Regular Expressions (Continued)

Precedencse Rules: The following conventions allow us to simplify our regexps considerably without introducing ambiguity:

- We leave out the outermost pair of parantheses. E.g., $(0+1)(11)^*$ is an abbreviation of $((0+1)(11)^*)$.
- Star operator has precedence over all operators. E.g., RS^* is an abbreviation of $R(S^*)$
- Concatenation has precedence over union. E.g., $RS^* + T$ is an abbreviation of $((RS^*) + T)$
- When the same binary operator is applied several times in a row, we can leave out the parantheses and assume the grouping is to the right. E.g., $11 + 01 + 10 + 11$ is an abbreviation of $(11 + (01 + (10 + 11)))$

Equivalence of Regexps: Regexps R and S are equivalent (denoted $R \equiv S$) iff they represent the same language (i.e., $L(R) = L(S)$), e.g., $b^*a(a+b)^* \equiv (a+b)^*ab^*$.

Theorem 1. *The general regexps R, S and T, the following equivalences hold:*

- **Comutativity of union:** $R + S \equiv S + R$
- **Associativity of union:** $(R + S) + T \equiv R + (S + T)$
- **Associativity of concatenation:** $(RS)T \equiv R(ST)$
- **Left distributivity:** $R(S + T) \equiv RS + RT$
- **Right distrbutivity:** $(S + T)R \equiv SR + TR$
- **Identity for union:** $R + \{\} \equiv R$
- **Identity for concatenation:** $\epsilon R \equiv R \equiv R\epsilon$
- **Annihilator for concatenation:** $\{\}R \equiv \{\} \equiv R\{\}$
- **Idempotence of Kleene star:** $R^{*^*} \equiv R^*$

Example 1. *We prove that $L(b^*a(a+b)^*) = L = \{$all strings of a's and b's that contain at least one $a\}$, by showing double inclusion (standard technique for proving set equality).*

Intuition: *Stating $L(b^*a(a+b)^*) = L$ amounts to making two separate claims.*

1. *Every string in $L(b^*a(a+b)^*)$ has at least one a (i.e., RE pattern does not include bad strings)*
2. *Every string with at least one a belongs to $L(b^*a(a+b)^*)$ (i.e., RE pattern includes every good string).*

Proof. Now, let's prove both parts.

1. ($L(b^*a(a+b)^*)$ subset of L): Let s be an arbitrary string in $L(b^*a(a+b)^*)$. This means $s = t \circ u \circ v$ for some strings $t \in L(b^*)$, $u \in L(a)$, and $v \in L((a+b)^*)$. Since there is only one string $a \in L(a)$, $u = a$ so $s = t \circ a \circ v$ and s is a string that contains at least one a, so $s \in L$.

2. (L subset of $L(b^*a(a+b)^*)$): Let s be an arbitrary string in L. This means that s contains at least one a, so it contains a first occurrence of a and can be broken up into three substrings: $s = r \circ a \circ t$, where r is some string that contains no a (maybe empty), a is the first occurrence of a in s, and t is some string of a's and b's. But then, $r \in L(b^*)$, $a \in L(a)$, and $t \in L((a+b)^*)$ so by definition, $s = r \circ a \circ t$ is in $L(b^*a(a+b)^*)$. □

Remark. *See textbook for other detailed examples.*

Nondeterministic Finite State Automata (NFA or NFSA)

Assume that you want to construct a DFA that accepts the following language

$$L = \{s \in \{a, b\}^* : \text{ s ends with babb}\}$$

The DFA for this language must remember the last 4 symbols processed. As we saw in our tutorial, this requires all possible combinations of the last 4 characters (16 of them). We should consider all possible combinations because in a DFA, a given state and the current input symbol uniquely determines the next state of the automaton. It is for this reason that such automata are called deterministic. But if we remove the determinism constraint, the following FSA accepts L:

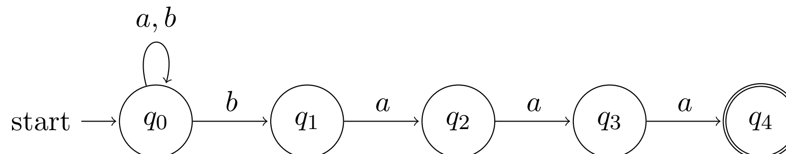

Figure 1:

Remark. *Note that if a string does not end with babb, then every attempt to follow transition out of q_0 (in Figure 1) ends up in empty set of states (one of the transitions won't work).*

Notice the simplicity of FSA in Figure 1. Such properties have lead to the definition of a variant of finite state automata, called nondeterministic finite state automata (NFA or NFSA). In these FSAs, given the current state, when the automaton reads an input symbol a, there may be several states to which it may go next (hence the nondeterminism).

NFA or NFSA: A nondeterministic finite state automaton is a quintuple $(Q, \Sigma, q_0, F, \delta)$, where Q is a fixed, finite, non-empty set of states. Σ is a fixed (finite, non-empty) alphabet ($Q \cap \Sigma = \{\}$). $q_0 \in Q$ is the initial state. $F \subseteq Q$ is the set of accepting ("final") states. $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is a transition function (i.e., $\delta(q, a)$ is the set of next states of the NFA when processing symbol a from state q)

Note: $\mathcal{P}(Q)$ is the power set of Q.

We can see that the definition of the NFA contains transitions like $\delta(q, \epsilon)$. These transitions are called **spontaneous** state transition or ϵ-transition, in which the NFA makes a transition from the current state to the next state without reading any input symbol. The NFA can be defined without the introduction of ϵ-transitions by extending the defintion of initial state to a set of states rather than a state. However, ϵ-transitions will allow us to simplify our notations and arguments in some cases (e.g., when we talk about closure properties).

