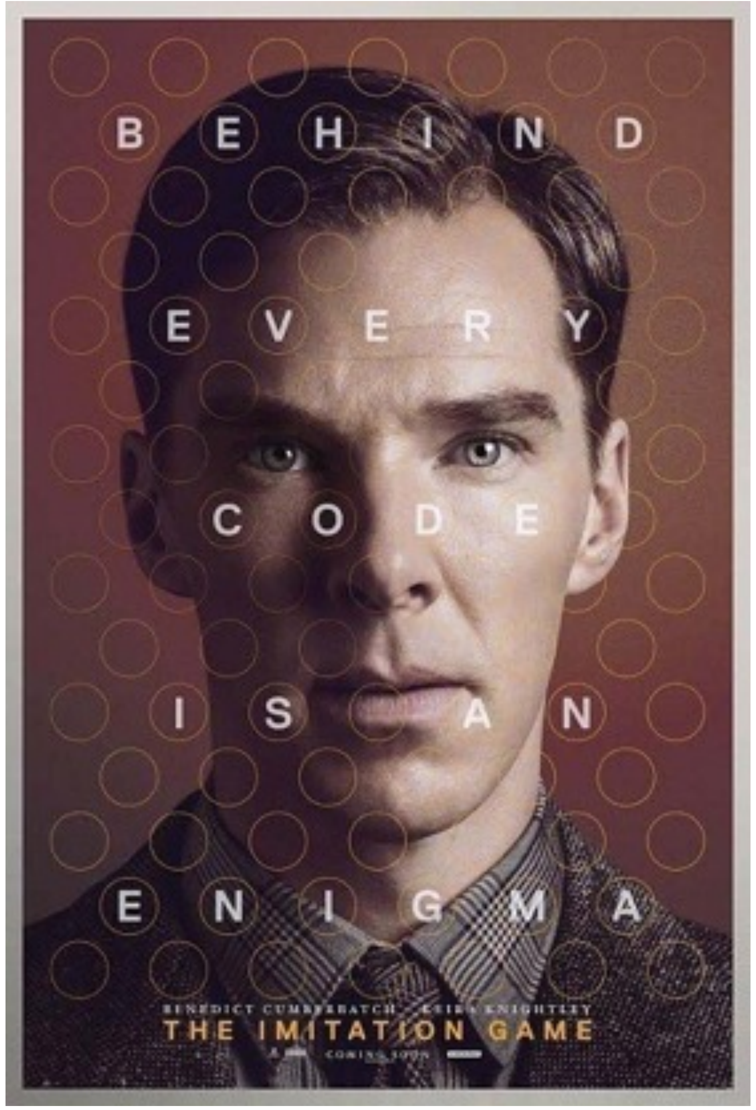
Automata & languages

A primer on the Theory of Computation



Laurent Vanbever

www.vanbever.eu

ETH Zürich (D-ITET)

September, 24 2015

Last week was all about

Deterministic Finite Automaton

We saw three main concepts

Regular Language

Formal definition

Closure

Regular Language

Formal definition

Closure

A language *L* is *regular* if some finite automaton recognizes it

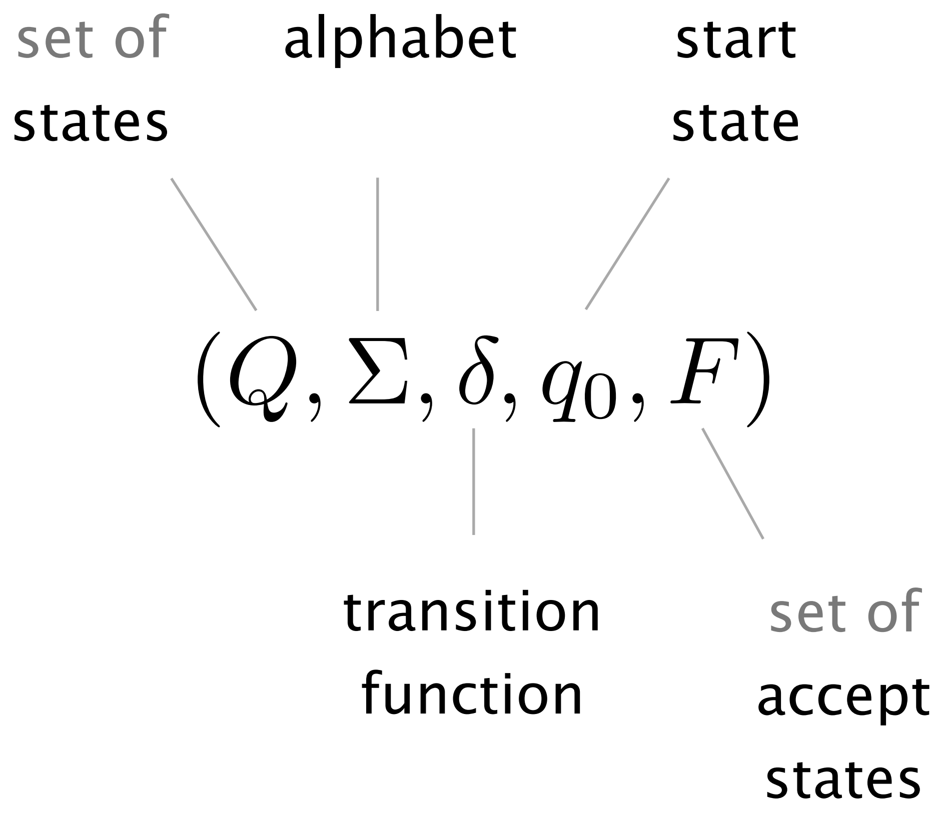
Regular Language

Formal definition

Closure

A finite automaton is a 5-tuple

$$(Q, \Sigma, \delta, q_0, F)$$



set of states

alphabet

start state

$$(Q, \Sigma, \delta, q_0, F)$$

transition function

set of accept states

Regular Language

Formal definition

Closure

If L_1 and L_2 are regular, then so are:

$L_1 \cup L_2$ $\quad$ $L_1 \cap L_2$ $\quad$ $\overline{L_1}$

$L_1 \oplus L_2$ $\quad$ $L_1 - L_2$

We still need to prove that regular languages are closed under concatenation and Kleene*

Finite Automata

Thu Sept 24

1 Closure (the end)

2 Equivalence

- DFA
- NFA
- Regular Expression

Back to concatenation

- Theorem

  The class of regular languages is closed under concatenation.

  If L_1 and L_2 are regular languages then so is $L_1 \bullet L_2$.

Back to concatenation

- Theorem

  The class of regular languages is closed under concatenation.

  If L_1 and L_2 are regular languages then so is $L_1 \bullet L_2$.

- Question:

  Can we apply the same trick as with the Unioner M which simultaneously runs M_1 and M_2 & accepts if either M_1 accepts or M_2?

