CS351
Regular Expressions

Regular expressions are a notation that you can think of similar to a programming language. In fact, regular expressions are quite fundamental in some programming languages like perl and applications like grep or lex. Regular expressions are similar to NFA and end up describing the same things we can express with a finite automaton. However, regular expressions are declarative in what strings are accepted, while automata are machines that accept strings.

Algebra of Regular Expressions

One of the powerful features of RE's is the definition of an algebra. Just as we can use arithmetic expressions on numbers (e.g., 5+3-4) we can perform similar operations on RE's. However, we must define what all of our symbols mean and do. First let's see some operations on languages, and then these will easily translate into operators for RE's.

1. The union of two languages L and M is the set of strings that are in both L and M. So if L = { 0, 1} and M = {111} then L ∪ M is {0, 1, 111}.

2. The concatenation of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. Concatenation is denoted by LM although sometimes we'll use L•M (pronounced "dot"). For example, if L = {0, 1} and M = {ε, 010} then LM is { 0, 1, 0010, 1010}.

3. The closure, star, or Kleene star of a language L is denoted L^* and represents the set of strings that can be formed by taking any number of strings from L with repetition and concatenating them.

   More specifically, L^0 is the set we can make selecting zero strings from L. L^0 is always { ε }.

   L^1 is the language consisting of selecting one string from L.

   L^2 is the language consisting of concatenations selecting two strings from L.
   …
   L* is the union of L^0, L^1, L^2, … L^∞

   For example, if L = {0 , 10} then
   L^0 = { ε }.
   L^1 = {0, 10 }
   L^2 = {00, 010, 100, 1010}
   L^3 = {000, 0010, 0100, 01010, 10010, 1000, 10100, 101010}
   … and L* is the union of all these sets, up to infinity.

In most cases L* is infinite, although there are some languages that have a finite closure. Consider the empty language Ø. $Ø^0 = \{\varepsilon\}$, $Ø^1 = \{\varepsilon\}$, ,,, so $Ø^* = \{\varepsilon\}$.

Definition of Regular Expressions

Say that **R** is a regular expression if **R** is:

1. **a** for some *a* in the alphabet ∑, standing for the language {a}
2. ε, standing for the language {ε}
3. Ø, standing for the empty language
4. R_1+R_2 where R_1 and R_2 are regular expressions, and + signifies union
5. R_1R_2 where R_1 and R_2 are regular expressions and this signifies concatenation
6. R^* where R is a regular expression and signifies closure
7. (R) where R is a regular expression, then a parenthesized R is also a regular expression

While it might seem that we are in danger of defining a RE circularly, definitions 1-3 form the basis and 4-7 the inductive definition. Note that the definitions from 4-7 use regular expressions that are smaller then the one being defined.

There is also a precedence among the operators, just as we have precedence among arithmetic operators.

Parentheses have the highest precedence, followed by *, concatenation, and then union.

Here are some examples:

- L(**001**) = {001}.
- L(**0+10***) = { 0, 1, 10, 100, 1000, 10000, … }
- L($\mathbf{0^*10^*}$) = {1, 01, 10, 010, 0010, …} i.e. {w | w has exactly a single 1}
- L(∑∑)* = {w | w is a string of even length}
- L((**0(0+1)**)*) = { ε, 00, 01, 0000, 0001, 0100, 0101, …}
- L((**0+ε)(1+ ε**)) = {ε, 0, 1, 01}
- L(1Ø) = Ø ; concatenating the empty set to any set yields the empty set.
- Rε = R
- R+Ø = R

Note that R+ε may or may not equal R (we are adding ε to the language)
Note that RØ will only equal R if R itself is the empty set.

Exercise: Write a regular expression for the set of strings that contains an even number of 1's over ∑={0,1}. Treat zero 1's as an even number.

Finite Automata and Regular Expressions

We've already claimed that FA and RE's are equivalent. To show this, we will show that we can take a DFA and express it as a RE. Then we will take a RE and show that we can convert it to an ε-NFA. Since the ε-NFA can be converted to a DFA and the DFA to an NFA, then RE will be equivalent to all the automata we have described.

From DFA to Regular Expression – Inductive Construction

We will look at two methods to go from a DFA to a RE. The first is an inductive construction, and then we'll look at state elimination.

Theorem: If L=L(A) for some DFA A, then there is a regular expression R such that L=L(R).

