Статья опубликована в журнале RSDN Magazine №2 за 2007 год

Недетерминированные конечные автоматы

Автор: Сергей Холодилов
The RSDN Group
http://vmk.ucoz.net/

Подвёл ты меня, Боролгин. А ведь я все деньги на тебя поставил.
.. <тут Боролгин сокрушается> ..
Не горюй, Боролгин. Я ещё и на орков поставил.

Арагорн в переводе Гоблина

Недетерминированные конечные автоматы – одна из моделей, используемых в теории вычислений. Вряд ли всё это когда-нибудь пригодится вам «по жизни»… но, чёрт возьми, математика – это интересно! Во всяком случае, для меня. А если уж она хоть как-то с программированием связана, то интересна вдвойне.

Я не претендую на математическую строгость, получилось что-то типа «популярной математики для чайников»… Но надо же с чего-то начинать. А причём здесь орки – поймёте по ходу дела :)

Просто конечные автоматы

Скорее всего, все более-менее знают, что такое конечные автоматы. Проблема в том, что я, например, знаю три варианта: конечные автоматы Мура, конечные автоматы Мили и «просто» конечные автоматы. Поскольку дальше нам потребуется вполне конкретное определение, имеет смысл ввести его здесь.

Итак, детерминированным конечным автоматом (ДКА) называется устройство, описываемое следующими параметрами:

- Q – конечное множество состояний.
- Σ – конечное множество входных символов.
- δ – функция перехода. Аргументы – состояние и входной символ, результат – состояние.
- q_0 – начальное состояние, принадлежит Q.
- F – множество допускающих состояний, является подмножеством Q.

И функционирующее следующим образом:

- Автомат начинает работу в состоянии q_0.
- Если автомат находится в состоянии q_i, а на вход поступает символ b, то автомат переходит в состояние $\delta(q_i, b)$.

ПРИМЕЧАНИЕ

Возможно, более простым вам покажется следующее определение: детерминированным конечный автоматом называется такой автомат, в котором при любой данной последовательности входных символов существует лишь одно состояние, в которое автомат может перейти из текущего. – *прим.ред.*

Работа ДКА заключается в распознавании цепочек символов, принадлежащих множеству Σ. Если, обработав цепочку, автомат оказался в допускающем состоянии, то цепочка считается допустимой, если нет, то нет. Таким образом, ДКА задаёт некоторый язык – множество допускаемых им цепочек, алфавит этого языка – множество Σ.

ПРИМЕЧАНИЕ

Это определение конечного автомата используется в теории вычислений. Автоматы Мура и Мили используются, в основном, при проектировании цифровой аппаратуры.

Конечные автоматы удобно изображать графически. На рисунке 1 приведён несложный ДКА, допускающий цепочки из 0 и 1, заканчивающиеся символом 0. Начальное состояние помечено входящей в него стрелочкой (да, этот здоровенный треугольник в душе – стрелочка), допускающее (в данном случае оно одно) – двойным кружком.

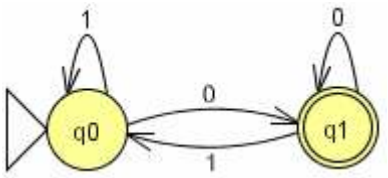


Рисунок 1. Простой детерминированный конечный автомат.

Второй вариант изображения автоматов – таблица переходов.

		0	**1**
->	**q_0**	q_1	q_0
*	**q_1**	q_1	q_0

Таблица 1. Тот же самый конечный автомат.

По горизонтали отложены символы входного алфавита, по вертикали – состояния, начальное состояние отмечено стрелочкой, а допускающие – звёздочками. Этот способ изображения менее нагляден, и приведён только в качестве примера, использовать его «для дела» я не буду.

Добавляем недетерминированность

Определение недетерминированного конечного автомата (НКА) практически полностью повторяет приведённое выше определение ДКА. Отличий всего два:

- δ – функция перехода. Аргументы – состояние и входной символ, результат – *множество состояний (возможно – пустое)*.
- Если автомат находится в состоянии q_i, а на вход поступает символ b, то автомат переходит *во множество состояний* $\delta(q_i, b)$. Если автомат находится во множестве состояний $\{q_i\}$, то он переходит во множество состояний, получаемое объединением множеств $\delta(q_i, b)$.

$$\{q_i\}^{next} = \bigcup \delta(q_j, b), \quad где\ q_j \in \{q_j\}^{current}$$

НКА тоже распознаёт цепочки символов, цепочка считается допустимой, если после её обработки множество состояний, в котором оказался автомат, содержит хотя бы одно допускающее. Таким образом, НКА также задаёт некоторый язык.

На рисунке 2 изображён простой НКА, допускающий цепочки из 0 и 1, заканчивающиеся на 00.

