Chapter 4

Regular expressions

Up until now we have been examining finite state automata in a variety of forms. Now we will examine a notation, called *Regular expressions* that can express the same languages as finite state automata, but in a more compact and user-friendly way.

A regular expression describes strings of characters in a compact way. Before we define formally how regular expressions work, let us see some examples:

sequence A regular expression ab stands for the character a followed by the character b. This idea can be carried out to arbitrary length, so (for example) $abcd$ is the character a followed by b followed by c followed by d.

alternation The regular expression $a \mid b$ stands for either the character a or the character b (but not both). The string $ab \mid cd$ stands for the string of characters ab or the string cd. Notice that sequence takes precedence over alternation. There can be multiple alternation operators, like this: $a \mid b \mid c$, which means a or b or c.

repetition The regular expression a^* stands for any one of the strings ϵ,a,aa, aaa,$aaaa$,... The expression ab^* stands for the any of the strings a,ab, abb,$abbb$,... Notice that repetition takes precedence over sequence.

grouping Parentheses can be used to group parts of regular expressions. For example the expression $(a \mid b)(c \mid d)$ represents the strings ac, ad, bc, and bd.

4.1 The operators of regular expressions

Regular expressions denote languages. For example, the regular expression: $01^* \mid 10^*$ denotes the language consisting of all strings that start with a single 0, and are followed by zero-or-more 1s, or start with a single 1, followed by zero-or-more 0s.

Let us now consider these operations more formally.

1. The concatenation of two languages L and M, denoted by LM is the set of strings that can be formed by taking any string in L, and concatenating it with any string in M. We usually denote concatenation by just putting the two languages in sequence, as we have previously shown.

   For example, if L is the language {1,01,10,101}, and M is the language {ϵ,0}, then LM is the language {1,01,10,101,010,100,1010}. The first four strings of LM are simply those of L concatenated with ϵ. The remaining three strings come from L concatenated with 0. Note that the string 10 is generated twice by this process, but appears only once in LM (because it is a *set*, not a *list*).

2. The union of two languages L and M, denoted by $L \mid M$, is the set of strings that is in L or M or both. For example if $L = \{01, 11, 100\}$ and $M = \{1, 101\}$, then $L \mid M = \{1, 01, 11, 100, 101\}$.

3. The closure (also called the *Kleene star*) of a language L, denoted by L^* represents the set of all strings that can be formed by taking any number of strings from L, with repetitions permitted, and concatenating them.

   For example, if $L = \{0, 1\}$, then L^* is the set of all strings of zero-or-more ones and zeros.

   if $L = \{0, 10\}$, then L^* is the set of all strings of ones and zeros where a one is always followed by a zero. e.g. 0,10,1010,10010, but not 0110. Formally:

$$L^* = \bigcup_{i \geq 0} L^i = L^0 \cup L^1 \cup L^2 \cdots$$

   where $L^0 = \{\epsilon\}$, $L^1 = \{L\}$, and $L^i = LL^{i-1}$ (i copies of L concatenated together).

Closure is tricky, so here are two examples to clarify the concept:

Example 1 If $L = \{0, 10\}$, then $L^0 = \{\epsilon\}$, since the set of all strings of length zero consists only of the string ϵ). $L^1 = L = \{0, 10\}$. $L^2 = LL = \{00, 010, 100, 1010\}$. $L^3 = LLL = \{000, 0010, 0100, 01010, 1000, 10010, 10100, 101010\}$, continuing in this fashion, we can compute $L^4, L^5 \cdots$. We compute L^* by taking the union of all these sets:

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cdots$$

$$L^* = \epsilon \cup \{0, 10\} \cup \{00, 010, 100, 1010\} \cup \{000, 0010, 0100, 01010, 1000, 10010, 10100, 101010\} \cup \cdots$$

$$L^* = \{\epsilon, 0, 10, 00, 010, 100, 1010, 000, 0010, 0100, 01010, 1000, 10010, 10100, 101010, \cdots\}$$

Example 2 Now consider a different language L which is the set of all strings of zero-or-more 0s. Clearly L is an infinite language, unlike the previous example which was finite. Despite this, it is not hard to derive L^*: $L^0 = \epsilon$, $L^1 = L$, $L^2 = L$, $L^3 = L$, and so on. So $L^* = L$.

