). С помощью все того же *De-expressionify Transition tool* мы осуществляем деконкатенацию подвыражения *cd*. По ходу выполнения замечаем, что здесь тоже понадобится добавление ε-переходов (от q_6 к q_8, от q_9 к q_{10}, от q_{11} к q_7). Только здесь они нужны для замены удаляемого оператора конкатенации, а не объединения, как в предыдущем случае. Результат – очевиден. Конфигурация в состоянии q_6 соответствует ситуации перед чтением символа (а также простого РВ) *c*, между q_8 и q_9, и аналогично – для *d*. Мы получили функциональный эквивалент конкатенации *c* и *d*.

Давайте попробуем добавить ε-переход от состояния q_6 к q_{10}, что будет означать переход из состояния перед прочтением символа *d* с пропуском символа *c*. Это, разумеется, неправильно, поэтому JFLAP выдаст сообщение об ошибке «A transition there is invalid», и требуемый переход создан не будет.

Проверка корректности переходов универсальна в том смысле, что конвертеру безразлично, какой оператор мы расщепляем. Однако у процесса деконкатенации имеются дополнительные ограничения. Скажем, если мы сначала попробуем создать вполне корректный переход от q_{11} к q_7 до двух других переходов, то будет получено сообщение «That may be correct, but the transitions must be connected in order», указывая на то, что такие переходы следует создавать в правильном порядке.

Далее можно работать с другими РВ-переходами. Инструментом *De-expressionify Transition tool* мы должны добиться того, что на дугах останутся одиночный символ либо ε. Если мы «кликнем» на какой-либо из них, будет получено сообщение «That's as good as it gets», указывая на то, что упрощение невозможно.

Двигаясь в направлении звездочки Клини, мы подобрались к «де-итерации», т.е. к упрощению оператора итерации. Пробуем что-нибудь сделать с РВ-переходом b^*, используя все тот же *De-expressionify Transition tool*. У звездочки Клини только один операнд – символ *b*, он расщепляется на новую порцию НКА. JFLAP указывает на то, что мы выполняем де-итерацию, и что нужно создать 4 спонтанных перехода.

Соединяем ε-переходами состояния q_4 к q_{12}, от q_{13} к q_5 (это позволит прочитать *b*), от q_4 к q_5 (это позволит прочитать 0 вхождений *b*), от q_5 к q_4 (это позволит повторить чтение *b*).

У нас остался несовместимым с НКА только один РВ-переход, содержащий

подвыражение в скобках – (q+a). Эти скобки можно рассматривать как оператор верхнего уровня, т.к. они указывают, что сначала должно быть вычислено их содержимое, и только потом могут быть продолжены другие вычисления.

Тем не менее, когда они окружают полное РВ, то в них необходимости нет.

Используя *De-expressionify Transition tool* над этим РВ-переходом, мы увидим, как круглые скобки исчезают, и остается только q+a. Как поступают при устранении оператора объединения, мы видели выше. Добавляем все ε-переходы, и JFLAP сообщает нам, что преобразование закончено, т.к. мы получили GFG, являющийся правильным НКА, и его можно затем экспортировать в отдельное окно.

Автоматизация преобразования РВ в КА

Преобразование может быть выполнено в автоматическом режиме. Вернемся к нашему исходному РВ. Снова выбираем пункт *Convert* : *Convert to NFA*. Затем нажимаем *Do Step*. Шаг конвертации заключается в преобразовании одиночного РВ-перехода. У нас он один, исходное РВ упрощается, и все ε-переходы добавляются без нашего участия.

Второй возможный вариант – нажатие кнопки *Do All*. Если в РВ не было ошибок, то результирующий НКА будет получен немедленно.

Алгоритм конвертации РВ в НКА

Шаг 1. Стартуем от РВ R.

Шаг 2. Создаем обобщенный граф переходов G с единственным начальным состоянием q_0, единственным конечным состоянием q_1 и единственным переходом между ними, помеченным исходным РВ R.