Proof: Suppose that A's states are numbered {1, 2, … n} for some integer n.
Let $R_{ij}^{(k)}$ denote the regular expression whose language is the set of labels of paths that go from state i to state j **without passing** through any state numbered above k. Note that there are no restrictions in states i and j.

When k=0, $R_{ij}^{(0)}$ for all i,j denotes the set of regular expressions for all direct transitions from states i to j.
When k=1, $R_{ij}^{(1)}$ for all i,j denotes the set of regular expressions for all direct transitions or if we make transition through state 1.
When k=2, $R_{ij}^{(2)}$ for all i,j denotes the set of regular expressions for all direct transitions or if we make transition through state 1 or 2.

When we get all the way up to k=n, the union of all $R_{ij}^{(n)}$ describes all transitions throughout the entire automaton and is an equivalent regular expression. The idea is somewhat similar to computing the all-pairs shortest path algorithm.

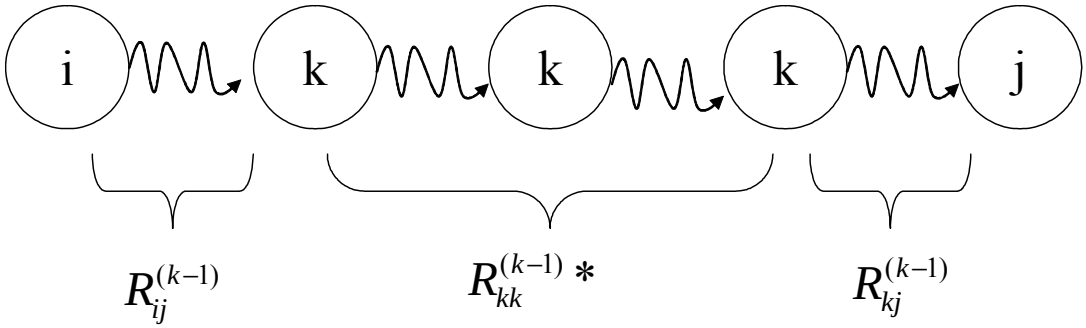
Basis: k=0. In this case, the path is either an arc or the null path (a single node).

If i≠j then $R_{ij}^{(0)}$ is the sum of all symbols *a* such that A has a transition from i to j on symbol *a*, or Ø if none.

If i=j then add ε to the above since we could stay in the same state with no input.

Induction: Assume that we have correctly developed expressions for $R_{ij}^{(k-1)}$. Then for $R_{ij}^{(k)}$: $R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}\left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)}$

We can depict the inductive step graphically:



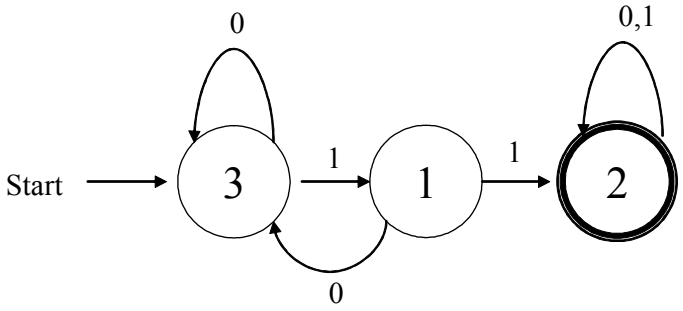
There are two cases here:

1. The path does not go through state k at all. In this case, the label of the path is in the language of $R_{ij}^{(k)}$
2. The path goes through state k at least once. We can break this path up into the part that goes from i to k, the part the possibly repeats back to state k, and then the part that goes from k to j. This is expressed in $R_{ik}^{(k-1)}\left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)}$

We union together both possibilities to get $R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}\left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)}$.

When k=n, the regular language of the automaton is then the sum (union) of all the expressions $R_{ij}^{(n)}$ where i is a start state and j is an accepting state.

