Equivalence of Regular Languages and FSMs

Read K & S 2.4
Read Supplementary Materials: Regular Languages and Finite State Machines: Generating Regular Expressions from Finite State Machines.
Do Homework 8.

Equivalence of Regular Languages and FSMs

Theorem: The set of languages expressible using regular expressions (the regular languages) equals the class of languages recognizable by finite state machines. Alternatively, a language is regular if and only if it is accepted by a finite state machine.

Proof Strategies

Possible Proof Strategies for showing that two sets, *a* and *b* are equal (also for iff):

1. Start with *a* and apply valid transformation operators until *b* is produced.

Example:
Prove:
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

$A \cap (B \cup C)$	$= (B \cup C) \cap A$	commutativity
	$= (B \cap A) \cup (C \cap A)$	distributivity
	$= (A \cap B) \cup (A \cap C)$	commutativity

2. Do two separate proofs: (1) a $\Rightarrow$ b, and (2) b $\Rightarrow$a, possibly using totally different techniques. In this case, we show first (by construction) that for every regular expression there is a corresponding FSM. Then we show, by induction on the number of states, that for every FSM, there is a corresponding regular expression.

For Every Regular Expression There is a Corresponding FSM

We'll show this by construction.

Example:

$a^*(b \cup \varepsilon)a^*$

Review - Regular Expressions

The regular expressions over an alphabet Σ^* are all strings over the alphabet $\Sigma \cup \{(,), \varnothing, \cup, *\}$ that can be obtained as follows:

1. $\varnothing$ and each member of Σ is a regular expression.
2. If α , β are regular expressions, then so is $\alpha\beta$.
3. If α , β are regular expressions, then so is $\alpha \cup \beta$.
4. If α is a regular expression, then so is α^*.
5. If α is a regular expression, then so is (α).
6. Nothing else is a regular expression.

We also allow ε and α^+, etc. but these are just shorthands for $\varnothing^*$ and $\alpha\alpha^*$, etc. so they do not need to be considered for completeness.

For Every Regular Expression There is a Corresponding FSM

Formalizing the Construction: The class of regular languages is the smallest class of languages that contains ∅ and each of the singleton strings drawn from Σ, and that is closed under

- Union
- Concatenation, and
- Kleene star

Clearly we can construct an FSM for any finite language, and thus for ∅ and all the singleton strings. If we could show that the class of languages accepted by FSMs is also closed under the operations of union, concatenation, and Kleene star, then we could recursively construct, for any regular expression, the corresponding FSM, starting with the singleton strings and building up the machine as required by the operations used to express the regular expression.

FSMs for Primitive Regular Expressions

An FSM for ∅:

An FSM for ε (∅*):

An FSM for a single element of Σ:

Closure of FSMs Under Union

To create a FSM that accepts the union of the languages accepted by machines M1 and M2:

1. Create a new start state, and, from it, add ε-transitions to the start states of M1 and M2.

Closure of FSMs Under Concatenation

To create a FSM that accepts the concatenation of the languages accepted by machines M1 and M2:

1. Start with M1.
2. From every final state of M1, create an ε-transition to the start state of M2.
3. The final states are the final states of M2.

Closure of FSMs Under Kleene Star

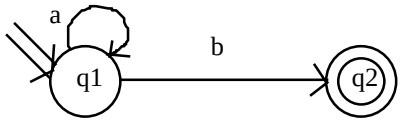
To create an FSM that accepts the Kleene star of the language accepted by machine M1:
1. Start with M1.
2. Create a new start state S0 and make it a final state (so that we can accept ε).
3. Create an ε-transition from S0 to the start state of M1.
4. Create ε-transitions from all of M1's final states back to its start state.
5. Make all of M1's final states final.

Note: we need a new start state, S0, because the start state of the new machine must be a final state, and this may not be true of M1's start state.

Closure of FSMs Under Complementation

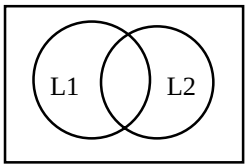
To create an FSM that accepts the complement of the language accepted by machine M1:
1. Make M1 deterministic.
2. Reverse final and nonfinal states.

A Complementation Example



Closure of FSMs Under Intersection

$L1 \cap L2 =$



Write this in terms of operations we have already proved closure for:

- Union
- Concatenation
- Kleene star
- Complementation

An Example

$(b \cup ab^*a)^*ab^*$

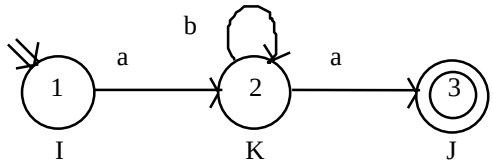
For Every FSM There is a Corresponding Regular Expression

Proof:
(1) There is a trivial regular expression that describes the strings that can be recognized in going from one state to itself ({ε} plus any other single characters for which there are loops) or from one state to another directly (i.e., without passing through any other states), namely all the single characters for which there are transitions.

