String Theory
(Computer Science Version)

Robert M. Keller
Harvey Mudd College
March 2013

Why are we doing this?

- Strings are **fundamental** to most formal systems (e.g. grammars).
- Strings **generalize natural numbers** (which are themselves very important).
- Strings are **easy** to work with compared to set-theoretic definitions of natural numbers.

Informal Definition of Strings

- A **string** is just a sequence $x_1x_2....x_n$ where $n \geq 0$, with each x_i being a member of some common **alphabet** Σ.
- The elements of a string are called “letters”.
- While infinite strings are sometimes used, for now we will be working only with **finite** strings.

Formal Definition of Strings

- A string over the set of letters Σ is any member of the following inductively defined set, named Σ^* :
  - The empty string, represented by ε, is in Σ^*.
  - If a string x is in Σ^*, and σ is in Σ, then σx is in Σ^*.
  - The only elements of Σ^* are those obtained by applying the above rules.
- By σx we mean a combination of σ with x in such a way that both can be recovered from the result. This can be informally thought of as the "followed by" operator (similar to $[\sigma \mid x]$ in Prolog).

Notes on the Empty String

- The empty string symbol ε (upper-case lambda) is a "meta symbol" and as such is never an ordinary letter in Σ.
  - Other symbols are often seen in place of ε:
    - λ (lower-case lambda)
    - Λ (upper-case lambda)

Strings vs. Natural Numbers

- Natural numbers N can be viewed as a special case of strings over a 1-letter alphabet.
- Let say the letter is just '1'.
- Then the connection is suggested by:

N	{1}*
0	ε
1	1
2	11
3	111
4	1111
n	1^n (n 1's in a row)

String Concatenation

- Concatenation means “chaining together”, i.e. following one string by another.
- Concatenation will be shown by juxtaposition:
  xy is y concatenated to x.
- Addition in N can be viewed as a special case of concatenation in {1}*.
- Some texts take concatenation as a primitive operation.

String Concatenation

- Concatenation is a function or binary operator on Σ^* and is defined inductively:
  - $\varepsilon y = y$ — basis
  - $(\sigma x)y = \sigma(xy)$ — induction step
- On the left-hand sides above is the thing we are defining, and on the right how we are defining it.
- This can be seen as analogous to defining *append* in rex or Scheme.

Why define by induction?

- Definition by induction is **precise**, compared to other alternatives.
- Definition by induction enables **proof** by induction.

String Axioms

- These axioms characterize strings, analogously to the way in which the Peano axioms characterize the natural numbers.
- In the following x, y, ... are implicitly quantified over strings
- σ, σ', ... are implicitly quantified over letters.

String Axioms (similar to Peano axioms)

- SA1: $(\forall x)\ (\forall \sigma)\ \sigma x \neq \varepsilon$
- SA2: $(\forall x, x')\ (\forall \sigma, \sigma')$
  $\sigma x = \sigma' x'$ implies $\sigma = \sigma'$ and $x = x'$
- SA3(Induction): Let P(x) be any formula with free variable x.

$$(P(\Lambda) \wedge \forall x\ (P(x) \rightarrow \forall \sigma\ P(\sigma x))) \rightarrow \forall x\ P(x)$$

Example of Inductive Proof

- ST1: $(\forall x)\ x\varepsilon = x$, in other words, ε concatenated to any string is just that string.

- For proof, we identify P(x) in SA3 with $x\varepsilon = x$.
- SA3 says that ST1 follows if we can show two things:
  - $P(\varepsilon)$, i.e. $\varepsilon\ \varepsilon = \varepsilon$. basis
  - $(\forall x)(P(x) \Rightarrow (\forall \sigma)P(\sigma x))$,
    i.e. $(\forall x)(x\varepsilon = x \Rightarrow (\forall \sigma)\ (\sigma x)\varepsilon = \sigma x)$ induction step
- How are these two statements shown?

Basis

- ST1’: $(\forall x)\ \varepsilon x = x$
- This in itself requires a secondary induction proof. Let Q(x) be $\varepsilon x = x$.
- Basis: Q(ε): $\varepsilon\varepsilon = \varepsilon$

Another Example of Inductive Proof

- ST2: $(\forall x, y, z)\ x(yz) = (xy)z$. In other words, concatenation is associative.
- Here we identify P(x) in SA4 with $x(yz) = (xy)z$.
- SA4 says that ST2 follows if we can show two things:
  - $P(\Lambda)$, i.e. $\varepsilon\ (yz) = (\varepsilon y)\ z$. — basis
  - $(\forall x)(P(x) \Rightarrow (\forall \sigma)P(\sigma x))$,

    i.e. $(\forall x)(x(yz) = (xy)z \Rightarrow (\forall \sigma)\ (\sigma x)(yz) = ((\sigma x)y)z$ — induction step
- How are these two statements shown?

Defining String Reversal

- We want to define inductively reversal (^{R} operator)
- Here's an intuitively-correct idea:
  - $\varepsilon^R = \varepsilon$ basis
  - $(\sigma x)^R = x^R(\sigma\varepsilon)$ induction step
- Note: We have to use $\sigma\varepsilon$ rather than just σ because $x^R\sigma$ is not defined. But since $\sigma\varepsilon$ is a string, we can concatenate.
- Note that we are using concatenation to define this operator. This could be considered somewhat heavy-handed.

Defining String Reversal

- Another option would be to use an auxiliary function r:
  - $x^R = r(x, \varepsilon)$, where
    - $r(\varepsilon, y) = y$ basis
    - $r(\sigma x, y) = r(x, \sigma y)$ induction step
- Here we haven't used any overt concatenation, and this makes us feel better.
- We could now try to prove that the two versions of reversal are equivalent. We'll table this.

Proving stuff about reversal (^{R} operator)

- ST3: $(xy)^R = y^R x^R$
- Try induction with P(x): $(\forall y)(xy)^R = y^R x^R$
- Basis: P(ε): $(\forall y)(\varepsilon y)^R = y^R \varepsilon^R$
- Induction step: $(\forall y)(xy)^R = y^R x^R$

  implies $(\forall \sigma)(\forall y)((\sigma x)y)^R = y^R (\sigma x)^R$
- LHS of = is: $((\sigma x)y)^R$
  $= (\sigma(xy))^R$
  $= (xy)^R (\sigma\varepsilon)$
  $= (y^R x^R) (\sigma\varepsilon)$
- RHS of = is: $y^R (\sigma x)^R$
  $= y^R (x^R(\sigma\varepsilon))$
  $= (y^R x^R) (\sigma\varepsilon)$

Justify each step

Free the Monoids!

- Algebraically, Σ^* is known as the "free monoid generated by Σ".
  (usually serves to deter the casual reader from going any further)
- Recall that a monoid is a set together with:
  - An associative operator
  - An identity for the operator
- Identify the components that justify calling Σ^* a monoid.

Natural Numbers

- Pick Σ to be any 1-letter alphabet. (e.g. use $\varnothing$ as a letter).
- Then Σ^* is essentially the set of natural numbers:
  - ε is like 0
  - σx is like x+1
- The string axioms are then equivalent to the Peano axioms.

- The induction rule is the usual mathematical induction principle.

Natural Numbers

- Addition is just concatenation over Σ^*.
- What are subtraction, multiplication, etc.?
- You'd need to define them inductively: more on this later.

Natural Numbers from Zip

- 0 is $\varnothing$ — basis
- n+1 is n $\cup$ {n} — induction step

for example

- 1 is $\varnothing \cup \{\varnothing\}$ which is $\{\varnothing\}$ which has 1 element
- 2 is $\{\varnothing\} \cup \{\{\varnothing\}\}$ which is $\{\varnothing, \{\varnothing\}\}$ which has 2 elements
- 3 is $\{\varnothing, \{\varnothing\}\} \cup \{\{\varnothing, \{\varnothing\}\}\}$ which is $\{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}$ which has 3 elements, etc.
- ...

- Let ω designate the set of natural numbers.

The Length | | of a String

- $|\varepsilon| = 0$ basis
- $|\sigma x| = x+1$ induction step

Some Properties of Length

- L1: $|xy| = |x| + |y|$
- L2: $|x^R| = |x|$

Machines and Languages

Robert Keller

March 2013

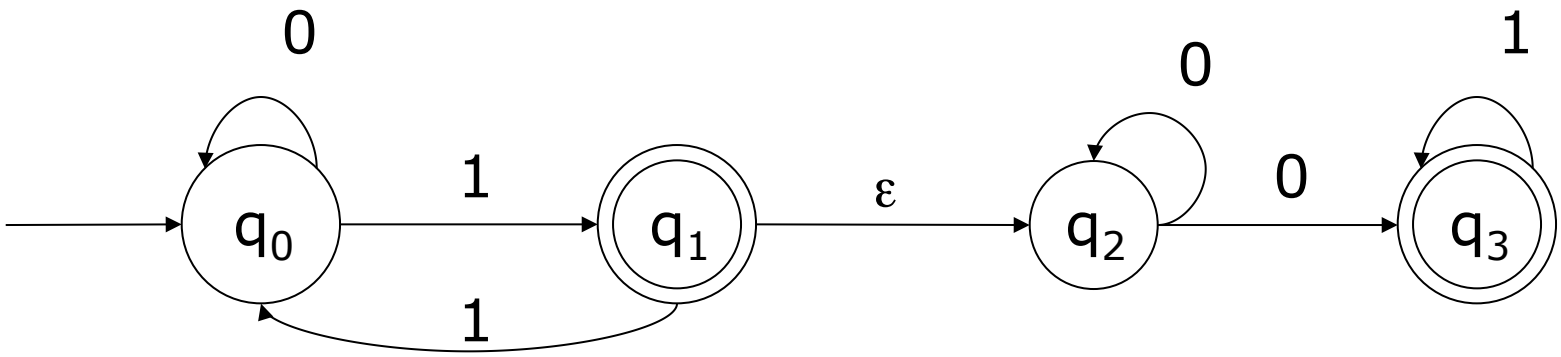
Machines

- By a **machine** over an alphabet Σ, we mean
  - a collection of **states** Q
    together with
  - a **transition relation** $\longrightarrow \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$
    - $q - \sigma \rightarrow q'$ means that (q, σ, q') is in the relation
  - an **initial state** $q_0 \in Q$
  - and a set of **accepting states** $F \subseteq Q$.

Behavior of a Machine

- A machine **starts** in state q_0.
- From a current state q it can **change state** to q' **with input** σ provided that $q - \sigma \rightarrow q'$.
- From a current state q it can **change state** to q' **spontaneously** provided that $q - \varepsilon \rightarrow q'$.
- The machine **accepts** a string $x \in \Sigma^*$ provided there is *some* path from q_0 to *some* $q \in F$ that **spells out** x.