Example: Consider the "clamping" automaton we saw earlier:



To convert this to a RE start with k=0:

$R_{11}^{(0)}$	ε
$R_{12}^{(0)}$	1
$R_{13}^{(0)}$	0
$R_{21}^{(0)}$	Ø
$R_{22}^{(0)}$	ε+0+1
$R_{23}^{(0)}$	Ø
$R_{31}^{(0)}$	1
$R_{32}^{(0)}$	Ø
$R_{33}^{(0)}$	ε+0

Next we compute $R_{ij}^{(1)} = R_{ij}^{(0)} + R_{ik}^{(0)}\left(R_{kk}^{(0)}\right)^{*} R_{kj}^{(0)}$

$R_{11}^{(1)}$	ε+(εε* ε)	= ε
$R_{12}^{(1)}$	1+ εε*1	= 1
$R_{13}^{(1)}$	0+ εε*0	= 0
$R_{21}^{(1)}$	Ø+ Øε*ε	= Ø
$R_{22}^{(1)}$	ε+0+1 + Øε*1	= ε+0+1
$R_{23}^{(1)}$	Ø+ Øε*0	= Ø
$R_{31}^{(1)}$	1+1ε*ε	= 1
$R_{32}^{(1)}$	Ø+1ε*1	= 11
$R_{33}^{(1)}$	ε+0+1ε*0	= ε+0+10

Next we compute $R_{ij}^{(2)} = R_{ij}^{(1)} + R_{ik}^{(1)}\left(R_{kk}^{(1)}\right)^{*} R_{kj}^{(1)}$

$R_{11}^{(2)}$	ε + 1(ε+0+1)*Ø	= ε
$R_{12}^{(2)}$	1 + 1(ε+0+1)*(ε+0+1)	= 1 + 1(ε+0+1)*
$R_{13}^{(2)}$	0 + 1(ε+0+1)*Ø	= 0
$R_{21}^{(2)}$	Ø + (ε+0+1) (ε+0+1)*Ø	= Ø
$R_{22}^{(2)}$	ε+0+1 + (ε+0+1) (ε+0+1)*(ε+0+1)	= (ε+0+1)*
$R_{23}^{(2)}$	Ø + (ε+0+1)(ε+0+1)*Ø	= Ø
$R_{31}^{(2)}$	1 + 11(ε+0+1)*Ø	= 1
$R_{32}^{(2)}$	11 + 11(ε+0+1)* (ε+0+1)	= 11(ε+0+1)*
$R_{33}^{(2)}$	ε+0+10 + 11(ε+0+1)*Ø	= ε+0+10

Finally we compute $R_{ij}^{(3)}$:

$R_{11}^{(3)}$	$\varepsilon + 0(\varepsilon+0+10)^*1$	
$R_{12}^{(3)}$	$1 + 1(\varepsilon+0+1)^* + 0(\varepsilon+0+10)^*11(\varepsilon+0+1)^*$	
$R_{13}^{(3)}$	$0 + 0(\varepsilon+0+10)^*(\varepsilon+0+10)$	$= 0+0(\varepsilon+0+10)^*$
$R_{21}^{(3)}$	$\emptyset + \emptyset(\varepsilon+0+10)^*1$	$= \emptyset$
$R_{22}^{(3)}$	$(\varepsilon+0+1)^* + \emptyset(\varepsilon+0+10)^*11(\varepsilon+0+1)^*$	$= (\varepsilon+0+1)^*$
$R_{23}^{(3)}$	$\emptyset + \emptyset(\varepsilon+0+10)^* (\varepsilon+0+10)$	$= \emptyset$
$R_{31}^{(3)}$	$1 + (\varepsilon+0+10)\ (\varepsilon+0+10)^*1$	$= 1+(\varepsilon+0+10)^*1$
$R_{32}^{(3)}$	$11(\varepsilon+0+1)^* + (\varepsilon+0+10)\ (\varepsilon+0+10)^*11(\varepsilon+0+1)^*$	
		$= (\varepsilon+0+10)^*11(\varepsilon+0+1)^*$
$R_{33}^{(3)}$	$\varepsilon+0+10 + (\varepsilon+0+10)\ (\varepsilon+0+10)^*\ (\varepsilon+0+10)^*$	$= (\varepsilon+0+10)^*$

Since the start state is 3 and the final state is 2, we really only need:
$R_{32}^{(3)} = (\varepsilon+0+10)^*11(\varepsilon+0+1)^* \qquad = (0+10)^*11(0+1)^*$