  M would have to accept if the input can be split into two pieces: one which is accepted by M_1 and the following one by M_2

Let’s play a bit

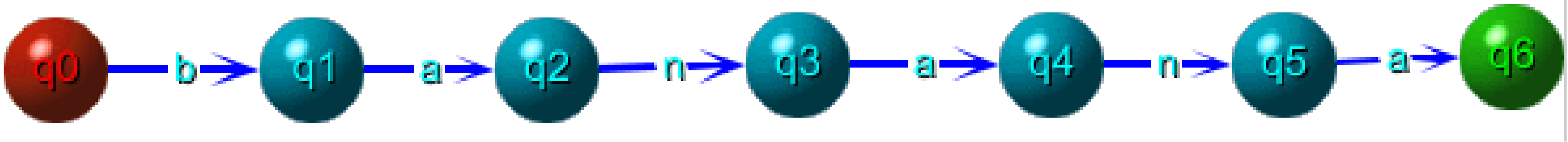
- First, find the DFA for
  - L_1 = { $x \in$ {0,1}* | x is any string}
  - L_2 = { $x \in$ {0,1}* | x is composed of one or more 0s}

- Then, find the DFA for
  - $L_1 \bullet L_2$ = { $x \in$ {0,1}* | x is any string that ends with at least one 0}

Introduction to Nondeterministic Finite Automata

- We have mostly played with Deterministic Finite Automata (DFA)
  - Every step of a computation follows from the preceding one in a unique way

- Nondeterministic Finite Automata (NFA) *generalizes* DFA
  - Several choices may exist for the next state at any point
  - We have already seen one example of a NFA:
    One in which not all transitions were defined

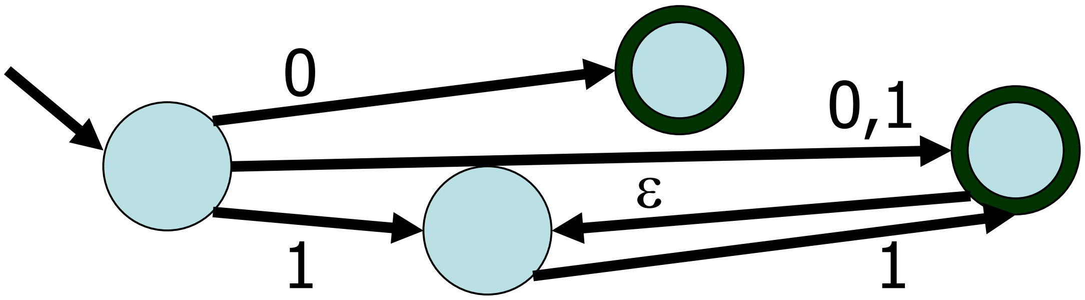


- Now, let's dive deeper

Introduction to Nondeterministic Finite Automata

- The static picture of an NFA is as a graph whose edges are labeled by Σ and by ε (called Σ_ε) and with start vertex q_0 and accept states F.

- Example:



NFA: Formal Definition.

- Definition:

  A nondeterministic finite automaton (NFA) is encapsulated by $M = (Q, \Sigma, \delta, q_0, F)$ in the same way as an FA, *except that*

  - Σ_ε is now the alphabet instead of Σ
  - δ has different domain and co-domain: $\delta: Q \times \Sigma_\varepsilon \rightarrow P(Q)$
  - $P(Q)$ is the power set of Q so that $\delta(q,a)$ is the set of all endpoints of edges from q which are labeled by a.

- Any state can have 0, 1 or more transitions for each symbol of the alphabet
  - including ε-transitions! Think of them as “free”; they don’t require to read

Formal Definition of an NFA: Dynamic

- Just as with FA’s, there is an implicit auxiliary tape containing the input string which is operated on by the NFA.

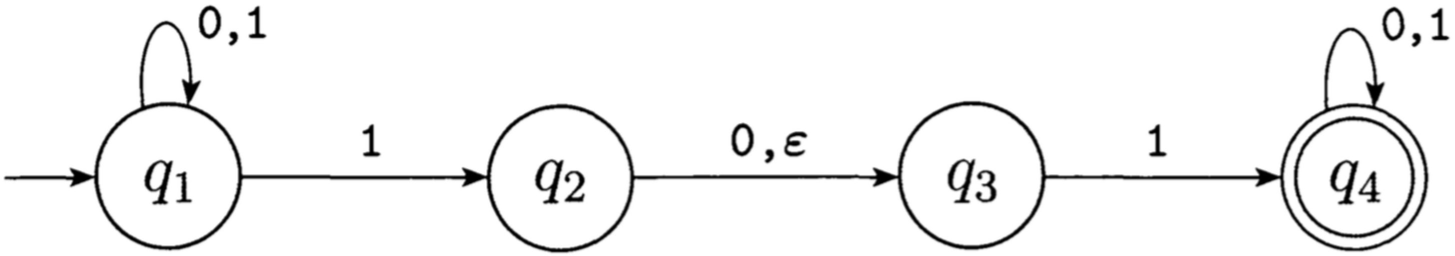
- As opposed to FA’s, NFA’s are parallel machines.
  The are able to be in several states at any given instant.

- The NFA reads the tape from left to right with each new character causing the NFA to go into another set of states.

- When the string is completely read, the string is accepted depending on whether the NFA’s final configuration contains an accept state.

Let’s play

- Let’s illustrate the computation performed by this NFA



- when the string 010110 is shown as input
- Bonus question: what is the language recognized by the automata?

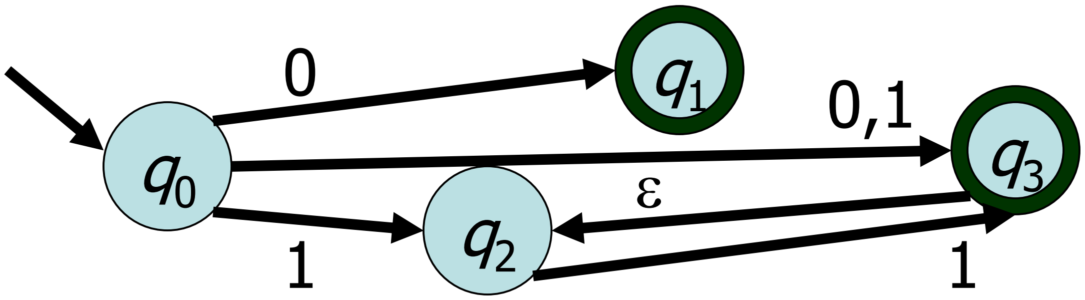
Formal Definition of an NFA: Dynamic

- A string *u* is accepted by an NFA *M* iff there *exists a path* starting at q_0 which is labeled by *u* and ends in an accept state.

- The language accepted by M is the set of all strings which are accepted by *M* and is denoted by *L*(*M*)
  - As following a label ε is for free (without reading an input symbol), you should delete all ε's when computing the label of a path.

Example

M_4:



Question: Which of the following strings is accepted?

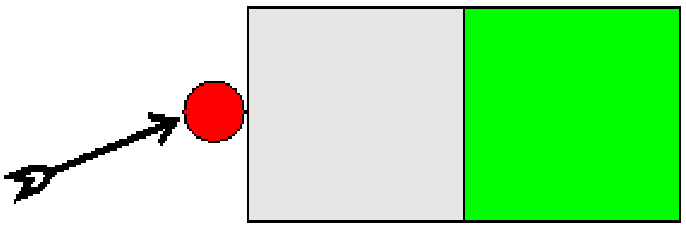
1. ε
2. 0
3. 1
4. 0111
5. 0011

Bonus question: what is the language recognized by the automata?

NFA’s and Regular Operations

- We will study how NFA’s interact with regular operations.

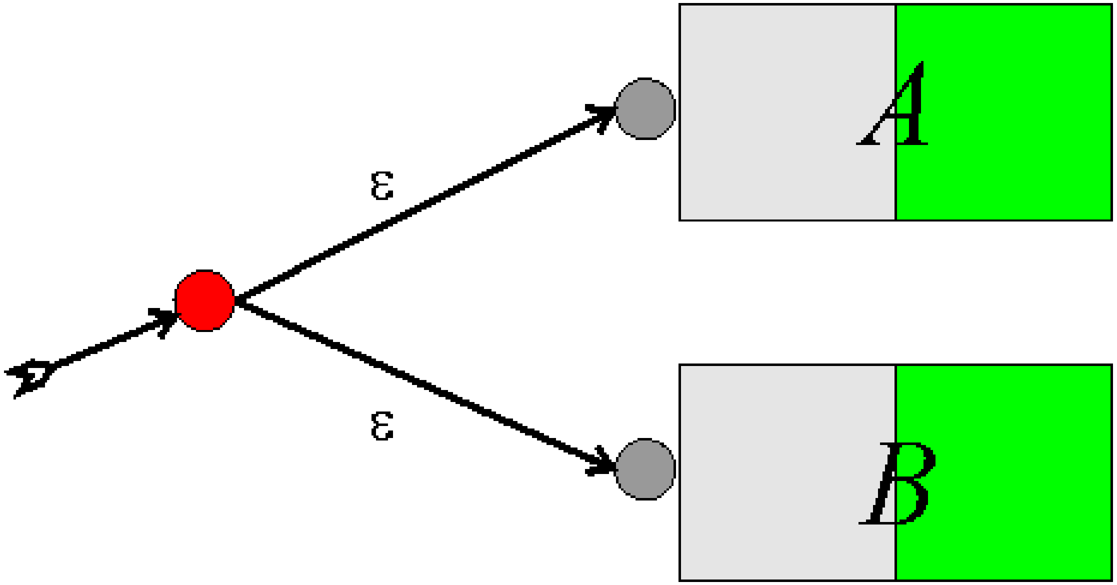
- We will use the following schematic drawing for a general NFA.



