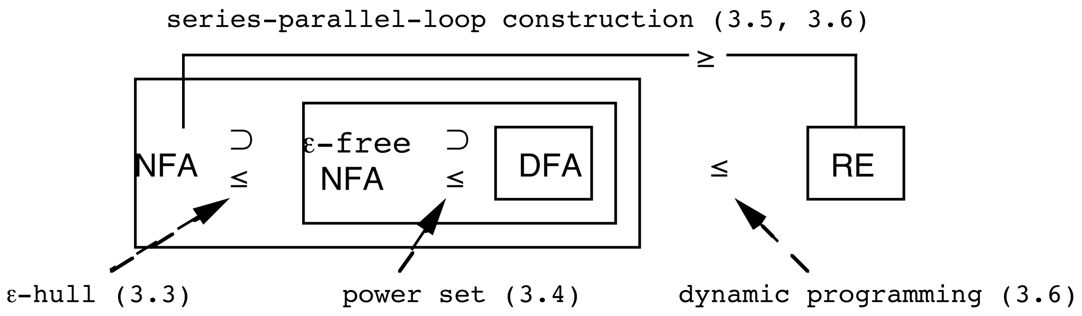
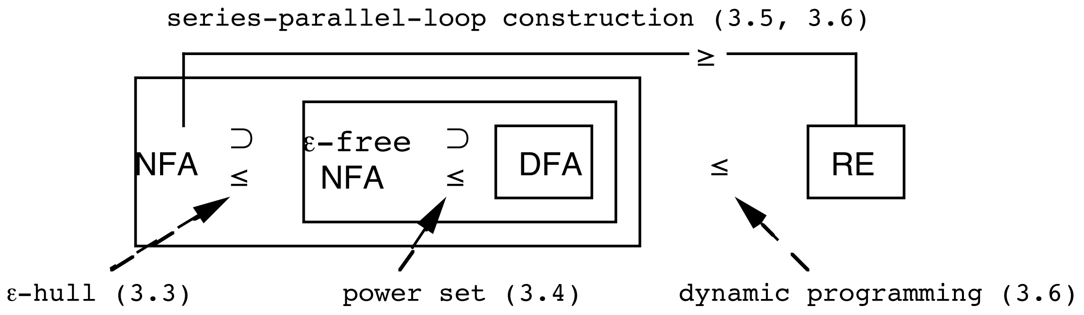

series-parallel-loop construction (3.5, 3.6)
NFA
ε-free NFA
DFA
RE
ε-hull (3.3)
power set (3.4)
dynamic programming (3.6)

jn 2004.04.16

3. Finite automata and regular languages: theory

Among the various formal languages that arise naturally in the theory of automata and phrase structure grammars, regular languages are the simplest and arguably the most useful. They are universally used in text searching applications, and they appear as important components of practically all the formal notations developed by computer science, such as those parts of programming languages that can be processed by lexical analysis. Regular languages can be defined in three distinct ways: as the languages accepted by deterministic finite automata (DFA), by non-deterministic finite automata (NFA), and by regular expressions (RE). Each of these formalisms sheds light on different properties of regular languages. The main goal of this chapter is to show the equivalence of DFAs, NFAs, and regular expressions, using the cyclic reduction shown below, where $X \leq Y$ means "whatever can be expressed by X, can also be expressed by Y". Since NFAs are a generalization of DFAs, denoted by s et inclusion $\supset$, it is clear that they are at least as powerful as DFAs - the interesting comparisons are those indicated by "$\leq$". From the equivalence of these different models of computation, all the major properties of regular languages follow readily.


series-parallel-loop construction (3.5, 3.6)
NFA
ε-free NFA
DFA
RE
ε-hull (3.3)
power set (3.4)
dynamic programming (3.6)

3.1 Varieties of automata: concepts, definitions, terminology

Notation: Alphabet A , e.g. $A = \{0, 1, ..\}$. Kleene star $A^* = \{ w \mid w = a1\ a2\ ...\ am, m \geq 0, ai \in A\}$.

$A^+ = AA^*$. Nullstring ε. Empty set $\{\}$ or $\emptyset$.

"Language" $L \subseteq A^*$. Set $S = \{s, s', .., s0, s1, ..\}$. Cardinality $|S|$. Power set $\mathbf{2^S}$. "$\neg$" not or complement.

Deterministic Finite Automaton (FA, DFA) $M = (S, A, f, s0, F)$
Set of states S, alphabet A, transition function $f: S \times A \to S$, initial state s0, accepting or final states $F \subseteq S$
Extend f from $S \times A \to S$ to $f: S \times A^* \to S$ as follows: $f(s, \varepsilon) = s$, $f(s, wa) = f(f(s, w), a)$ for $w \in A^*$
Df: **M accepts w** $\in A^*$ iff $f(s0, w) \in F$. Set $L \subseteq A^*$ accepted by M: $\mathbf{L(M)} = \{ w \mid f(s0, w) \in F\}$.