(2) Using (1) as the base case, we can build up a regular expression for an entire FSM by induction on the number assigned to possible intermediate states we can pass through. By adding them in only one at a time, we always get simple regular expressions, which can then be combined using union, concatenation, and Kleene star.

Key Ideas in the Proof

Idea 1: Number the states and, at each induction step, increase by one the states that can serve as intermediate states.

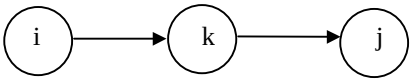


Idea 2: To get from state I to state J without passing through any intermediate state numbered greater than K, a machine may either:
1. Go from I to J without passing through any state numbered greater than K-1 (which we'll take as the induction hypothesis), or
2. Go from I to K, then from K to K any number of times, then from K to J, in each case without passing through any intermediate states numbered greater than K-1 (the induction hypothesis, again).

So we'll start with no intermediate states allowed, then add them in one at a time, each time building up the regular expression with operations under which regular languages are closed.

The Formula

Adding in state k as an intermediate state we can use to go from i to j, described using paths that don't use k:



R(i, j, k) = R(i, j, k - 1) /* what you could do without k

∪

R(i, k, k-1) /* go from i to the new intermediate state without using k or higher

∘

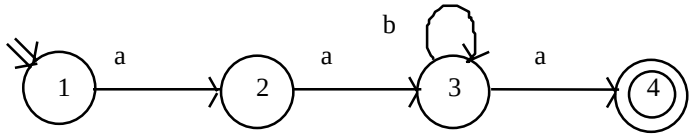
R(k, k, k-1)* /* then go from the new intermediate state back to itself as many times as you want

∘

R(k, j, k-1) /* then go from the new intermediate state to j without using k or higher

Solution: $\bigcup R(s, q, N) \ \forall q \in F$

An Example of the Induction



Going through no intermediate states:

(1,1,0) = ε (1,2,0) = a (1, 3, 0) = ∅ (2,3,0) = a (3,3,0) = ε ∪ b (3,4,0) = a

Allow 1 as an intermediate state:

Allow 2 as an intermediate state:

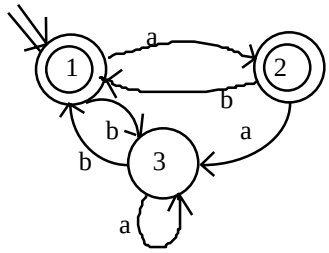
(1, 3, 2) = (1, 3, 1) ∪ (1, 2, 1)(2, 2, 1)*(2, 3, 1)
= ∅ ∪ a ε* a
= aa

Allow 3 as an intermediate state:

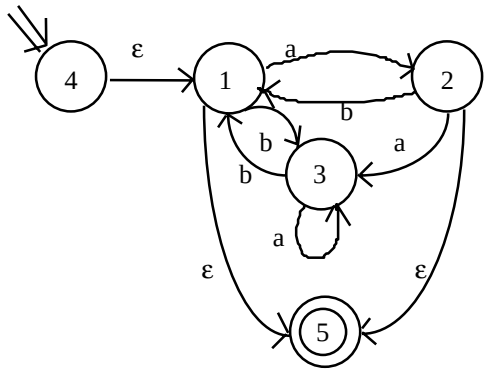
(1, 3, 3) = (1, 3, 2) ∪ (1, 3, 2)(3, 3, 2)*(3, 3, 2)
= aa ∪ aa (ε ∪ b)* (ε ∪ b)
= aab*

(1, 4, 3) = (1, 4, 2) ∪ (1, 3, 2)(3, 3, 2)*(3, 4, 2)
= ∅ ∪ aa (ε ∪ b)* a
= aab*a

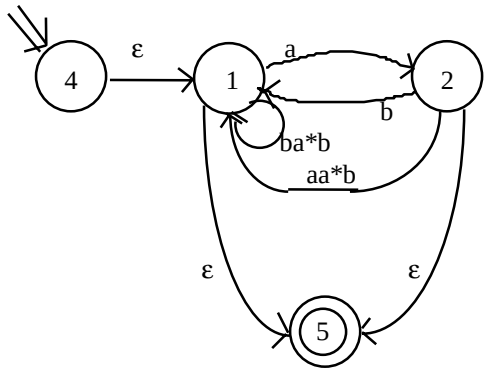
An Easier Way - See Packet



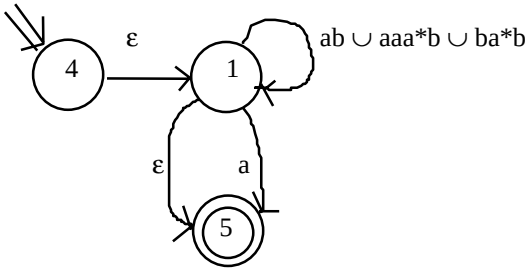
(1) Create a new initial state and a new, unique final state, neither of which is part of a loop.