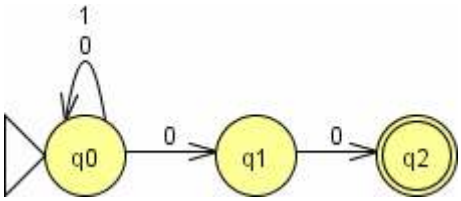


Рисунок 2. Простой недетерминированный конечный автомат.

Этот же автомат в виде таблицы:

		0	**1**
->	**q_0**	$\{q_0, q_1\}$	$\{q_0\}$
	q_1	$\{q_2\}$	(
*	**q_2**	(	(

Таблица 2. Тот же самый недетерминированный конечный автомат.

Разберёмся, как он работает.

Подход №1

Допустим, на вход автомату поступила цепочка «100100».

До	**Вход**	**Описание**	**После**
		Автомат начинает работу в множестве состояний $\{q_0\}$	$\{q_0\}$
$\{q_0\}$	1	Из состояния q_0 по символу 1 существует только один переход, в q_0 же.	$\{q_0\}$
$\{q_0\}$	0	Из состояния q_0 по символу 0 существует два перехода, в q_0 и в q_1.	$\{q_0, q_1\}$
$\{q_0, q_1\}$	0	Из состояния q_0 по символу 0 существует два перехода, в q_0 и в q_1, из состояния q1 – один переход, в q_2. Поскольку автомат находится в двух состояниях, множества объединяются.	$\{q_0, q_1, q_2\}$
$\{q_0, q_1, q_2\}$	1	Автомат находится в трёх состояниях, но из q_1 и из q_2 не существует переходов по символу 1 (т.е. значение функции перехода из этих состояний по входному символу 1 – пустое множество). В итоге остаётся только q_0.	$\{q_0\}$
$\{q_0\}$	0	И т.д.	$\{q_0, q_1\}$
$\{q_0, q_1\}$	0	И т.п. Так как в получившемся множестве состояний есть q_2 – допускающее состояние, автомат признаёт цепочку корректной.	$\{q_0, q_1, q_2\}$

Таблица 3. Обработка цепочки 100100.

Подход №2

Менее формальное, но немного более наглядное представление обработки той же цепочки тем же автоматом изображено на рисунке 3.

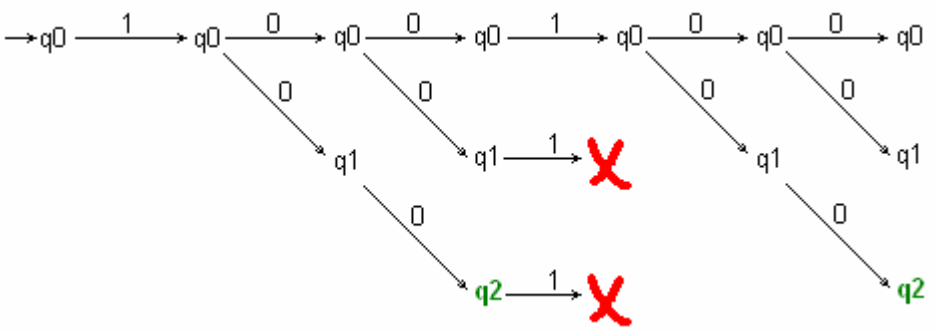


Рисунок 3. Обработка цепочки 100100

В данном случае мы несколько удаляемся от определения НКА (множества состояний не представлены так явно). Картинка основана на следующей идее: каждый раз, когда по входному символу возможен переход в несколько состояний, порождается новая ветка вычислений, когда переходов нет – ветка «засыхает». Если хоть одна из «живых» веток ведёт в допускающее состояние – цепочка допущена.

В общем, подходы дают аналогичные результаты, за исключением одной мелочи. Слегка изменённый НКА, изображённый на рисунке 4, допускает любую цепочку символов {0, 1}, содержащую два нуля подряд. Если каждый раз честно порождать новую ветку, то при обработке цепочки «1001001….» получится дерево, изображённое на рисунке 5. Понятно, что две нижние ветки полностью совпадают (они представляют одинаковые состояния автомата, получают одинаковые входы, значит и результаты будут одинаковые), более того, каждый раз, когда в цепочке будет встречаться 00, будет порождаться ещё одна точно такая же ветка.