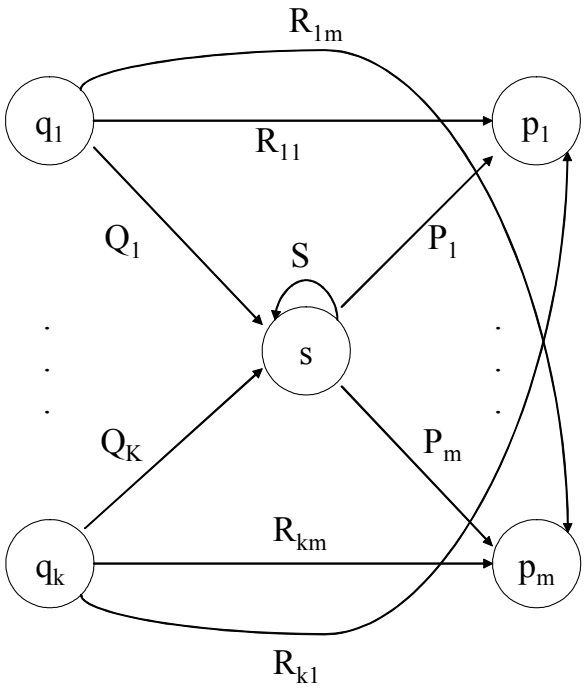
You may have been able to eyeball the original automaton and just come up directly with this expression ad-hoc by visually tracing all possible ways to get from the start state to the goal state. While this solution works, be careful to include all paths from the start to the goal. For example, (0*+(10)*)11 may include several ways to get the goal, but it doesn't include all ways to get to the goal (we could repeat additional 0 and 1's for example).

From DFA to Regular Expression – State Elimination

Another way of converting a DFA to a RE is through state elimination. This technique is often easier then the above (which requires looking at an exponentially large number of expressions based on the number of states).

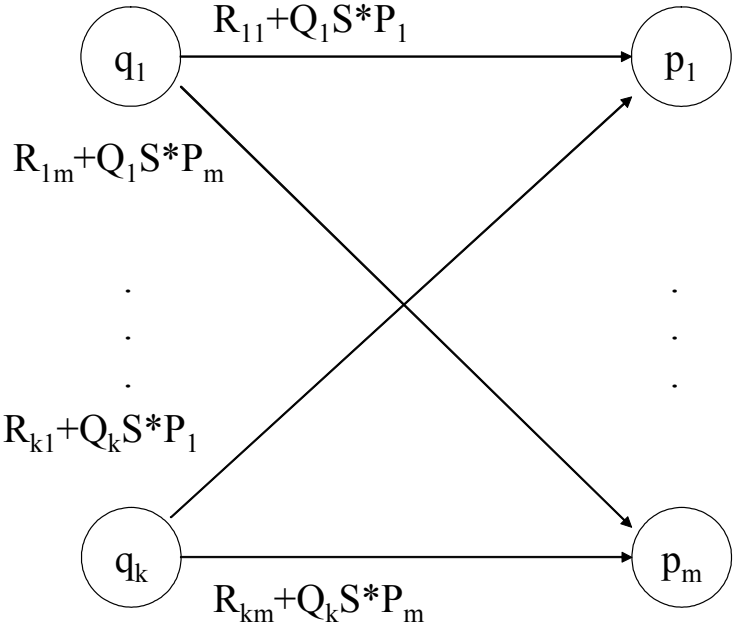
This approach involves eliminating states of the automaton and replacing the edges with regular expressions that includes the behavior of the eliminated states. When we get down to the situation with just a start and final node, then we can directly express the RE.

Consider the figure below, which shows a generic state s about to be eliminated. The labels on all edges are regular expressions.



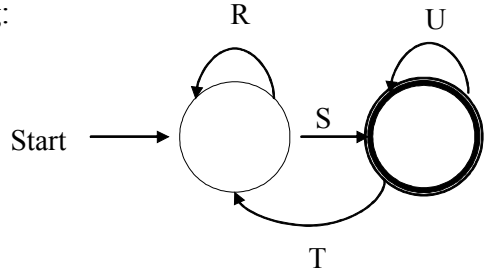
This graph shows all of the relevant edges to consider when we want to remove state S. To remove s, we must make labels from each q_i to p_1 up to p_m that include the paths we could have made through s.

