The Coke Vending Machine

- Vending machine dispenses soda for $0.45 a pop.
- Accepts only dimes ($0.10) and quarters ($0.25).
- Eats your money if you don’t have correct change.

- You’re told to “implement” this functionality.

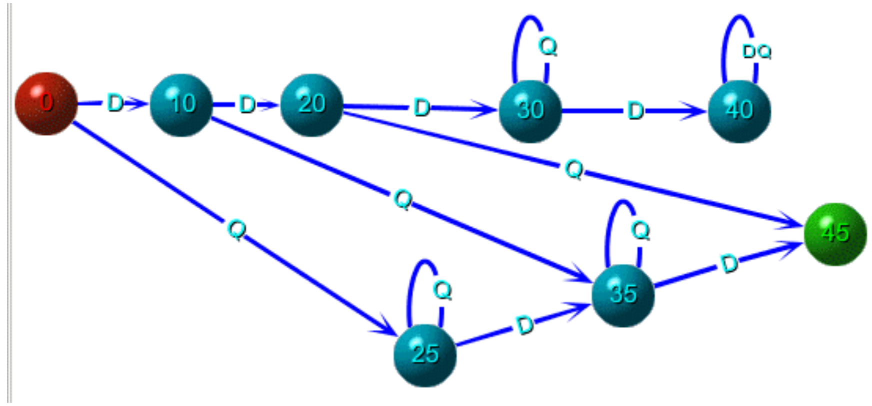
Vending Machine Java Code

```
Soda vend(){
  int total = 0, coin;
  while (total != 45){
      receive(coin);
      if ((coin==10 && total==40)
      ||(coin==25 && total>=25))
            reject(coin);
      else
            total += coin;
  }
  return new Soda();
}
```

Why this was overkill…

- Vending machines have been around long before computers.
  - Or Java, for that matter.
- Don’t really need int’s.
  - Each int introduces 2^{32} possibilities.
- Don’t need to know how to add integers to model vending machine
  - total += coin.
- Java grammar, if-then-else, etc. complicate the essence.

Vending Machine “Logics”



Why was this simpler than Java Code?

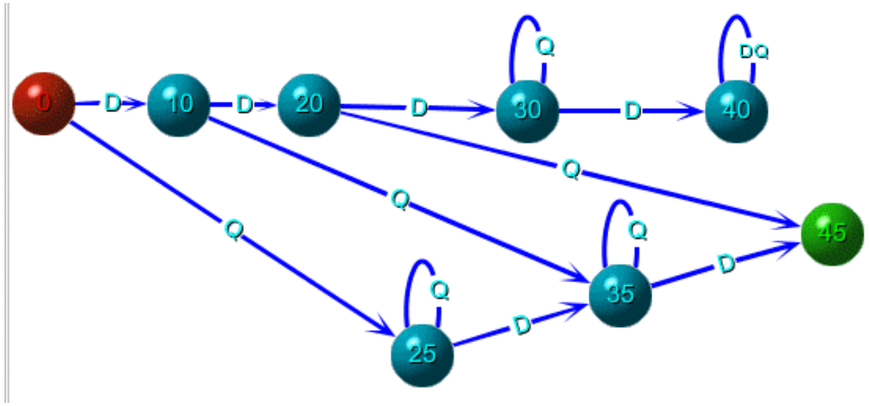
- Only needed two coin types “D” and “Q”
  - symbols/letters in alphabet
- Only needed 7 possible current total amounts
  - states/nodes/vertices
- Much cleaner and more aesthetically pleasing than Java lingo

- Next: generalize and abstract…

Alphabets and Strings

- Definitions:

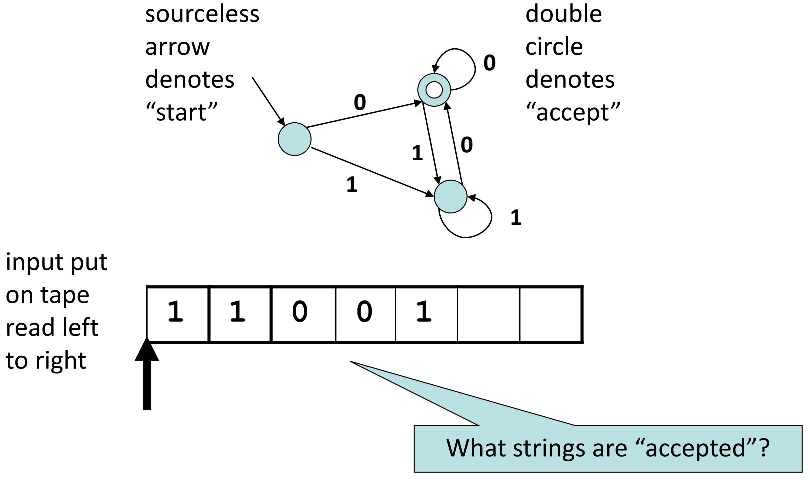
- An alphabet Σ is a set of symbols (characters, letters).
- A string (or word) over Σ is a sequence of symbols.
  - The empty string is the string containing no symbols at all, and is denoted by ε.
  - The length of the string is the number of symbols, e.g. |ε| = 0.



Questions:

1) What is Σ?
2) What are some good or bad strings?
3) What does ε signify here?

Finite Automaton Example



Formal Definition of a Finite Automaton

A finite automaton (FA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$,where

- Q is a finite set called the states
- Σ is a finite set called the alphabet
- $\delta: Q \times \Sigma \to Q$ is the transformation function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accept states (a.k.a. final states).

Formal Definition of a Finite Automaton

A finite automaton (FA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$,where

- Q is a finite set called the states
- Σ is a finite set called the alphabet
- $\delta: Q \times \Sigma \to Q$ is the transformation function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accept states (a.k.a. final states).

- The “input string” and the tape containing it are implicit in the definition. The definition only deals with the *static* view.

  Further explaining is needed for understanding how FA’s interact with their input.

Accept States

- How does an FA operate on strings?

  Imagine an auxiliary tape containing the string.

  The FA reads the tape from left to right with each new character causing the FA to go into another state.

  When the string is completely read, the string is accepted depending on whether the FA’s final state was an accept state.

Accept States

- How does an FA operate on strings?

  Imagine an auxiliary tape containing the string.

  The FA reads the tape from left to right with each new character causing the FA to go into another state.

  When the string is completely read, the string is accepted depending on whether the FA’s final state was an accept state.

- Definition: A string *u* is accepted by an automaton M iff (*if and only if*) the path starting at q_0 which is labeled by *u* ends in an accept state.

Accept States

- How does an FA operate on strings?

  Imagine an auxiliary tape containing the string.

  The FA reads the tape from left to right with each new character causing the FA to go into another state.

  When the string is completely read, the string is accepted depending on whether the FA's final state was an accept state.

- Definition: A string u is accepted by an automaton M iff (*if and only if*) the path starting at q_0 which is labeled by u ends in an accept state.

