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

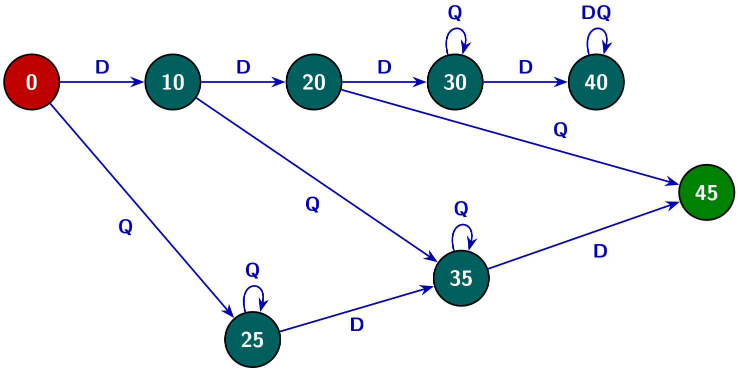
```
Soda vend(){
  int total = 0, coin;
  while (total != 45){
      receive(coin);
      if ((coin==10 && total==40)
      ||(coin==25 && total>=25))
            reject(coin);
      else
            total += coin;
  }
  return new Soda();
}
```

Overkill?!?

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"



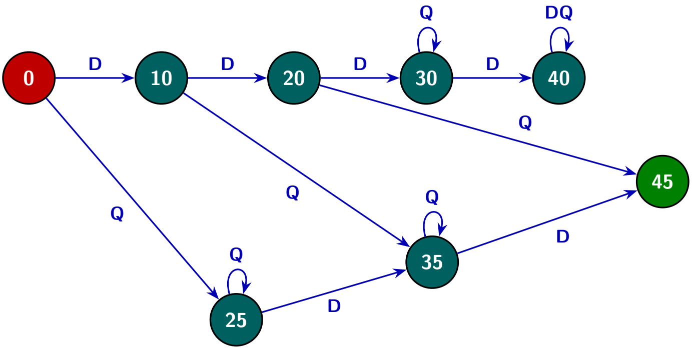
Why was this simpler than Java Code?

- 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. $|\varepsilon| = 0$.

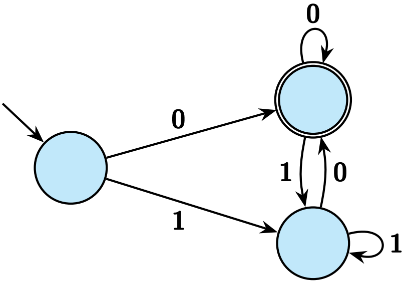


Questions:

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

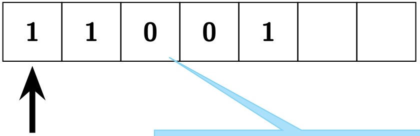
Finite Automaton Example

sourceless arrow denotes "start"



double circle denotes "accept"

input put on tape read left to right



What strings are "accepted"?

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**
- $\delta\colon Q \times \Sigma \to Q$ 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**
- $\delta \colon Q \times \Sigma \to Q$ 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.

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.

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 the path starting at q_0 which is labeled by u ends in an accept state.

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 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.

Definition: Language

- 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.

Definition: Language

- 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.
  - Pretend 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.
  - 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:
    $\{ 11, 111, 11111, 1111111, 11111111111, \ldots \}$
    $= \{1^2, 1^3, 1^5, 1^7, 1^{11}, 1^{13}, \ldots\} = \{ 1^p \mid p \text{ is a prime number} \}$
  - 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. One 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. One 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

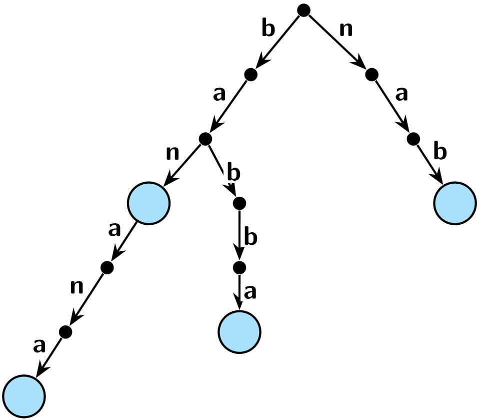
- Theorem: **All finite languages are regular.**
- Proof:

  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

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



Building Bigger Languages

- We now know: every **finite** language is regular.
- But the interesting languages are **infinite** – prime numbers, palindromes, "all strings ending in 11", . . . – and a tree of all words does not work for them.