These are shown below:



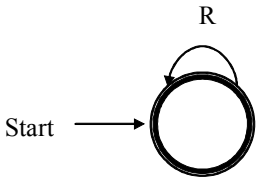
The steps to construct the RE from the finite automaton are as follows:

1. Starting with intermediate states and then moving to accepting states, apply the above reduction process to produce an equivalent automaton with regular expression labels on the edges. The result will be a one or two state automaton with a start state and accepting state.
2. If the two states are different, we will have an automaton that looks like the following:



   We can describe this automaton as: (R+SU*T)*SU*

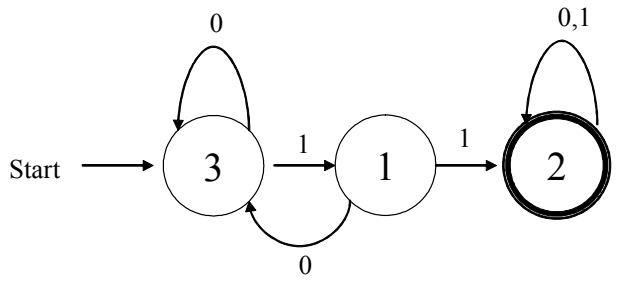
3. If the start state is also an accepting state, then we must also perform a state elimination from the original automaton that gets rid of every state but the start state. This leaves the following:



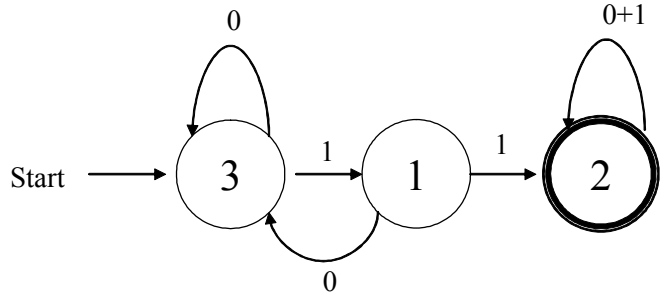
   We can describe this automaton as simply R*.

4. If there are n accepting states, we must repeat the above steps for each accepting states to get n different regular expressions, R_1, R_2, … R_n. Each time we turn any other accepting states to non-accepting states. The desired regular expression for the automaton is then the union of each of the n regular expressions: $R_1 \cup R_2 \ldots \cup R_N$

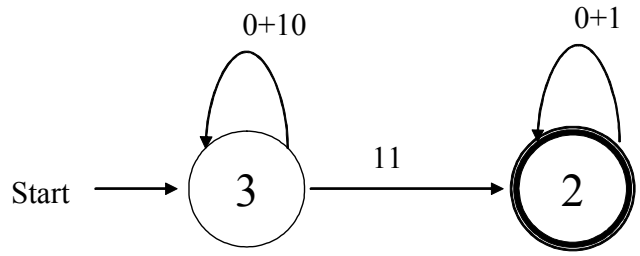
Example: Convert the same automaton we used in the inductive construction to a regular expression but using state elimination:



First convert the edges to RE's:



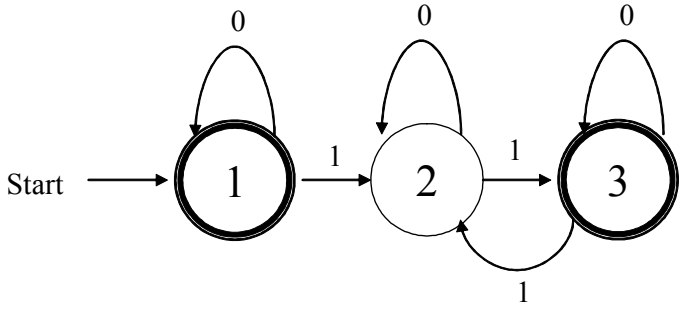
Next eliminate state 1. Note we must include the loop from 3 back to 3 via state 1:



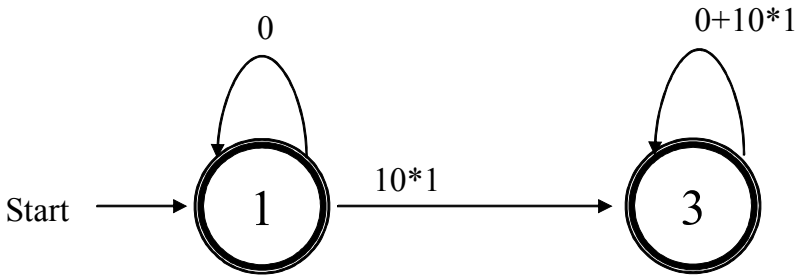
The RE is then: (0+10)*11(0+1)*

Note that it is possible to come up with a different RE using state elimination than the inductive method However, the two different regular expressions describe the same language.

