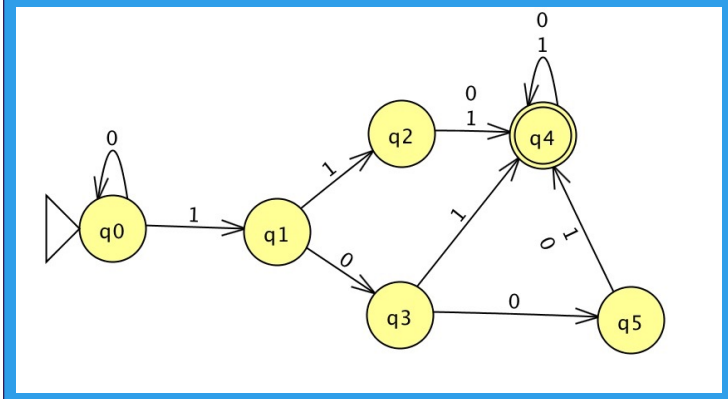
Theory of Computation and Programming Languages



NFAs

CS51 – Spring 2026

Today we will continue with automata.

Formal definition of DFAs

- A Deterministic Finite Automaton (DFA) is a finite state machine that accepts or rejects finite strings of symbols and produces the same unique computation for each unique input string. For any given finite input string, the DFA will halt and either accept or reject the string. A DFA, M, is said to recognize a language, $L(M)$, which is the set of all strings that M accepts.
- Formally, a DFA is described by a 5-tuple $(Q, \Sigma, \delta, q0, F)$
  - Q is a finite set of states
  - Σ is a finite set of input symbols also known as the alphabet
  - δ is a state transition function ($\delta : Q \times \Sigma \rightarrow Q$)
  - $q0$ is the start state ($q \in Q$)
  - F is a set of accept states ($F \subseteq Q$).

2

Remember from our last class meeting, we encountered DFAs, simple state machines that have a finite set of states, a start state, a set of accept states, and for each character in an alphabet, they tell us for each state where we would transition.

Computing on a DFA

- We have a string as input on a tape.
- We start at the beginning of the string.
- We read a symbol from the tape and transition to the state indicated by the model.
- If we end in a final state (i.e. when we get to the end of the string we are in a final state), we accept the string.
- Otherwise, if when we get to the end of the string on the tape and we're in a non-final state we reject.
- All strings accepted by a DFA define a language.

3

As we saw, computing on a DFA unfolds as follows. We assume that the string we want to determine whether it belongs to a language is given to us as input on a tape. We start at the beginning of the string, that is the first character or first cell of the tape. We read a symbol from the tape and transition to the state as indicated by our DFA, that is we follow the arrow that matches the letter we just read. If we end up in a final state, that is we read the whole string and we are in a final state, we accept the string. If we ran out of characters and don't end up in a final state, then we reject that string as it does not belong to L. All strings accepted by a DFA define a specific language that that DFA describes

DFAs over numbers

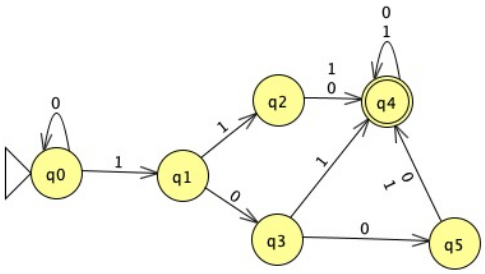
- We can use any alphabet we want
- If we use 1's and 0's we can interpret them as binary numbers!

4

Last time, all the examples we encountered where over alphabets with letters. We can use any alphabet we want though, including alphabets over numbers. For example, if we work only with the numbers 0 and 1 we could interpret them as binary numbers!

Practice time

- Assuming we work on 0s and 1s that are interpreted as bits, what does the following DFA do?