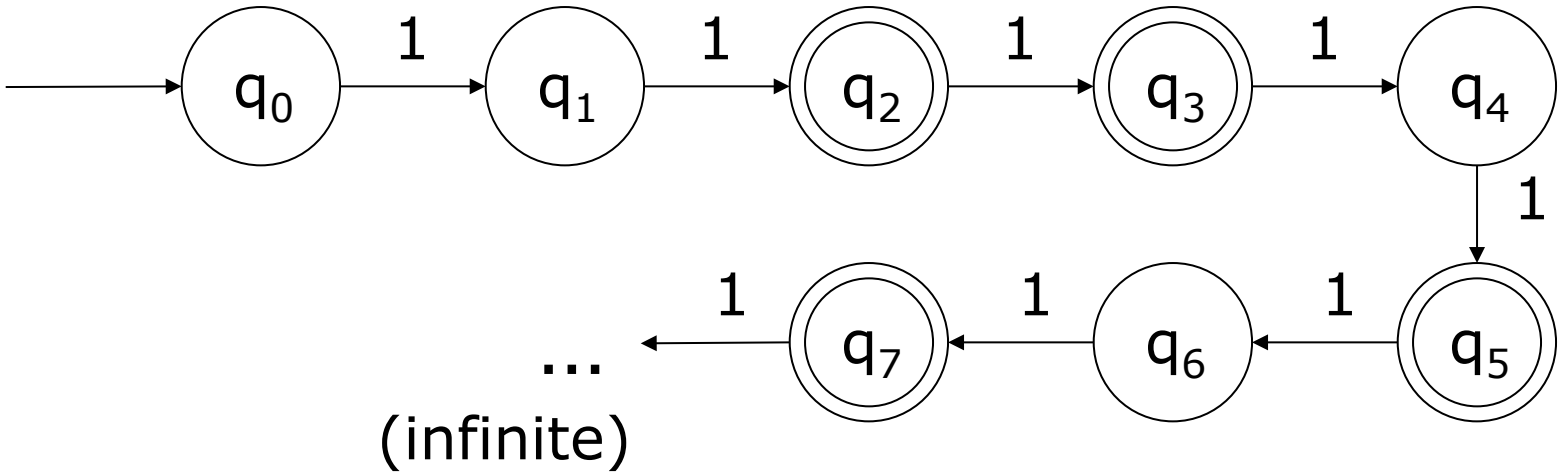
Example of a Machine



Accepted: 0 0 1 0 0 1

Not accepted: 1 0 1 0

Another Example of a Machine



Accepted: 1^p where p is prime

Not accepted: 1^q where q is composite

Languages for Machines

- If M is a machine, then L(M) is the **language accepted by** M, defined as the set of finite strings spelled out by all paths from the initial state to some accepting state.

The Language of a Machine *State*

- If q is a state, then the language L_q is defined to be the set of strings spelled out in going from q to some accepting state.

- Hence the language for the initial state is the language for the machine.

Equivalence of States

- Two states are defined to be **equivalent**

  $q \equiv q'$

  iff their languages are **equal**:

  $L_q = L_{q'}$.

Equivalence Relations Review

- $\equiv$ is an equivalence relation, meaning:
  - $\forall q \in Q\ (\ q \equiv q\ )$ [We drop the $\in Q$ for brevity below.]
  - $\forall q\ \forall q'\ (\ q \equiv q' \rightarrow q' \equiv q\ )$
  - $\forall q\ \forall q'\ \forall q''\ (\ (\ q \equiv q' \wedge q' \equiv q''\ ) \rightarrow q \equiv q''\ )$

Partitions

- Every equivalence relation determines a partition on Q.
- A partition is a set of subsets of Q such that:
  - No two subsets intersect.
  - The union of the subsets is all of Q.
- The partition determined by $\equiv$ is given by $\{\{q' \mid q' \equiv q\} \mid q \in Q\}$.
- The elements of the partition, sets of the form $\{q' \mid q' \equiv q\}$ are called the **equivalence classes** of $\equiv$.

Partition Determines Relation

- Every partition determines an equivalence relation:

  If P is a partition, then define: $q' \equiv q$ iff $\exists S \in P$ ($q \in S$ and $q' \in S$).

- Verify that the 3 equivalence relation properties hold.

Rank of a Partition

- The rank of a partition is just the number of equivalence classes.
- If the state set is finite, the rank of the corresponding equivalence partition is also finite.
- If the state set is infinite, the rank could still be finite.

Machines for Languages

- A language $L \subseteq \Sigma^*$ can be viewed as a machine:
  - The states are elements of Σ^*.
  - The initial state is ε.
  - The transitions are $x - \sigma \rightarrow x\sigma$.
  - The accepting states are the elements of L.
  - As with any machine, there are languages L_x for each x in Σ^*. Incidentally, $L_x = \{ z \mid xz \in L \}$

Equivalence of Strings modulo a Language

- With respect to a language L, two strings are equivalent

  $x \equiv_L y$

  iff the languages of their corresponding states are equal:

  $L_x = \{ z \mid xz \in L \} = \{ z \mid yz \in L \} = L_y$

Congruence

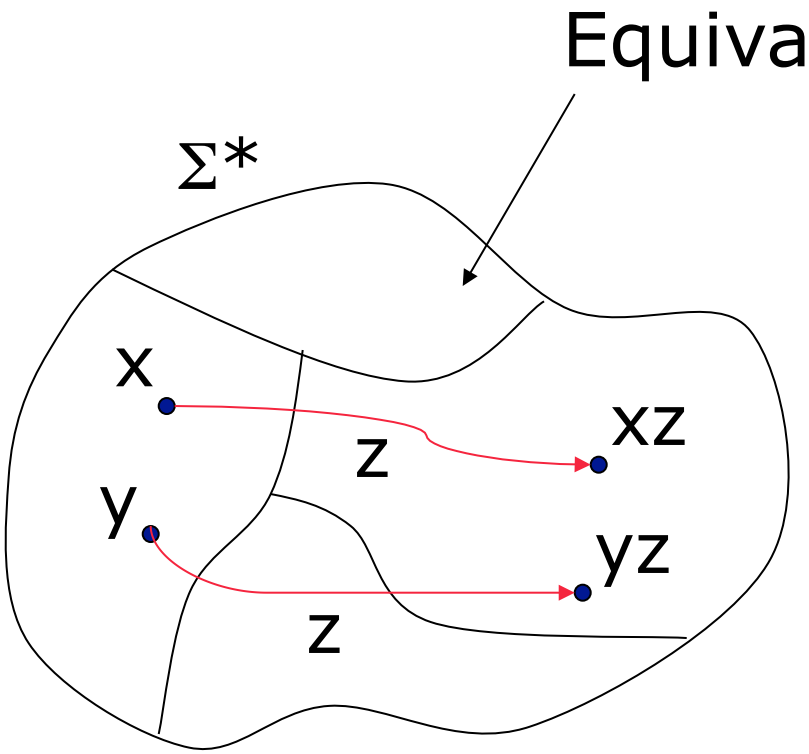
- The relation $\equiv_L$ has the additional property of being a **congruence**:

  $x \equiv_L y$ implies $\forall z \in \Sigma^*$ $(xz \equiv_L yz)$.

- By induction, a necessary and sufficient condition for $\equiv_L$ to be a congruence is:

  $x \equiv_L y$ implies $\forall \sigma \in \Sigma$ $(xz \equiv_L yz)$.

Congruence Pictured



Applying the same input sequence to congruent states yields congruent states.

Partition of the congruence

Finite-State Machines (FSMs)

- A machine is finite-state iff its state set is finite.

Determinism

- A machine is deterministic iff:
  - There are no spontaneous state changes, and
  - For each $q, q', q'' \in Q$ and $\sigma \in \Sigma$

    if $q - \sigma \rightarrow q'$ and $q - \sigma \rightarrow q''$, then $q' = q''$.

    In other words, there is a **partial function** $\delta: Q \times \Sigma \rightarrow Q$ such that $q - \sigma \rightarrow q'$ iff $\delta(q, \sigma) = q'$.

    "Partial" means that $\delta(q, \sigma)$ could be **undefined** for some pairs (q, σ).

DFA

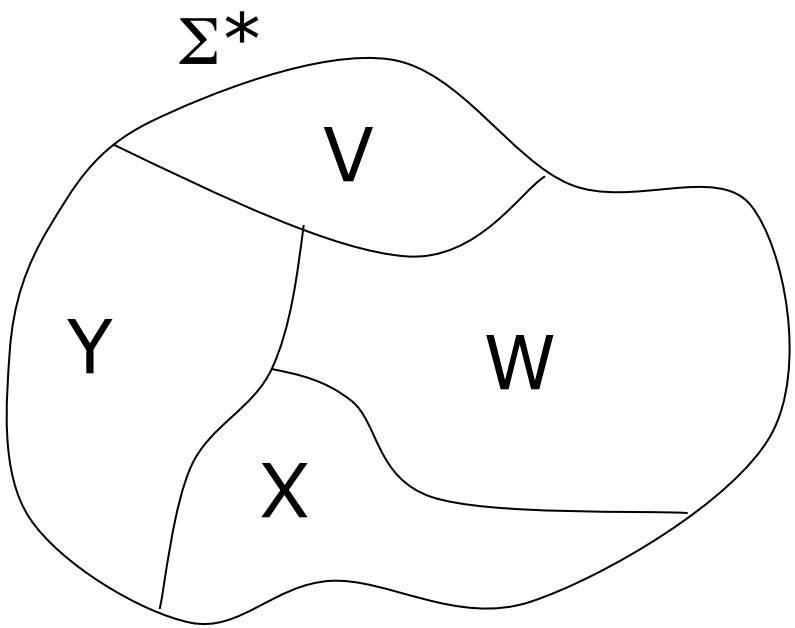
- A finite-state machine that is also deterministic is called a DFA (for deterministic finite-state acceptor).

Finite-State Languages

- Say a language L is finite-state iff it is accepted by some DFA.
- The equivalence partition for a finite-state language is guaranteed to be finite rank.
- Conversely, if the equivalence partition for a language is finite rank, then there is a DFA that accepts that language.

Language in terms of Equivalence Classes

- Consider a finite-state language.
- Its equivalence partition must be finite-rank.
- The language itself must be the union of some of the equivalence classes.



e.g. V∪Y could be the language

Myhill-Nerode Theorem

- $L \subseteq \Sigma^*$ is a finite-state language iff
- L is the union of some equivalence classes of some congruence relation on of finite rank.

Proof of Myhill-Nerode

- Suppose L is a finite-state language.
- Let M be a DFA accepting L.
- Let M be the a DFA.
- Let δ be the state-transition function described earlier, i.e. $\delta(q, \sigma) = q'$ means there is a transition from q to q' via letter σ.
- Extend $\delta: Q \times \Sigma \rightarrow Q$ to $\delta^*: Q \times \Sigma^* \rightarrow Q$, as follows:
  - $\forall q \in Q$ $\quad \delta^*(q, \varepsilon) = q$
  - $\forall q \in Q\ \forall x \in \Sigma^*\ \forall \sigma \in \Sigma$ $\quad \delta^*(q, \sigma x) = \delta^*(\delta(q, \sigma), x)$
- Claim: $x \equiv_L y$ iff $\delta^*(q_0, x) \equiv \delta^*(q_0, y)$.

Lemma

- $\forall z \in \Sigma^* \; \forall x \in \Sigma^*$
  $\forall q \in Q \; \delta^*(q, xz) = \delta^*(\delta^*(q, x), z)$

- Proof is by induction on x.

- Basis $x = \varepsilon$: $\delta^*(q, \varepsilon) = q$ and $xz = z$, so
  $\delta^*(q, xz) = \delta^*(q, z) = \delta^*(\delta^*(q, \varepsilon), z)$

Lemma

- Induction step:

  Assume $\forall q \in Q\ \delta^*(q, xz) = \delta^*(\delta^*(q, x), z)$.

  Show $\forall q \in Q\ \forall \sigma \in \Sigma\ \ \delta^*(q, (\sigma x)z) = \delta^*(\delta^*(q, \sigma x), z)$.