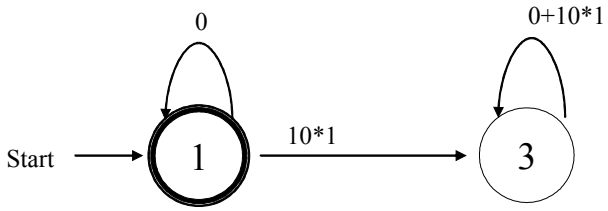
Here is another example, an automaton that accepts an even number of 1's:



Eliminate state 2:

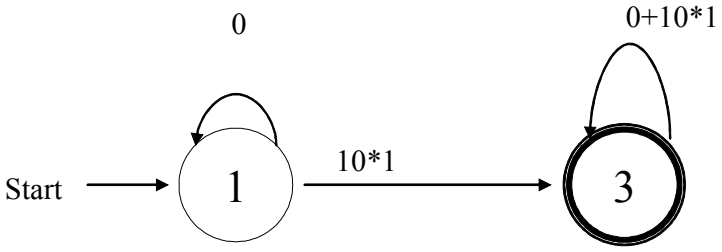


We have two accepting states, let's turn off state 3 first:



This is described by simply 0*; we can ignore going to state 3 since we would "die".

Next turn off state 1:



This is described by 0*10*1(0+10*1)*

The final regular expression is the union of both expressions which is then:

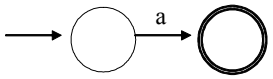
0* + 0*10*1(0+10*1)*

Converting Regular Expressions to Automata

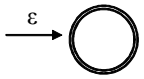
Finally, given a RE we can convert it to an automata. It is easiest to convert a RE to an ε-NFA using the constructions that are shown below. From the resulting ε-NFA we could then construct a DFA if we choose.

For the basis of defining an automata for primitive regular expressions:

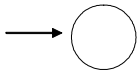
R=a



R=ε

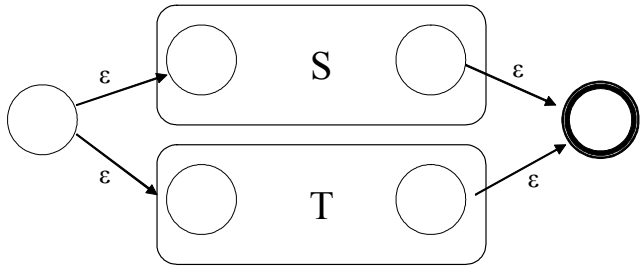


R=Ø

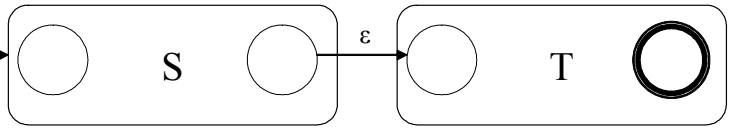


For more complex regular expressions:

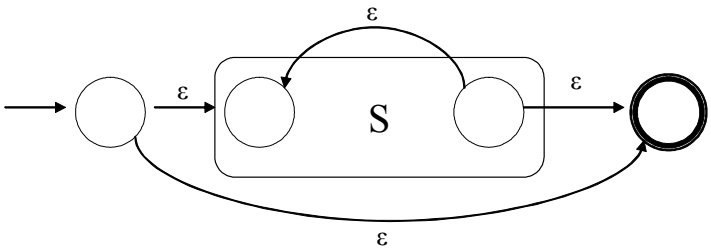
R=S+T



R=ST



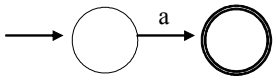
R=S*



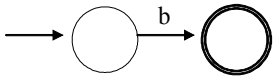
Given a regular expression, we simply piece it together using these constructions from the individual parts until we have an ε-NFA that represents the original RE.

Example: Convert R= (ab+a)* to an NFA. We proceed in stages below, starting with primitives and working our way up.