4.2 Precedence of operators

Like all algebras, the regular expression operators have an assumed order of "precedence", which means that operators are associated with their operands in a particular order. We are familiar with precedence from high-school algebra: The expression $ab + c$ groups the product before the sum, so the expression means $(a \times b) + c$. Similarly, when we encounter two operators that are the same, we group from the left, so $a - b - c$ means $(a - b) - c$, and *not* $a - (b - c)$. For regular expressions the order of evaluation is:

repetition. The star ($*$)operator has highest precedence. It applies to the smallest string of symbols to its left, that is a well-formed expression.

sequence. The concatenation operator has the next level of precedence. After grouping all the stars to their operands, we group all concatenation operators to their operands. All expressions that are adjacent (without an operator in between) are concatenated. Since concatenations is associative, it does not matter what order we group the operands, though it is conventional to group from the left. Thus abc is grouped as $(ab)c$.

alternation. The alternation operator (|) has the lowest precedence. The remaining operators are grouped with their operands. Alternation is also associative, so the order of grouping does not matter, but again the convention is to group from the left.

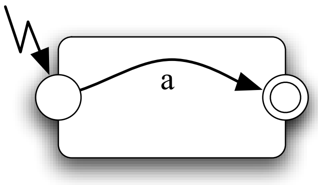
For example, applying the rules to the expression $01^* \mid 10 \mid 1^*0$, we group the star first, giving $0(1^*) \mid 10 \mid (1^*)0$. Then we group the sequence operators, giving $(0(1^*)) \mid (10) \mid ((1^*)0)$. Finally we group the alternation operators, to get: $(((0(1^*))) \mid (10)) \mid ((1^*)0)$.

4.3 Converting finite automata to regular expressions

Regular expressions and finite automata define the same class of languages, so it is possible to transform a DFA into a regular expression, and a regular expression into a DFA. We will ignore the transformation from DFA to regular expression, since this is not often used.

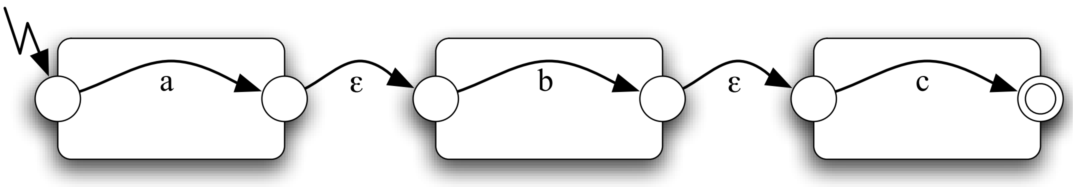
We can transform a regular expression to an ϵ-NFA, in a sequence of trivial steps, as we will now see. First we will need to construct some "building blocks" for constructing the NFA.

The simplest regular expression, a, can be converted to a two-state NFA, with a start-state and a final state, by writing the character a on the transistion, like this:



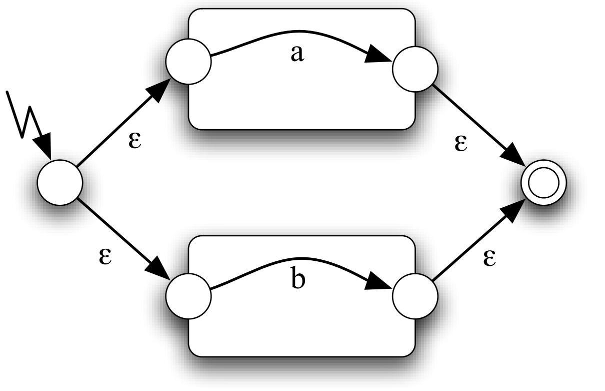
It is clear that this NFA accepts a string consisting only of the character a.