  But $\delta^*(q, (\sigma x)z)$

$= \delta^*(q, \sigma(xz))$	associativity of concat.
$= \delta^*(\delta(q, \sigma), xz)$	definition of δ^*
$= \delta^*(\delta^*(\delta(q, \sigma), x), z)$	induction hypothesis
$= \delta^*(\delta^*(q, \sigma x), z)$	definition of δ^*

Proof of Claim

- $x \equiv_L y$ iff (by definition of $\equiv_L$)
- $L_x = L_y$ iff
- $\forall z \in \Sigma^*$ $(xz \in L \leftrightarrow yz \in L)$ iff
- $\forall z \in \Sigma^*$ $(\delta^*(q_0, xz) \in F \equiv \delta^*(q_0, yz) \in F)$, where F is the set of accepting states of M iff
- $\forall z \in \Sigma^*$
  $(\delta^*(\delta^*(q_0, x), z) \in F \leftrightarrow \delta^*(\delta^*(q_0, y), z) \in F$ iff

- $\delta^*(q_0, x) \equiv \delta^*(q_0, y)$

Impact of Claim

- The claim shows that two **strings** are equivalent iff the **states** to which the machine is taken when reading those state are also equivalent.
- But the set of states is **finite**, so the set of equivalence classes, i.e. the partition, must be finite as well.
- Hence the partition on Σ^* is also **finite**, it being in one-one correspondence with the partition on states.

Claim: If $\equiv_L$ has finite-rank, then there is a DFA accepting L.

- The DFA M is simply constructed as follows:
  - The states of M are the equivalence classes of $\equiv_L$.
  - Let [x] denote the equivalence class of $x \in \Sigma^*$.
  - The initial state of M is $[\varepsilon]$.
  - The accepting states are [x] for $x \in L$.
  - The transitions are defined by the function:
    $\forall x \in \Sigma^* \; \forall \sigma \in \Sigma \quad \delta([x], \sigma) = [x\sigma]$
  - δ is well defined because $\equiv_L$ is a congruence relation.

Finite-State Automata

Robert M. Keller
Harvey Mudd College
March 2013

Automata

- Colloquially, an **automaton** (plural “automata”) is an autonomous device (such as a robot or wind-up toy).

- In CS, the term has a more specific meaning: that of an abstract **mathematical machine** that can perform a specific function.

Uses of Automata

- There are many uses, one of which is to specify algorithms for accepting languages.

- An automaton **accepts a language** if it can tell, for any given input string, whether or not the string is in the language.

Example: Compilers, etc.

- Every compiler contains an automaton, that tells whether or not the input string is well-formed, i.e. is in the language that it compiles.

- Every pattern search program is effectively an automaton for recognizing patterns.

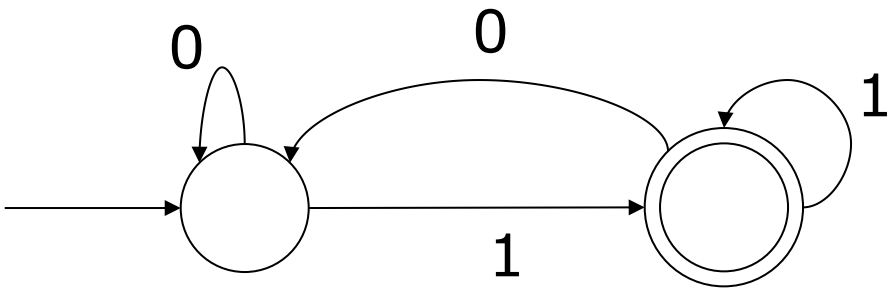
Finite-State Automata (FSA or DFA, they are the same)

- An automaton is finite-state if its behavior is representable by transitions between a states in a finite set, some of which are designated accepting and others not.

- Each automaton has a designated start state.

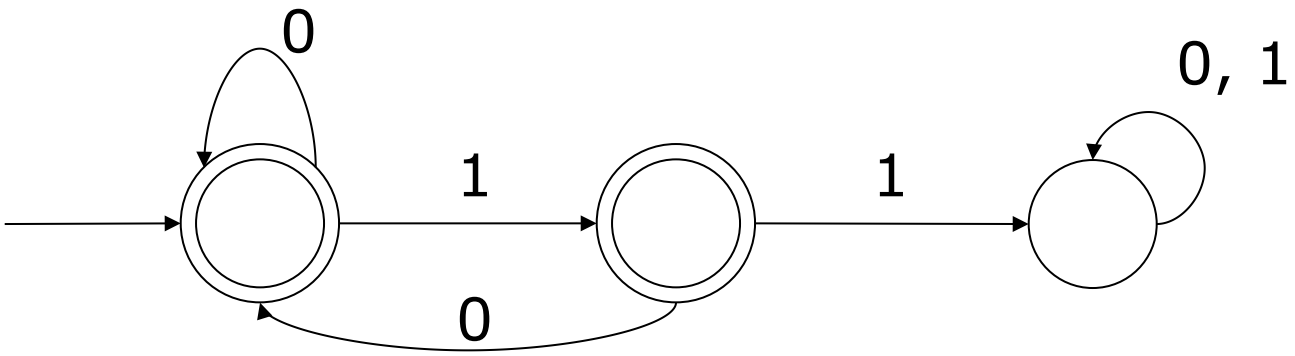
Examples of FSA

- An FSA capable of accepting exactly the strings ending with 1.



Examples of FSA

- An FSA capable of accepting exactly the strings containing no two consecutive 1's.



Thing to Check

- For each combination of a state and a symbol, there should be exactly one arrow leaving the state with that symbol.
- This is the “deterministic” (“D”) in DFA.
- If this property does not hold, better fix it; your automaton might be wrong.

Application

- One way to implement a search is to construct, perhaps on the fly, an automaton that accepts the corresponding language, then simulate the automaton on the given input.

Two Ways to Define Specific Languages

- Give an FSA that accepts the language.
- Give a regular expression for the language.

Remarkable Fact

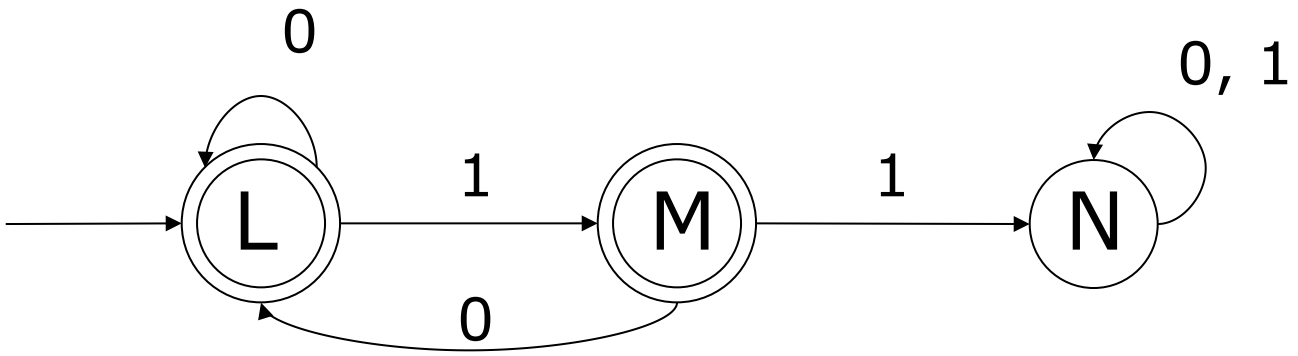
- The preceding two ways are equivalent.
- Equivalent here means that the two methods define the same family of languages.

Application of this Theory

- Sometimes it's easier to give an automaton for a language.
- Sometimes it's easier to give a regular expression.
- It would be nice to be able to go from one to the other more-or-less freely.

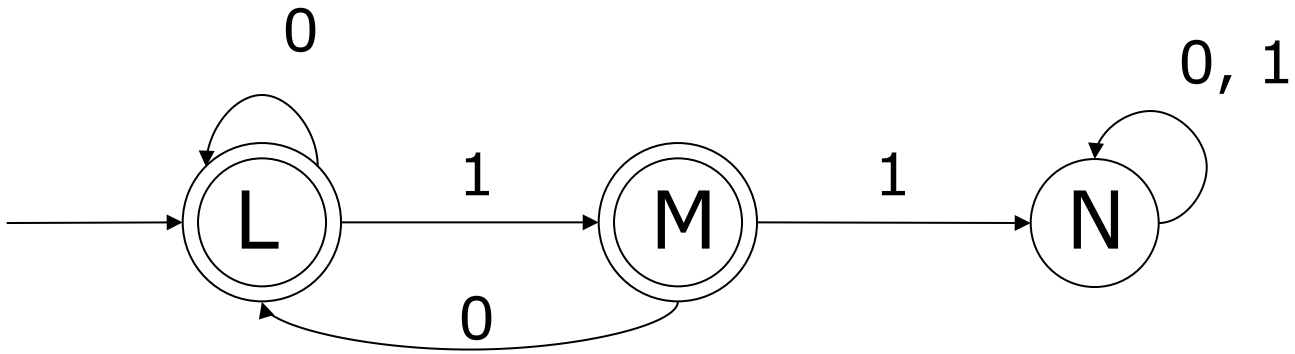
Regular Expression from DFA

- Label the States



- Identify each state with the set of paths from the start state to it. This set is a language.
- The language accepted by the DFA is the **union** of the paths to each of the accepting states, in this case L ∪ M.

Deriving Closed Forms



- View the acceptor as a set of **regular-expression equations**:
  - L = L0 ∪ M0 ∪ ε
  - M = L1
  - N = M1 ∪ N(0 ∪1)
- The ε is on the RHS of the **starting state** only.
- We want to solve (e.g. using Arden's Rule) for L and M, and take the union of the solutions.

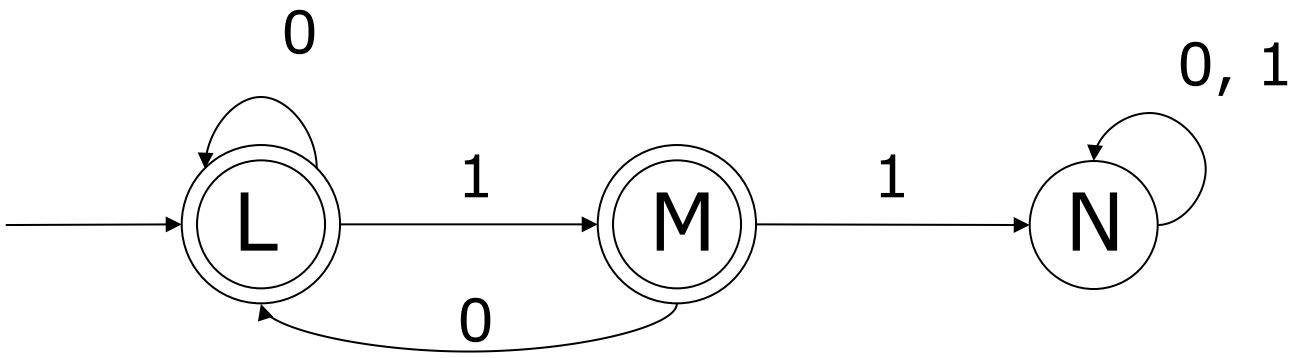
Solving RE Equations

- **Solve** for L and M:
  - L = L0 ∪ M0 ∪ ε
  - M = L1
  - N = M1 ∪ N(0 ∪1)

- **Substitution** Operation:
  - A LHS variable can be replaced with its RHS, so replacing M in the L equation:
  - L = L0 ∪ L10 ∪ ε, or more simply
  - L = L(0 ∪ 10) ∪ ε
- **Elimination** Operation:
  - An equation of the form L = LA ∪ B has the solution L = BA*, so:
  - L = ε(0 ∪ 10)*, or more simply L = (0 ∪ 10)*
- Substitution again:
  - M = L1
  - M = (0 ∪ 10)*1

Conclusion

- The language accepted by the DFA below is
  - L ∪ M
  - which is (0 ∪ 10)* ∪ (0 ∪ 10)*1
  - or more simply
  - (0 ∪ 10)*(ε ∪ 1)



DFA $\Rightarrow$ RE Algorithm

- Express the FSA as a set of RE equations
  - Each state is a variable.
  - Each variable is equated to a union of expressions showing how to get to that state in one step from other states.
  - The start state has ε on the RHS as well.
- Solve the RE equations for the variables:
  - The variables, along with their equations, are solved for one at a time.
  - Choose a variable for elimination.
  - Express that variable in terms of the remaining variables only, using the * operator ($L = LA \cup B$ has the solution $L = BA^*$).
  - Substitute the solution for all occurrences of the variable in the remaining equations.
  - Repeat the above steps until no variables remain.
- Work backward, substituting the solutions found for other variables, until each variable is expressed in closed form.