- The red circle stands for the start state q_0, the green portion represents the accept states F, the other states are gray.

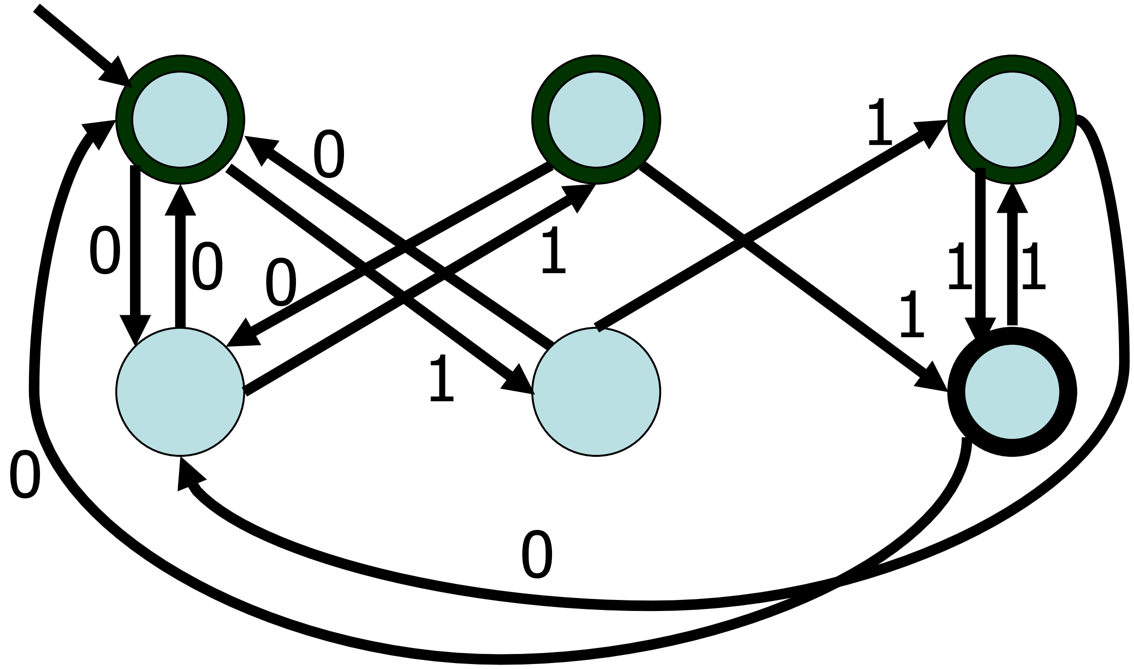
NFA: Union

- The union $A \cup B$ is formed by putting the automata A and B in parallel. Create a new start state and connect it to the former start states using ε-edges:



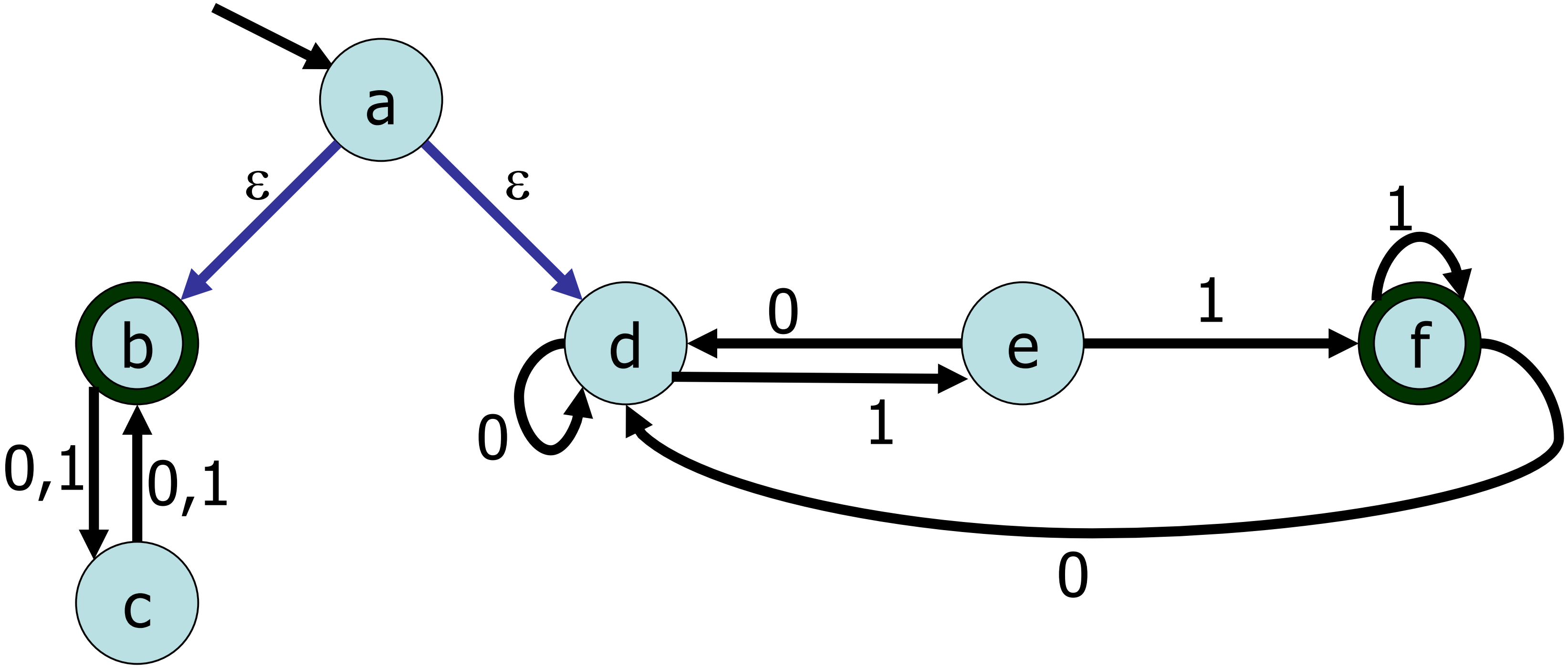
Remember our union example?