Шаг 3. Хотя существует некоторый переход t, принадлежащий G, из состояния q_i в состояние q_j, и помеченный выражением S, которое состоит более чем из одного символа. Пусть φ – это оператор верхнего уровня для выражения S, и пусть $[\alpha_1, \alpha_2, ..., \alpha_\psi]$ – это упорядоченный список операндов оператора φ (т.к. скобки и звездочка Клини имеют один операнд, то в этом случае $\psi = 1$).

а) Если φ – это круглые скобки, заменяем t на РВ-переход по α_1 из состояния q_i в состояние q_j.

б) Если φ – это оператор итерации, то создаем в G два новых состояния q_x и q_y, а также переход по α_1 между ними, и затем четыре ε-перехода: из q_i в состояние q_x, из q_y в q_j, из q_i в q_j, из q_j в q_i.

в) Если φ – это оператор объединения, то удаляем t, и для всех k от 1 до ψ выполняем следующее: создаем два новых состояния q_{xk} и q_{yk}; создаем переход по α_k между q_{xk} и q_{yk}; затем два ε-перехода: из q_i в состояние q_{xk}, из q_{yk} в q_j.

г) Если φ – это оператор конкатенации, то удаляем t, и для всех k от 1 до ψ выполняем следующее: создаем два новых состояния q_{xk} и q_{yk}; создаем переход по α_k между q_{xk} и q_{yk}; если $\psi > 0$ создаем ε-переход из q_{yk-1} в состояние q_{xk}. Наконец, создаем два ε-перехода: из q_i в состояние q_{x1}, из q_ψ в q_j.

Шаг 4. Мы получаем GTG, являющийся правильным НКА.

Преобразование КА в РВ

Такое преобразование следует простой логике, которая во многом напоминает описанную выше конвертацию КА в РВ. Оно начинается с КА, который мы рассматриваем в качестве GTG. Затем мы последовательно удаляем состояния, генерируя эквавалентный GTG. Процесс генерации заканчивается, когда останутся единственные начальное и единственное конечное состояния. JFLAP далее использует формулу для представления упрощенного GTG как РВ.

КА, который мы будем преобразовывать в РВ, представлен на экране. Алгоритм конвертации требует в первую очередь, чтобы КА трансформировался в такой GTG,

который мы упомянули выше. Кроме того, начальное и конечное состояния совпадать не должны, и должен быть только один переход между каждой парой состояний q_i и q_j, причем i может быть равно j.

В нашем КА пока имеются два конечных состояния, одно из которых по совместительству еще и начальное, поэтому нам нужно преобразовать КА к требуемому виду.

Чтобы начать конвертацию, мы выбираем пункт *Convert : Convert FA to RE*. Система сообщает нам о необходимости обеспечить единственное конечное состояние. Мы добавляем новое конечное состояние q_3, к которому следует создать два ε-перехода от прежних конечных состояний. Этот промежуточный результат очевиден.

Теперь выполним еще одно требование алгоритма – единственность переходов для любой пары состояний, т.е. не может быть более одного перехода от q_i к q_j. Рассмотрим два циклических перехода в q_1 по d и e. Мы можем удовлетворить это требование путем замены этих двух переходов на РВ-переход $d+e$, указывая на то, что переход возможно по d или e.

Выбираем инструмент *Transition Collapser tool*, который должен преобразовать все переходы от q_i и q_j в одиночные переходы, где метки удаляемых переходов разделены оператором объединения. Получен еще один промежуточный результат. Теперь наш GTG уже не является правильным КА. Описанный инструмент должен использоваться, если более одной пары состояний имеют более одного перехода.

Если некоторого перехода больше не существует, то нужно создать пустой переход (это не то же самое, что ε-переход; здесь переход по символу, обозначающему пустое множество). Создадим такой переход от q_0 и q_2.