- This definition is somewhat informal. To really define what it means for a string to label a path, you need to break u up into its sequence of characters and apply δ repeatedly, keeping track of states.

Language

- Definition:

  The **language** accepted by an finite automaton M is the set of all strings which are accepted by M. The language is denoted by $L(M)$.
  We also say that M recognizes L(M), or that M accepts L(M).

- Think of all the possible ways of getting from the start to any accept state.

Language

- Definition:

  The **language** accepted by an finite automaton M is the set of all strings which are accepted by M. The language is denoted by $L(M)$.
  We also say that M recognizes L(M), or that M accepts L(M).

- Think of all the possible ways of getting from the start to any accept state.

- We will eventually see that not all languages can be described as the accepted language of some FA.

- A language L is called a **regular language** if there exists a FA M that recognizes the language L.

Designing Finite Automata

- This is essentially a creative process…

- "You are the automaton" method
  - Given a language (for which we must design an automaton).
  - Pretending to be automaton, you receive an input string.
  - You get to see the symbols in the string one by one.
  - After each symbol you must determine whether string seen so far is part of the language.
  - Like an automaton, you don't see the end of the string, so you must always be ready to answer right away.

- Main point: What is crucial, what defines the language?!

Find the automata for…

1) $\Sigma = \{0,1\}$,
   Language consists of all strings with odd number of ones.

2) $\Sigma = \{0,1\}$,
   Language consists of all strings with substring "001",
   for example 100101, but not 1010110101.

More examples in the book and in the exercises…

Definition of Regular Language

- Recall the definition of a regular language:

  A language *L* is called a regular language if there exists a FA *M* that recognizes the language L.

- We would like to understand what types of languages are regular.
  Languages of this type are amenable to super-fast recognition.

Definition of Regular Language

- Recall the definition of a regular language:

  A language *L* is called a regular language if there exists a FA *M* that recognizes the language L.

- We would like to understand what types of languages are regular.
  Languages of this type are amenable to super-fast recognition.

- Are the following languages regular?
  - Unary prime numbers: { 11, 111, 11111, 1111111, 11111111111, … }
    $= \{1^2, 1^3, 1^5, 1^7, 1^{11}, 1^{13}, \ldots\} = \{ 1^p \mid p \text{ is a prime number} \}$
  - Palindromic bit strings: $\{\varepsilon, 0, 1, 00, 11, 000, 010, 101, 111, \ldots\}$

Finite Languages

- All the previous examples had the following property in common:
  ***infinite* cardinality**

- Before looking at infinite languages, we should look at finite languages.

Finite Languages

- All the previous examples had the following property in common: ***infinite* cardinality**

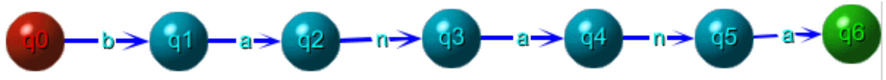
- Before looking at infinite languages, we should look at finite languages.

- Question:

  Is the singleton language containing one string regular?
  For example, is the language {banana} regular?

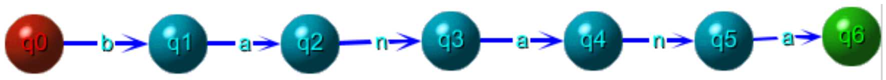
Languages of Cardinality 1

- Answer: Yes.



Languages of Cardinality 1

- Answer: Yes.



- Question: Huh? What's wrong with this automaton?!?
  What if the automation is in state q_1 and reads a "b"?

Languages of Cardinality 1

- Answer: Yes.

- Question: Huh? What's wrong with this automaton?!?
  What if the automation is in state q_1 and reads a "b"?

- Answer:

  This a first example of a nondeterministic FA. The difference between a deterministic FA (DFA) and a nondeterministic FA (NFA) is that every DFA state has one exiting transition arrow for each symbol of the alphabet.

Languages of Cardinality 1

- Answer: Yes.

- Question: Huh? What's wrong with this automaton?!?
  What if the automation is in state q_1 and reads a "b"?

- Answer:

  This a first example of a nondeterministic FA. The difference between a deterministic FA (DFA) and a nondeterministic FA (NFA) is that every DFA state has one exiting transition arrow for each symbol of the alphabet.

- Question: Is there a way of fixing it and making it deterministic?

Arbitrary Finite Number of Finite Strings

- Theorem: All finite languages are regular.

Arbitrary Finite Number of Finite Strings

- Theorem: All finite languages are regular.

- Proof:

  One can always construct a tree whose leaves are word-ending. Make word endings into accept states, add a fail sink-state and add links to the fail state to finish the construction.

  Since there's only a finite number of finite strings, the automaton is finite.

Arbitrary Finite Number of Finite Strings

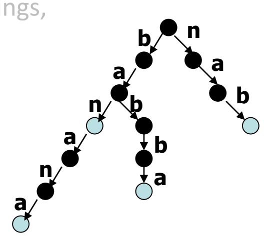
- Theorem: All finite languages are regular.

- Proof:

  One can always construct a tree whose leaves are word-ending. Make word endings into accept states, add a fail sink-state and add links to the fail state to finish the construction.

  Since there's only a finite number of finite strings, the automaton is finite.

- Example for {banana, nab, ban, babba}:



Regular Operations

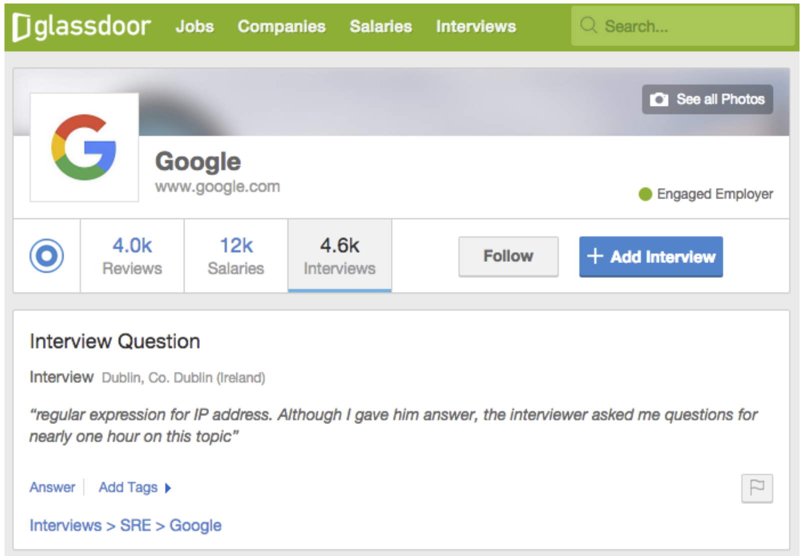
- You may have come across the regular operations when doing advanced searches utilizing programs such as **emacs**, **egrep**, **perl**, **python**, etc.