Another Example

- **Solve**:
  - $L = L1 \cup M0 \cup N0 \cup \varepsilon$
  - $M = L0 \cup M1 \cup N1$
  - $N = L1 \cup M1 \cup N0$

- Note that these equations don't really correspond to a deterministic machine, but it doesn't matter.
- Eliminate N, using $N = (L1 \cup M1)0^*$
  - $L = L1 \cup M0 \cup (L1 \cup M1)0^*0 \cup \varepsilon$
  - $M = L0 \cup M1 \cup (L1 \cup M1)0^*1$
- Regroup:
  - $L = L(1 \cup 10^*0) \cup M(0 \cup 10^*0) \cup \varepsilon$
  - $M = L(0 \cup 10^*1) \cup M(1 \cup 10^*1)$

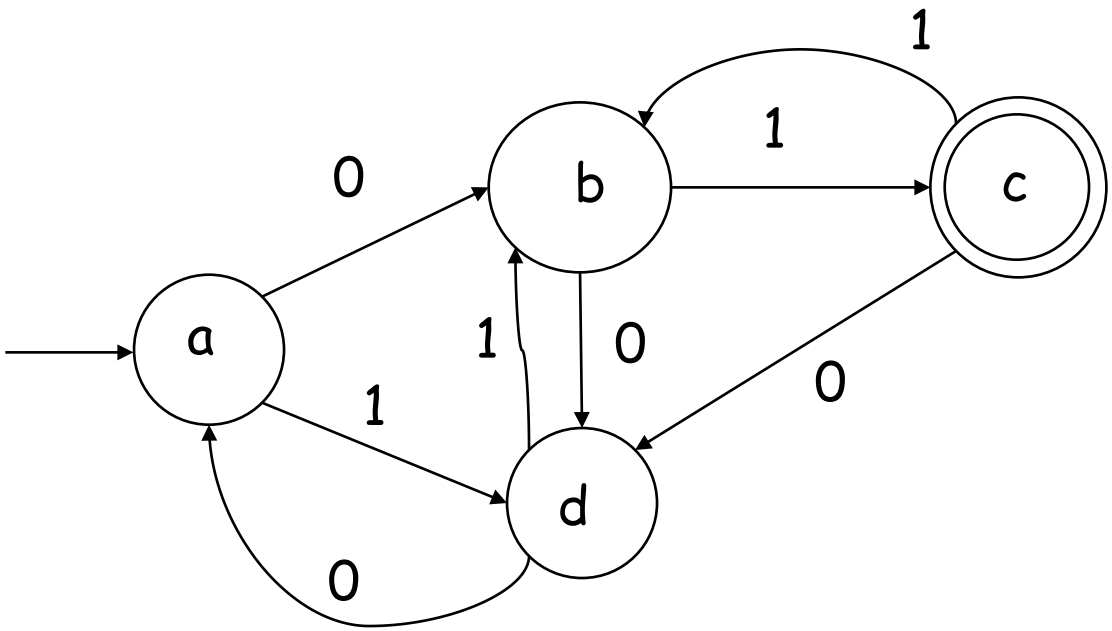
Solution, continued

- Solving:
  - $L = L(1 \cup 10^*0) \cup M(0 \cup 10^*0) \cup \varepsilon$
  - $M = L(0 \cup 10^*1) \cup M(1 \cup 10^*1)$
- Eliminate M using $M = L(0 \cup 10^*1)(1 \cup 10^*1)$, giving:
  - $L = L(1 \cup 10^*0) \cup L(0 \cup 10^*1)(1 \cup 10^*1)(0 \cup 10^*0) \cup \varepsilon$
- Regrouping:
  - $L = L((1 \cup 10^*0) \cup (0 \cup 10^*1)(1 \cup 10^*1)(0 \cup 10^*0)) \cup \varepsilon$
- Solving:
  - $L = ((1 \cup 10^*0) \cup (0 \cup 10^*1)(1 \cup 10^*1)(0 \cup 10^*0))^*$
- Working backward:
  - $M = ((1 \cup 10^*0) \cup (0 \cup 10^*1)(1 \cup 10^*1)(0 \cup 10^*0))^* (0 \cup 10^*1)(1 \cup 10^*1)$
  - $N = (L1 \cup M1)0^* = \ldots$

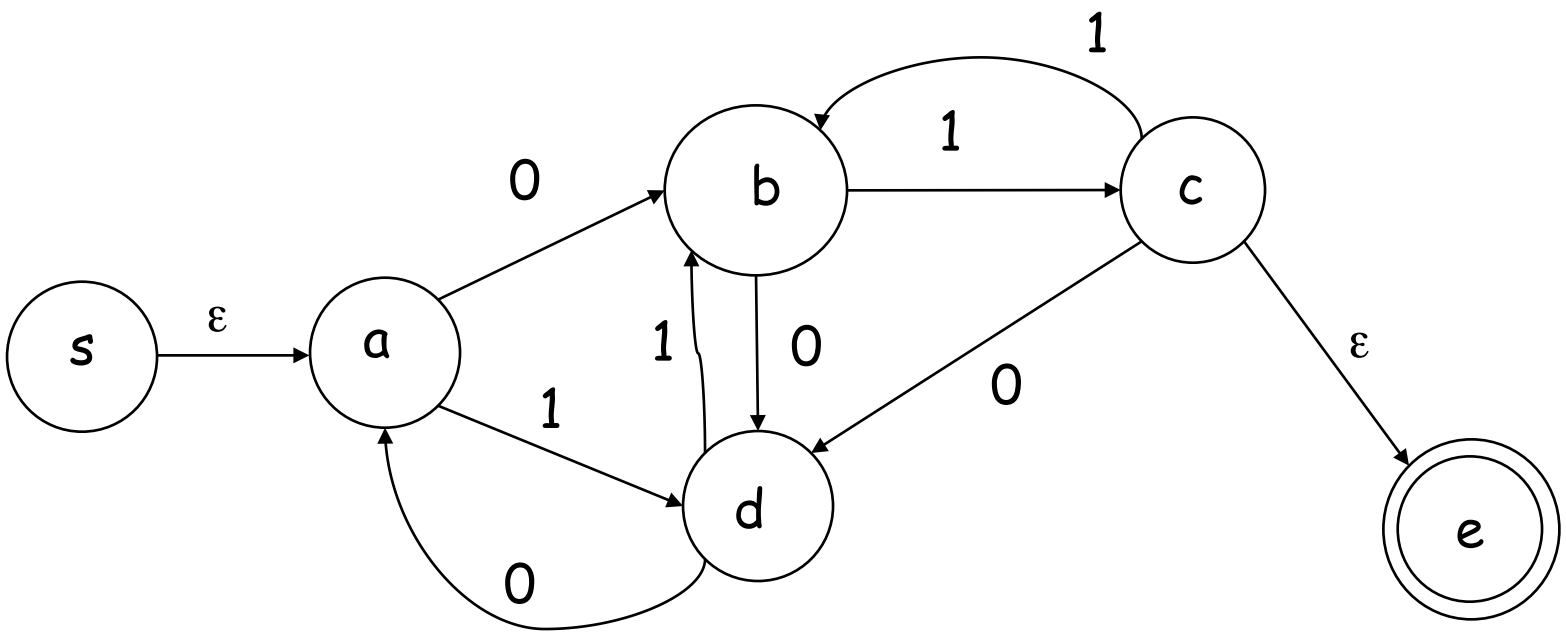
Graphical Alternative Viewpoint

- The DFA is interpreted graphically as a set of regular-expression equations.
- After an initial setup, nodes are eliminated one at a time, replacing paths through them with regular expressions.
- At completion, there is one arc between a pair of nodes, labeled with the regular expression for the language of the DFA.

DFA → RE Example



Step 1: Add Isolated Start and End States

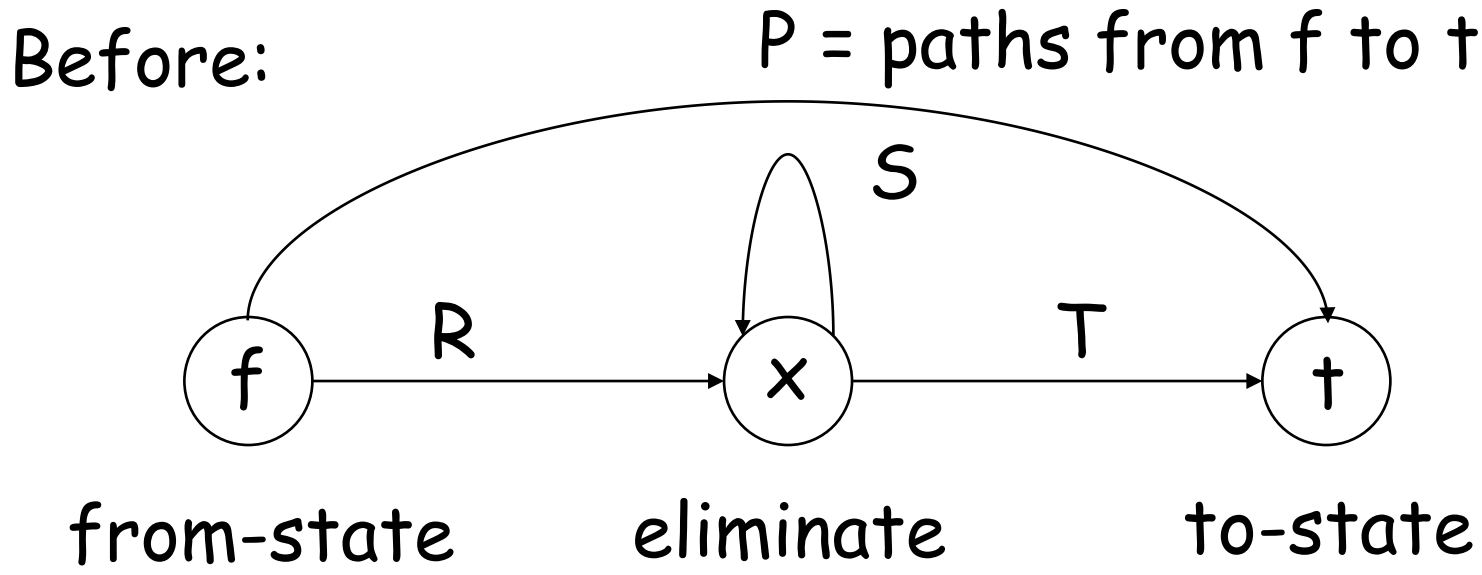


Ultimate goal

Elimination Step

- Pick a node for elimination (other than start and end).
- Union to the regular expression of **each pair** of nodes having a path through the chosen node an additional expression component representing those paths.