Non-deterministic Finite Automaton (NFA) **with ε-transitions: $\mathbf{f: S \times (A \cup \{\varepsilon\}) \to 2^S}$**.
Special case: NFA without ε-transitions: $\mathbf{f: S \times A \to 2^S}$.

Extend $f: S \times A^* \to 2^S$: $f(s, \varepsilon)$ = ε-hull of s = all states reachable from s via ε-transitions (including s);
$f(s, wa) = \cup f(s', a)$ for $s' \in f(s, w)$.

Extend f further $f: 2^S \times A^* \to 2^S$ as follows: $f(s1, .., sk, a) = \cup f(si, a)$ for $i = 1, .., k$.
Df: **M accepts w** $\in A^*$ iff $f(s0, w) \cap F \neq \{\}$. Notice: w is accepted iff $\exists$ some w-path from s0 fo F.
Set $\mathbf{L \subseteq A^*}$ accepted by M: L(M) $= \{ w \mid f(s0, w) \cap F \neq \{\}\}$.

The operation of a non-deterministic machine can be interpreted in 2 equivalent ways:
- Oracle: For any given word w, if there is any sequence of transitions that leads to an accepting state, the machine will magically find it, like a sleep walker, avoiding all the paths that lead to a rejecting state.
- Concurrent computation: The machine spawns multiple copies of itself, each one tracing its own root-to-leaf

path of the tree of all possible choices. If any copy reaches an accepting state, it broadcasts success. Non-determinism yields an **exponential increase in computing power**!

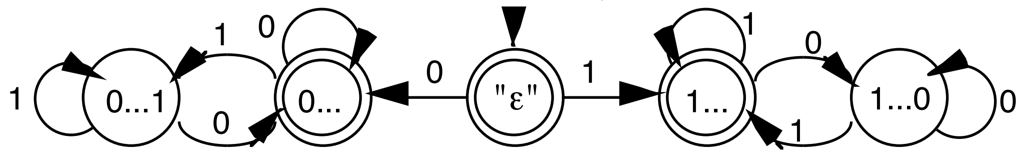
Df: Two FAs (of any type) are **equivalent** iff they accept the same language.

HW 3.1:
Design a) an NFA and b) a DFA that accepts the language $L = (0 \cup 1)^*01$ of all strings that terminate in 01.
Given an arbitrary string w over {0, 1}, describe a general rule for constructing
a) an NFA N(w) and b) a DFA M(w) that accepts the language $L(w) = (0 \cup 1)^*w$.

3.2 Examples: the case for non-determinism

Ex 1: **FAs "can't count"**, or more precisely, a DFA M can count up to a constant which is at most equal to the number |M| of its states. The fixed, finite memory capacity is the most important property of FAs, and it severely limits their computing power. As an example, we show that no FA can recognize $L = \{0^k1^k \mid k > 0\}$. By way of contradiction, assume ∃ FA M that accepts L, and denote M's number of states by |M| = n. In the course of accepting $w = 0^n1^n$, as M reads the prefix 0^n, it goes thru n+1 states s0, s1, .., sn. By the "pigeon hole principle", some 2 states si, sj in this sequence must be equal, si = sj, i < j. Thus, M cannot distinguish the prefixes 0^i and 0^j, and hence also accepts, incorrectly, $w' = 0^{n-(j-i)}1^n$, and many other ill-formed strings. Contradiction, QED.

Ex 2: **Careful about "can't count"!** Let $L = \{w \in \{0 \cup 1\}^* \mid \#(01) = \#(10)\}$. In any w, "ups" = 01 and "downs" = 10 alternate, so #(01) = #(10) ± 1. Solution: L = {w | first character = last character }.

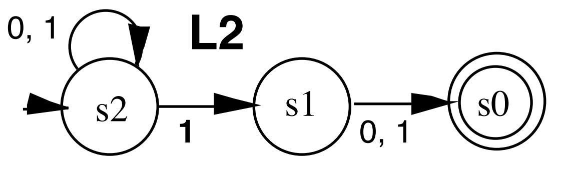


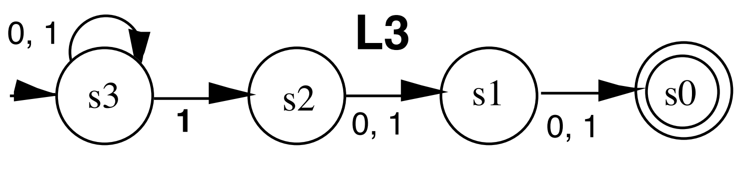
Exercise: We saw a 2-state fsm serial adder. Show that there is no fsm multiplier for numbers of arbitrary size.