- There are four basic operations we will work with:
  - Union
  - Concatenation
  - Kleene-Star
  - Kleene-Plus (which can be defined using the other three)

Regular Operations – Summarizing Table

Operation	Symbol	UNIX version	Meaning
Union	∪	|	Match one of the patterns
Concatenation	•	*implicit in* UNIX	Match patterns in sequence
Kleene-star	*	*	Match pattern 0 or more times
Kleene-plus	+	+	Match pattern 1 or more times

Regular Operations matters!



Regular operations: Union

- In UNIX, to search for all lines containing vowels in a text one could use the command
  - **`egrep -i `a|e|i|o|u'`**
  - Here the pattern "*vowel*" is matched by any line containing a vowel.
  - A good way to define a pattern is as a set of strings, i.e. a language. The language for a given pattern is the set of all strings satisfying the predicate of the pattern.

Regular operations: Union

- In UNIX, to search for all lines containing vowels in a text one could use the command
  - `egrep -i `a|e|i|o|u'`
  - Here the pattern "*vowel*" is matched by any line containing a vowel.
  - A good way to define a pattern is as a set of strings, i.e. a language. The language for a given pattern is the set of all strings satisfying the predicate of the pattern.
- In UNIX, a pattern is implicitly assumed to occur as a substring of the matched strings. In our course, however, a pattern needs to specify the whole string, not just a substring.

Regular operations: Union

- In UNIX, to search for all lines containing vowels in a text one could use the command
  - `egrep -i `a|e|i|o|u'`
  - Here the pattern "*vowel*" is matched by any line containing a vowel.
  - A good way to define a pattern is as a set of strings, i.e. a language. The language for a given pattern is the set of all strings satisfying the predicate of the pattern.
- In UNIX, a pattern is implicitly assumed to occur as a substring of the matched strings. In our course, however, a pattern needs to specify the whole string, not just a substring.
- Computability: Union is exactly what we expect.
  If you have patterns A = {aardvark}, B = {bobcat}, C = {chimpanzee},
  the union of these is $A \cup B \cup C$ = {aardvark, bobcat, chimpanzee}.

Regular operations: Concatenation

- To search for all consecutive double occurrences of vowels, use:
  - **`egrep -i `(a|e|i|o|u)(a|e|i|o|u)'`**
  - Here the pattern "*vowel*" has been repeated. Parentheses have been introduced to specify where exactly in the pattern the concatenation is occurring.

Regular operations: Concatenation

- To search for all consecutive double occurrences of vowels, use:
  - `egrep -i `(a|e|i|o|u)(a|e|i|o|u)'`
  - Here the pattern "*vowel*" has been repeated. Parentheses have been introduced to specify where exactly in the pattern the concatenation is occurring.
- Computability: Consider the previous result:
  L = {aardvark, bobcat, chimpanzee}.

  When we concatenate L with itself we obtain:
  $L \bullet L$ = {aardvark, bobcat, chimpanzee} $\bullet$ {aardvark, bobcat, chimpanzee} =
  {aardvarkaardvark, aardvarkbobcat, aardvarkchimpanzee,
  bobcataardvark, bobcatbobcat, bobcatchimpanzee,
  chimpanzeeaardvark, chimpanzeebobcat, chimpanzeechimpanzee}

Regular operations: Kleene-*

- We continue the UNIX example: now search for lines consisting purely of vowels (including the empty line):
  - ``egrep -i `^(a|e|i|o|u)*$’``
  - Note: ^ and $ are special symbols in UNIX regular expressions which respectively anchor the pattern at the *beginning* and *end* of a line. The trick above can be used to convert any Computability regular expression into an equivalent UNIX form.

Regular operations: Kleene-*

- We continue the UNIX example: now search for lines consisting purely of vowels (including the empty line):
  - ``egrep -i `^(a|e|i|o|u)*$’``
  - Note: ^ and $ are special symbols in UNIX regular expressions which respectively anchor the pattern at the *beginning* and *end* of a line. The trick above can be used to convert any Computability regular expression into an equivalent UNIX form.

- Computability: Suppose we have a language B = {ba, na}.
  Question: What is the language B*?

Regular operations: Kleene-*

- We continue the UNIX example: now search for lines consisting purely of vowels (including the empty line):
  - ``egrep -i `^(a|e|i|o|u)*$’``
  - Note: ^ and $ are special symbols in UNIX regular expressions which respectively anchor the pattern at the *beginning* and *end* of a line. The trick above can be used to convert any Computability regular expression into an equivalent UNIX form.

- Computability: Suppose we have a language B = {ba, na}.
  Question: What is the language B*?

- Answer: B * = { ba,na }* = {ε, ba, na, baba, bana, naba, nana, bababa, babana, banaba, banana, nababa, nabana, nanaba, nanana, babababa, babababana, … }

Regular operations: Kleene-+

- Kleene-+ is just like Kleene-* except that the pattern is forced to occur *at least once*.
- UNIX: search for lines consisting purely of vowels (not including the empty line):
  - ``egrep -i `^(a|e|i|o|u)+$’``

Regular operations: Kleene-+

- Kleene-+ is just like Kleene-* except that the pattern is forced to occur *at least once*.
- UNIX: search for lines consisting purely of vowels (not including the empty line):
  - `egrep -i `^(a|e|i|o|u)+$'`

- Computability:

  Suppose we have a language B = {ba, na}.
  What is B^+ and how does it defer from B^*?

Regular operations: Kleene-+

- Kleene-+ is just like Kleene-* except that the pattern is forced to occur *at least once*.
- UNIX: search for lines consisting purely of vowels (not including the empty line):
  - `egrep -i `^(a|e|i|o|u)+$'`

- Computability:

  Suppose we have a language B = {ba, na}.
  What is B^+ and how does it defer from B^*?

  B^+ = {ba, na}$^+$ = {ba, na, baba, bana, naba, nana, bababa, babana, banaba, banana, nababa, nabana, nanaba, nanana, babababa, babababana, … }

  The only difference is the absence of ε

Closure of Regular Languages

- When applying regular operations to regular languages, regular languages result. That is, regular languages are closed under the operations of *union*, *concatenation*, and *Kleene-**.

- Goal: Show that regular languages are *closed* under regular operations. In particular, given regular languages L_1 and L_2, show:

1. $L_1 \cup L_2$ is regular,
2. $L_1 \bullet L_2$ is regular,
3. L_1^* is regular.

- No.'s 2 and 3 are deferred until we learn about NFA's.
- However, No. 1 can be achieved immediately.