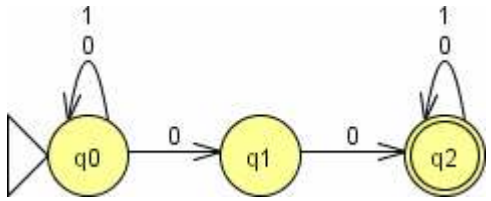


Рисунок 4. Немного изменённый НКА

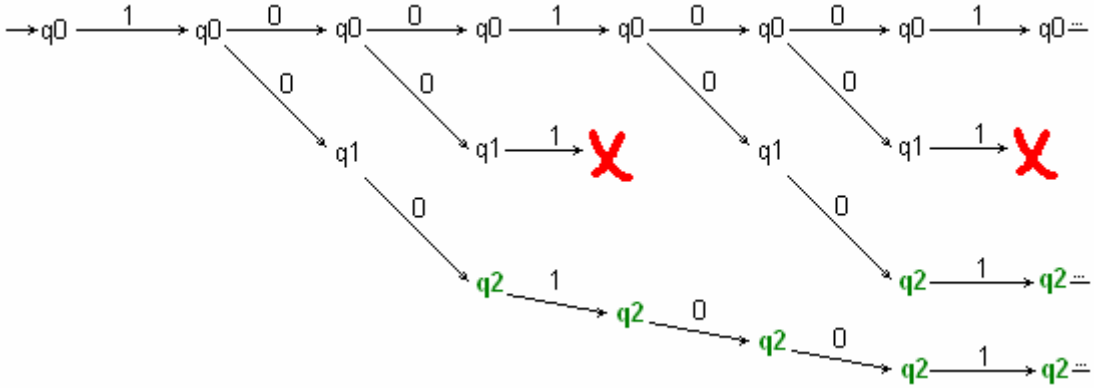


Рисунок 5. Обработка цепочки 1001001.

Можно добавить слияние одинаковых веток, но это уменьшит наглядность. В подходе, основанном на множествах, слияние происходит автоматически – множество не может содержать два одинаковых элемента.

Подход №3

Наименее формальный подход к описанию работы НКА основан на том, что он «угадывает». Вернёмся к автомату на рисунке 2 и цепочке «100100».

До	Вход	Описание	После

		Автомат начинает работу в состоянии $\{q_0\}$	q_0
q_0	1	Из состояния q_0 по символу 1 существует только один переход, в q_0 же.	q_0
q_0	0	Из состояния q_0 по символу 0 существует два перехода, в q_0 и в q_1. Но! *Автомат угадал*, что эта последовательность нулей – ещё не конец цепочки. Поэтому остаёмся в q_0.	q_0
q_0	0	И т.д.	q_0
q_0	1	И т.п.	q_0
q_0	0	А вот теперь *автомат угадал*, что это завершение, и что следующим входным символом тоже будет 0 (иначе нет смысла переходить в q_1 – из него нет переходов по 1). Переход в q_1	q_1
q_1	0	Из q_1 только один переход – в q_2. Цепочка допущена.	q_2

Таблица 4. Обработка цепочки 100100.

То есть, фактически, мы взяли дерево из подхода №2, и обрезали все ветки, кроме одной – наиболее успешной.

… и эпсилон-переходы

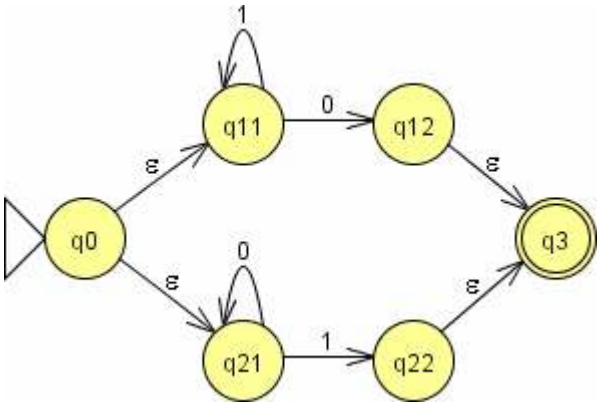
Небольшое полезное расширение стандартного НКА – ε-НКА, или НКА с эпсилон-переходами.

«Эпсилон-переходом» называется переход между состояниями, который может быть выполнен автоматом «просто так», без входного символа. На графах и в таблицах такие переходы обычно помечаются символом ε.

ПРИМЕЧАНИЕ

В замечательной программе JFLAP, скриншоты из которой используются в качестве рисунков, ε-переходы почему-то обозначаются символом λ, но я нашёл в себе силы поправить картинки.

На рисунке 6 изображён пример ε-НКА.



Как это ни удивительно, но на этом немного странном примере продемонстрировано одно из наиболее полезных свойств ε-переходов – возможность простого объединения нескольких автоматов в один. В данном случае объединяемыми автоматами являются НКА, изображённые на рисунке 7, а результирующий автомат допускает цепочки, допустимые хотя бы одним из них.