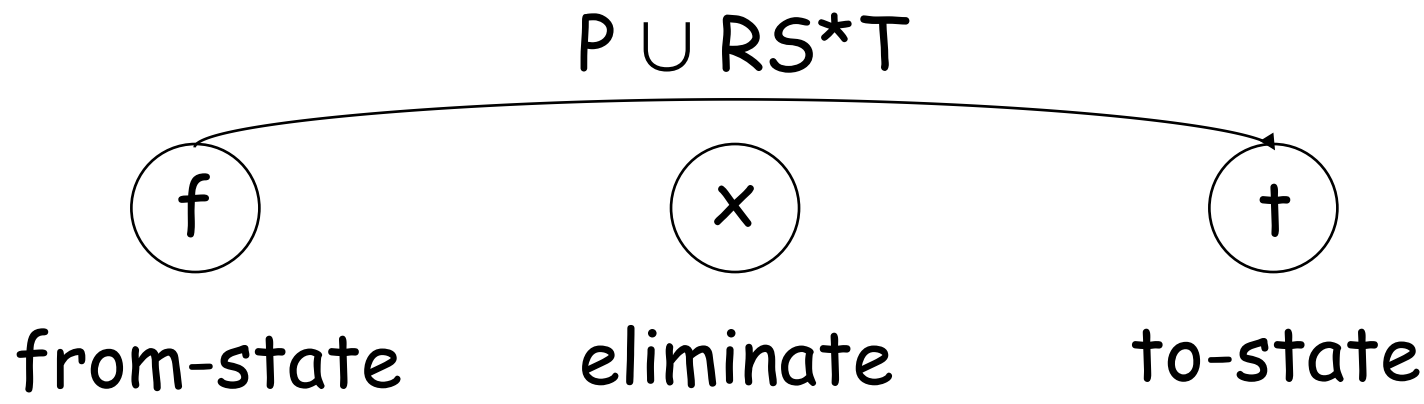
Elimination Step Illustrated



added paths from f to t:

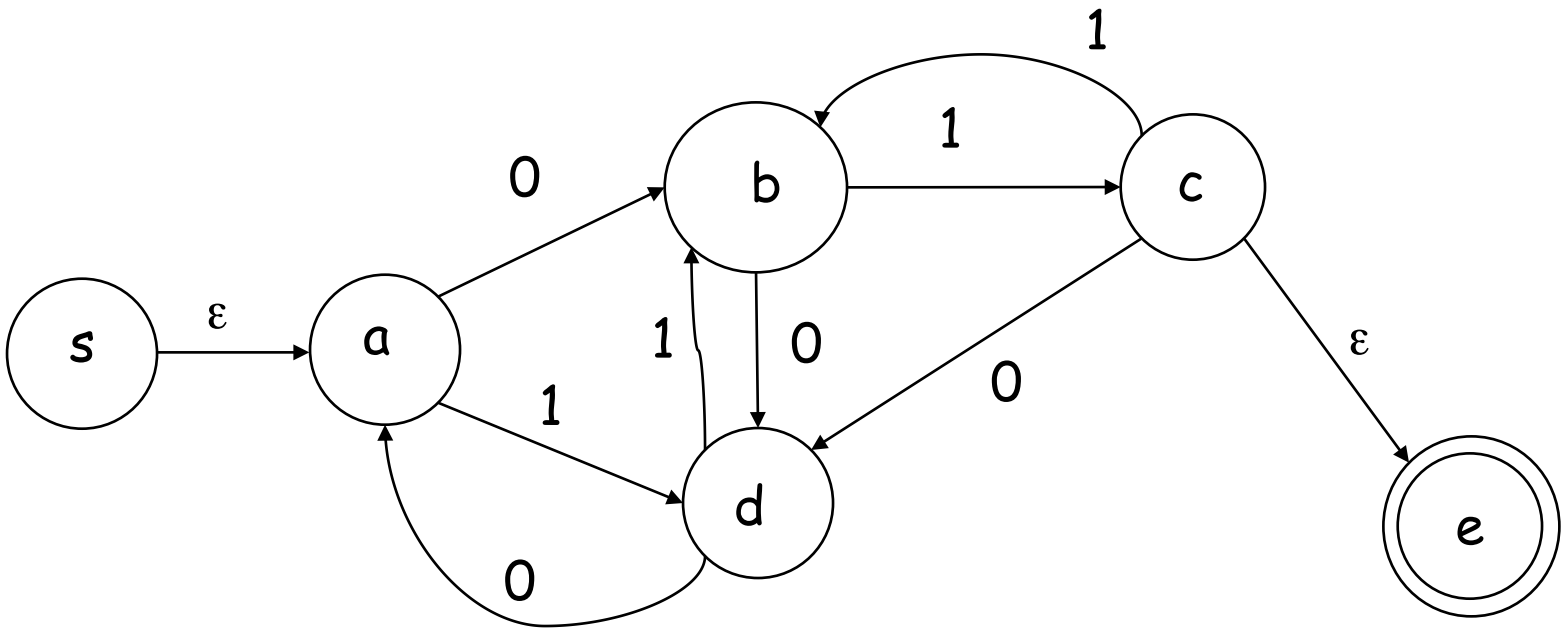
Elimination Step Illustrated

After:

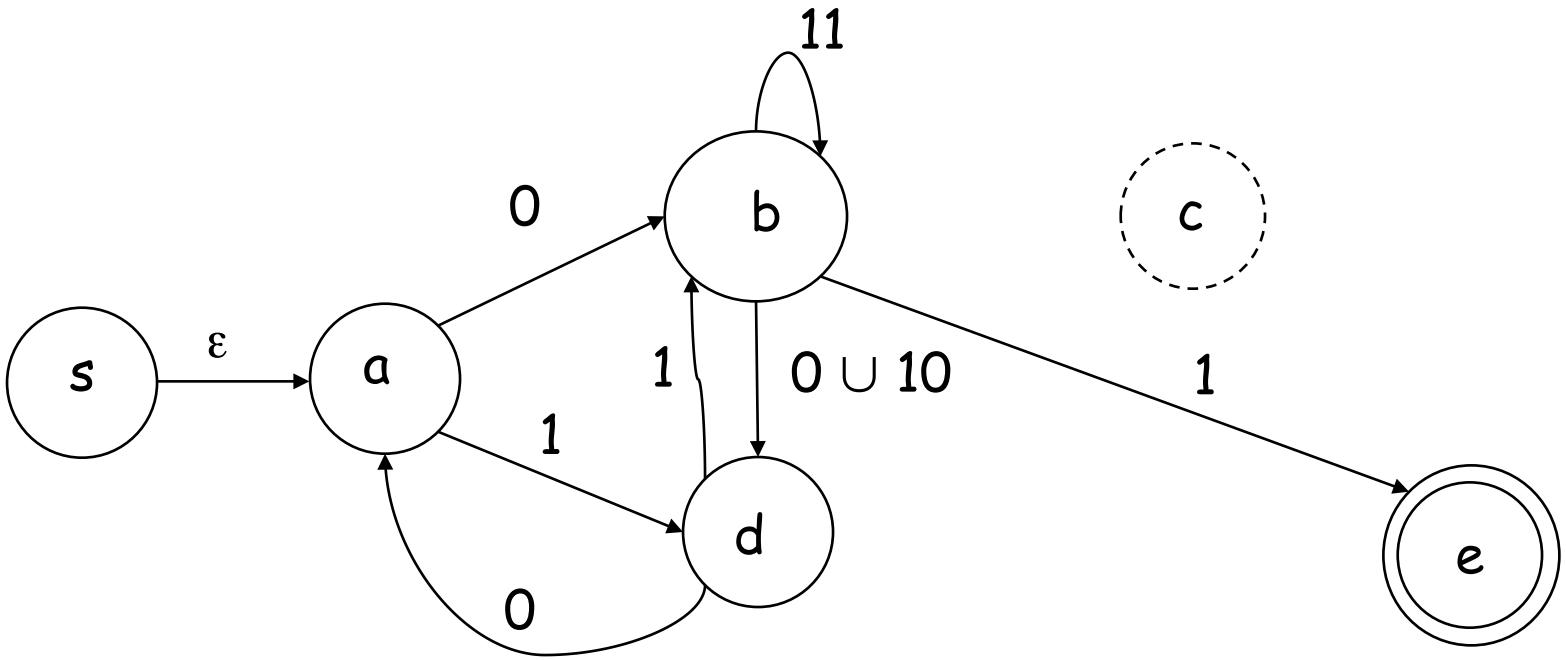


This has to be done for **all** pairs f, t
including the case where f = t.