Union Example

- Problem: Draw the FA for

$$L = \{ x \in \{0,1\}^* \mid |x|=\text{even or } x \text{ ends with } 11\}$$

Let's start by drawing M_1 and M_2, the automaton recognizing L_1 and L_2

- L_1 = { $x \in \{0,1\}^*$ | x has even length}

- L_2 = { $x \in \{0,1\}^*$ | x ends with 11 }

Cartesian Product Construction

- We want to construct a finite automaton M that recognizes any strings belonging to L_1 or L_2.
- Idea: Build M such that it simulates *both* M_1 and M_2 simultaneously and accept if either of the automatons accepts

Cartesian Product Construction

- We want to construct a finite automaton M that recognizes any strings belonging to L_1 or L_2.
- Idea: Build M such that it simulates *both* M_1 and M_2 simultaneously and accept if either of the automatons accepts
- Definition: The Cartesian product of two sets A and B, denoted by $A \times B$, is the set of all ordered pairs (a,b) where $a \in A$ and $b \in B$.

Formal Definition

- Given two automata $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$
- Define the unioner of M_1 and M_2 by: $M_\cup = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_1, q_2), F_\cup)$

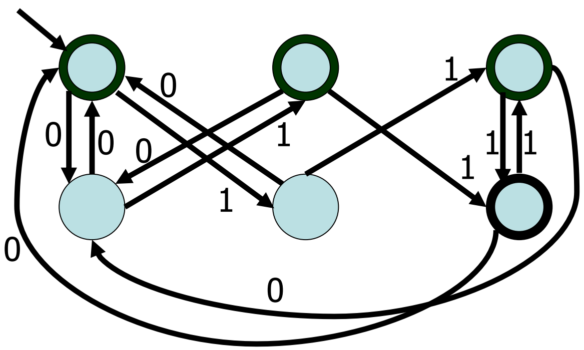
  - where the accept state (q_1, q_2) is the combined start state of both automata

  - where $F_\cup$ is the set of ordered pairs in $Q_1 \times Q_2$with at least one state an accept state. That is: $F_\cup = Q_1 \times F_2 \cup F_1 \times Q_2$

  - where the transition function δ is defined as $\delta\big((q_1, q_2), j\big) = \big(\delta_1(q_1, j), \delta_2(q_2, j)\big) = \delta_1 \times \delta_2$

Union Example: $L_1 \cup L_2$

- When using the Cartesian Product Construction:

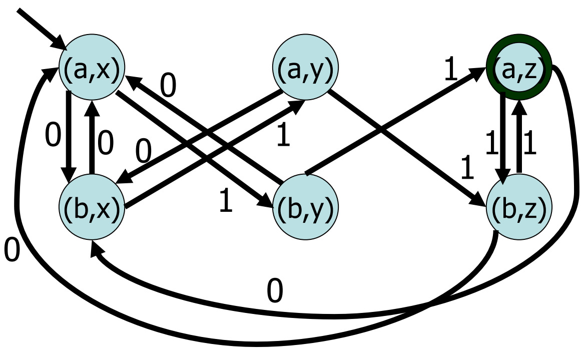


Other constructions: Intersector

- Other constructions are possible, for example an intersector:
- Accept only when both ending states are accept states. So the only difference is in the set of accept states. Formally the intersector of M_1 and M_2 is given by
$M_\cap = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{0,1}, q_{0,2}), F_\cap)$, where $F_\cap = F_1 \times F_2$.

Other constructions: Intersector

- Other constructions are possible, for example an intersector:
- Accept only when both ending states are accept states. So the only difference is in the set of accept states. Formally the intersector of M_1 and M_2 is given by
$M_\cap = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{0,1}, q_{0,2}), F_\cap)$, where $F_\cap = F_1 \times F_2$.



Other constructions: Difference

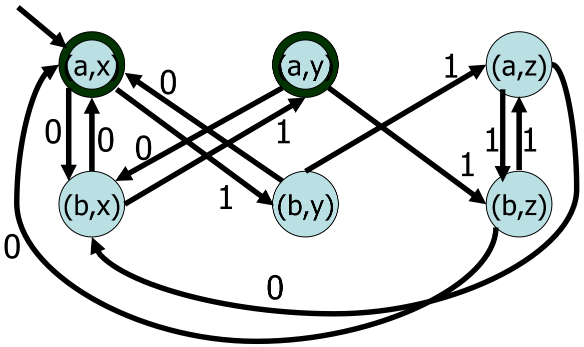
- The difference of two sets is defined by $A - B = \{x \in A \mid x \notin B\}$
- In other words, accept when first automaton accepts and second does not

$$M_- = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{0,1}, q_{0,2}), F_-),$$
where $F_- = F_1 \times Q_2 - Q_1 \times F_2$

Other constructions: Difference

- The difference of two sets is defined by $A - B = \{x \in A \mid x \notin B\}$
- In other words, accept when first automaton accepts and second does not

$$M_{-} = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{0,1}, q_{0,2}), F_{-}),$$
where $F_{-} = F_1 \times Q_2 - Q_1 \times F_2$



Other constructions: Symmetric difference

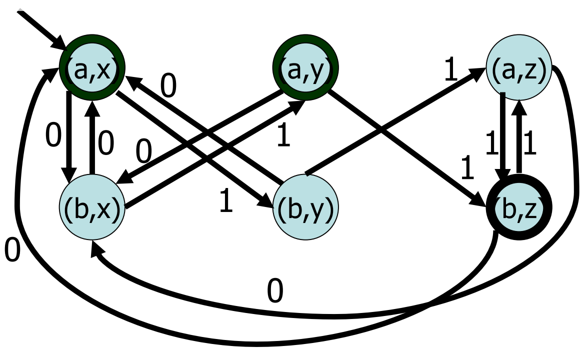
- The symmetric difference of two sets A, B is $A \oplus B = A \cup B - A \cap B$
- Accept when exactly one automaton accepts:

$M_{\oplus} = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{0,1}, q_{0,2}), F_{\oplus})$, where $F_{\oplus} = F_{\cup} - F_{\cap}$

Other constructions: Symmetric difference

- The symmetric difference of two sets A, B is $A \oplus B = A \cup B - A \cap B$
- Accept when exactly one automaton accepts:

$M_{\oplus} = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{0,1}, q_{0,2}), F_{\oplus})$, where $F_{\oplus} = F_{\cup} - F_{\cap}$



Complement

- How about the complement? The complement is only defined with respect to some universe.
- Given the alphabet Σ, the *default universe* is just the set of all possible strings Σ^*. Therefore, given a language L over Σ, i.e. $L \subseteq \Sigma^*$ the complement of L is $\Sigma^* - L$
- Note: Since we know how to compute set difference, and we know how to construct the automaton for Σ^* we can construct the automaton for $\overline{L}$.