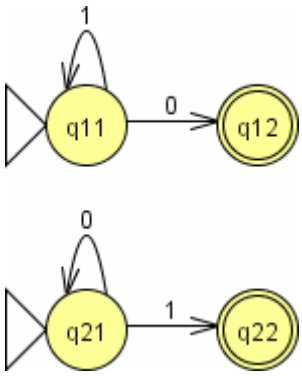


Рисунок 7. Составные части автомата, показанного на рисунке 6.

Кроме того, обратите внимание, что никаких дополнительных ограничений на ε-переходы не накладывается. То есть:

- из одного состояния ε-НКА может выходить сколько угодно как обычных, так и ε-переходов;
- может существовать цепочка из нескольких последовательных ε-переходов;
- автомат может проходить цепочку целиком, частично или не проходить ее вообще (если у ε-НКА есть другие варианты поведения).

В общем, граф/таблица переходов автомата выглядят так, как будто во входном алфавите появился ещё один символ – ε, а его работа так, будто в любом месте обрабатываемой цепочки (в начале, в конце, между символами) находится произвольное количество этих символов. При этом для каждого состояния есть как минимум один переход по ε – в себя.

… и более формально

Введём понятие *ε-замыкание*.

- ε-замыканием состояния q_i называется множество состояний ε-НКА, в которые из q_i можно попасть по цепочке ε-переходов. Как минимум, в это множество входит само q_i.
- Функцию, аргументом которой является состояние, а значением – соответствующее ε-замыкание, назовём *eclose*.

Функцию eclose можно определить так:

$$eclose(q_i) = \{q_i\} \cup \{eclose(q_j) \mid \exists\ \varepsilon\text{–переход от } q_i \text{ к } q_j\}$$

А теперь мы можем строго определить функционирование ε-НКА.

- Автомат начинает работать во множестве состояний $eclose(q_0)$.
- Если автомат находится во множестве состояний $\{q_i\}$, то он переходит во множество состояний, получаемое ε-замыканием всех состояний из объединения множеств $\delta(q_i, a)$.

$$\{q_i\}^{next} = \bigcup eclose(q_j), \quad где\ q_j \in \bigcup \delta(q_k, a), \quad где\ q_k \in \{q_k\}^{current}$$

И почему это круто

Допустим, вам нужен автомат, распознающий двоичные, десятичные и шестнадцатеричные числа в стандартном ассемблерном формате (я понимаю, что такие ситуации происходят в вашей жизни нечасто, но я и так сломал голову в поисках не слишком искусственного примера). То есть, допустимая строчка должна описываться одним из следующих утверждений:

- Состоит из одной или более цифр (0-9), может завершаться символом «d» (например, «1234»).
- Состоит из одного или более символов «0», «1», завершается символом «b» (например, «0100b»).
- Начинается с десятичной цифры, потом идёт любое количество символов 0-9 и a-f, завершается символом «h» (например, «0ab4h»).

Решение показано на рисунке 8.