DFAs are conceptually straightforward: "when in state s you read input a, go to state s'(and perhaps produce some output". NFAs can be considerably trickier, as the number of action sequences they might produce explodes. Three main reasons argue for introducing non-determinism despite the added complications. The most general reason is that non-deterministic algorithms are interesting and important, and the theory should be able to model them. A second reason becomes apparent when developing the complexity theory towards the end of this course, where the "P vs. NP" question relates the two most important complexity classes. The third reason becomes apparent in the course of this chapter: NFAs are a very convenient technique to desing FAs and prove their properties.

Ex 3: NFA's clairvoyance yields an exponential reduction of the size of the state space as compared to DFAs.

Consider $L_k = (0 \cup 1)^* 1 (0 \cup 1)^{k-1}$ i.e. all strings whose k-th last bit is 1. A NFA accepts L_k using only k+1 states (as shown for k = 2 and k = 3) by "guessing" where the tail-end k bits start.

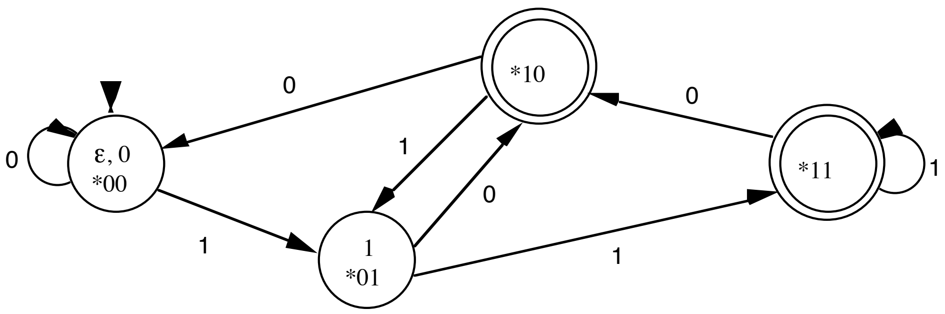




A DFA that accepts L_k must contain a shift register k bits long, hence has at least 2^k states. This shows that, in general, simulating a NFA by a DFA requires an **exponential increase in the size of the state space.**

The following DFA $M(L_2)$ has a state for each of the 2-bit suffixes 00, 01, 10, 11. Each state s corresponds to a "language", i.e. a set of words that lead M from its initial state to s. The short strings ε , 0, 1 can be associated with some set of "long" strings with the following semantics:

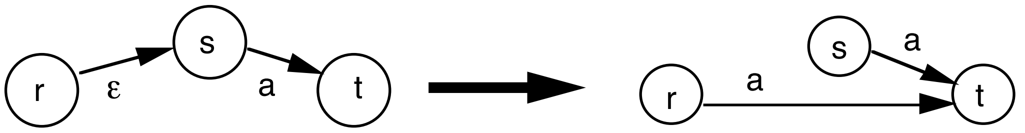
ε, 0, *00:	no "1" has been seen that might be useful as the next-to-last bit
1, *01:	the current bit is a "1", if this turns out to be the next-to-last bit, we must accept
*10:	accept if this is the end of the input; if not, go to state *00 or *01 depending on the next bit read
*11:	accept if this is the end of the input; if not, go to state *10 or *11 depending on the next bit read



This DFA M(L_2) is a subset of of a DFA M(L_2) with 8 states that is obtained from the general construction of section 3.4 to simulate a NFA N by some DFA M.

3.3 Spontaneous transitions: convenient, but not essential

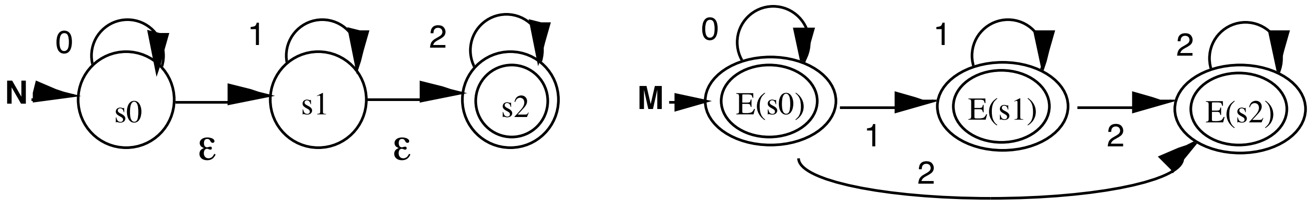
Lemma (ε-transitions):
Any NFA N with ε-transitions can be converted to an equivalent NFA M without ε-transitions.
The basic idea is simple, as illustrated below. An ε-transitions from r to s implies that anything that N can do starting in s, such as an a-transition from s to t, N can already do starting in r. After adding appropriate new transitions, such as the a-transition from r to t, the ε-transition from r to s can be deleted.