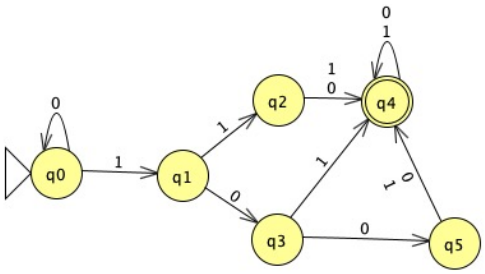


5

For example, if we assume that we work with 0s and 1s in our alphabet and that we interpret these as bits, what do you think the following DFA do?

Answer

- Determines if an input string of 0s and 1s, when interpreted as a binary number, is greater than or equal to 5.



6

It determines if an input string of 0s and 1s, when interpreted as a binary number, is greater than or equal to 5.

Practice time

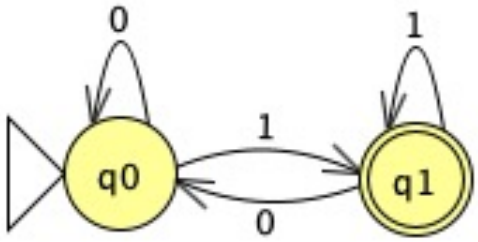
- Can you write a DFA that determines if a number is odd?

7

Your turn. Can you write a DFA that determines if a number is odd?

Answer

- Can you write a DFA that determines if a number is odd?
- Remember, odd numbers end in 1.



8

The key idea here is to remember that odd numbers end in 1 so we'd end up with this DFA.

Deterministic finite automata

- A Deterministic Finite Automaton (DFA) is a finite state machine that accepts or rejects finite strings of symbols and produces the **same unique computation** for each unique input string. For any given finite input string, the DFA will halt and either accept or reject the string. A DFA, M, is said to recognize a language, $L(M)$, which is the set of all strings that M accepts.
- This unique computation where we know what state we can go to given an input is what the term deterministic means.
- In particular, for any state of a DFA, the state transition function specifies **exactly one** state for each input symbol in the alphabet.

9

If we give a string to a DFA it guarantees the same unique computation for it. This unique computation where we know what state we can go to given an input is what the term deterministic means. In particular, for any state of a DFA, the state transition function specifies **exactly one** state for each input symbol in the alphabet.

Non-deterministic finite automata

- Non-deterministic finite automata (NFAs) are generalizations of the DFA concept.
- In an NFA, for a given state and input symbol, we can go to **zero**, one or **more** states (rather than just a single one for DFAs).
- Tend to be a bit easier to create than DFAs
- An NFA accepts a string if *some* path exists through the DFA based on the input string that end in the final state.

10

There is another type of automaton called an NFA or non-deterministic finite automaton which generalizes the idea of DFAs. What's different in NFAs is that for a given state and input symbol, we can go to zero, one (as before in DFAs) or more (!) states. NFAs are easier to create than DFAs since we don't have to think of transitions for each character or we can have more than one. An NFA accepts a string if *some* path exists through the DFA based on the input string that end in the final state.

NFA Example 1

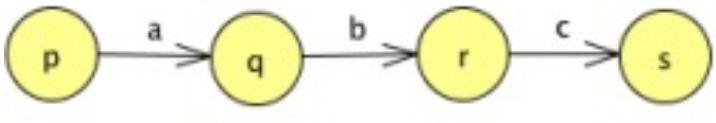
- Suppose our alphabet is {a, b, c} and we are interested in the set of all words that contain the string *abc* or that end with the string *ac*.
- Some examples of such words are:
  - w1 = aaabc
  - w2 = babcabcac
    - Note that w2 has the substring abc in multiple locations *and* also ends with ac.
  - w3 = aaac
- On the other hand, the string w4 = abbccaab is not in the desired set.
- Let's try to construct a finite machine that accepts such words.