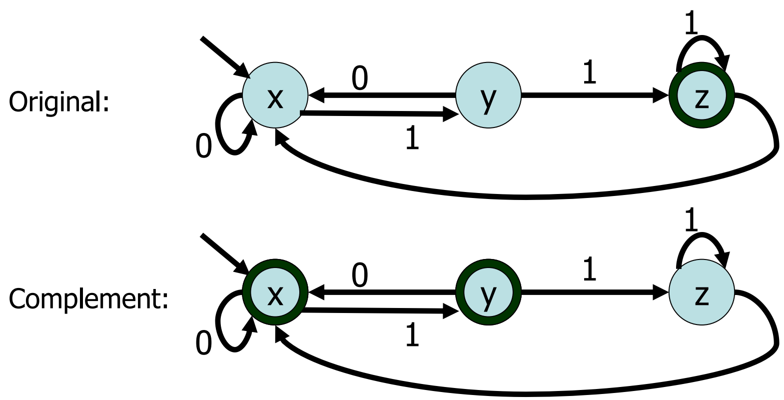
Complement

- How about the complement? The complement is only defined with respect to some universe.
- Given the alphabet Σ, the *default universe* is just the set of all possible strings Σ^*. Therefore, given a language L over Σ, i.e. $L \subseteq \Sigma^*$ the complement of L is $\Sigma^* - L$
- Note: Since we know how to compute set difference, and we know how to construct the automaton for Σ^* we can construct the automaton for $\bar{L}$.
- Question: Is there a simpler construction for $\bar{L}$?

Complement

- How about the complement? The complement is only defined with respect to some universe.
- Given the alphabet Σ, the *default universe* is just the set of all possible strings Σ^*. Therefore, given a language L over Σ, i.e. $L \subseteq \Sigma^*$ the complement of L is $\Sigma^* - L$
- Note: Since we know how to compute set difference, and we know how to construct the automaton for Σ^* we can construct the automaton for $\bar{L}$.
- Question: Is there a simpler construction for $\bar{L}$?
- Answer: Just switch accept-states with non-accept states.

Complement Example



Boolean-Closure Summary

- We have shown constructively that regular languages are closed under boolean operations. I.e., given regular languages L_1 and L_2 we saw that:

  1. $L_1 \cup L_2$ is regular,
  2. $L_1 \cap L_2$ is regular,
  3. $L_1 - L_2$ is regular,
  4. $L_1 \oplus L_2$ is regular,
  5. $\bar{L_1}$ is regular.

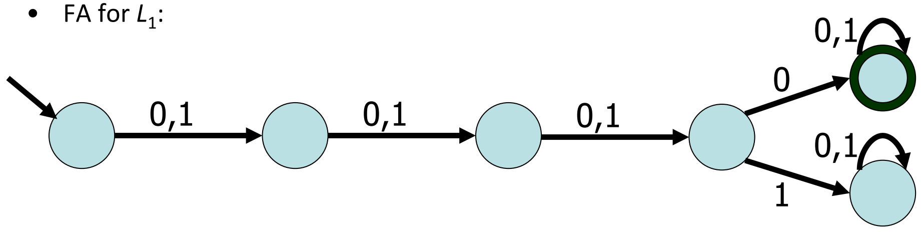
- No. 2 to 4 also happen to be regular operations. We still need to show that regular languages are closed under concatenation and Kleene-*.
- Tough question: Can't we do a similar trick for concatenation?

Back to Nondeterministic FA

- Question: Draw an FA which accepts the language

  $L_1 = \{ x \in \{0,1\}^* \mid 4^{th} \text{ bit from left of } x \text{ is } 0 \}$

- FA for L_1:



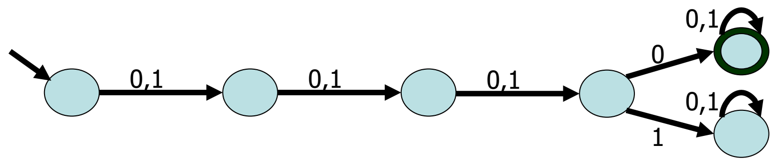
- Question: What about the 4^{th} bit from the *right*?

- Looks as complicated: $L_2 = \{ x \in \{0,1\}^* \mid 4^{th} \text{ bit from right of } x \text{ is } 0 \}$

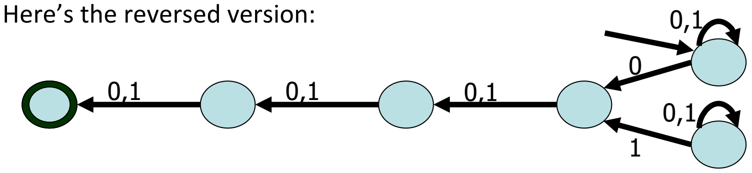
Weird Idea

- Notice that L_2 is the reverse L_1.

- I.e. saying that 0 should be the 4^{th} from the left is reverse of saying that 0 should be 4^{th} from the right. Can we simply reverse the picture (reverse arrows, swap start and accept)?!?



- Here's the reversed version:



Discussion of weird idea

1. Silly unreachable state. *Not pretty, but allowed in model.*

2. Old start state became a crashing accept state. *Underdeterminism. Could fix with fail state.*

3. Old accept state became a state from which we don't know what to do when reading 0. *Overdeterminism. Trouble.*

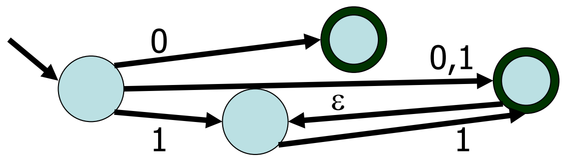
4. (Not in this example, but) There could be more than one start state! *Seemingly outside standard deterministic model.*

- Still, there is something about our automaton. It turns out that NFA's (=Nondeterministic FA) are actually quite useful and are embedded in many practical applications.
- Idea, keep more than 1 active state if necessary.

Introduction to Nondeterministic Finite Automata

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

- Example:

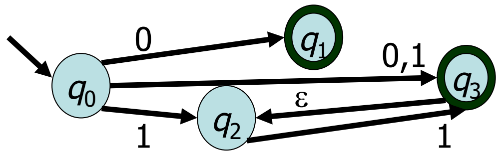


- Any labeled graph you can come up with is an NFA, as long as it only has *one start state*. Later, even this restriction will be dropped.

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 δ has different domain and co- domain: $\delta: Q\times\Sigma_\varepsilon \to P(Q)$
- Here, $P(Q)$ 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.
- Example, for NFA of the previous slide:

$\delta(q_0,0) = \{q_1,q_3\}$,
$\delta(q_0,1) = \{q_2,q_3\}$,
$\delta(q_0,\varepsilon) = \varnothing$,
...
$\delta(q_3,\varepsilon) = \{q_2\}$.

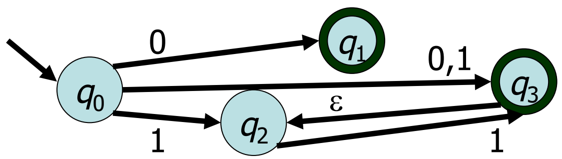


Formal Definition of an NFA: Dynamic

- 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 – 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.
- Definition: 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 $L(M)$.
  - Following a label ε is for free (without reading an input symbol). In computing the label of a path, you should delete all ε's.
  - The only difference in acceptance for NFA's vs. FA's are the words "*there exists*". In FA's the path always exists and is unique.