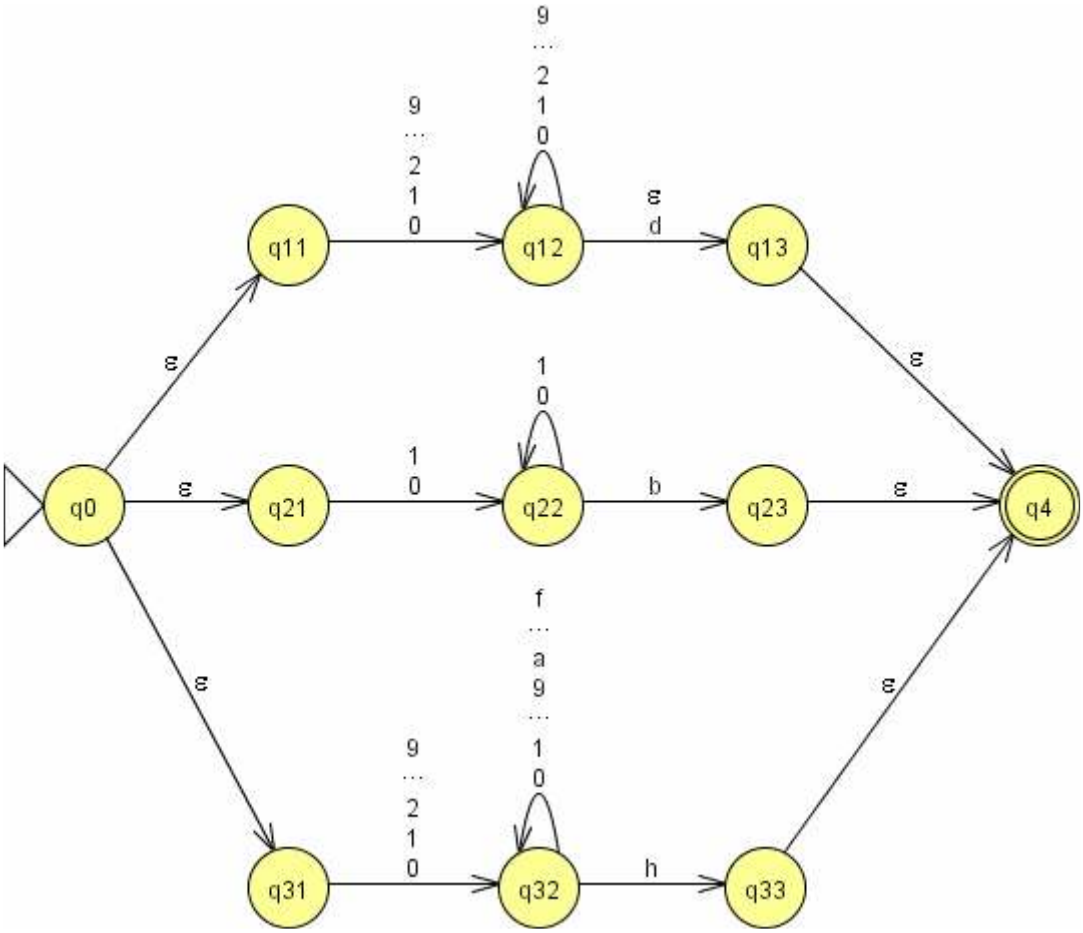


Рисунок 8. e-НКА, распознающий числа в ассемблерном формате

Что в первую очередь радует в этом конечном автомате – то, что он *понятен* и *изменяем*. Посмотрев на граф, можно практически сразу сказать, что делает автомат, и убедиться, что он соответствует ТЗ. А если задание поменяется, граф будет достаточно просто изменить. Например, если добавляется ещё одна система счисления, нужно просто нарисовать соответствующий автоматик и присоединить его к начальному и допускающему состояниям исходного автомата ε-переходами.

Выполняющий аналогичную задачу ДКА приведён на рисунке 9 (что за цифры под состояниями – разберёмся позже, пока не обращайте внимания), он тоже не слишком сложен, более того, в данном случае даже получилось меньше состояний (а вот переходов больше, 78 против 60). Но сравните: насколько ДКА более запутан! Попробуйте, глядя на него, понять, что делает этот автомат, проверить, правильно ли он это делает, попробуйте добавить или убрать систему счисления, ещё как-нибудь изменить ТЗ…