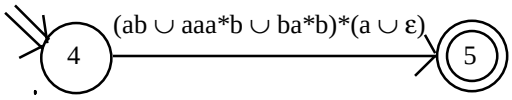


(2) Remove states and arcs and replace with arcs labelled with larger and larger regular expressions. States can be removed in any order, but don't remove either the start or final state.



(Notice that the removal of state 3 resulted in two new paths because there were two incoming paths to 3 from another state and 1 outgoing path to another state, so 2×1 = 2.) The two paths from 2 to 1 should be coalesced by unioning their regular expressions (not shown).





Thus, the equivalent regular expression is:

(ab ∪ aaa*b ∪ ba*b)*(a ∪ ε)

Using Regular Expressions in the Real World (PERL)

Matching floating point numbers:

-? ([0-9]+(\.[0-9]*)? | \.[0-9]+)

Matching IP addresses:

([0-9]+ (\ . [0-9]+) {3})

Finding doubled words:

\< ([A-Za-z]+) \s+ \1 \>

From Friedl, J., Mastering Regular Expressions, O'Reilly,1997.

Note that some of these constructs are more powerful than regular expressions.

Regular Grammars and Nondeterministic FSAs

Any regular language can be defined by a regular grammar, in which all rules

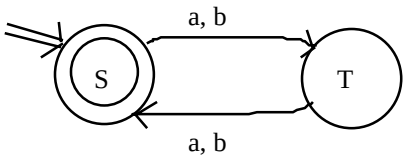
- have a left hand side that is a single nonterminal
- have a right hand side that is ε, a single terminal, a single nonterminal, or a single terminal followed by a single nonterminal.

Example: $L=\{w \in \{a, b\}^* : |w| \text{ is even}\}$

$((aa) \cup (ab) \cup (ba) \cup (bb))^*$

$S \rightarrow \varepsilon$
$S \rightarrow aT$
$S \rightarrow bT$

$T \rightarrow a$
$T \rightarrow b$
$T \rightarrow aS$
$T \rightarrow bS$



An Algorithm to Generate the NDFSM from a Regular Grammar

1. Create a nonterminal for each state in the NDFSM.
2. s is the start state.
3. If there are any rules of the form $X \rightarrow w$, for some $w \in \Sigma$, then create an additional state labeled #.
4. For each rule of the form $X \rightarrow w\ Y$, add a transition from X to Y labeled w ($w \in \Sigma \cup \varepsilon$).
5. For each rule of the form $X \rightarrow w$, add a transition from X to # labeled w ($w \in \Sigma$).
6. For each rule of the form $X \rightarrow \varepsilon$, mark state X final.
7. Mark state # final.

Example 1 - Even Length Strings

$S \rightarrow \varepsilon$
$S \rightarrow aT$
$S \rightarrow bT$

$T \rightarrow a$
$T \rightarrow b$
$T \rightarrow aS$
$T \rightarrow bS$

Example 2 - One Character Missing

$S \rightarrow \varepsilon$
$S \rightarrow aB$
$S \rightarrow aC$
$S \rightarrow bA$
$S \rightarrow bC$
$S \rightarrow cA$
$S \rightarrow cB$

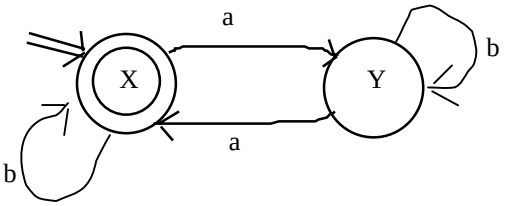
$A \rightarrow bA$
$A \rightarrow cA$
$A \rightarrow \varepsilon$
$B \rightarrow aB$
$B \rightarrow cB$
$B \rightarrow \varepsilon$

$C \rightarrow aC$
$C \rightarrow bC$
$C \rightarrow \varepsilon$

An Algorithm to Generate a Regular Grammar from an NDFSM

1. Create a nonterminal for each state in the NDFSM.
2. The start state becomes the starting nonterminal
3. For each transition $\delta(T, a) = U$, make a rule of the form $T \rightarrow aU$.
4. For each final state T, make a rule of the form $T \rightarrow \varepsilon$.

Example:



Conversion Algorithms between Regular Language Formalisms