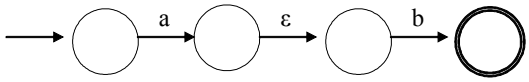
a



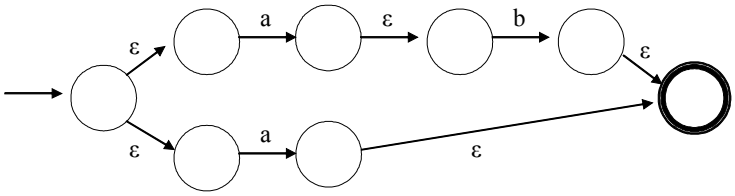
b



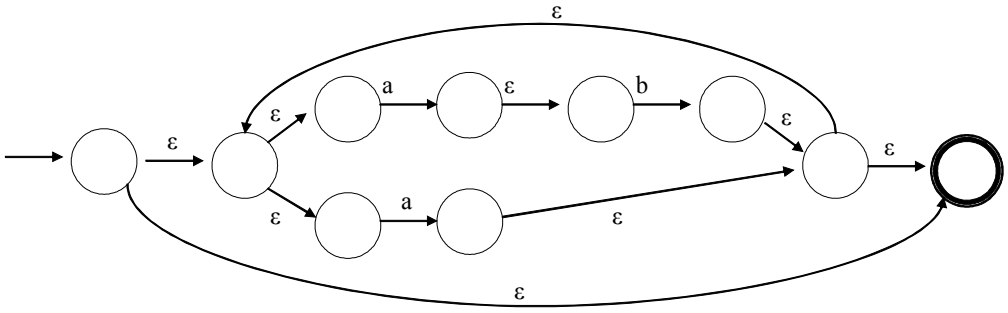
ab



ab+a



(ab+a)*



What have we shown? That regular expressions and finite state automata are really two different ways of expressing the same thing. In some cases you may find it easier to start with one and move to the other; for example, many people find it easier to construct the NFA for accepting the language of an even number of one's and convert that to a RE than to directly express the language as a RE.

Algebraic Laws for Regular Expressions

Just like we have an algebra for arithmetic, we also have an algebra for regular expressions. Note that while there are some similarities to arithmetic algebra, it is a bit different with regular expressions.

The following laws hold:

Commutative law for union: $L + M = M + L$
Associative law for union: $(L + M) + N = L + (M + N)$
Associative law for concatenation: $(LM)N = L(MN)$

Note that there is no commutative law for concatenation, i.e. $LM \neq ML$

The identity for union is: $L + Ø = Ø + L = L$
The identity for concatenation is: $Lε = εL = L$
The annihilator for concatenation is: $ØL = LØ = Ø$

Left distributive law: $L(M + N) = LM + LN$
Right distributive law: $(M + N)L = LM + LN$
Idempotent law: $L + L = L$

The following are laws involving closure:

$(L^*)^*$ = L^* , i.e. closing an already closed expression does not change the language
$Ø^* = ε$
$ε^* = ε$
$L^+ = LL^* = L^*L$ (more of a definition than a law)
$L^* = L^+ + ε$
$L? = ε + L$ (more of a definition than a law)

Checking A Law

Suppose we are told that the law $(R + S)^* = (R^*S^*)^*$ holds for regular expressions. How would we check that this claim is true?

We can use the "concretization" test:

- Think of R and S as if they were single symbols, rather than placeholders for languages, i.e., R = {0} and S = {1}.
- Test whether the law holds under the concrete symbols. If so, then this is a true law, and if not then the law is false.

For our example, then the left side is clearly any sequence of 0's and 1's. The right side also denotes any string of 0's and 1's, since 0 and 1 are each in L(0*1*).

This test is necessary (i.e., if the test fails, then the law does not hold.) and sufficient (if the test succeeds, the law holds). The book has a fairly simple argument for why, when the "concretized" expressions denote the same language, then the languages we get by substituting any languages for the variables are also the same.

However, extensions of the test beyond regular expressions may fail. Consider the "law" $L \cap M \cap N = L \cap M$.

This is clearly false, for example, let $L=M=\{a\}$ and $N=\emptyset$. But if $L=\{a\}$ and $M = \{b\}$ and $N=\{c\}$ then $L \cap M$ does equal $L \cap M \cap N$ which is empty. The test would say this law is true, but it is not because we are applying the test beyond regular expressions. We'll see soon various languages that do not have corresponding regular expressions.