Df: the ε-closure E(s) of a state s is the set of states that can be reached from s following ε-transitions.
In the example above: E(r) = {r, s}, E(s) = {s}, E(t) = {t}. The significance of this concept is due to the fact that whenever N reaches s, it might also reach any state in E(s) without reading any input.

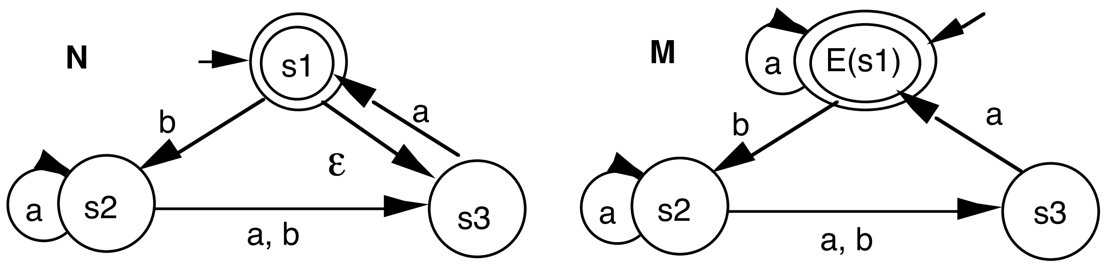
The following example Ex4 illustrates the general construction that transforms a NFA N with ε-transitions (at left) into an equivalent NFA M without ε-transitions (at right).

Ex 4: L = { $0^i\ 1^j\ 2^k$ | i, j, k ≥ 0} = 0*1*2*. This language is typical of the structure of communication protocols. A message consists of a prefix, a body, and a suffix, in this order. If any part may be of arbitrary length, including zero length, the language of legal messages has the structure of L.



Replace each state s of N by a state E(s) of M: E(s0) = {s0, s1, s2}, E(s1) = {s1, s2}, E(s2) = {s2}. This reflects the fact that whenever N got to s0, it might spontaneously have proceeded to s1 or to s2. E(s0) is the starting state of M. Any state E(s) of M that contains an accepting state of N must be made an accepting state of M - in our example, all the states of M are accepting! Finally, we adjust the transitions:
since E(s0) = {s0, s1, s2}, M's transition function f assigns to f(E(s0), 2) the union of all of N's transitions outgoing from s0, s1, s2: f(s0, 0), f(s1, 1), f(s2, 2).
Despite the fact that all of M's states are accepting, M only accepts the strings in L. The first symbol '2' of the string '21', for example, leads M from E(s0) to E(s2), but E(s2) cannot process the secons symbol '1', hence '21' is not accepted.

Ex 5: NFA N converted to an ε-free NFA M. E(s1) = {s1, s3}. L = (a ∪ ba* (a ∪ b) a)*