Example

M_4:

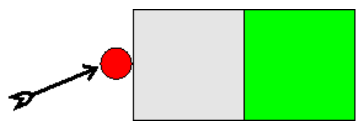


Question: Which of the following strings is accepted?

1. ε
2. 0
3. 1
4. 0111

NFA's vs. Regular Operations

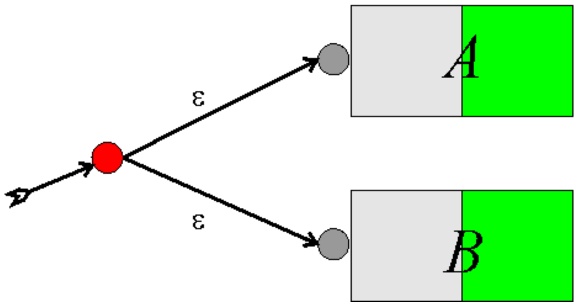
- On the following few slides we will study how NFA's interact with regular operations.
- We will use the following schematic drawing for a general NFA.



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

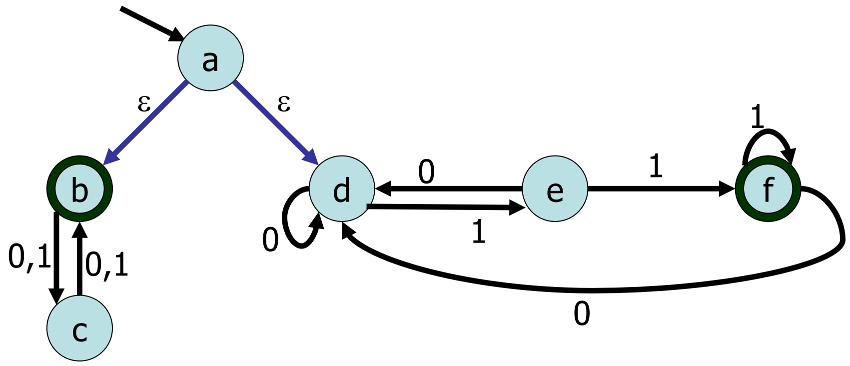
NFA: Union

- 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:



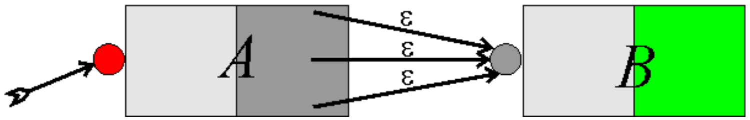
Union Example

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



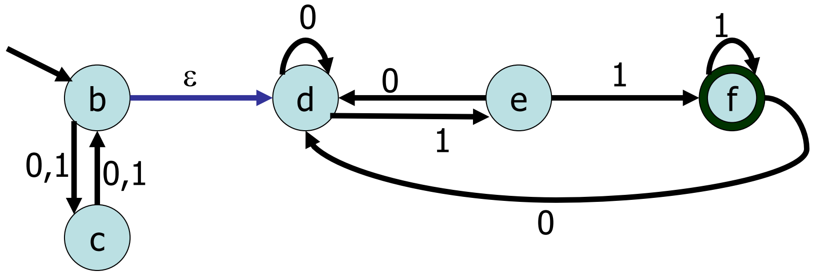
NFA: Concatenation

- The concatenation $A \bullet B$ 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

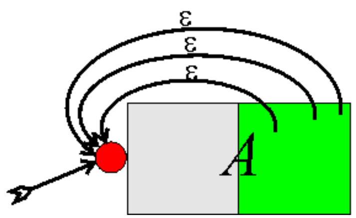
- L = {x has even length} • {x ends with 11}



- Remark: This example is somewhat questionable…

NFA’s: Kleene-+.

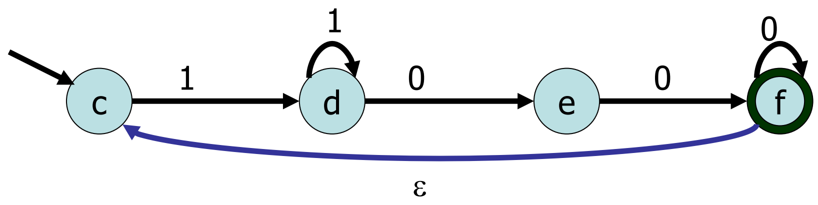
- The Kleene-+ A^+ is formed by creating a feedback loop. The accept states connect to the start state via ε-edges:



Kleene-+ Example

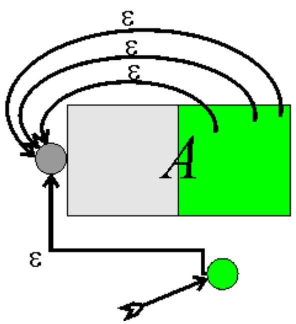
$L = \{$ x is a streak of one or more 1’s followed by a streak of two or more 0’s $\}^+$

$= \{$ x starts with 1, ends with 0, and alternates between one or more consecutive 1’s and two or more consecutive 0’s $\}$



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. More formally,
- 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^+ and L_1^*. In fact, the accepting NFA’s can be constructed in linear time.
- This is almost what we want. 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.

Regular Expressions (REX)

- We are already familiar with the regular operations. Regular expressions give a way of symbolizing a sequence of regular operations, and therefore 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:

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

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	$r_1\cup r_2$	$L(r_1)\cup L(r_2)$	$r_1\|r_2$
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:

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

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

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

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: $(0\cup1)^*(0^7\cup1^3(0\cup1)^*1^3)(0\cup1)^*$
  - An even briefer valid answer is: $\Sigma^*(0^7\cup1^3\Sigma^*1^3)\Sigma^*$
  - The *official* answer using only the standard regular operations is:

    $(0\cup1)^*(0000000\cup111(0\cup1)^*111)(0\cup1)^*$
  - 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

Regular Expressions: A different view…

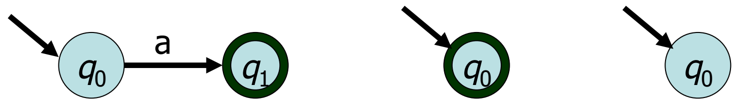
- Regular expressions are just strings. Consequently, the set of all regular expressions is a set of strings, so by definition is a language.
- Question: Supposing that only union, concatenation and Kleene-* are considered. What is the alphabet for the language of regular expressions over the base alphabet Σ ?
- Answer: $\Sigma \cup \{$ (,), $\cup$, * $\}$