- Compare this obtained using the Cartesian Product Construction:



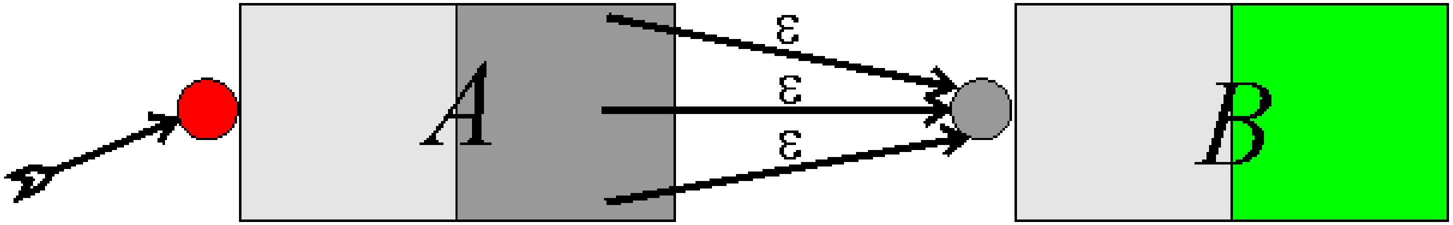
With this NFA which *recognizes the same language*

- L = {x has even length} ∪ {x ends with 11}



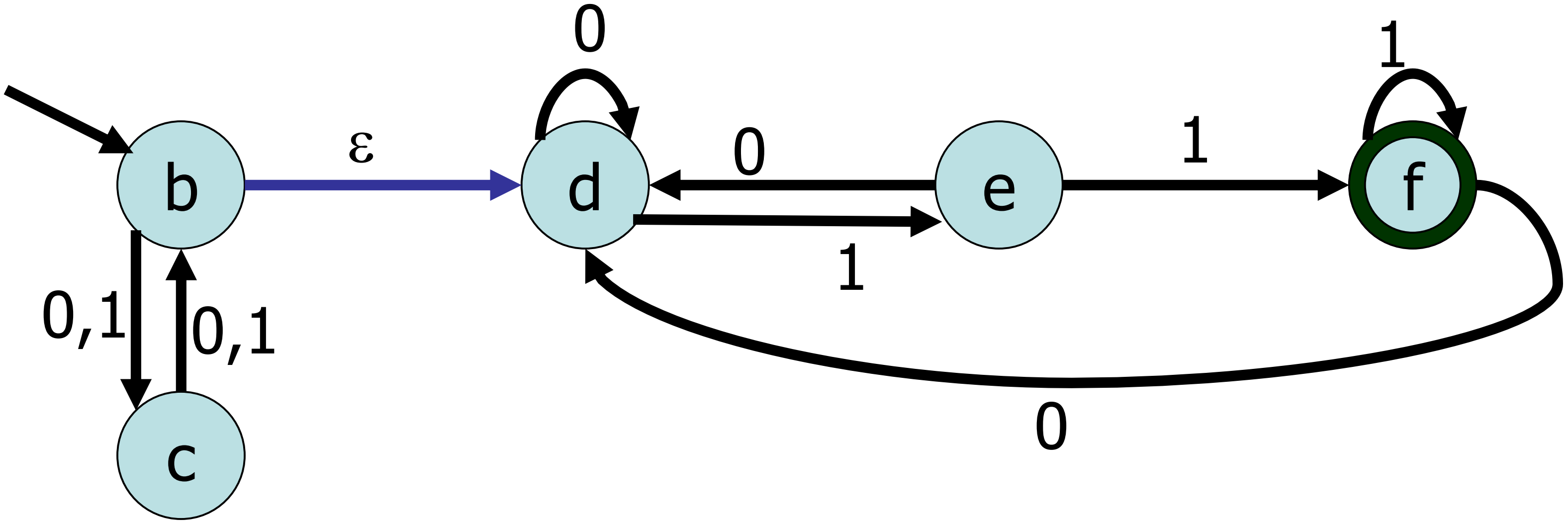
Now, let's deal with concatenation!

- The concatenation *A*•*B* is formed by putting the automata in serial
- The start state comes from *A* while the accept states come from *B*.
- *A*'s accept states are turned off and connected via ε-edges to *B* 's start state:



Concatenation Example

- L = {x has even length} • {x ends with 11}



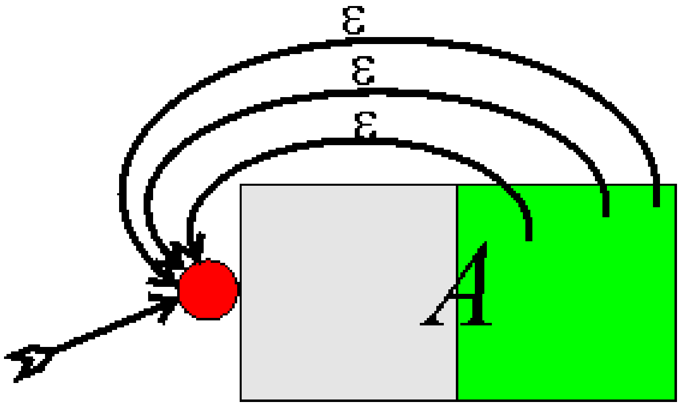
- This example is somewhat questionable… Why?

Now let's go back to our first example

- Given the DFA for
  - L_1 = { $x \in \{0,1\}^*$ | x is any string}
  - L_2 = { $x \in \{0,1\}^*$ | x is composed of one or more 0s}
- Draw the NFA for
  - $L_1 \bullet L_2$ = { $x \in \{0,1\}^*$ | x is any string that ends with at least one 0}

NFA’s: Kleene-+.

- The Kleene-+ A^+ is formed by creating a feedback loop.
- The accept states connect to the start state via ε-edges:

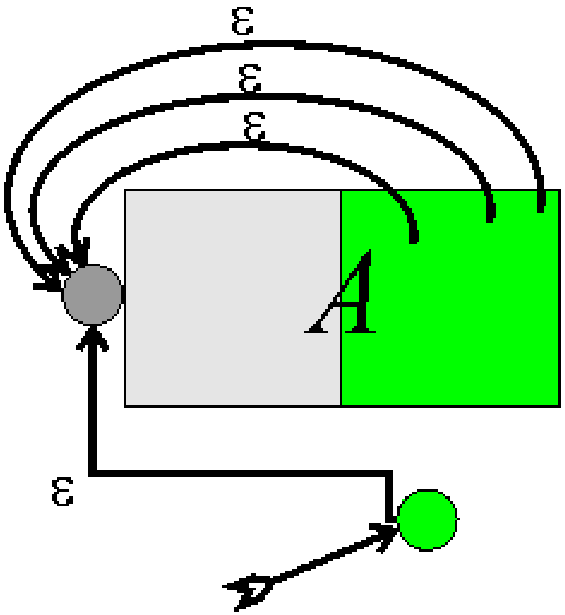


Kleene-+ Example

$L = \left\{ \begin{array}{c} x \text{ is a streak of one or more 1's followed} \\ \text{by a streak of two or more 0's} \end{array} \right\}^+$

NFA's: Kleene-*

- The construction follows from Kleene-+ construction using the fact that A^* is the union of A^+ with the empty string.
- Just create Kleene-+ and add a new start accept state connecting to old start state with an ε-edge:



Closure of NFA under Regular Operations

- The constructions above all show that NFA's are *constructively* closed under the regular operations.

- Theorem: If L_1 and L_2 are accepted by NFA's, then so are:
  - $L_1 \cup L_2$
  - $L_1 \bullet L_2$,
  - L_1^+
  - L_1^*

  The accepting NFA's can be constructed in linear time.

Closure of NFA under Regular Operations

- The constructions above all show that NFA's are *constructively* closed under the regular operations.

- Theorem: If L_1 and L_2 are accepted by NFA's, then so are:
  - $L_1 \cup L_2$
  - $L_1 \bullet L_2$,
  - L_1^+
  - L_1^*

  The accepting NFA's can be constructed in linear time.

- If we can show that all NFA's can be converted into FA's this will show that FA's , and hence regular languages, are closed under the regular operations.

NFA → FA ?!?

- The regular languages were defined to be the languages accepted by FA's, which are by default, *deterministic*.
- It would be nice if NFA's could be "determinized" and converted to FA's
- If so, we could proof that regular languages are closed under regular operations
- Let's try this next.