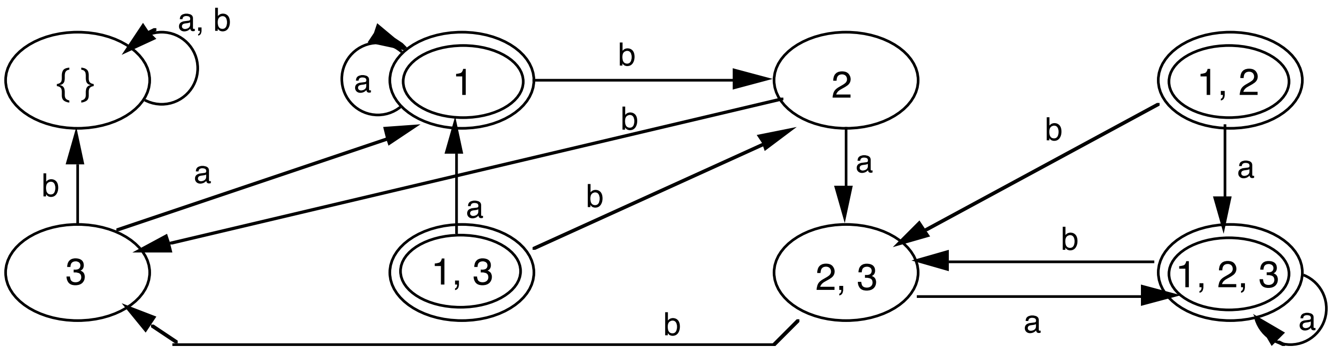


3.4 DFA simulating NFA: the power set construction.

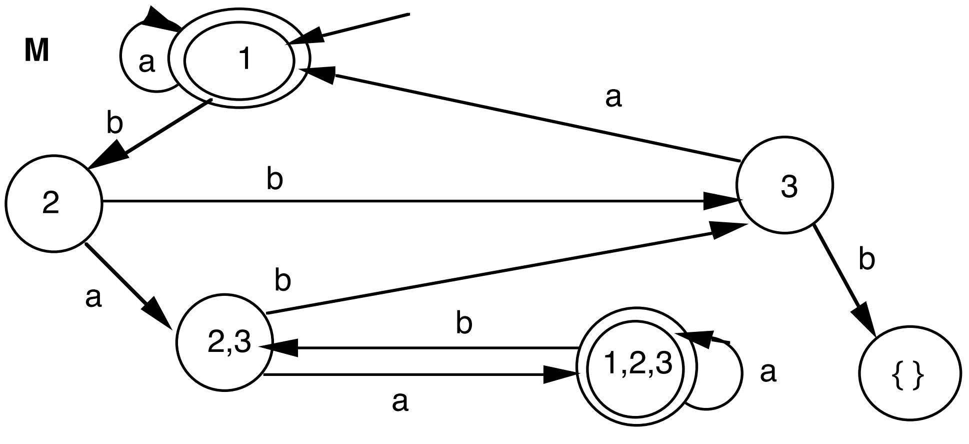
Thm (equivalence NFA-DFA): Any NFA N can be converted into an equivalent DFA M.

Pf: Thanks to Lemma on ε-transitions, assume without loss of generality that N = (S, A, f, s0, F) has no ε-transitions. Define M = (2^S, A, f', {s0}, F') as follows. 2^S is the power set of S, {s0}the initial state. F' consists of all those subsets R ⊆ S that contain some final state of N i.e. R ∩ F ≠ {}. f': 2^S x A -> 2^S is defined as:
for R ∈ 2^S and a ∈ A, f'(R, a) = { s ∈ S | s ∈ f(r, a) for some r ∈ R}.
N and M are equivalent due the following invariant: for all x ∈ A*, f'({s0}, x) = f(s0, x). QED

Ex (modified from Sipser p57-58). Convert the NFA of Ex 5 (at right) in section 3.3 to an equivalent DFA.



The power set construction tends to introduce unreachable states. These can be eliminated using a transitive closure algorithm. Alternatively, we generate states of M only as the corresponding subsets of S are being reached, thus combining transitive closure with the construction of the state space. For ease of comparison, let's redraw the DFA above without unreachable states, with a layout similar to Ex5. The non-determinism in example Ex5: f(s2, a) = {s2, s3} is resolved by introducing states {s2, s3} and { s1, s2, s3}:



3.5 Regular expressions

Df: Given an alphabet A, the class R(A) of regular expressions over A is obtained as follows:
Primitive expressions: a for ever a ∈ A, ε (nullstring), Ø (empty set).
Compound expressions: if R, R' are regular expressions, (R ∪ R'), (R ∘ R'), (R*) are regular expressions.
Convention on operator priority: * > ∘ > ∪. Use parentheses as needed to define structure.
A regular expression denotes a regular set by associating the expression operators *, ∘ , ∪ with the set operations Kleene star, catenation, and union, respectively.

Thm: A set L is regular iff L is described by some regular expression.

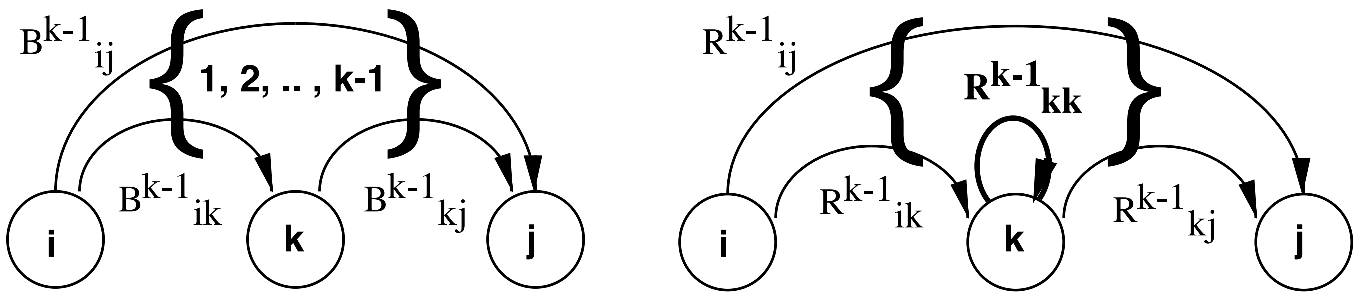
Pf <=: Convert a given regular expression R into an NFA N. Use trivial NFAs for the primitive expressions, the constructions used in the Closure Thm for the compound expressions. QED