REX → NFA

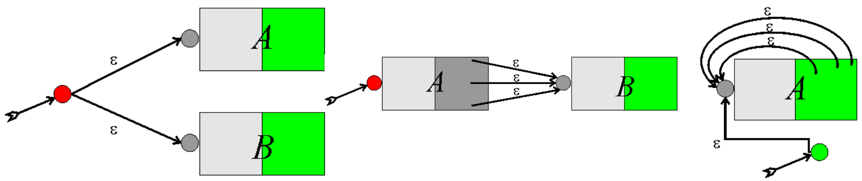
- Since NFA's are closed under the regular operations we immediately get
- Theorem: Given any regular expression *r* there is an NFA *N* which simulates *r*. That is, the language accepted by *N* is precisely the language generated by *r* so that $L(N) = L(r)$. Furthermore, the NFA is constructible in 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 $\varnothing$. 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:



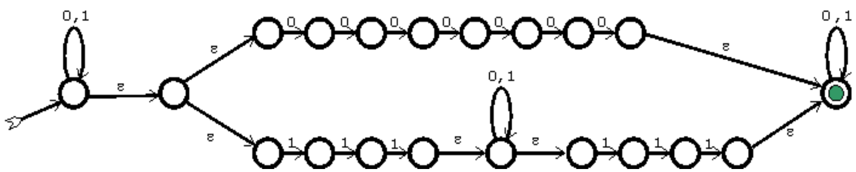
REX → NFA exercise: Find NFA for $(ab \cup a)^*$

REX → NFA: Example

- Question: Find an NFA for the regular expression

  (0∪1)*(0000000∪111(0∪1)*111)(0∪1)*

  of the previous example.



REX → NFA → FA ?!?

- The fact that regular expressions can be converted into NFA's means that it makes sense to call the languages accepted by NFA's "regular."

- However, 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, for then the definition of "regular" languages, as being FA-accepted would be justified.

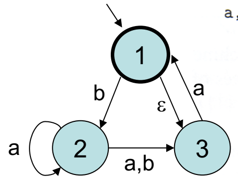
- Let's try this next.

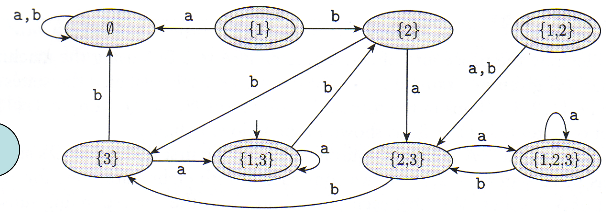
NFA’s have 3 types of non-determinism

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: We might keep track of all parallel active states as the input is being called out. If at the end of the input, one of the active states happened to be an accept state, the input was accepted.
- Example, consider the following NFA, and its deterministic FA.





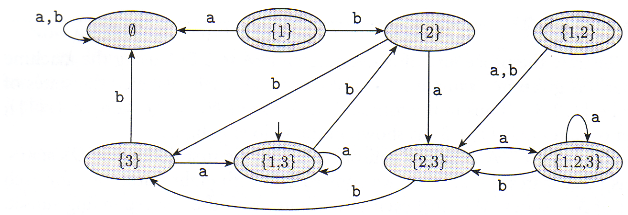
One-Slide-Recipe to Derandomize

- 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}, if and only if there is an epsilon-edge-only-path pointing from any of the states a,b,c to states d,e,f (a.k.a. transitive closure). We do the very same for the starting state: starting state of FA = {starting state of NFA, all NFA states that can recursively be reached from there}
- Accepting states of the FA are all states that include a accepting NFA state.

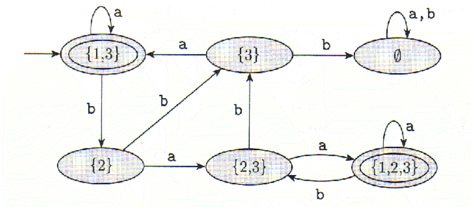
Remarks

- 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.

Automata Simplification

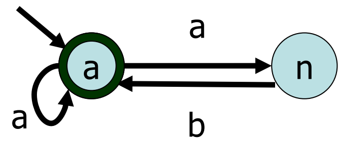


- The FA can be simplified. States {1,2} and {1}, for example, cannot be reached. Still the result is not as simple as the NFA.



Derandomization Exercise

- Exercise: Let's derandomize the simplifed two-state NFA from slide 1/70 which we derived from regular expression $(ab \cup a)^*$

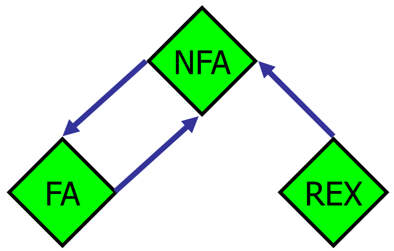


REX → NFA → FA

- 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.

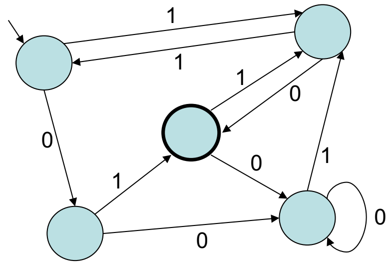
REX → NFA → FA → REX …

- 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:



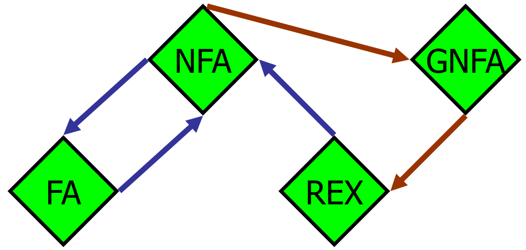
NFA → REX is simple?!?

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



REX → NFA → FA → REX …

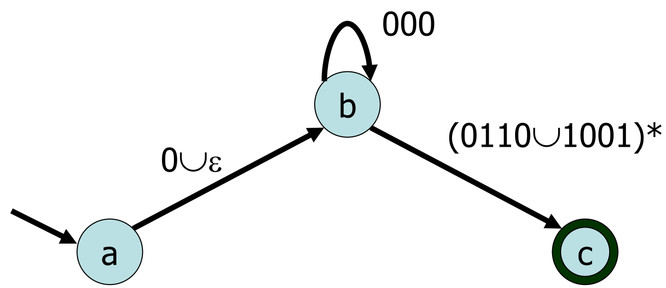
- In converting NFA's to REX's we'll introduce the most generalized notion of an automaton, the so called "Generalized NFA" or "GNFA". In converting into REX's, we'll first go through a GNFA:



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.

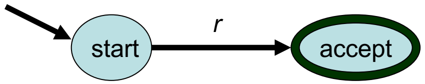
GNFA Example



- 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

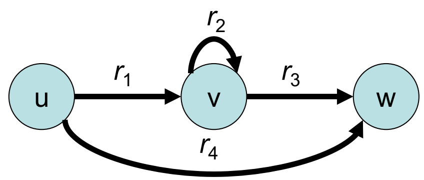
1. Construct a GNFA from the NFA.
   - A. If there are more than one arrows from one state to another, unify them using "∪"
   - B. Create a unique start state with in-degree 0
   - C. Create a unique accept state of out-degree 0
   - D. [If there is no arrow from one state to another, insert one with label Ø]
2. Loop: As long as the GNFA has strictly more than 2 states:
   Rip out arbitrary interior state and modify edge labels.