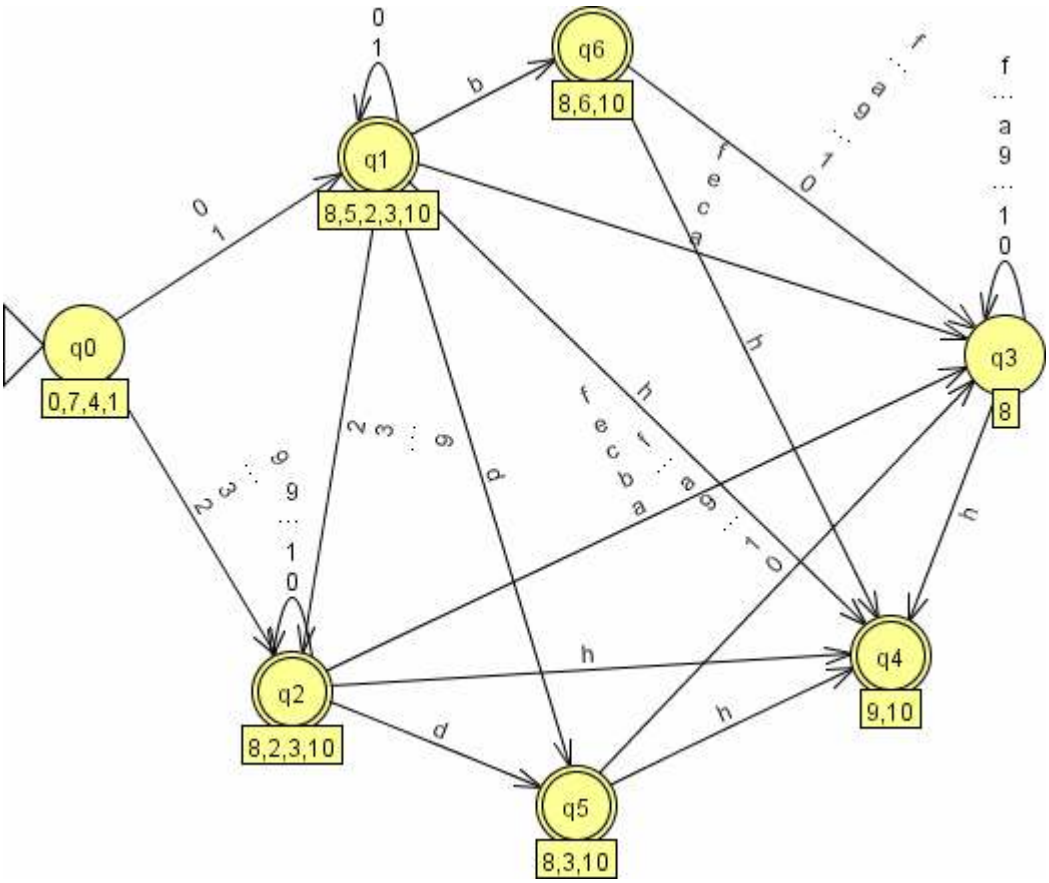


Рисунок 9. ДКА, распознающий числа в ассемблерном формате.

ПРИМЕЧАНИЕ

Помимо прочего, это ещё и не совсем правильный ДКА. В «воистину правильном» ДКА из каждого состояния есть переход по каждому символу входного алфавита, определением для ДКА не предусмотрено ситуации пустого множества состояний и «засыхания ветки вычислений».

Поэтому для полного соответствия нужно добавить в автомат ещё одно состояние, в которое должны сходиться все переходы по «непредусмотренным» символам, а выхода из этого состояния не будет – по любому символу следует переход в себя. Иногда такие состояния называют «дьявольскими».

В общем, мораль такая: с точки зрения проектирования конечных автоматов, ε-НКА – язык более высокого уровня, чем ДКА. Вычислительные возможности ε-НКА, просто НКА и ДКА совпадают – для любого ε-НКА *теоретически* можно построить ДКА, описывающий тот же язык (этим мы скоро займёмся), но проектировать ε-НКА гораздо, гораздо проще и удобнее.

Реализация методом «в лоб»

На детерминированном компьютере проще всего реализовывать НКА, основываясь на его определении. То есть, хранить множество текущих состояний, а при обработке очередного входного символа, переходить из каждого состояния в соответствующее множество и объединять результаты.

Примерно так:

- Таблица, с оригинальным именем table, по ключу, состоящему из состояния и символа, возвращает множество состояний.
- Множества объединяются оператором «+=».
- Множество current содержит текущие состояния.
- input – обрабатываемый в данный момент входной символ.

Обработка входного символа:

```
Set<State> tmp = new Set<State>(); // Создаём пустое множество

foreach (State s in current)
{
  // Объединяем результаты всех переходов
  tmp += table[new Key(s, input)];
}

// Сохраняем результат
current = tmp;
```

Производительность

На первый взгляд, количество операций, необходимое для обработки одного символа можно оценить как O(M), где M – количество состояний автомата. Но эта оценка верна, только если вам удастся реализовать объединение множеств за константное время, не зависящее ни от размера первого множества, ни от размера второго.

Типичная сложность реализации объединения множеств линейно зависит от размера второго множества. Поскольку оценить мы его можем только как O(M), в результате сложность обработки одного символа – $O(M^2)$, обработка строки длиной N – $O(N*M^2)$.

Можно переписать код, например, так:

```
SetsUnion<State> tmp = new SetsUnion<State>(); // Создаём объединение множеств

foreach (State s in current)
{
  // Объединяем результаты всех переходов
  // Операция + выполняется за константное время,
  // она просто добавляет множество в список
  tmp += table[new Key(s, input)];
}

// Сохраняем результат, при этом объединяем все множества в одно
// Если не объединять, будут сложности с итерацией по множеству
current = tmp.normalize();
```

Но и в этом случае $O(M^2)$ никуда не уходит, просто прячется внутри метода normalize.

Правда, это, конечно, оценка худшего случая – если все состояния соединены друг с другом всеми возможными переходами, в среднем всё будет не так печально.

e-переходы

При желании таким же несложным способом можно реализовать и ε-НКА – нужно перед началом обработки данных и после обработки каждого символа подавать на вход автомату ε, пока множество состояний не перестанет изменяться. Примерно так:

```
for (;;)
{
  Set<State> tmp = new Set<State>(); // Создаём пустое множество

  foreach (State s in current)
  {
    // Объединяем результаты всех переходов по эпсилону
    tmp += table[new Key(s, empty_input)];
```

```
  }

  if (current == tmp)
  {
    // Ничего не поменялось - ура, выходим
    break;
  }

  // Сохраняем результат
  current = tmp;
}
```

Это ужасающе неэффективно, но зато работает.

Немного подумав, можно вспомнить про формальное определение ε-НКА через ε-замыкания, и сообразить, что множество $eclose(q_i)$ – фиксировано, и его нужно вычислить только один раз, причём до начала работы автомата.