Pf =>(McNaughton, Yamada 1960. Compare: Warshall's transitive closure algorithm, 1962):
Let DFA M = (S, A, f, s1, F) accept L. S = { s1, .. , sn}. Define R^k_{ij} = the set of all strings w that lead M from state si to state sj **without passing thru any state with label > k**.
Initialize: R^0_{ij} = { a | f(si, a) = sj } for i ≠ j. R^0_{ii} = { a | f(si, a) = si ∪ { ε }.
Induction step: $R^k_{ij} = R^{k-1}_{ij} \cup R^{k-1}_{ik} \ (R^{k-1}_{kk})^* \ R^{k-1}_{kj}$
Termination: R^n_{ij} = the set of all strings w that lead M from state si to state sj without any restriction.
L(M) = $\cup R^n_{1j}$ for all sj ∈ F. The right hand side is a regular expression that denotes L(M). QED

Intuitive verification. Remember Warshall's transitive closure and Floyd's "all distances" algorithms.

Warshall	Floyd
$B^0_{ij} = A_{ij}$ adjacency matrix, B^0_{ii} = true	$B^0_{ij} = A_{ij}$ edge length matrix, $B^0_{ii} = 0$
$B^k_{ij} = B^{k-1}_{ij}$ or (B^{k-1}_{ik} and B^{k-1}_{kj})	$B^k_{ij} = \min (B^{k-1}_{ij} , B^{k-1}_{ik} + B^{k-1}_{kj})$
$B^n_{ij} = C_{ij}$ connectivity matrix	$B^n_{ij} = D_{ij}$ distance matrix

In Warshall's and Floyd's algorithms, cycles are irrelevant for the issue of connectedness and harmful for computing distances. Regular expressions, on the other hand, describe **all** paths in a graph (state space), in particular the infinitely many cyclic paths. Thus, we add a loop R^{k-1}_{kk} in the Fig. at right, and insert the regular expression $(R^{k-1}_{kk})^*$ between R^{k-1}_{ik} and R^{k-1}_{kj}.



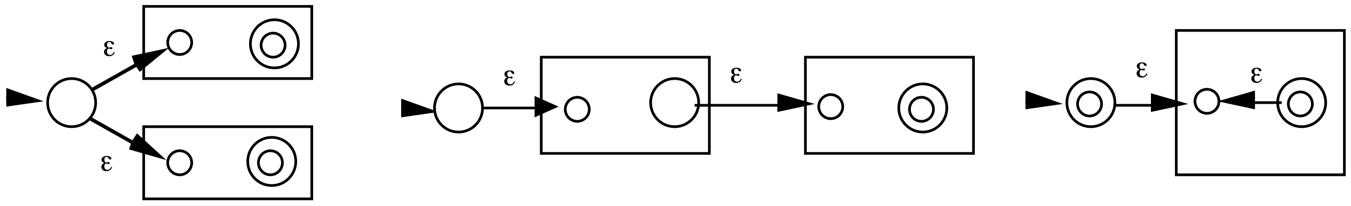
3.6 Closure of the class of regular sets under union, catenation, and Kleene star

Df: **A language (or set) L ⊆ A* is** called **regular iff L is accepted by some FA.**
It turns out that all FAs (DFA or NFA, with or without ε-transitions) are equivalent w.r.t. "accepting power".

Given L, L' ⊆ A*, define union L ∪ L', catenation L ∘ L' = { v = ww' | w ∈ L, w' ∈ L'}.
Define $L^0 = \{ \varepsilon \}$, $L^k = L \circ L^{k-1}$ for k > 0. Kleene star: $L^* = \cup (k = 0 \text{ to } \infty)\ L^k$.

Thm (closure under the regular operations):

If L, L' ⊆ A* are regular sets, L ∪ L', L ∘ L' and L* are regular sets.
Pf: Given FAs that accept L, L' respectively, construct NFAs to accept L ∪ L', L' ∘ L' and L* as shown. The given FAs are represented as boxes with starting state at left (small) and one accepting state (representative of all others) at right (large). In each case we add a new starting state and some ε-transitions as shown.