Building Bigger Languages

- We now know: every **finite** language is regular.
- But the interesting languages are **infinite** – prime numbers, palindromes, “all strings ending in 11”, . . . – and a tree of all words does not work for them.
- **Idea:** don’t build each language from scratch. Instead, **build bigger languages out of languages we already have** – just like arithmetic builds numbers from numbers with $+$ and $\times$.

Building Bigger Languages

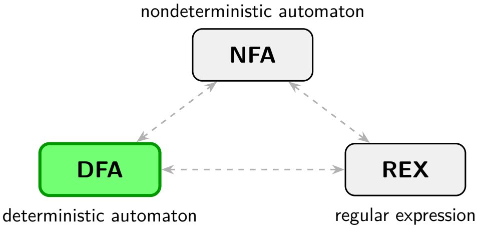
- We now know: every **finite** language is regular.
- But the interesting languages are **infinite** – prime numbers, palindromes, "all strings ending in 11", . . . – and a tree of all words does not work for them.
- **Idea:** don't build each language from scratch. Instead, **build bigger languages out of languages we already have** – just like arithmetic builds numbers from numbers with $+$ and $\times$.
- For languages, the natural building blocks are:
  - put two languages together (**union**),
  - glue their strings one after the other (**concatenation**),
  - repeat (**star**).

  These are the **regular operations**.

Regular Operations

- There are four **basic operations** we will work with:
  - Union
  - Concatenation
  - Kleene-Star
  - Kleene-Plus (which can be defined using the other three)
- Two questions will drive the rest of this chapter:
  1. Starting from regular languages, do these operations **keep us regular**?
     (If yes: a recipe for building infinite regular languages from finite pieces.)
  2. Do we reach **every** regular language this way?
- **Preview:** you have likely used these operations already: UNIX regular expressions (`egrep`, `perl`, ...) are a notation for exactly them. We define regular expressions properly later in the chapter.

Where Are We?

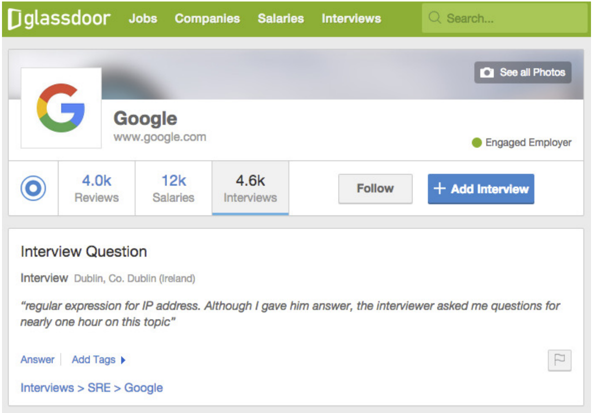


- A regular language can be described in three ways: by a **DFA**, by an **NFA**, or by a **regular expression**.
- **Program of this chapter:** show that all three are **equivalent** — they describe exactly the same languages.
- **Now:** the regular operations – our tools for building bigger languages. First question: does building keep us regular (**closure**)?

Regular Operations – Summarizing Table

Operation	Symbol	UNIX version	Meaning
Union	$\cup$	$\|$	Match one of the patterns
Concatenation	$\bullet$	*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 $A = \{\text{aardvark}\}$, $B = \{\text{bobcat}\}$, $C = \{\text{chimpanzee}\}$,
  the union of these is $A \cup B \cup C = \{\text{aardvark, bobcat, chimpanzee}\}$.

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: $L \bullet L = \{$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 $B = \{\text{ba}, \text{na}\}$. 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 $B = \{\text{ba, na}\}$. Question: What is the language B^*?