Remark. *The power of NFA is that, by definition, NFA accepts a string iff set of states reached at the end contains at least one accepting state. It is like saying that NFA has unlimited parallelism.*

Subset Construction: Given a NFA $M = (Q, \Sigma, q_0, F, \delta)$, we can construct a DFA $M' = (Q', \Sigma, q'_0, F', \delta')$ that accepts the same language as M as follows:

- $Q' = P(Q)$
- $q'_0 = \mathcal{E}(q_0)$ (i.e., the set of all states reachable from the initial state of the given NFSA via ϵ-transitions only)
- $F' = \{q' \in Q' : q' \cap F \neq \emptyset\}$ (i.e., all states that contain an accepting state of the given NFSA)
- For any $q' \in Q$ and $a \in \Sigma$, $\delta'(q', a) = \cup_{q_x \in q'} \left(\cup_{q_y \in \delta(q_x, a)} \mathcal{E}(q_y)\right)$ where $\mathcal{E}(q_y)$ is the set of states reachable from q_y following any number of ϵ transitions.

This construction is called the subset construction, because each state of M' is a set of states of M

Example 2. *Consider the following NFA:*

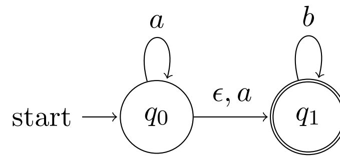

Figure 2: NFA corresponding to regexp a^*b^*

The corresponding DFA using the subset construction is:

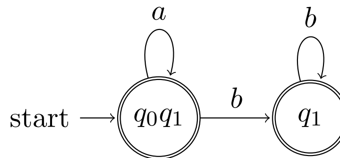

Figure 3: Correponding DFA of NFA in Figure 2

Remark. *Although NFA may introduce unlimited parallelism. But it is not a practical model!*

Closure properties

Let's construct a FSA that accepts the language

$$L = \{s \in \{a, b\}^* : s \text{ contains three } a\text{'s in a row and an even number of } b\text{'s } \}$$

Another way to express L is to say

$$L = \{s : s \text{ contains } aaa\} \cap \{s : s \text{ contains even many } b\text{'s}\}$$

Each of these sub-languages correspond to a FSA as follows:

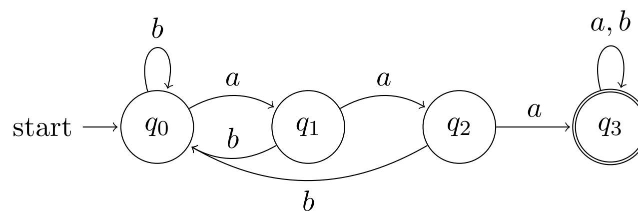

Figure 4: FSA for $\{s : s \text{ contains } aaa\}$ (FSA1)

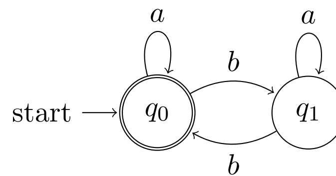

Figure 5: FSA for $\{s : s \text{ contains even many } b\text{'s}\}$ (FSA2)

Now, let's try to combine the states in FSA1 and FSA2 so that the resulting states can track the states in both of the aforementioned FSAs at the same time. The resulting FSA will look like follows (q_{xy} is a state that represents state q_x in FSA1 and state q_y in FSA2):

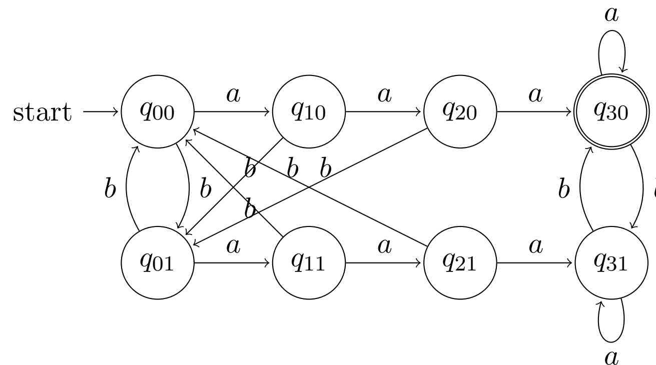

Figure 6: FSA for $\{s : s \text{ contains even many } b\text{'s}\}$ (FSA2)

It should be obvious now that the only accepting state in this FSA should be q_{30} in which we have seen three a's and even number of b's.

The aforementioned example demonstrates a powerful design technique by which we can combine FSAs that accept languages to obtain an FSA that accepts the resulting language of the combination.

Closure Property: Let R and S represent two languages that are accepted by FSA_R and FSA_S respectively. If an operation that is applied to R and S results in a language T for which there exists a FSA (FSA_T) that decides language T, we say that the class of languages accepted by FSA is closed under this operation

Theorem 2. *The class of languages that are accepted by FSA is closed under* **complementation, union, intersection, concatenation** *and the* **Kleene star** *operation. In other words, if L and L' are languages that are accepted by FSA, then so are all of the following: $\bar{L}$, $L \cap L'$, $L \cup L'$, $L \circ L'$ and $L^{\circledast}$.*

Regular Languages

Theorem 3. *Let L be a language. The following statements are equivalent:*

1. *$L = L(A)$ for some NFA A*
2. *$L = L(A')$ for some DFA A'*
3. *$L = L(R)$ for some regexp R*

We are not going to prove this theorem. However, we are going to talk about the main ideas of the proof. You can look at Sections 7.4.2 and Sections 7.6 in the textbook for a foraml treatment of this theorem.