In addition to closure under the regular operations union, catenation, and Kleene star, we have:

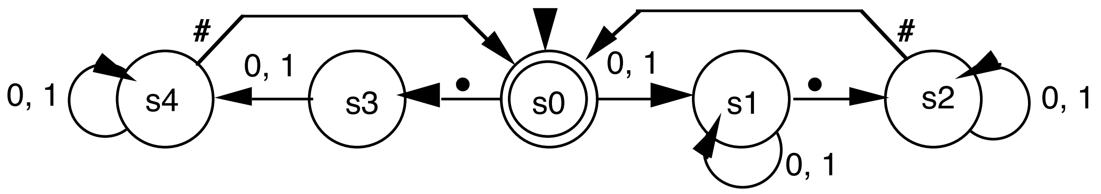
Thm: if L is regular, the complement ¬L is also regular.
Pf: Take a **D**FA M = (S, A, f, s0, F) that accepts L. M' = (S, A, f, s0, S-F) accepts ¬L. QED

Thm: If $L, L' \subseteq A^*$ are regular, the intersection $L \cap L'$ is also regular. Pf: $L \cap L' = \neg(\neg L \cup \neg L')$. QED

Thus, the class of regular languages over an alphabet A forms a Boolean algebra.

3.7 DFAs and right invariant equivalence relations. State minimization.

Ex "real constants": $A = \{0, 1, \bullet, \#\}$. $L = \big(((0 \cup 1)^+ \bullet (0 \cup 1)^* \cup (0 \cup 1)^* \bullet (0 \cup 1)^+) \# \big)^*$
Interpret a word in L as a sequence of real constants with a mandatory binary point •, e.g 0•1, 11•, •1.
A constant must have at least one bit 0 or 1, the binary point alone is excluded. Constants are separated by #.
To get a DFA, imagine that the transitions not shown in the figure all lead to a non-accepting trap state s5.



State identification, equivalent states: Given the state diagram of a DFA M, devise an experiment to
1) determine the current state of M, or 2) to distinguish two given states r, s.

Ex: In order to identify s0, feed ε into M - no other state is accepting. '•#' uniquely identifies s1.
'#' distinguishes between s3 and s4. No experiment distinguishes s2 from s4: s2 and s4 are equivalent.
Equivalent states can be merged to obtain a smaller FA M' equivalent to M.

Df: States r and s of M are equivalent (indistinguishable) iff for all $w \in A^*$, $f(r, w) \in F \Leftrightarrow f(s, w) \in F$.
It turns out that in order to prove 2 states equivalent, it suffices to test all words w of length $|w| \leq n = |S|$.

Before proving this result, consider a dynamic programming algorithm to identify non-equivalent state pairs. We start with the observation that all states might be equivalent. As pairs of non-equivalent states are gradually being identified, we record for each such pair s, r a shortest witness that distinguishes s and r. We illustrate this algorithm using the example of the FA "real constants" above.

At left in the figure below, all state pairs $si \neq sj$ are marked that can be distinguished by some word of length 0. This distinguishes accepting states from non-accepting states, and ε is a shortest witness. Unmarked slots identify pairs that have not yet been proven distinguishable. For each of these unmarked pairs r, s, and all $a \in A$, check whether the pair f(r, a), f(s, a) has been marked distinguishable. If so, mark r, s distinguishable with a shortest witness w = aw', where w' is inherited from f(r, a), f(s, a). When computing the entry for s1, s3 at right, for example, notice that f(s1, B) = s1, f(s3, B) = s4. Since s1, s4 have already been proven distinguishable by w' = #, s1, s3 are distinguishable by w = B#.
Checking the last unmarked pair s2, s4 at right yields no new distinguishable pair: f(s2,#) = f(s4,#) = s0; f(s2, •) = f(s4, •) = trap state s5; f(s2,B) = s2, f(s4,B) = s4, but s2, s4 have not yet been proven distinguishable. This terminates the process with the information that s2, s4 are equivalent and can be merged. Distinguishable states obviously cannot be merged -> this is a state minimization algorithm.

	s1	s2	s3	s4
s0	ε	ε	ε	ε
s1				
s2				
s3				

|w| = o

	s1	s2	s3	s4
s0	ε	ε	ε	ε
s1		#		#
s2			#	
s3				#

|w| = 1

	s1	s2	s3	s4
s0	ε	ε	ε	ε
s1		#	0#	#
s2			#	
s3				#

|w| = 2

Hw 3.2.: Invent another interesting example of a DFA M with equivalent states and apply this dynamic programming algorithm to obtain an equivalent M' with the minimum number of states.
Hw 3.3: Analyze the complexity of this dynamic programming algorithm in terms of |S| = n and |A|.
Hw 3.4: Prove the following Thm: If states r, s are indistinguishable by words w of length $|w| \leq n = |S|$, r and s are equivalent. Hint: use the concepts and notations below, and prove the lemmas.