- Answer: $B^* = \{\text{ba,na}\}^* = \{\varepsilon$, ba, na, baba, bana, naba, nana, bababa, babana, banaba, banana, nababa, nabana, nanaba, nanana, babababa, bababana, $\ldots\}$

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 $B = \{\text{ba, na}\}$.
  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 $B = \{\text{ba, na}\}$.
  What is B^+ and how does it defer from B^*?
- $B^+ = \{\text{ba, na}\}^+ = \{$ba, na, baba, bana, naba, nana, bababa, babana, banaba, banana, nababa, nabana, nanaba, nanana, babababa, bababana, $\ldots\}$

  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

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

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

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:
  $M_\cup = (Q_1 \times Q_2,\ \Sigma,\ \delta_1 \times \delta_2,\ (q_1, q_2),\ F_\cup)$

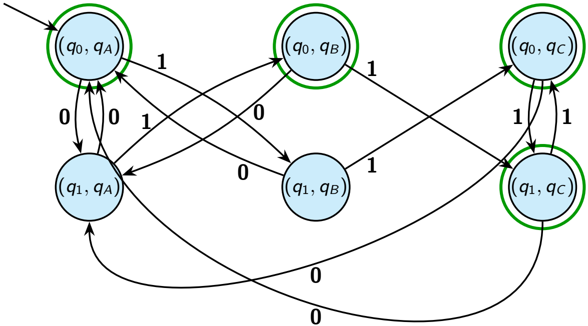
  - where the start state (q_1, q_2) is the combined start state of both automata

  - 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
  $\delta((q_1, q_2), j) = (\delta_1(q_1, j), \delta_2(q_2, j)) = \delta_1 \times \delta_2$

Union Example: $L_1 \cup L_2$

- When using the **Cartesian Product Construction**:

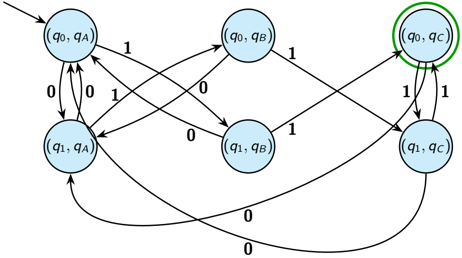


Other constructions: Intersector

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

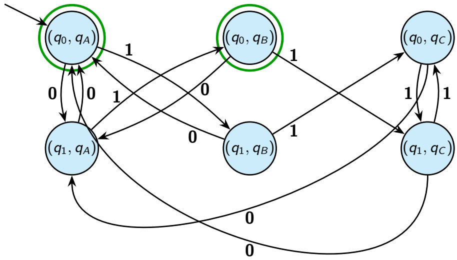


Other constructions: Difference

- 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 $F_{-} = F_1 \times Q_2 - Q_1 \times F_2$.

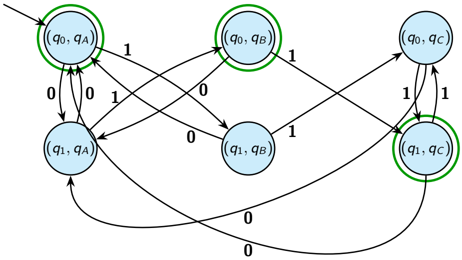


Other constructions: Symmetric difference

- 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

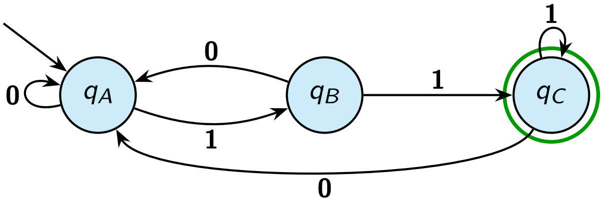
- How about the **complement**? The complement is only defined with respect to some universe.
- Given the alphabet Σ, the default universe is just the set of all possible strings Σ^*. Therefore, given a language L over Σ, i.e. $L \subseteq \Sigma^*$ the complement of L is $\Sigma^* - L$
- Note: Since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

Complement

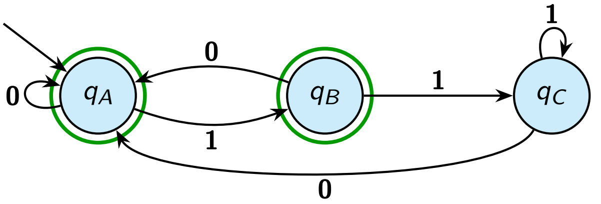
- How about the **complement**? The complement is only defined with respect to some universe.
- Given the alphabet Σ, the default universe is just the set of all possible strings Σ^*. Therefore, given a language L over Σ, i.e. $L \subseteq \Sigma^*$ the complement of L is $\Sigma^* - L$
- Note: Since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

Complement Example

Original:



Complement:



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. $\overline{L_1}$ is regular.
- Only No. 1 is a **regular operation** (2–5 are boolean operations, which we get for free). We still need to show closure under concatenation and Kleene-*.
- Tough question: Can't we do a similar trick for concatenation?