NFA's have 3 types of non-determinism

Nondeterminism type	Machine Analog	δ -function	Easy to fix?	Formally
Under-determined	Crash	No output	yes, fail-state	$\lvert\delta(q,a)\rvert = 0$
Over-determined	Random choice	Multi-valued	no	$\lvert\delta(q,a)\rvert > 1$
ε	Pause reading	*Redefine alphabet*	no	$\lvert\delta(q,\varepsilon)\rvert > 0$

Determinizing NFA’s: Example

- Idea: convert the NFA into an equivalent DFA that simulates it

- In the DFA, we keep track of all possible active states as the input is being read. If at the end, one of the active states is an accept state, the input is accepted.

One-Slide-Recipe to Derandomize

- Instead of the states in the NFA, we consider the power-states in the FA. (If the NFA has n states, the FA has 2^n states.)

- First we figure out which power-states will reach which power-states in the FA. (Using the rules of the NFA.)

One-Slide-Recipe to Derandomize

- Instead of the states in the NFA, we consider the power-states in the FA. (If the NFA has n states, the FA has 2^n states.)

- First we figure out which power-states will reach which power-states in the FA. (Using the rules of the NFA.)

- Then we must add all epsilon-edges: We redirect pointers that are initially pointing to power-state {*a,b,c*} to power-state {*a,b,c,d,e,f*} iif
  there is an epsilon-edge-only-path pointing from any of the states *a,b,c*
  to states *d,e,f* (transitive closure).

One-Slide-Recipe to Derandomize

- Instead of the states in the NFA, we consider the power-states in the FA. (If the NFA has n states, the FA has 2^n states.)

- First we figure out which power-states will reach which power-states in the FA. (Using the rules of the NFA.)

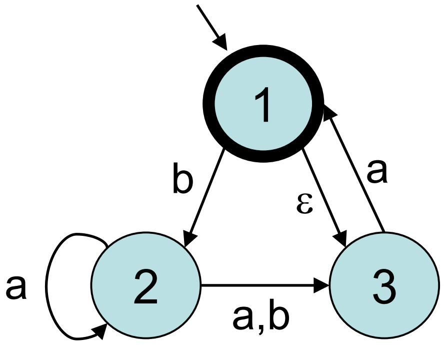
- Then we must add all epsilon-edges: We redirect pointers that are initially pointing to power-state {*a,b,c*} to power-state {*a,b,c,d,e,f*} iif
  there is an epsilon-edge-only-path pointing from any of the states *a,b,c*
  to states *d,e,f* (transitive closure).

- We do the same for the starting state:
  starting state of DFA = {starting state of NFA +
  all NFA states that can recursively be reached from there}

- FA's accepting states are all states that include an accepting NFA state.

Determinizing NFA's: Example

- Let's derandomize the following NFA



Remarks

- The previous recipe can be made totally formal. More details can be found in the reading material.

- Just following the recipe will often produce a too complicated FA. Sometimes obvious simplifications can be made. In general however, this is not an easy task.

NFA → FA

- Summary: Starting from any NFA, we can use subset construction and the epsilon-transitive-closure to find an equivalent FA accepting the same language. Thus,

- Theorem: If L is any language accepted by an NFA, then there exists a constructible [deterministic] FA which also accepts L.

- Corollary: The class of regular languages is closed under the regular operations.

- Proof: Since NFA’s are closed under regular operations, and FA’s are by default also NFA’s, we can apply the regular operations to any FA’s and determinize at the end to obtain an FA accepting the language defined by the regular operations.

Back to Regular Expressions (REX)

- We just saw that DFA and a NFA have actually the same expressive power, they are equivalent.

- What about regular expressions?

Back to Regular Expressions (REX)

- Regular expressions enable to symbolize a sequence of regular operations, and so a way of generating new languages from old.

- For example, to generate the regular language {banana,nab}* from the atomic languages {a},{b} and {n} we could do the following:

$$((\{b\}\bullet\{a\}\bullet\{n\}\bullet\{a\}\bullet\{n\}\bullet\{a\})\cup(\{n\}\bullet\{a\}\bullet\{b\}))^*$$

- Regular expressions specify the same in a more compact form:

$$(\mathrm{banana}\cup\mathrm{nab})^*$$

Regular Expressions (REX)

- Definition: The set of regular expressions over an alphabet Σ and the languages in Σ^* which they generate are defined recursively:
  - Base Cases: Each symbol $a \in \Sigma$ as well as the symbols ε and $\varnothing$ are regular expressions:
    - a generates the atomic language $L(a) = \{a\}$
    - ε generates the language $L(\varepsilon) = \{\varepsilon\}$
    - $\varnothing$ generates the empty language $L(\varnothing) = \{\ \} = \varnothing$
  - Inductive Cases: if r_1 and r_2 are regular expressions so are $r_1 \cup r_2$, $(r_1)(r_2)$, $(r_1)^*$ and $(r_1)^+$:
    - $L(r_1 \cup r_2) = L(r_1) \cup L(r_2)$, so $r_1 \cup r_2$ generates the union
    - $L((r_1)(r_2)) = L(r_1) \bullet L(r_2)$, so $(r_1)(r_2)$ is the concatenation
    - $L((r_1)^*) = L(r_1)^*$, so $(r_1)^*$ represents the Kleene-*
    - $L((r_1)^+) = L(r_1)^+$, so $(r_1)^+$ represents the Kleene-+

Regular Expressions: Table of Operations including UNIX

Operation	Notation	Language	UNIX
Union	$r_1 \cup r_2$	$L(r_1) \cup L(r_2)$	$r_1 \vert r_2$
Concatenation	$(r_1)(r_2)$	$L(r_1) \bullet L(r_2)$	$(r_1)(r_2)$
Kleene-*	$(r)^*$	$L(r)^*$	$(r)*$
Kleene-+	$(r)^+$	$L(r)^+$	$(r)+$
Exponentiation	$(r)^n$	$L(r)^n$	$(r)\{n\}$

Regular Expressions: Simplifications

- Just as algebraic formulas can be simplified by using less parentheses when the order of operations is clear, regular expressions can be simplified. Using the pure definition of regular expressions to express the language {banana,nab}* we would be forced to write something nasty like

$$((((b)(a))(n))(((a)(n))(a))\cup(((n)(a))(b)))*$$

- Using the operator precedence ordering *, • , ∪ and the associativity of • allows us to obtain the simpler:

$$(\mathrm{banana}\cup\mathrm{nab})*$$

- This is done in the same way as one would simplify the *algebraic* expression with re-ordering disallowed:

$$((((b)(a))(n))(((a)(n))(a))+(((n)(a))(b)))^4 = (\mathrm{banana}+\mathrm{nab})^4$$

Regular Expressions: Example

- Question: Find a regular expression that generates the language consisting of all bit-strings which contain a streak of seven 0's or contain two disjoint streaks of three 1's.
  - Legal: 010000000011010, 01110111001, 111111
  - Illegal: 11011010101, 10011111001010, 00000100000

- Answer: $(0\cup 1)^*(0^7\cup 1^3(0\cup 1)^*1^3)(0\cup 1)^*$
  - An even briefer valid answer is: $\Sigma^*(0^7\cup 1^3\Sigma^*1^3)\Sigma^*$
  - The *official* answer using only the standard regular operations is:

    $$(0\cup 1)^*(0000000\cup 111(0\cup 1)^*111)(0\cup 1)^*$$

  - A brief UNIX answer is:

    ```
    (0|1)*(0{7}|1{3}(0|1)*1{3})(0|1)*
    ```

Regular Expressions: Examples

1) 0^*10^*

2) $(\Sigma\Sigma)^*$

3) $1^*\emptyset$

4) $\Sigma = \{0,1\}$, {w | w has at least one 1}

5) $\Sigma = \{0,1\}$, {w | w starts and ends with the same symbol}

6) {w | w is a numerical constant with sign and/or fractional part}
   - E.g. 3.1415, -.001, +2000

REX → NFA

- Since NFA’s are closed under the regular operations we immediately get

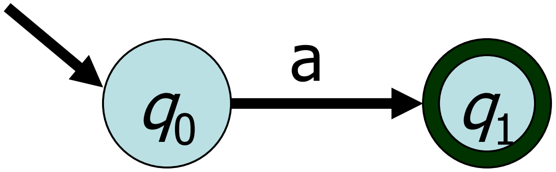
- Theorem:

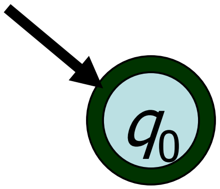
  Given *any* regular expression *r* there is an NFA *N* which simulates *r*. The language accepted by *N* is precisely the language generated by *r:* so that $L(N) = L(r)$.