A sequence of simple regular expressions abc, can be handled by converting each character in the expression to a two-state NFA, and then joining the NFAs togetherwith ϵ transitions, like this:



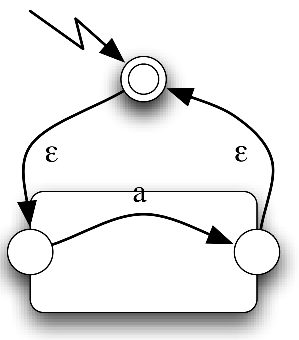
It is immediately obvious that this NFA accepts a string consisting only of the sequence of characters abc.

A regular expression of the form $a \mid b$, can be handled by building two simple NFAs, to handle the a and b, and then running them in parallel. Again, we use ϵ transistions to glue the parts together, like this:

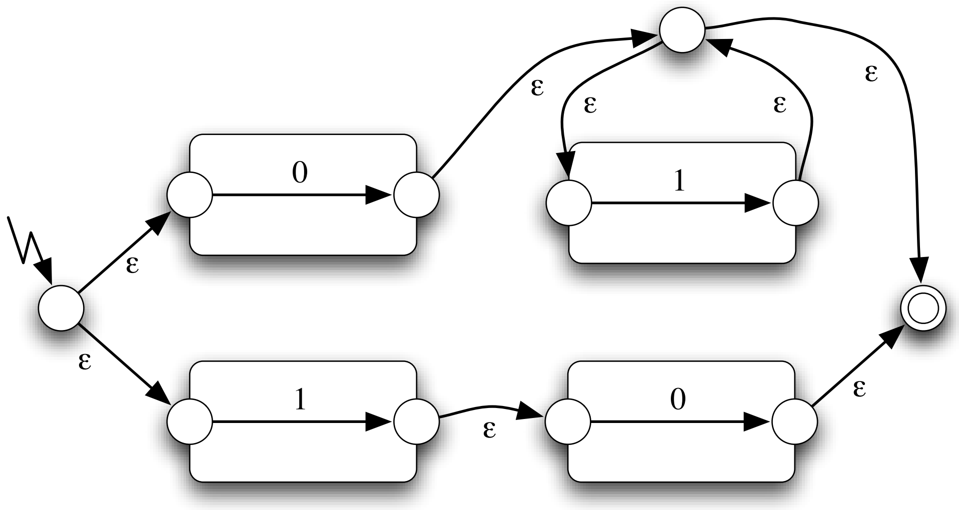


Once again, it is immediately clear that this NFA will accept a string consisting of either a or b.

Finally, to handle the regular expression a^*, we simply add a pair of ϵ transitions to the NFA for a. One of the transitions allows us to completely bypass the NFA (thus allowing "zero-times"), and the other transition allows an infinite number of repetitions (thus allowing "-or-more times"). The diagram therefore looks like this:

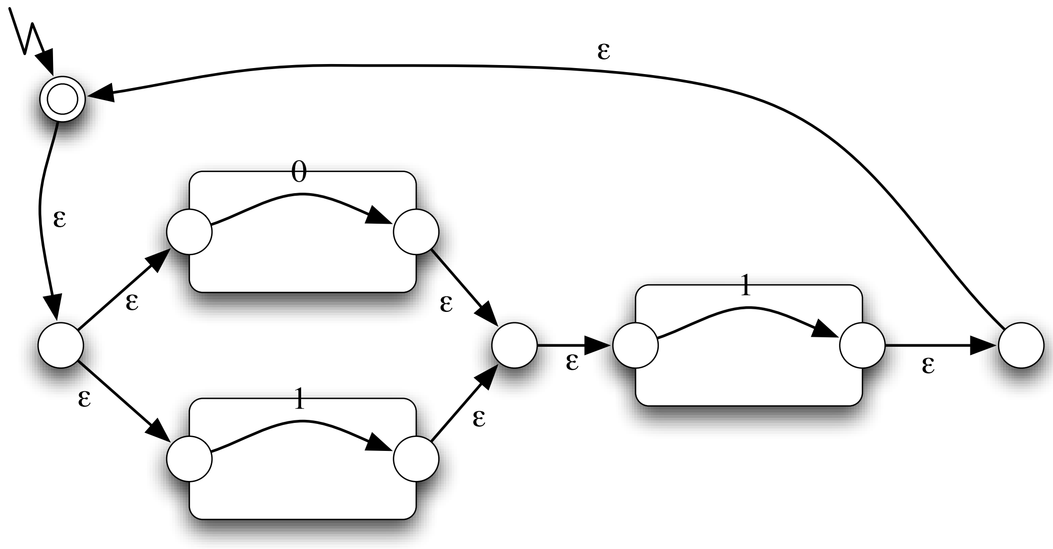


We can now compose these diagrams to handle any regular expression. For example, conside the regular expression $01^* \mid 10$. If we insert parentheses, to make the meaning clear, we get $((0(1^*)) \mid (10)$ which can immediately be converted to this diagram:



It is now a simple matter of using the techniques we already know to reduce this ϵ-NFA to a deterministic finite-state automaton.

Similarly, the expression $((0 \mid 1)1)^*$ can be converted to this diagram:



4.3.1 What is the use of this transformation?

Regular expressions are a very compact way of *expressing* a language. DFAs are a very efficient way of *implementing* a language recogniser. The transformation we have just described trivially converts a regular expression into an ϵ-NFA, and we already know how to convert any NFA into a DFA. Thus we now have a highly efficient way of turning any regular expression into an executable program.

There are numerous tools available, that do this job. One example is *Jlex*, which you may have used during a compiler-construction course.

4.4 Regular expressions in Unix

The regular expressions described earlier have enough expressive power to describe any regular language. However some expressions can become a bit clumsy. Unix extends the set of operators that are available, to make the task of expressing a language even easier. The additional operators available in Unix are:

any The character . (dot) means "any character".

list The sequence of characters $[abcde]$ means $a \mid b \mid c \mid d \mid e$. This saves about half the typing, since we don't need to type the $|$.

We can also write $[b - e]$, to represent the consecutive sequence of characters beginning at b and ending at e. For example $[a - z]$ means any lower-case letter. [A-Za-z0-9] means any letter or digit. This is a *big* saving in typing. If we want to include a $-$ sign in the list, we just put it first or last in the list. We could describe a signed (one-digit) integer with $[-+][0 - 9]$.

There are few predefined patterns: $[:digit:]$ means "any digit", $[:alpha:]$ means any alphabetic character, and $[:alnum:]$ means "any letter or digit". To use these patterns, they *must* appear in a list: $[[:digit:]]$ matches a single digit.

optional The character ? placed after an expression means "zero-or-one-of". For example the pattern $(+|-)?[[:digit:]][[:digit:]]*$, describes an integer number with an optional preceding sign. We could also express this as: $[+-]?[[:digit:]][[:digit:]]*$.

repetition. There is an additional repetition operator, $+$, that means “one-or-more-of”. The regular expression $R+$ has the same meaning as $RR*$. We can now express our previous definition of signed integer as $[+-]?[[:digit:]]+$.

multiple The operator {n} (where n is a positive integer) placed after a regular expression means “n-copies-of”. For example the expression $(ab)\{3\}$ has the same meaning as $ababab$.

nomagic Clearly, some of the characters we have special meanings (such as $*$ [? .). We can tell Unix to treat these characters *without* their magic interpretation by preceding them with a backslash character ($\backslash$). For example, the regular expression $\backslash * a\backslash ?$ will only match the string of three characters $*a?$.

Here are some example Unix patterns:

Integer A signed integer can be specified by : $[+-][0-9]+$, or equivalently $[+-][[:digit:]]+$.

Identifier A typical programming language identifier can be specified by: $[a-zA-Z][a-zA-Z0-9\_]*$, or this: $[[:alpha:]][[:alnum:]\_]*$.