11

Let's see an example with an NFA. Suppose our alphabet is {a, b, c} and we are interested in the set of all words that contain the string *abc* or that end with the string *ac*. Think about it, a word like aaabc works. babcabcac also works because it has the substring abc AND also ends with ac. aaac also works. But abbccaab is not a string we would accept. Let's try to build an automaton that accepts such words.

NFA Example 1

- Suppose we want the string *abc* anywhere in the word.
- We want the following configuration to be part of our finite machine:



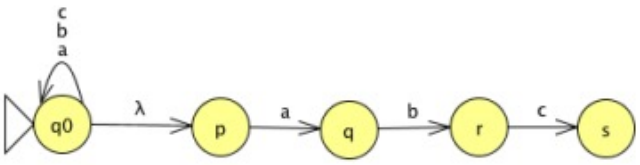
- We can move from p to s with the string abc.
- Since this occurrence of abc can be anywhere in the word, we want to be able to *reach* the state p at any step.
- Also, once we reach s, we want to accept the word with any combination of input symbols following this occurrence of abc.

12

First, suppose we want the string abc anywhere in the word. In terms of states and transitions, we want the configuration shown below to be a part of our finite machine. We can move from p to s with the string abc. Since this occurrence of abc can be anywhere in the word, we want to be able to *reach* the state p at any step. Also, once we reach s, we want to accept the word with any combination of input symbols following this occurrence of abc.

NFA Example 1

- We introduce an initial state q0.
- The transitions labeled a, b and c at q0 can read any combination of the symbols in the alphabet.
- The transition labeled λ allows one to "jump" from q0 to p without reading an input symbol.
- Thus, we may "guess" where an abc might occur to make this jump.

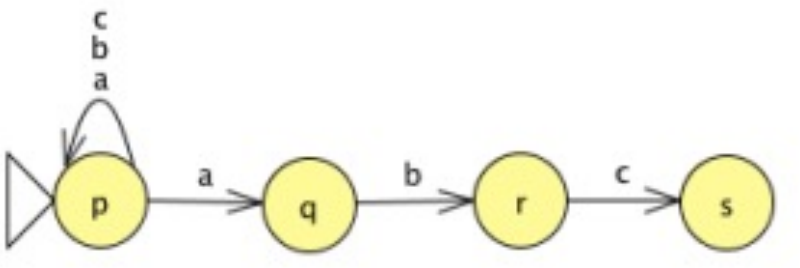


13

We will continue by introducing an initial state q0. Note that if we read a b or c we stay at q0. But weirdly, we can also transition to p by reading nothing. This is denoted my lambda which allows us to jump to p by reading the empty string. In a sense, we can kinda guess where an abc might occur to make the jump from q0 to p.

NFA Example 1

- Equivalently without the λ transition:
  - State p is the initial state. We observe that there are two transitions labeled a from the state p. One stays in p while the other goes to q.
  - This nondeterministic behavior allows one to make the jump to the state q when looking for the substring abc.

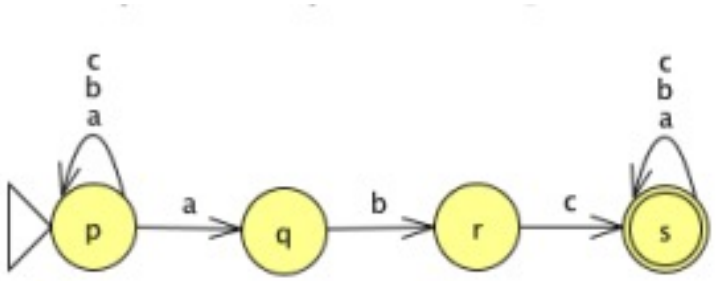


14

We could also have modeled this automaton without using a lambda tranistion. We could have said that p is the initial state. When you read b and c you stay in p. But when you read a you may stay at p or you may go to q. This nondeterministic behavior allows one to make the jump to the state q when looking for the substring abc.

NFA Example 1

- Next, we introduce transitions a, b and c at state s and make s an accept state.
- Thus, any word containing the string abc is accepted.