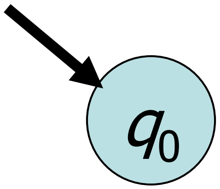
  The NFA is constructible in linear time.

REX → NFA

- *Proof*: The proof works by induction, using the recursive definition of regular expressions. First we need to show how to accept the base case regular expressions $a \in \Sigma$, ε and $\emptyset$. These are respectively accepted by the NFA's:

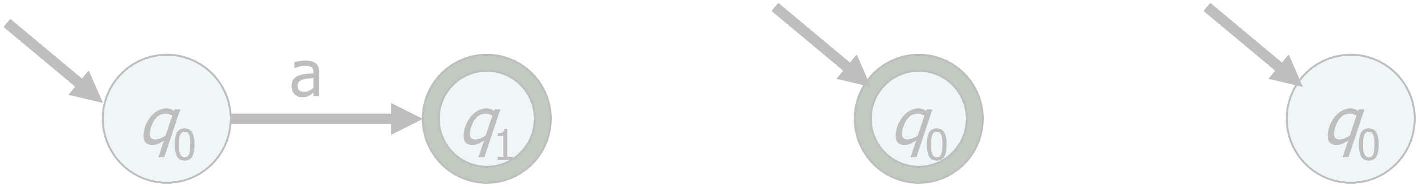




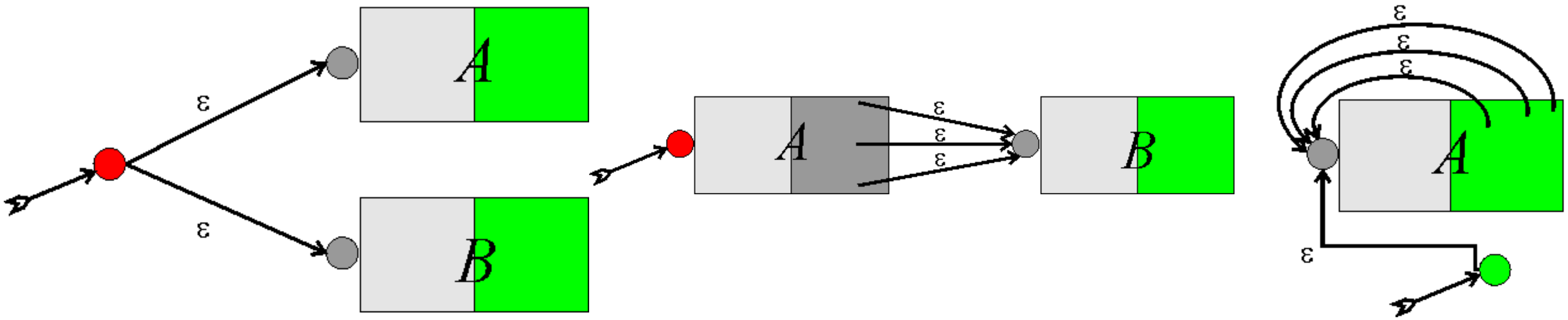


REX → NFA

- *Proof*: The proof works by induction, using the recursive definition of regular expressions. First we need to show how to accept the base case regular expressions $a \in \Sigma$, ε and $\varnothing$. These are respectively accepted by the NFA's:



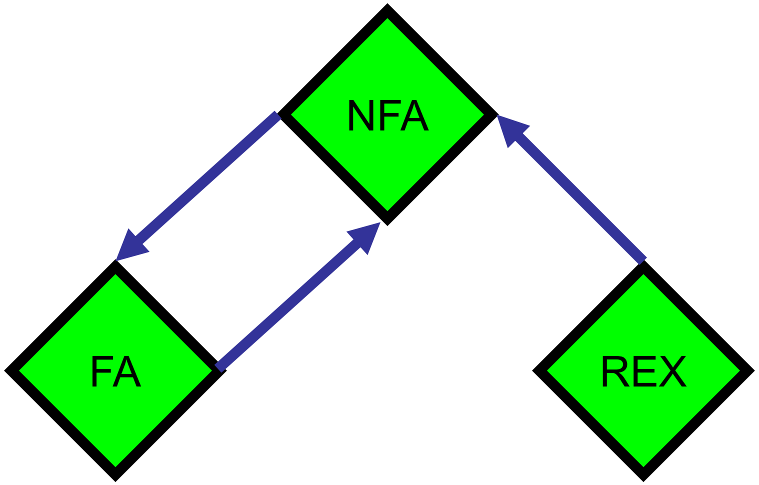
- Finally, we need to show how to inductively accept regular expressions formed by using the regular operations. These are just the constructions that we saw before, encapsulated by:



REX → NFA exercise: Find NFA for $(ab \cup a)^*$

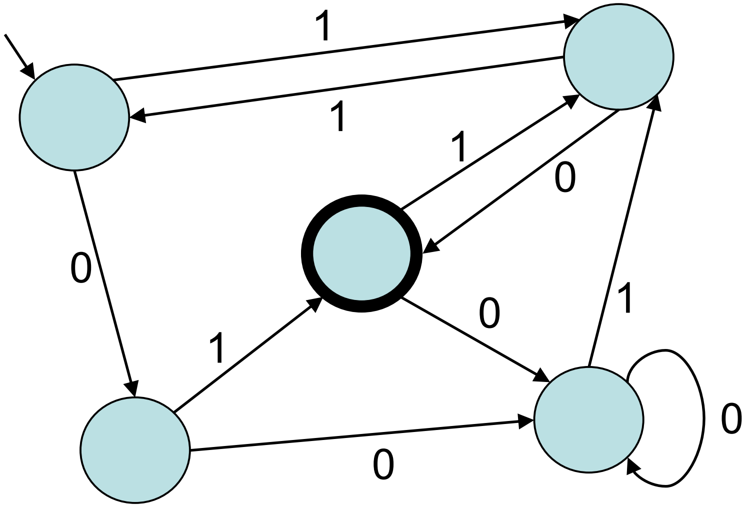
REX → NFA → FA → REX …

- We are one step away from showing that FA's ≈ NFA's ≈ REX's; i.e., all three representation are equivalent. We will be done when we can complete the circle of transformations:



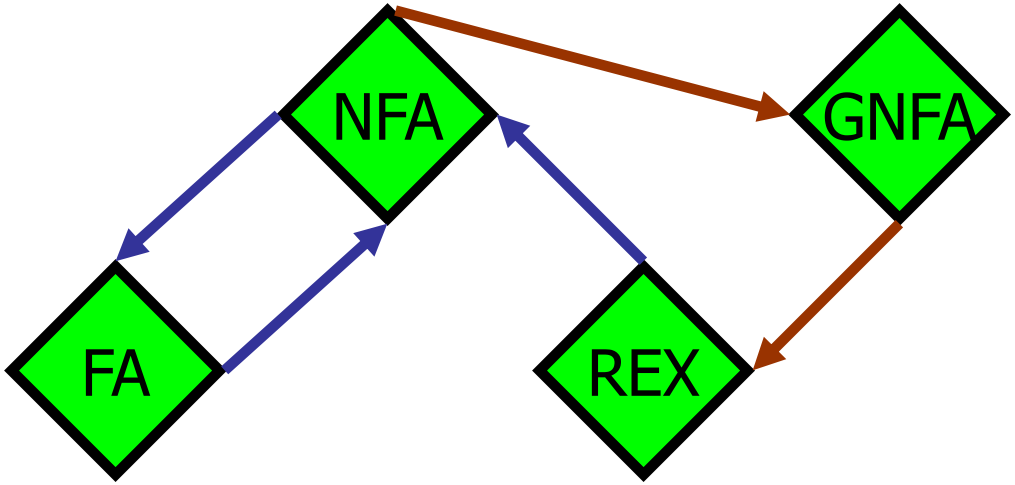
NFA → REX is simple?!?