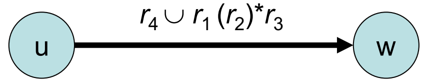
3. The answer is the unique label r.

NFA → REX: Ripping Out.

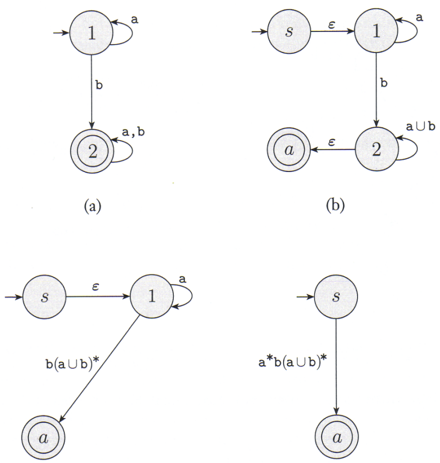
- Ripping out is done as follows. If you want to rip the middle state v out (for all pairs of neighbors u,w)…



- … 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:

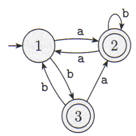


FA → REX: Example



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

FA → REX: Exercise



(a)

Summary: FA ≈ NFA ≈ REX

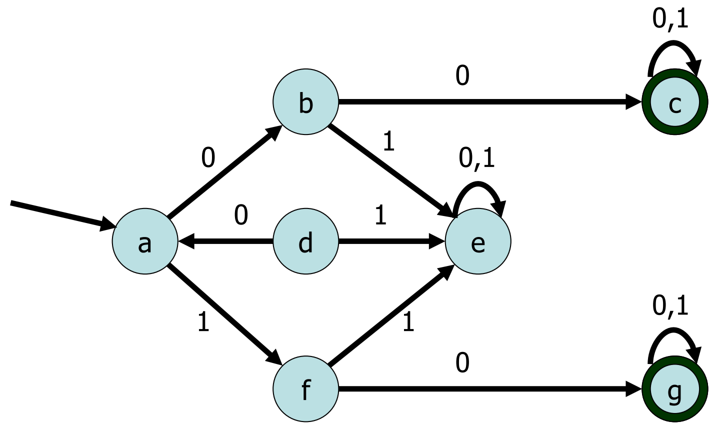
- This completes the demonstration that the three methods of describing regular languages are:

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

- We have learnt that 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:

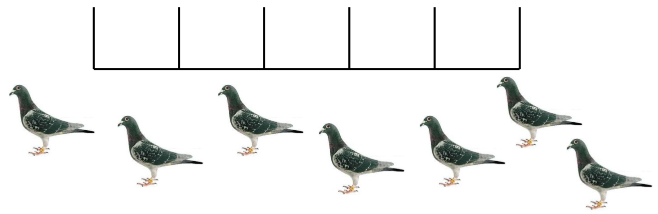


Minimization

- Definition: An automaton is irreducible if
  - it contains no useless states, and
  - no two distinct states are equivalent.
- By just 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 ∈ L iff vx ∈ 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.

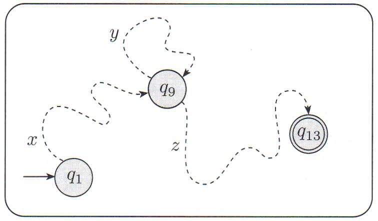
Pigeonhole principle

- Consider language L, which contains word w ∈ L.
- Consider an FA which accepts L, with n < |w| states.
- Then, when accepting w, the FA must visit at least one state twice.
- This is according to the pigeonhole (a.k.a. Dirichlet) principle:
  - If m>n pigeons are put into n pigeonholes, there's a hole with more than one pigeon.
  - That's a pretty fancy name for a boring observation...



Languages with unbounded strings

- Consequently, regular languages with unbounded strings can only be recognized by FA (finite! bounded!) automata if these long strings loop.



- The FA can enter the loop once, twice, …, and not at all.
- That is, language L contains all {xz, xyz, xy^2z, xy^3z, …}.

Pumping Lemma

- Theorem: Given a regular language L, there is a number p (called the pumping number) such that any string in L of length $\geq p$ is pumpable within its first p letters.
- In other words, for all $u \in L$ with $|u| \geq p$ we can write:
  - $u = xyz$ (x is a prefix, z is a suffix)
  - $|y| \geq 1$ (mid-portion y is non-empty)
  - $|xy| \leq p$ (pumping occurs in first p letters)
  - $xy^iz \in L$ for all $i \geq 0$ (can pump y-portion)
- If, on the other hand, there is no such p, then the language is not regular.

Pumping Lemma Example

- Let L be the language $\{0^n1^n \mid n \geq 0\}$
- Assume (for the sake of contradiction) that L is regular
- Let p be the pumping length. Let u be the string 0^p1^p.
- Let's check string u against the pumping lemma:
- "In other words, for all $u \in L$ with $|u| \geq p$ we can write:
  - $u = xyz$ (x is a prefix, z is a suffix)
  - $|y| \geq 1$ → $y = 0^+$ (mid-portion y is non-empty)
  - $|xy| \leq p$ (pumping occurs in first p letters)
  - $xy^iz \in L$ for all $i \geq 0$ (can pump y-portion)"

→ Then, xz or xyyz is not in L. Contradiction!

Let's make the example a bit harder…

- Let L be the language {w | w has an equal number of 0s and 1s}
- Assume (for the sake of contradiction) that L is regular
- Let p be the pumping length. Let u be the string 0^p1^p.
- Let's check string u against the pumping lemma:
- "In other words, for all $u \in L$ with $|u| \geq p$ we can write:
  - $u = xyz$ (x is a prefix, z is a suffix)
  - $|y| \geq 1$ (mid-portion y is non-empty)
  - $|xy| \leq p$ (pumping occurs in first p letters)
  - $xy^iz \in L$ for all $i \geq 0$ (can pump y-portion)"

Harder example continued

- Again, y must consist of 0s only!
- Pump it there! Clearly again, if xyz $\in$ L, then xz or xyyz are not in L.

- There's another alternative proof for this example:
  - 0*1* is regular.
  - $\cap$ is a regular operation.
  - If L regular, then L $\cap$ 0*1* is also regular.
  - However, L $\cap$ 0*1* is the language we studied in the previous example (0^n1^n). A contradiction.

Now you try…

- Is $L_1 = \{ww \mid w \in (0 \cup 1)^*\}$ regular?

- Is $L_2 = \{1^n \mid n \text{ being a prime number }\}$ regular?