В результате, получаем вот такой код:

```
Set<State> tmp = new Set<State>(); // Создаём пустое множество

foreach (State s in current)
{
  // Объединяем замыкания всех текущих состояний
  tmp += closingsTable[new Key(s)];
}

// Сохраняем результат
current = tmp;
```

Этот кусок тоже имеет сложность $O(M^2)$.

Реализация преобразованием в ДКА

Теория

Рассмотрим произвольный НКА с тремя состояниями – q_0, q_1, q_2.

Независимо от своей внутренней структуры, в каждый конкретный момент этот НКА может находиться в одном из следующих множеств состояний:

- ((пустое множество)
- $\{q_0\}$
- $\{q_1\}$
- $\{q_2\}$
- $\{q_0, q_1\}$
- $\{q_0, q_2\}$
- $\{q_1, q_2\}$
- $\{q_0, q_1, q_2\}$

И всё, других вариантов нет. Причём переход между множествами состояний чётко детерминирован – это просто объединение значений функции перехода для каждого из состояний, и сами значения и их объединения тоже находятся в этом списке.

Обобщая наблюдения до произвольного НКА с любым количеством состояний, мы можем определить следующий ДКА:

- его состояниями будут множества состояний НКА (цифры под состояниями ДКА на рисунке 9 обозначали соответствующие ему состояния НКА).

$$q_j^{дка} = \{q_i^{нка}\}$$

- входной алфавит – такой же, как у НКА
- функция перехода будет «правильным образом» сопоставлять множеству и входному символу другое множество.

$$\delta^{дка}(q_j^{дка}, a) = \bigcup \delta^{нка}(q_i^{нка}, a), \quad где\ q_i^{нка} \in q_j^{дка}$$

- начальным состоянием будет множество, состоящее только из начального состояние НКА.

$$q_0^{дка} = \{q_0^{нка}\}$$

- допускающими будут те состояния ДКА, которые содержат хотя бы одно допускающее состояние НКА.

$$F^{дка} = \{q_j^{дка} \mid q_j^{дка} \cap F^{нка} \neq \varnothing\}$$

Идея преобразования основана на том, что множество подмножеств конечного множества состояний – конечно, то есть, как бы НКА не крутился, он всегда находится в одном из *конечного* множества состояний.

ПРИМЕЧАНИЕ

Если мысль о том, что множества состояний одного автомата являются состояниями другого автомата, пока что плохо влезает в вашу голову, можно представить то же самое менее абстрактно. Для НКА с тремя состояниями, обозначим состояния ДКА следующими строчками:

* -x-

* -q0-

* -q1-

* -q2-

* -q0-q1-

* -q0-q2-

* -q1-q2-

* -q0-q1-q2-

Назначим среди этих состояний правильное начальное (оно должно соответствовать начальному состоянию НКА) и допускающие, нарисуем таблицу для функции переходов, получим ДКА.

Для НКА с большим количеством состояний – по аналогии.

С использованием индукции достаточно просто доказывается, что сконструированный ДКА описывает тот же язык (допускает точно те же цепочки символов), что и исходный НКА. Эта задача оставляется читателю в качестве упражнения :)

СОВЕТ

Индукционное предположение – в процессе обработки цепочки ДКА всегда находится в состоянии, соответствующем текущему множеству состояний НКА. База очевидна, для доказательства индукционного перехода нужно посмотреть на определение НКА и сравнить алгоритм вычисления следующего множества состояний с алгоритмом работы функции перехода нашего ДКА. А после всего этого сравнить, в каких случаях автоматы допускают цепочки.

Добавляем e-переходы

Конструирование ДКА, соответствующего заданному ε-НКА, немного отличается – нужно в нескольких местах заменить слова «состояние q_i» на «ε-замыкание состояния q_i». Вот эти места:

- функция перехода ДКА будет «правильным образом» сопоставлять множеству и входному символу другое множество.

$$\delta^{дка}(q_j^{дка},a)=\bigcup eclose(q_i^{нка}),\quad где\ q_i^{нка}\in\bigcup\delta^{нка}(q_k^{нка},a),\quad где\ q_k^{нка}\in q_j^{дка}$$

- начальным состоянием ДКА будет ε-замыкание начального состояние ε-НКА.

$$q_0^{дка}=eclose(q_0^{нка})$$

Доказательство идентичности описываемых автоматами языков аналогично доказательству для обычного НКА.

ПРИМЕЧАНИЕ

Таким образом, можно считать доказанным, что любой язык, описываемый ε-НКА, может быть описан обычным ДКА. А поскольку обратное очевидно (ДКА это просто частный случай ε-НКА), вычислительная мощность ДКА и ε-НКА совпадают.