- Then FA → REX even simpler!
- Please solve this simple example:



REX → NFA → FA → REX …

- In converting NFA's to REX's we'll introduce the most generalized notion of an automaton, the so called "Generalized NFA" or "GNFA". In converting into REX's, we'll first go through a GNFA:



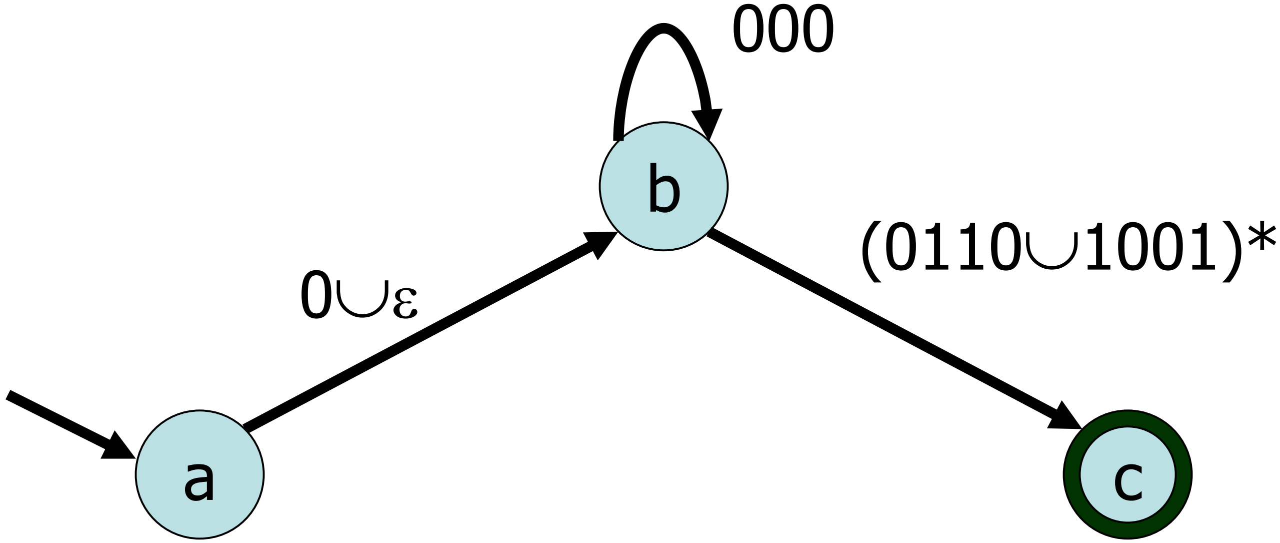
GNFA’s

- Definition: A generalized nondeterministic finite automaton (GNFA) is a graph whose edges are labeled by regular expressions,
  - with a unique start state with in-degree 0, but arrows to every other state
  - and a unique accept state with out-degree 0, but arrows from every other state (note that accept state ≠ start state)
  - and an arrow from any state to any other state (including self).

- A GNFA accepts a string *s* if there exists a path *p* from the start state to the accept state such that *w* is an element of the language generated by the regular expression obtained by concatenating all labels of the edges in *p.*

- The language accepted by a GNFA consists of all the accepted strings of the GNFA.

GNFA Example



- This is a GNFA because edges are labeled by REX's, start state has no in-edges, and the *unique* accept state has no out-edges.

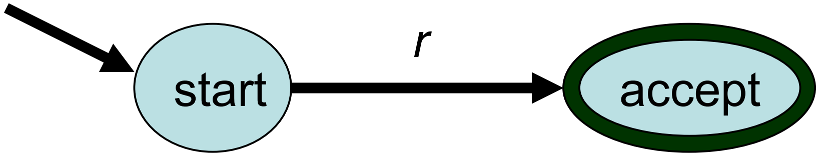
- Convince yourself that 000000100101100110 is accepted.

NFA → REX conversion process

1. Construct a GNFA from the NFA.
   - A. If there are more than one arrows from one state to another, unify them using “∪”
   - B. Create a unique start state with in-degree 0
   - C. Create a unique accept state of out-degree 0
   - D. [If there is no arrow from one state to another, insert one with label Ø]

2. Loop: As long as the GNFA has strictly more than 2 states:

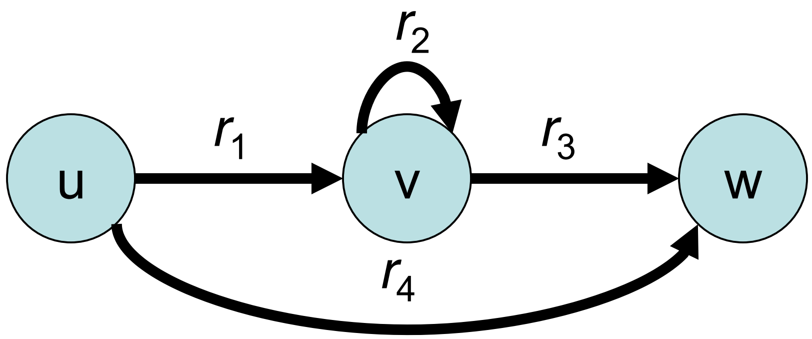
   Rip out arbitrary interior state and modify edge labels.



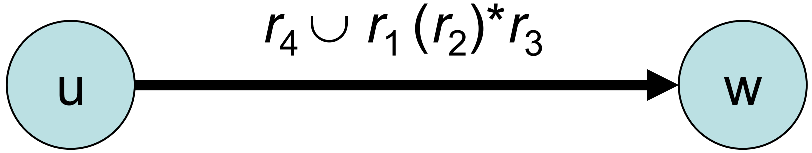
3. The answer is the unique label *r*.

NFA → REX: Ripping Out.