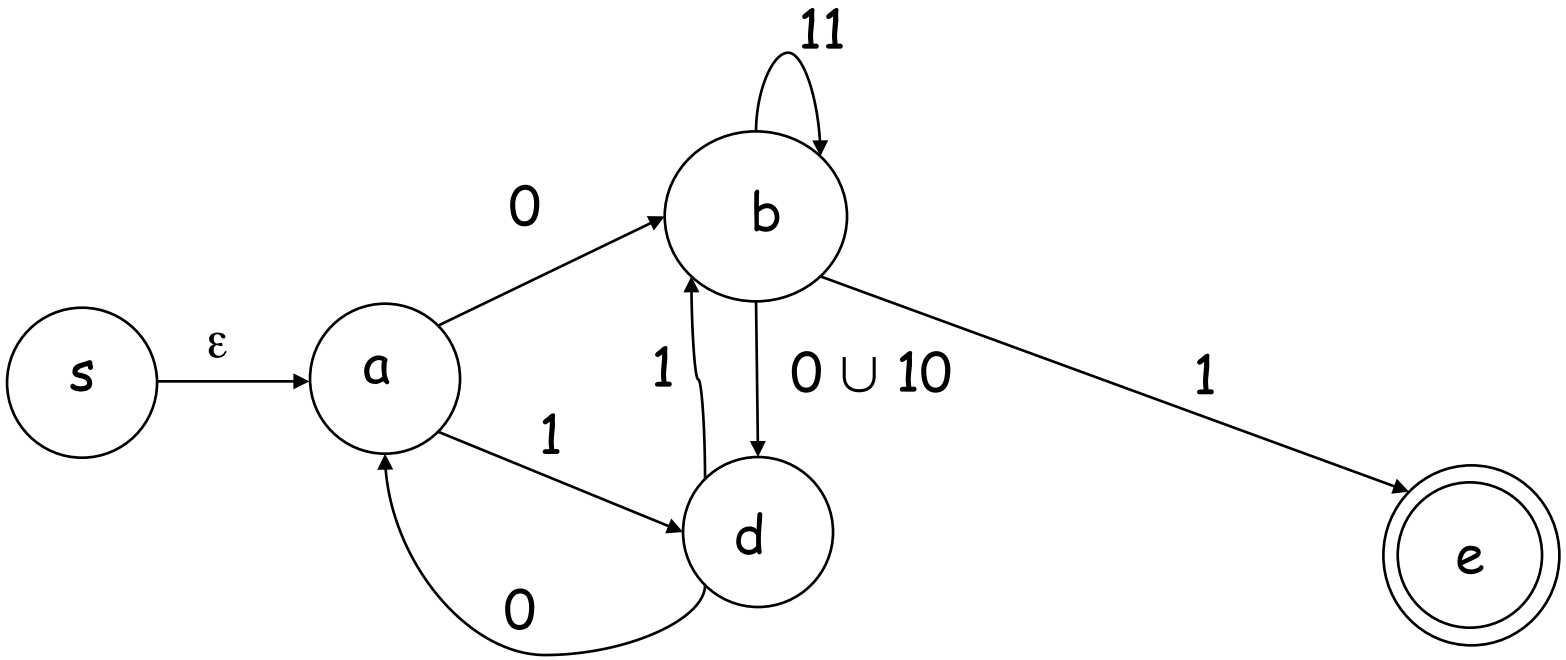
Eliminate c



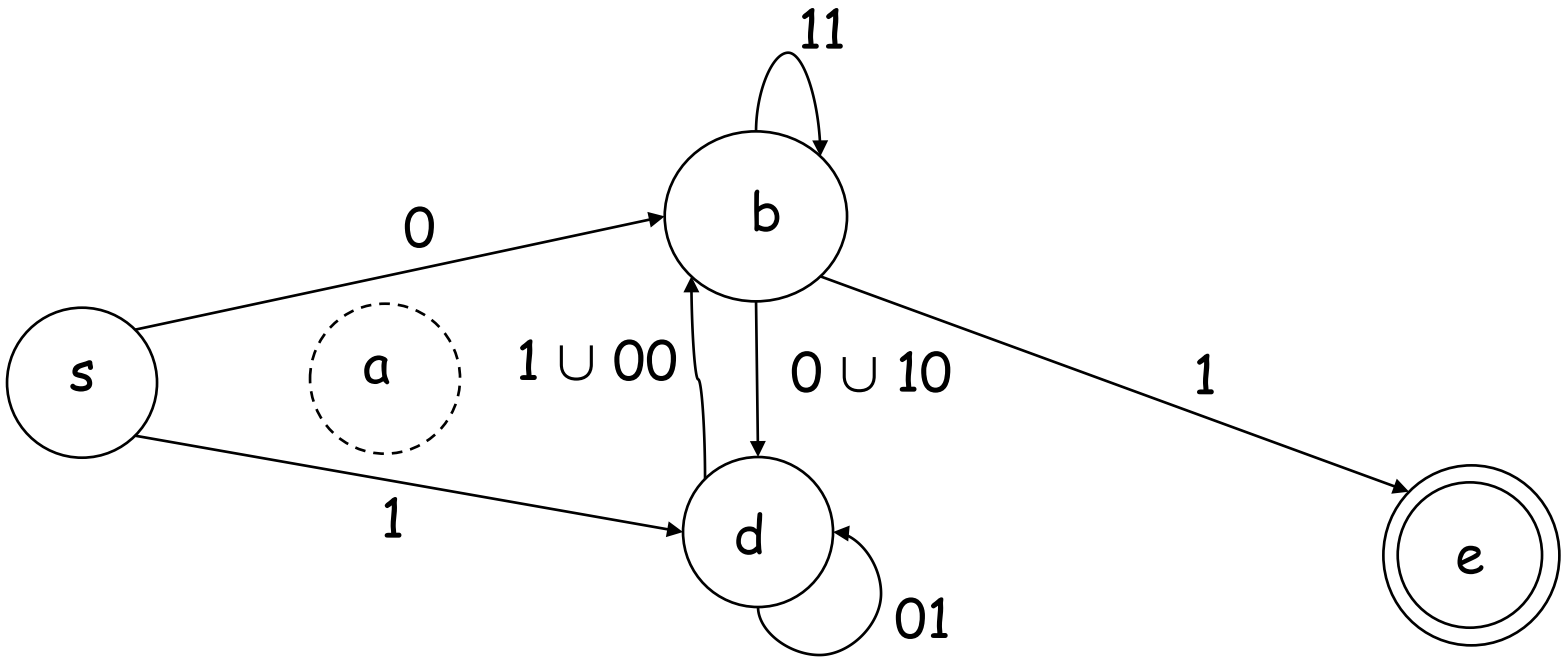
c Eliminated



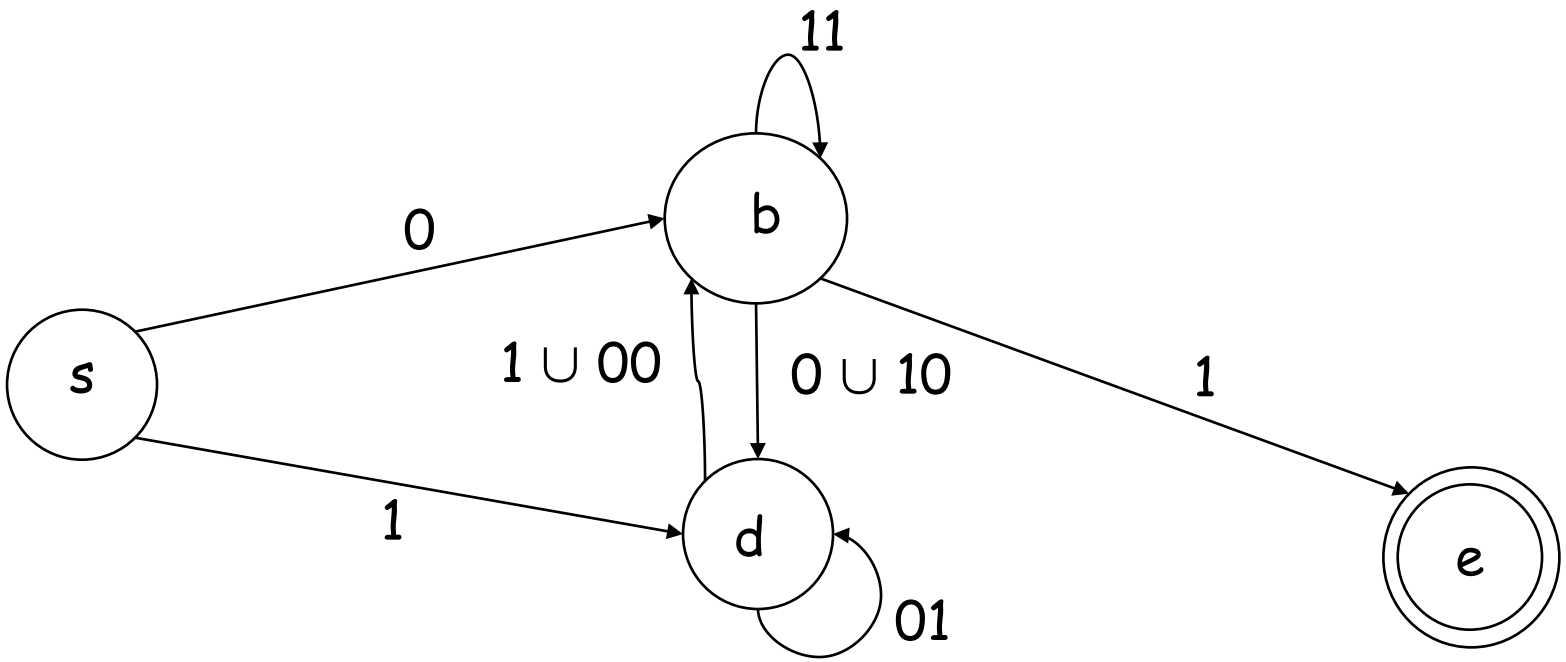
Eliminate a



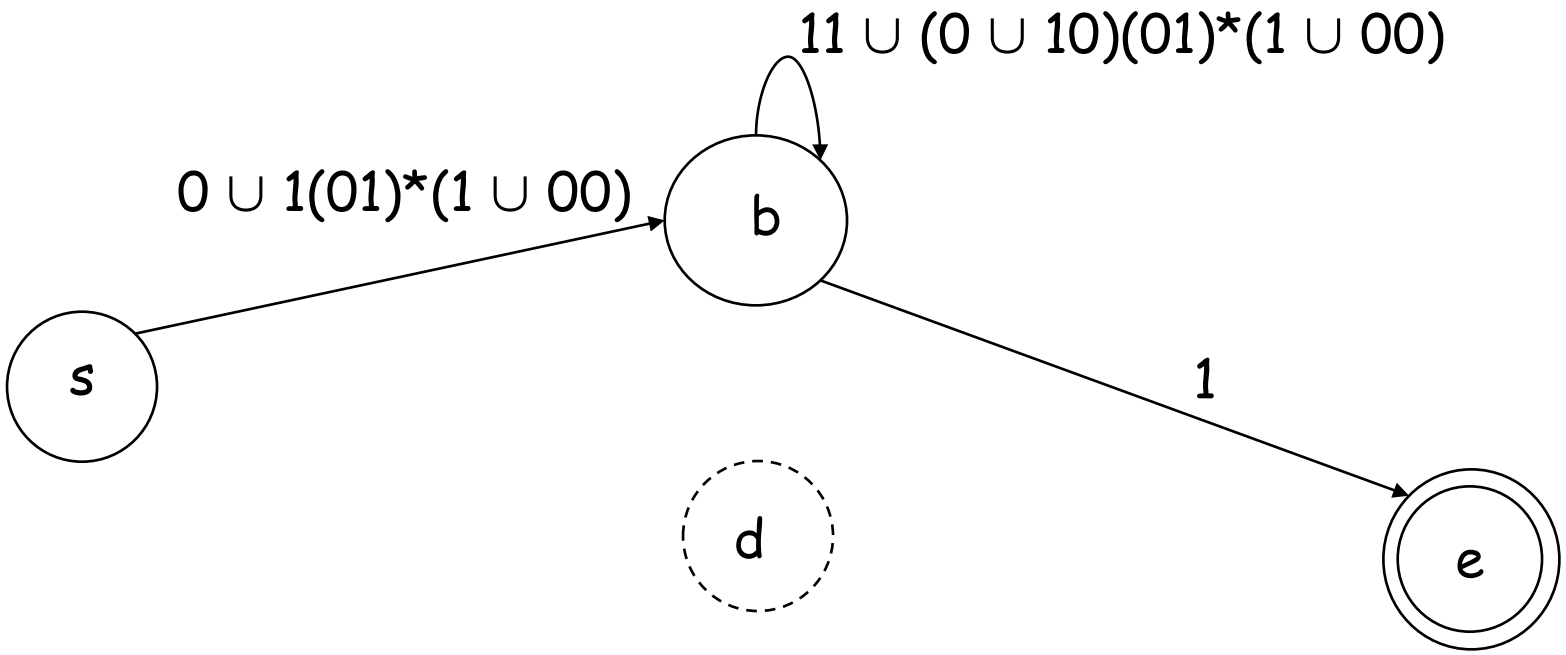
a Eliminated



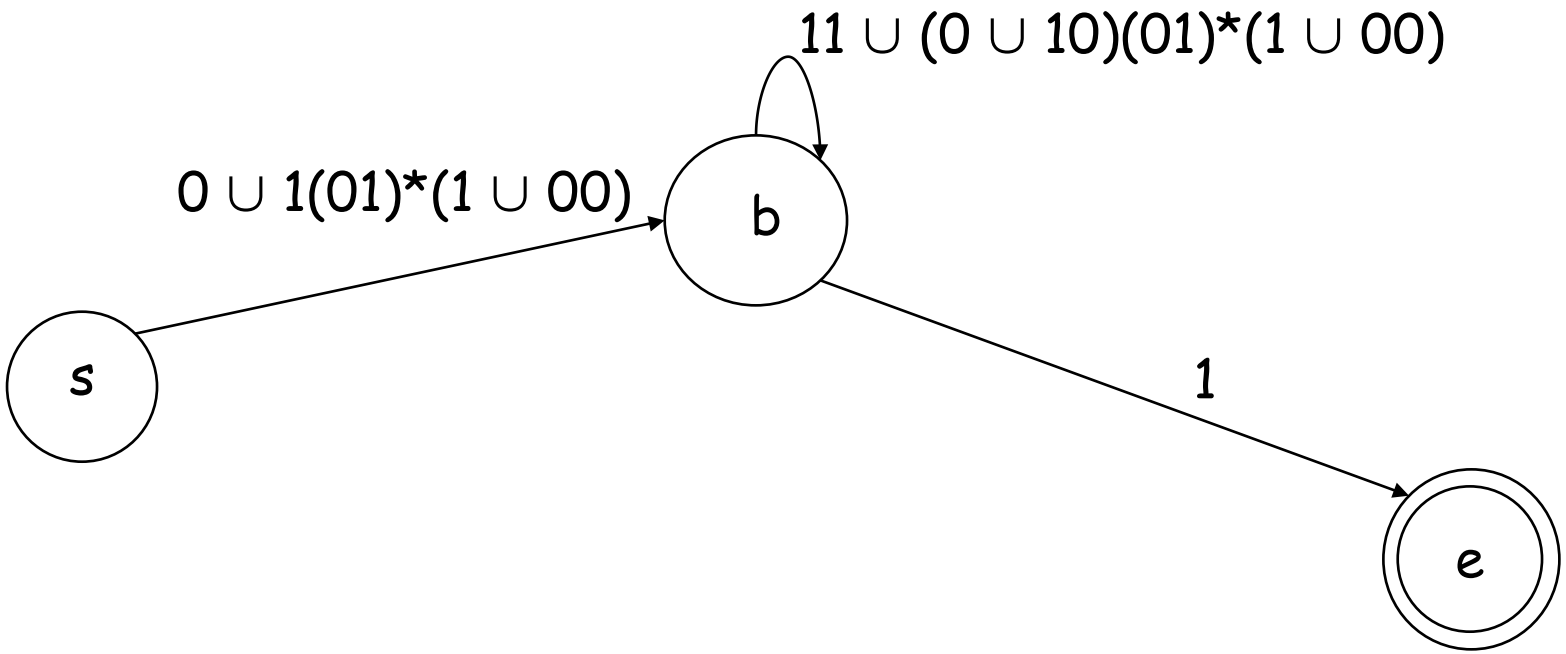
Eliminate d



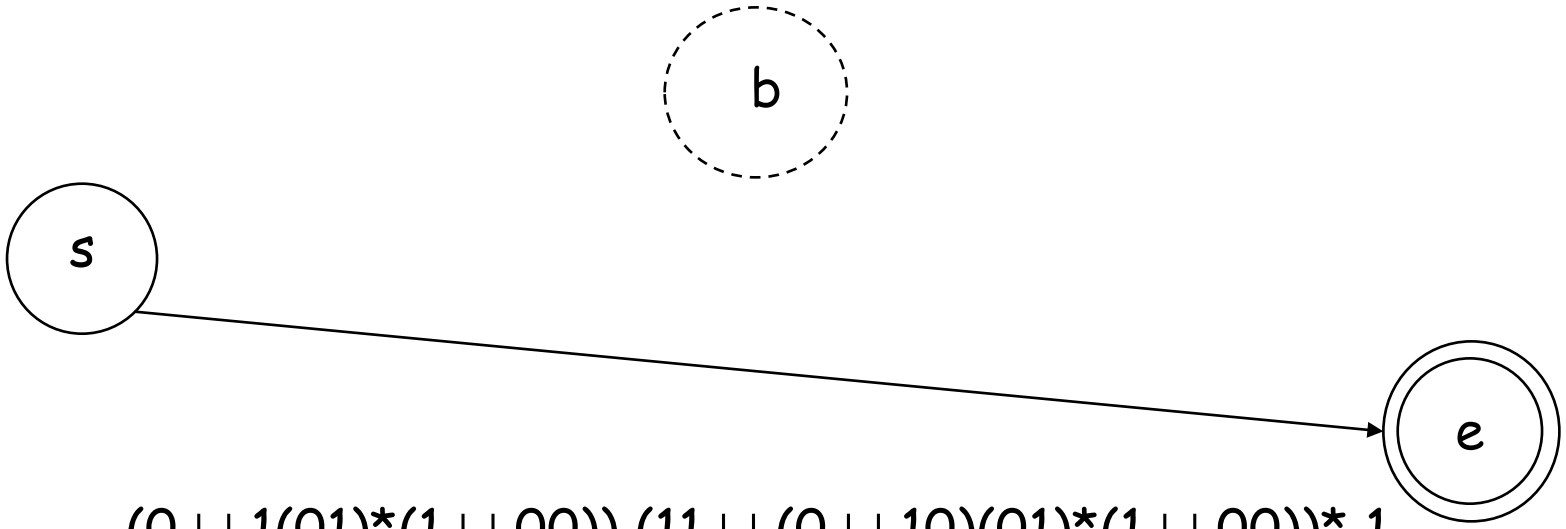
d Eliminated



Eliminate b



b Eliminated (= done)



(0 ∪ 1(01)*(1 ∪ 00)) (11 ∪ (0 ∪ 10)(01)*(1 ∪ 00))* 1

Summary so far

- The language accepted by an DFA is a regular language.
- We haven't yet shown that the converse is true.

Non-Deterministic Finite-State Automata

Robert M. Keller
Harvey Mudd College
March 2013

What is Non-Determinism?

- Can an automaton have choice or free-will?
- Non-determinism supposes it can.

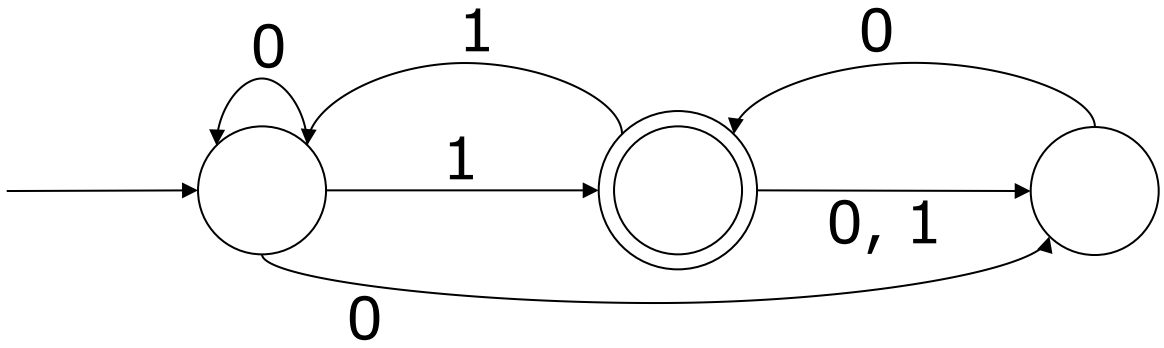
Uses of Non-Determinism?

- Modeling asynchronous events, wherein we don't know the order in which things happen.
- Modeling certain kinds of parallel computing.
- As a mathematical construction device.

NFA's

- NFA = Non-deterministic finite-state automaton
- From any given state, we can have **more than one** transition for the same letter.
- From any given state, we can have **no** transitions for a given letter.