Df "r, s are indistinguishable by words of length ≤ k":
$r \sim_k s$ for $k \geq 0$ iff for all $w \in A^*$, $|w| \leq k$: $f(r, w) \in F \Leftrightarrow f(s, w) \in F$
Important properties of of the equivalence relations $\sim_k$:
Lemma (inductive contruction): $r \sim_k s$ iff $r \sim_{k-1} s$ and for all a: $f(r, a) \sim_{k-1} f(s, a)$
Lemma (termination): If $\sim_k = \sim_{k-1}$, $\sim_k = \sim_m$ for all $m > k$.
Thm: If r, s are indistinguishable by words w of length $|w| \leq n = |S|$, r and s are equivalent.

An algebraic approach to state minimization

Given any $L \subseteq A^*$, define the equivalence relation (reflexive, symmetric, transitive) R_L over A^*:
$x R_L y$ iff All $z \in A^*$, $xz \in L \Leftrightarrow yz \in L$. I.e., either xz and yz both in L, or xz and yz both in $\neg L$
Notice: R_L is "right invariant": $x R_L y \Rightarrow$ All $z \in A^*$, $xz R_L yz$.
Intuition: $x R_L y$ iff the prefixes x and y cause all pairs xz, yz to share [non-]membership status w.r.t. L.

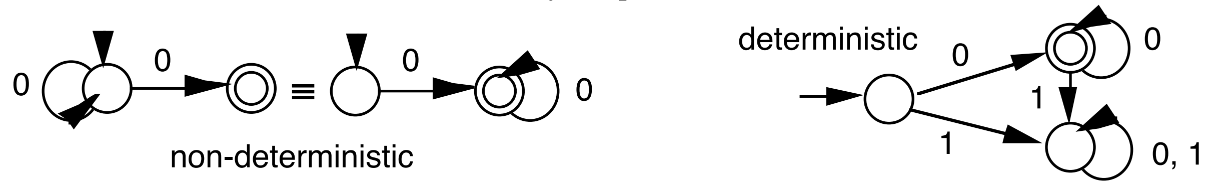
Given DFA M, define equivalence relation R_M over A^*: $x R_M y$ iff $f(s0, x) = f(s0, y)$. R_M is right invariant.

Df: index of equivalence relation R = # of equivalence classes of R.

Thm (regular sets and equivalence relations of finite index). The following 3 statements are equivalent:
1) $L \subseteq A^*$ is accepted by some DFA M
2) L is the union of some of the equivalence classes of a right invariant equivalence relation of finite index
3) R(L) is of finite index.

Thm: The minimum state DFA accepting L is unique up to isomorphism (renaming states).

In contrast, minimum state **N**FAs are not necessarily unique. Ex $A = \{0, 1\}$, $L = 0^+$:



3.8 Odds and ends about regular languages and FAs

The "pumping lemma" (iteration lemma):
For any regular $L \subseteq A^*$ there is an integer $n > 0$ (the "pumping length") with the following property:
any $w \in L$ of length $|w| \geq n$ can be sliced into 3 parts $w = xyz$ satisfying the following conditions:
1) $|xy| \leq n$, 2) $|y| > 0$, **3) for all $i \geq 0$, $x y^i z \in L$.**

Pf: Consider any DFA M that accepts L, e.g. the minimum state DFA M(L). Choose $n = |S|$ as the pumping length. Feed any $w \in L$ of length $|w| \geq n$ into M. On its way from s0 to some accepting state, M goes through $|w| + 1 \geq n + 1$ states. Among the first n+1 of these states, s0, s1, .., sn, .., there must be a duplicate state si = sj for some $i < j$, with a loop labeled y leading from si back to si. Thus, xz, xyz, xyyz, ... are all in L. QED

The pumping lemma is used to prove, by contradiction, that some language is **not** regular.
Ex: $L = \{ 0^i 1^j \mid i < j \}$ is **not** regular. Assume L is regular. Let n be L's pumping length. Consider $w = 0^n 1^{n+1}$, $w \in L$. Even if we don't know how to slice w, we know $|xy| \leq n$ and hence $y = 0^k$ for some $k > 0$. But then $0^n 1^{n+1}, 0^{n+k} 1^{n+1}, 0^{n+2k} 1^{n+1}$, .. are all $\in L$, contradicting the definition $L = \{ 0^i 1^j \mid i < j \}$. L is not regular, QED.

End of Ch3