Существенное различие между GTG и КА состоит в том, что в последнем переходы описывают единственную цепочку, а переходы в GTG описывают наборы цепочек. В нашем случае мы создаем переходы по пустому множеству строк. Это похоже на создание ε-перехода, поэтому JFLAP предупреждает нас о создании пустых переходов. Как водится, он нас информирует о том, как много еще пустых переходов требуется сделать. Всего их нужно 7, но один мы уже сделали (полный список – q_0 и q_2, q_1 и q_3, q_2 и q_0, q_3 и q_0, q_3 и q_1, q_3 и q_2, а также циклический переход в q_3). Результирующий GTG можно посмотреть в JFLAP.

Теперь у нашего обобщенного графа одно заключительное состояние, не совпадающее с начальным состоянием и в каждой паре состояний в точности один переход. На следующем шаге мы должны итеративно удалять каждое состояние в GTG, за исключением начального и конечного. По мере удаления этих состояний, мы настраиваем оставшиеся переходы так, чтобы гарантировать эквивалентность GTG на каждом шаге.

Состояния можно сворачивать в любом порядке, но мы будем делать это в заданном порядке. Выбираем инструмент *State Collapser tool*, и затем щелкаем по состоянию q_2. Будет показано окно, которое перед сворачиванием состояния проинформирует нас о новых метках переходов. Пусть r_{ij} – это РВ на переходе от q_i в q_j. По правилам, если мы удаляем q_k, то для всех состояний q_i и q_j, где $i \neq k$ и $j \neq k$, r_{ij} заменяется на $r_{ij} + r_{ik}r_{kk}^{*}r_{kj}$. Иными словами, мы компенсируем удаление состояния q_k тем, что инкапсулируем на пути от q_i в q_j эффект от перехода от q_i в q_k (т.е. r_{ik}), затем цикл по q_k (r_{kk}^{*}), и в конце – переход из q_k в q_j (т.е. r_{kj}). И отметим, что пустое множество удовлетворяет следующему отношению: если r – РВ, то $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^{*} = \varepsilon$.

Выберем в таблице строку, которая описывает циклический переход в q_1 - $d+e+ca^{*}c$. Переходы, из которых будет сформирован новый переход, подсвечиваются в GTG. У нас два пути, которые должны объединяться в РВ-переход: 1) цикл в q_1 – $d+e$; 2) из q_1 в q_1 через q_2 – $ca^{*}c$. Более формально, $r_{1,1} = d+e$, $r_{1,2} = r_{2,1} = c$, $r_{2,2} = a$. Значит, новый переход $r_{1,1} + r_{1,2}r_{2,2}^{*}r_{2,1}$, что нам собственно и демонстрирует JFLAP.

Правила для операций над пустым множеством более загадочны. Выберем строку, описывающую новый переход из q_0 в q_1. У нас два пути, которые должны объединяться в

РВ-переход: 1) цикл в $q_0 - b$; 2) из q_0 в q_1 через $q_2 - \varnothing a^* c$. Более формально, $r_{0,1} = b$, $r_{0,2} = \varnothing$, $r_{2,2} = a$, $r_{2,1} = c$. Значит, новый переход $r_{0,1} + r_{0,2} r_{2,2}^* r_{2,1} = b + \varnothing a^* c$. Конкатенация любого РВ с пустым множеством дает пустое множество, поэтому $\varnothing a^* c = \varnothing$ и $b + \varnothing a^* c = b + \varnothing$. Объединение любого РВ с пустым множеством дает то же самое РВ, поэтому получаем b – РВ на новом переходе от q_0 в q_1.

Можно пройтись по другим заменам, чтобы увидеть конкретные формулы для РВ для размещения на переходах. Затем мы можем нажать *Finalize*. Будут произведены все замены, а состояние q_2 – удалено. Тот же процесс мы произведем над q_1. Всего – 4 замены, но некоторые метки достаточно длинные. Когда будем удалено q_1, мы увидим более красивую картинку.

В этот момент у нашего GTG есть два состояния – одно начальное и одно заключительное.