- We can have free transitions that occur without using any letter. (These are labeled with ε.)
- We can have multiple starting states, freely chosen.

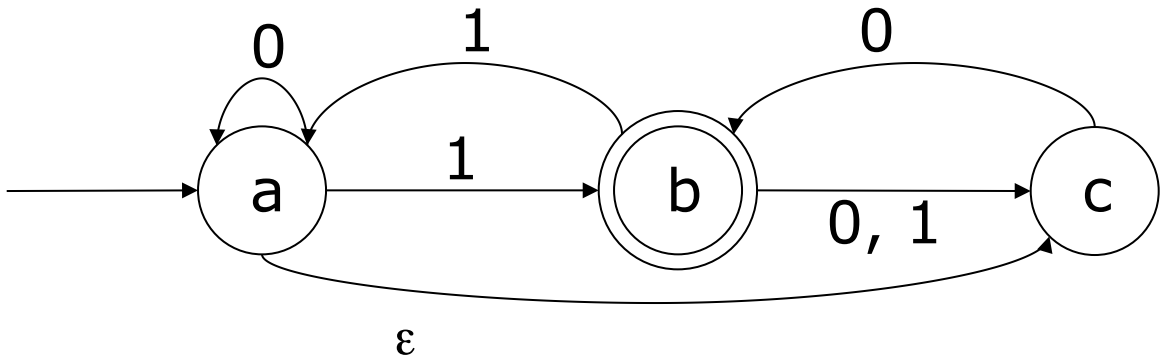
Random Example of an NFA



Paths in an NFA

- Let x be a **string** over the alphabet of N.
- Let q and q' be states of N.
- We say there is a "**path from q to q' with label x**" provided that there is a series of arcs connecting q to q' and the labels on that series, when concatenated, form x.
- Note that ε-transitions can be concatenated and do not change the string to which they are concatenated.

Paths in an NFA (not identical to previous)



Some Paths

1 from a to b

ε from a to c

0 from a to c

00 from a to b

11 from a to a

1 from b to c

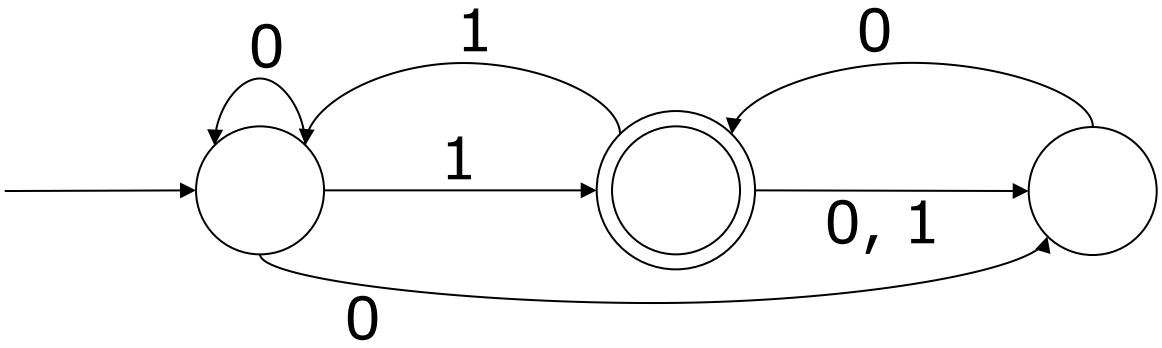
010 from a to a

. . .

Acceptance by an NFA

- An NFA accepts a string x if there is **a** path from **some** start state to **some** accepting state with label x.
- If there is no such path, then the NFA does not accept the string.
- The **language** accepted by an NFA is the set of all strings accepted by the NFA.

Original Example of an NFA



Strings accepted:
1
00
01
000
001
100
110
111
. . .