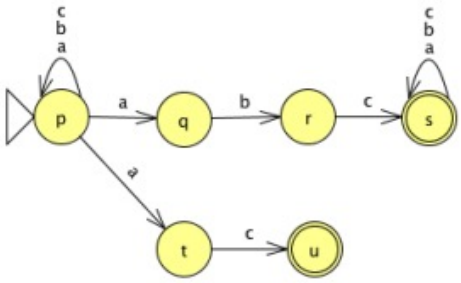


15

Don't forget that we need to have an accept state. This NFA will accept any word that contains the string abc.

NFA Example 1

- To accept words that end with ac, we introduce two more states and transitions as shown below.
- This is a Nondeterministic Finite Automaton (NFA) that accepts our desired set of words.

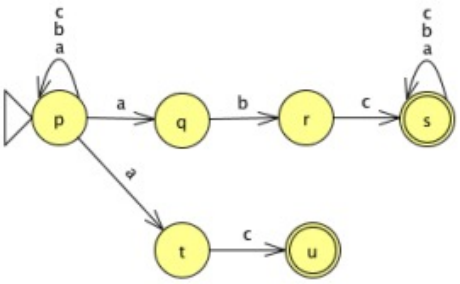


16

We still miss the part of accepting words that end with ac. For that, we will introduce two more states. Now from p, when reading a, we can stay as is, go to q or go to t. This NFA accepts the words of the language we defined it. Note that in contrast to DFAs, if we look at any state we don't have to have a single transition for every single alphabet character.

NFA Example 1

- We note several properties of this NFA:
- There are three transitions from state p labeled a. Another way of saying this is that the transition labeled a from state p goes to the set {p, q, t}.
- There are no transitions labeled a or c from the state q. Another way of saying this is that the transitions labeled a and c from state q go to the empty set ϕ.
- There can be transitions labeled λ.

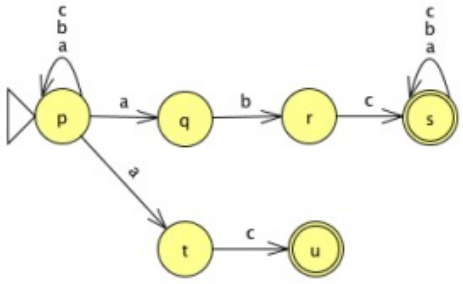


17

For example, we can sa that from state p, if we read a we go to the set {p, q, t}. If we look at q, we could say that if we read a or c we go to the empty set. We can also have transitions labeled as lambda that allow us to move to another state without reading a character.

NFA Example 1

- Properties such as above distinguish an NFA from a DFA.
- Another consequence of nondeterminism is that there may be several possible paths for a given input string.
- For example, for the input w = ababc, there are several configurations in the NFA.
- an NFA accepts an input if there is *at least one* accepting configuration.



18

This is what makes NFAs different than DFAs. Another consequence of nondeterminism is that there may be several possible paths for a given input string. Remember, an NFA accepts an input if there is *at least one* accepting configuration.

Formal definition of NFAs

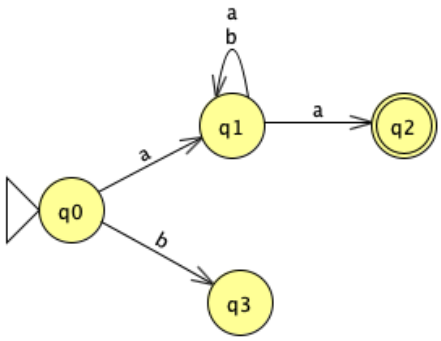
- Formally, a NFA is described by a 5-tuple $(Q, \Sigma, \delta, q0, F)$
  - Q is a finite set of states
  - Σ is a finite set of input symbols also known as the alphabet
  - δ is a state transition function ($\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow P(Q)$), where $P(Q)$ is the powerset, that is the set of all subsets of Q.
  - $q0$ is the start state ($q0 \in Q$)
  - F is a set of accept states ($F \subseteq Q$).
- An input w is accepted by an NFA if there is at least configuration for w that ends in an accept state, that is a state that belongs to F.