Просто хотелось отдельно отметить этот важный теоретический результат :)

Алгоритм

Состояния ДКА

Для начала оценим количество состояний «теоретического» ДКА. Если НКА имеет M состояний, то состояниями ДКА будут все подмножества множества $\{q_0, \dots q_{M-1}\}$. Поскольку каждое из q_i может входить или не входить в подмножество, мы получаем 2^M состояний ДКА.

Такое количество состояний немного пугает, но, к счастью, часто подавляющее большинство состояний оказывается недостижимым (то есть, не существует последовательности переходов, которая приводит автомат в такое состояние из начального), поэтому их можно просто отбросить, и это никак не повлияет на описываемый ДКА язык.

Но часто – не значит всегда. Пример НКА, ДКА которого будет содержать 2^{M-1} состояний, приведён на рисунке 10. Вставляя «в хвост» НКА дополнительные состояния, можно неограниченно увеличивать количество состояний ДКА.

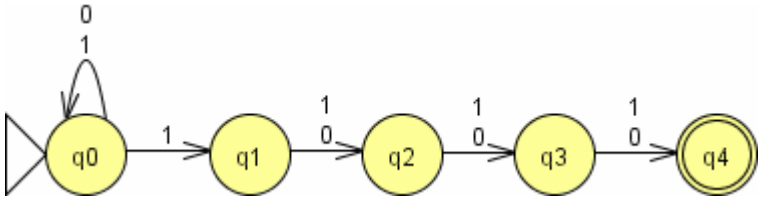


Рисунок 10. Неудачный НКА.

Генерация ДКА

Возможны два подхода:

- Сначала сгенерировать все состояния ДКА, которые только могут потребоваться, установить между ними связи, а потом избавляться от недостижимых состояний.
- Сразу же генерировать только достижимые состояния.

Мы пойдём вторым путём. Основная идея алгоритма:

- Начинаем с начального состояние ДКА – eclose(q_0). Оно достижимо по определению.
- Если состояние достижимо, то все состояния, в которые из него можно попасть, тоже достижимы.
- Из состояния q_i можно попасть в те состояния, которые являются значением $\delta(q_i, x)$, где δ – функция перехода ДКА, а x принадлежит множеству входных символов. Перебрав все входные символы, получим все такие состояния.
- После чего применяем к полученным состояниям тот же принцип. Остановка – когда все сгенерированные состояния рассмотрены подобным образом.

Код

Генерация ДКА:

```
// Добавляем начальное состояние в список новых
newStates.add(eclose(q0));

while (!newStates.empty())
{
  // Извлекаем очередное новое состояние списка
  MultiState tmp = newStates.popfront();

  if (dfaStates.Contain(tmp))
  {
    // Оно уже есть в списке достижимых
    continue;
  }

  // Это новое состояние, ещё не рассмотренное
  dfaStates.add(tmp);

  foreach (Symbol a in inputSymbols)
  {
    // Для каждого входного символа генерируем новое состояние
    MultiState newTmp = tmp.transit(a).eclose();
    // добавляем его в список «новых» состояний
    newStates.add(newTmp);
    // и устанавливаем связь в таблице переходов ДКА
    dfaTransitTable.add(new Key(tmp, a), newTmp);
```

```
    }
}
```

Работа ДКА:

```
// Сохраняем результат
current = dfaTransitTable[new Key(current, input)];
```

Производительность

Оценка алгоритма генерации ДКА неутешительна. Количество операций линейно зависит от количества состояний ДКА, а верхняя оценка для них – 2^M. В среднем будет получше, но тоже ничего хорошего – генерация новых состояний, то есть внутренняя часть цикла, это $O(M^2*S)$ операций (где S – количество символов).

Но зато, после того как ДКА сгенерирован, он обрабатывает любой символ за константное время, а строчку длиной N – за O(N).

Заключение

.. а дальше? ..

Пластилиновая ворона

Возможно (я на это надеюсь!), вам кажется, что всё описано недостаточно строго, или же что описано ужасающе мало. Этих и многих других недостатков лишена замечательная книжка «Введение в теорию автоматов, языков и вычислений», авторы Джон Хопкрофт, Раджив Мотвани, Джеффри Ульман (издательство Вильямс, 2002). Очень рекомендую, правда, вряд ли вы её найдёте в магазинах, разве что на русском выпустят третье издание.

Значительно проще найти книгу «Классика программирования: алгоритмы, языки, автоматы, компиляторы», Мозговой М.В., Наука и Техника, 2006. Тоже хорошая книжка, она менее фундаментальна и строга, ближе к программированию, и содержит куски кода на C#.

Ну а играть с автоматами проще всего в программе JFLAP, которая уже упоминалась выше.