Strings not accepted:
ε
0
10
11
010
011
101
. . .

Observation

- A DFA can be viewed as a special case of an NFA.
- If L is a language accepted by a DFA, then L is also accepted by an NFA.

The NFA Theorem

- If L is accepted by an NFA, there is also a DFA that accepts L.

Proof of the NFA Theorem (called the "Subset Construction")

- Suppose we have an NFA N. We want to construct a DFA D that accepts the same language exactly.

- Let S be the set of states of N. Choose as the states of D the set P(S) = the **set of all subsets** of the states of N. Since S is finite, so is P(S).

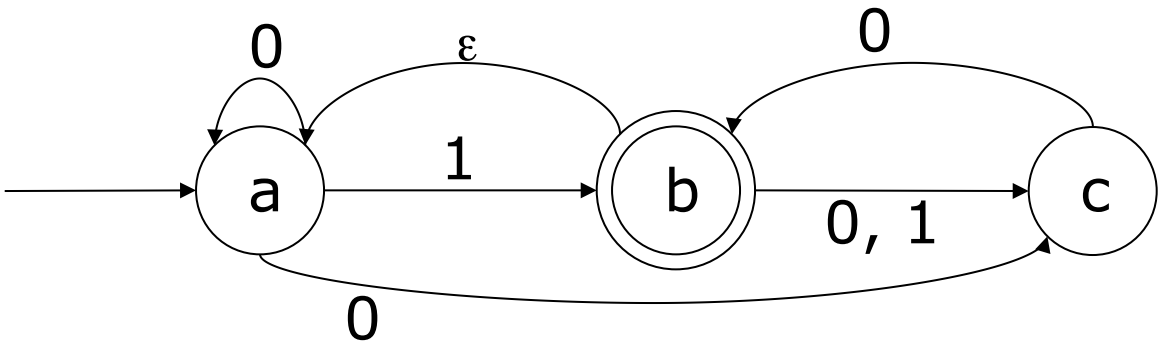
- (In practice, we won't usually need **all subsets**.)

Transitions of D

- Let R be a state of D. So $R \subseteq S$.
- We must define a transition from R for each symbol σ in our alphabet.
- To designate the **next state** from R given symbol σ, use:

  $R_\sigma = \{q' \in S \mid (\exists q \in R)\ \exists$ path from q to q' with label $\sigma\}$

Examples of DFA Transitions



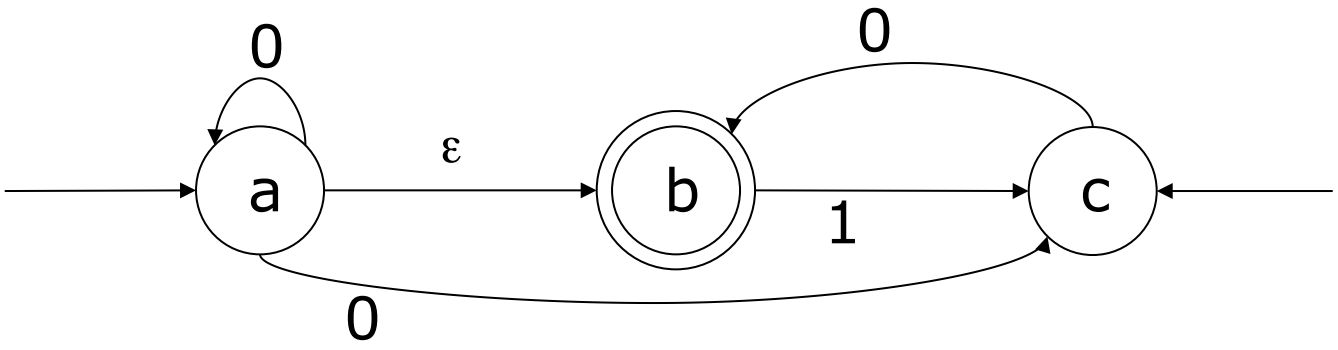
$\{a, b\}_0 = \{a, c\}$
$\{a, b\}_1 = \{a, b, c\}$
$\{b\}_0 = \{a, c\}$
$\{b\}_1 = \{c\}$
$\{c\}_0 = \{a, b\}$
$\{c\}_1 = \varnothing$

Initial State of D

- Letting S_0 be the set of initial states of N, the initial state of D is defined to be the subset

$S_0 \cup \{q' \in S \mid (\exists q \in S_0)$ $\exists$ path from q to q' with label $\varepsilon\}$

Examples of Initial State



The initial state of D is {a, b, c}.

Conclusion of the NFA Theorem

- The preceding construction shows how we capture all paths of the original NFA N in a DFA D.

- The only part remaining is defining the **accepting states** of D. These will be the subsets of S that **contain at least one accepting state** of N.

Interpretation of the DFA Construction

- The DFA constructed in the proof of the theorem can be interpreted as an automaton that simulates all possible choices of the NFA **in parallel**.

- This idea will have at least one other application (to be seen).

A Computational Shortcut

- By beginning with the initial state of the DFA and working from there, we can eliminate the need to generate all states in P(S) in most cases.

- In other words, we need only generate the states **reachable from** the initial state of D.

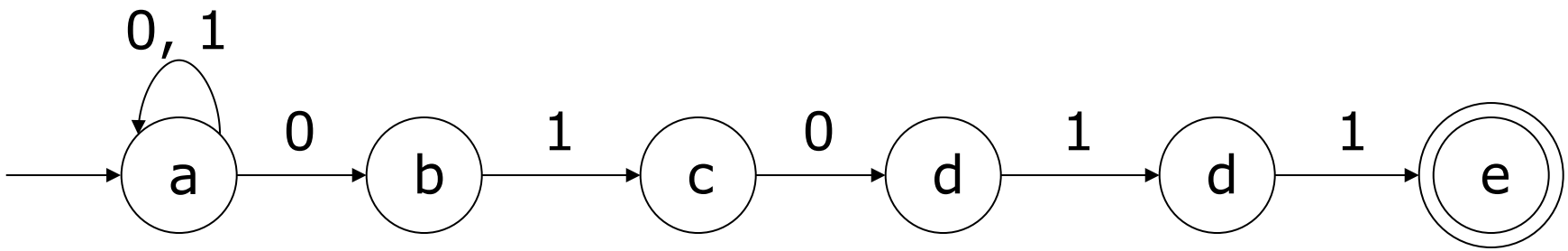
Application of the Construction

- Suppose we want to construct a DFA that will accept the language of **strings ending with a given string**, for example: 01011.

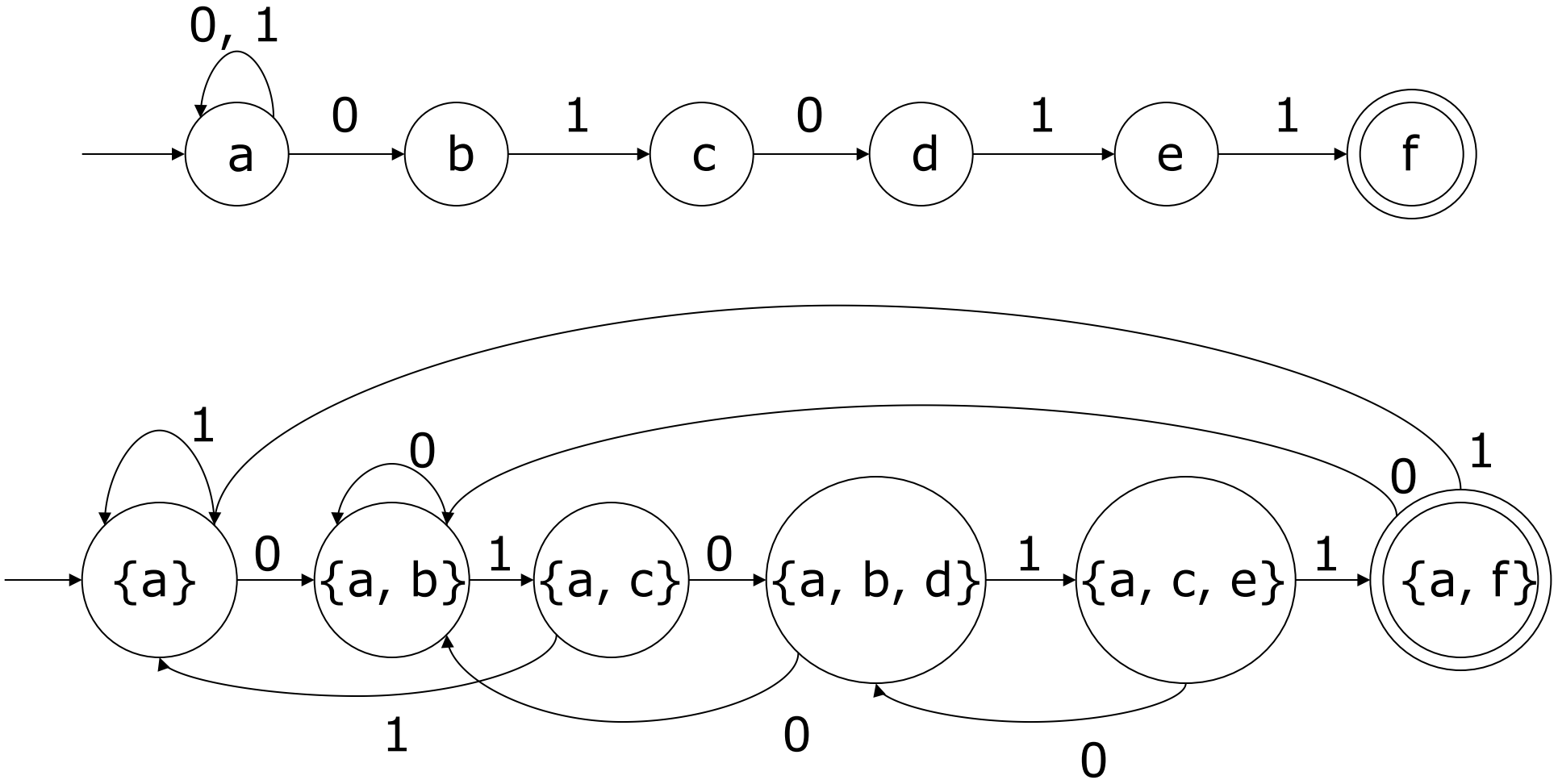
- The tricky part here is that if we get as input, for example, 01010, while this is not accepted, the last part 010 can possibly be used as the initial part of a string that is, for example 0101011.

- By constructing an appropriate NFA and converting it, it is easy to get the DFA right.

An NFA for {0, 1}* 01011



Constructing a DFA for {0, 1}* 01011



Notes on the Previous Example

- Not all subsets were reachable. Those that weren't were not generated.
- The number of states is the same. This is a coincidence; it may be more or fewer.
- The transition structure is more complex in the DFA. This is typical.

Algorithmic Application

- The principle underlying the illustrated method for text matching was the insight and basis for two text searching algorithms:
  - **Knuth-Morris-Pratt**: Search for a single string
  - **Aho-Corasick**: Search for a finite set of strings

Another Possibility for Search

- Rather than go through the DFA construction and simulate the result to do search a text, it is possible to simulate the NFA directly.
- In this case there would be a **set** of “current states” rather than a single one.
- Just use **multiple state pointers** for the simulation rather than a single one.

More applications of the subset construction are forthcoming.

- Closure properties
- DFA from regular expressions