Пусть r_{xy} – это РВ на переходе от q_x к q_y. Для GTG в такой форме, где q_i – это начальное состояние, а qj – конечное, эквивалентным РВ является выражение, заданное уравнением 4.1.

$$r = (r_{ii}^* r_{ij} r_{jj}^* r_{ji})^* r_{ii}^* r_{ij} r_{jj}^* \qquad (4.1)$$

Наше преобразование закончено, и JFLAP отображает РВ 4.2, полученное с учетом (4.1)

$$(a + b(d + e + ca^* c)^* b)^* (\varepsilon + b(d + e + ca^* c)^* ca^*) \qquad (4.2)$$

Это РВ можно экспортировать в отдельное окно.

Алгоритм конвертации КА в РВ

Шаг 1. Стартуем от КА, который рассматриваем как обобщенный граф G.

Шаг 2. Пусть F – это множество заключительных состояний G, а q_0 – начальное состояние. Если $|F| > 1$ или $F = \{q_0\}$, то создаем новое состояние q_f , производим все ε-переходы для каждого q_i из F от q_i к q_f, и делаем q_f – единственным конечным состоянием.

Шаг 3. Пусть S – это множество всех состояний G. Для каждой пары (q_i, q_j) из SxS пусть $L = \{l_1, l_2, ..., l_n\}$ будет множеством всех РВ на переходах от q_i к q_j. Пусть $e = \varnothing$, если $|L| = 0$, и $e = l_1 + l_2 + ... + l_n$, в противном случае. Заменяем все переходы от q_i к q_j с единственным переходом между ними по выражению e.

Шаг 4. Пусть T – это множество всех нестартовых и незаключительных состояний G. Пусть r_{xy} – это выражение на переходе от q_x к q_y. Для каждого q_k из T и каждой пары (q_i, q_j) из $(T\text{-}\{q_k\})$x$(T\text{-}\{q_k\})$ заменяем выражение r_{ij} на $r_{ij} + r_{ik} r_{kk}^* r_{kj}$ и удаляем q_k из G.

Шаг 5. У G теперь два требуемых состояния, а эквивалентное РВ теперь $r = (r_{00}^* r_{0f} r_{ffj}^* r_{f0})^* r_{00}^* r_{0f} r_{ff}^*$.

Литература к лекции 4

1. Гилл, А. Введение в теорию конечных автоматов / А. Гилл. – М.: Наука, 1966. – 272 с.
2. Кузнецов, А.С. Теория вычислительных процессов [Текст] : учеб. пособие / А. С. Кузнецов, М. А. Русаков, Р. Ю. Царев ; Сиб. федерал. ун-т. - Красноярск: ИПК СФУ, 2008. – 184 с.
3. Короткова, М.А. Математическая теория автоматов. Учебное пособие / М.А. Короткова. – М.: МИФИ, 2008. – 116 с.
4. Молчанов, А. Ю. Системное программное обеспечение. 3-е изд. / А.Ю. Молчанов. – СПб.: Питер, 2010. – 400 с.
5. Теория автоматов / Э. А. Якубайтис, В. О. Васюкевич, А. Ю. Гобземис, Н. Е. Зазнова, А. А. Курмит, А. А. Лоренц, А. Ф. Петренко, В. П. Чапенко // Теория вероятностей. Математическая статистика. Теоретическая кибернетика. — М.: ВИНИТИ, 1976. — Т. 13. — С. 109–188. — URL http://www.mathnet.ru/php/getFT.phtml?

jrnid=intv&paperid=28&what=fullt&option_lang=rus

6. Серебряков В. А., Галочкин М. П., Гончар Д. Р., Фуругян М. Г. Теория и реализация языков программирования — М.: МЗ-Пресс, 2006 г., 2-е изд. - http://trpl7.ru/t-books/TRYAP_BOOK_Details.htm
7. Finite State Machine Generator - http://sourceforge.net/projects/genfsm/
8. Введение в схемы, автоматы и алгоритмы - http://www.intuit.ru/studies/courses/1030/205/info