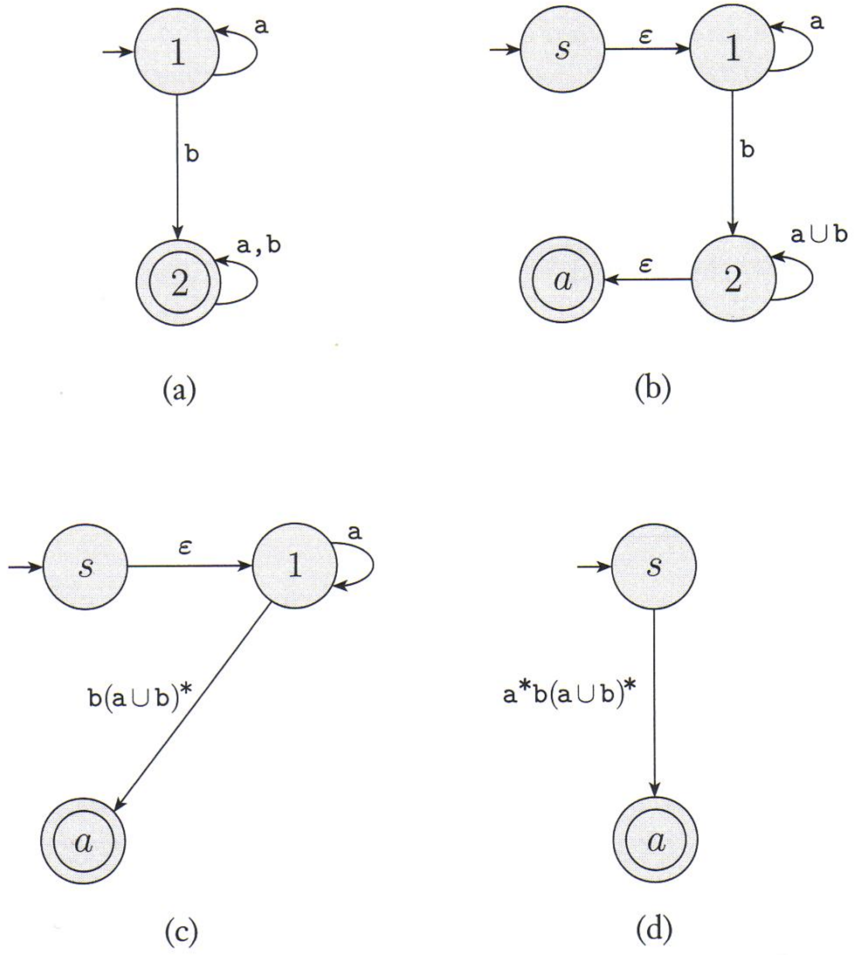
- Ripping out is done as follows. If you want to rip the middle state *v* out (for all pairs of neighbors u,w)…



- … then you'll need to recreate all the lost possibilities from *u* to *w*. I.e., to the current REX label r_4 of the edge (*u*,*w*) you should add the concatenation of the (*u*,*v*) label r_1 followed by the (*v*,*v*)-loop label r_2 repeated arbitrarily, followed by the (*v*,*w*) label $r_{3.}$. The new (*u*,*w*) substitute would therefore be:

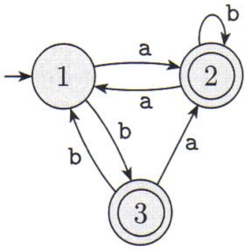


FA → REX: Example



(a) (b) (c) (d)

FA → REX: Exercise



(a)

Summary: FA ≈ NFA ≈ REX

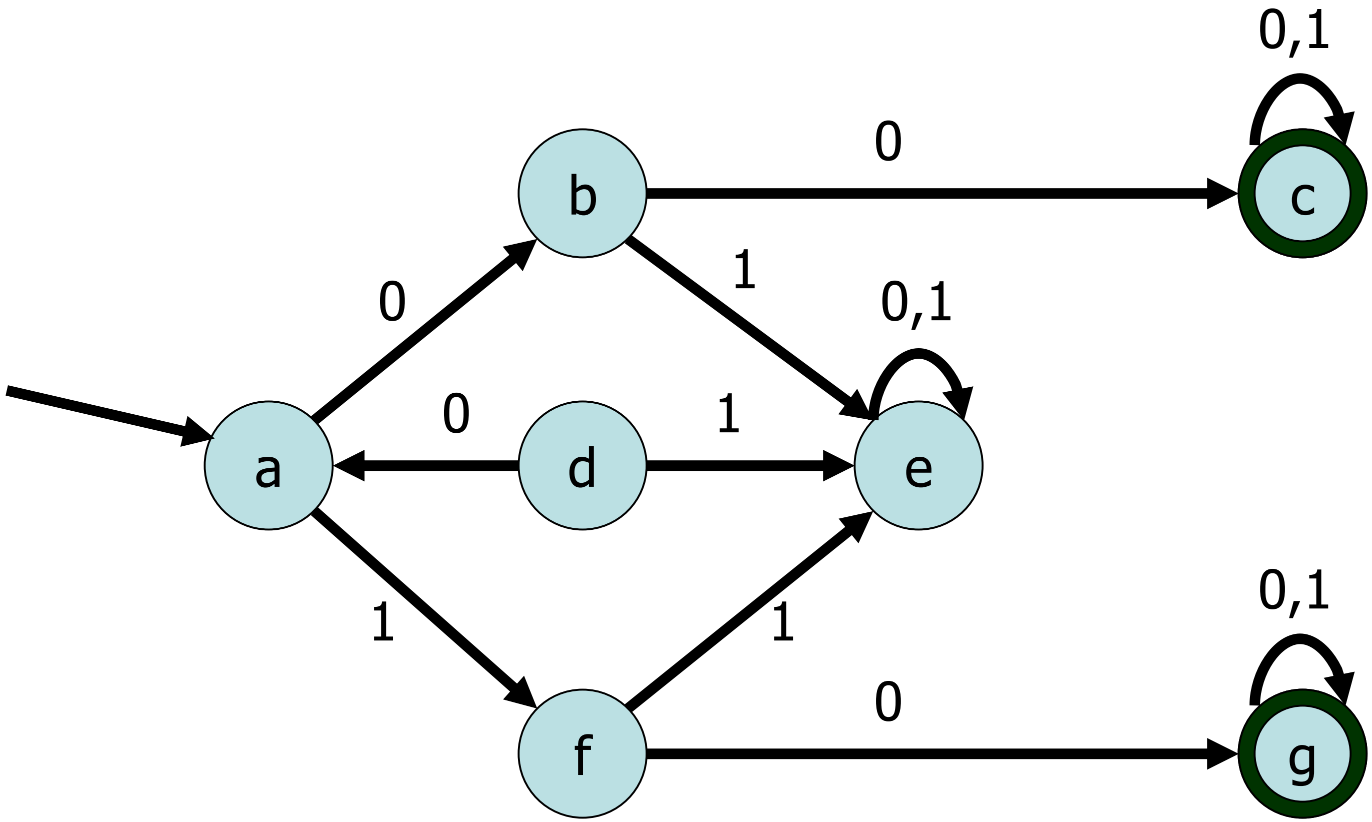
- This completes the demonstration that the three methods of describing regular languages are:

1. Deterministic FA’s
2. NFA’s
3. Regular Expressions

All these are equivalent!

Remark about Automaton Size

- Creating an automaton of small size is often advantageous.
  - Allows for simpler/cheaper hardware, or better exam grades.
  - Designing/Minimizing automata is therefore a funny sport. Example:



Minimization

- Definition: An automaton is irreducible if
  - it contains no useless states, and
  - no two distinct states are equivalent.

Minimization

- Definition: An automaton is irreducible if
  - it contains no useless states, and
  - no two distinct states are equivalent.

- By following these two rules, you can arrive at an “irreducible” FA. Generally, such a local minimum does not have to be a global minimum.

- It can be shown however, that these minimization rules actually produce the global minimum automaton.

Minimization

- Definition: An automaton is irreducible if
  - it contains no useless states, and
  - no two distinct states are equivalent.

- By following these two rules, you can arrive at an "irreducible" FA. Generally, such a local minimum does not have to be a global minimum.

- It can be shown however, that these minimization rules actually produce the global minimum automaton.

- The idea is that two prefixes *u*,*v* are indistinguishable iff for all suffixes *x*, *ux* $\in$ L iff *vx* $\in$ L.

  If u and v are distinguishable, they cannot end up in the same state. Therefore the number of states must be at least as many as the number of pairwise distinguishable prefixes.

Three tough languages

1) $L_1 = \{0^n1^n \mid n \geq 0\}$

2) L_2 = {w | w has an equal number of 0s and 1s}

3) L_3 = {w | w has an equal number of occurrences of
   01 and 10 as substrings}

Three tough languages

1) $L_1 = \{0^n 1^n \mid n \geq 0\}$

2) L_2 = {w | w has an equal number of 0s and 1s}

3) L_3 = {w | w has an equal number of occurrences of 01 and 10 as substrings}

- In order to fully understand regular languages, we also must understand their limitations!