19

We can now formally define an NFA by updating our definition for the state transition function.

NFA Example 2

- Accepts strings that start and end with a.
- a(a|b)*a

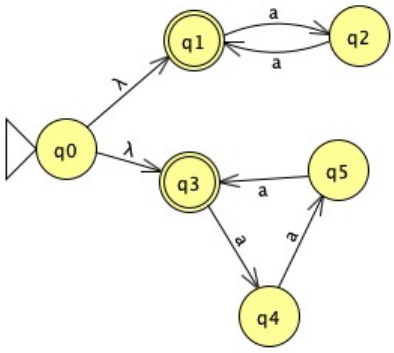


20

Let's look at a few NFAs. What do you think this NFA does? Accepts strings that start and end with a. Can you think of a regular expression that describes it? a(a|b)*a

NFA Example 3

- Accepts strings of a's that have lengths divisible by 2 OR 3
- (aa)*|(aaa)*

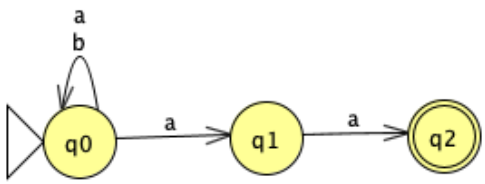


21

How about this one? Accepts strings of a's that have lengths divisible by 2 or 3. The regular expression would be (aa)*|(aaa)*

NFA Example 4

- Accepts strings of a's that end in 2 a's
- (a|b)*aa

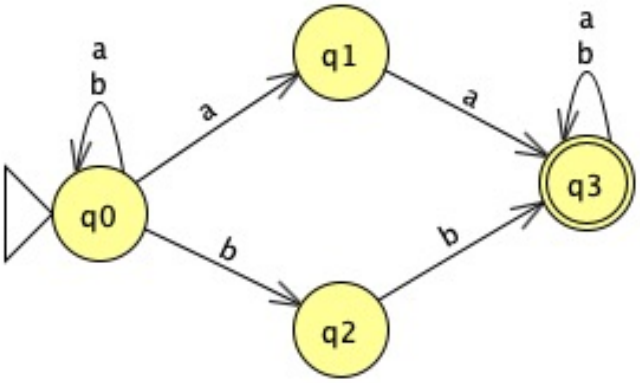


22

This one accepts strings of a's that end in 2 a's. The regular expression is (a|b)*aa

NFA Example 5

- Accepts strings that either have aa or bb as a substring.
- (a|b)*(aa|bb)(a|b)*

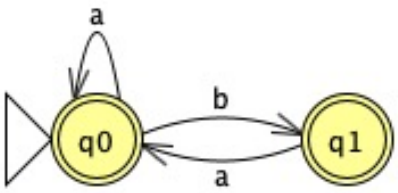


23

This one accepts strings that either have aa or bb as a substring. Regular expression is (a|b)*(aa|bb)(a|b)*

NFA Example 6

- Accepts any string of a's and b's that doesn't have two adjacent b's.
- a*b(aa*b)*a*

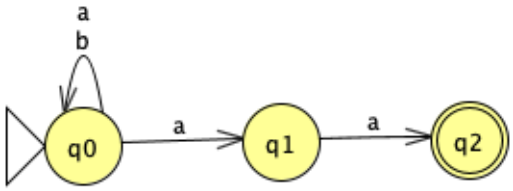


24

This one accepts any string of a's and b's that doesn't have two adjacent b's. The regular expression is a*b(aa*b)*a*

NFA Example 7

- Consider the following NFA.
- Is aaaaa in the language?
- Yes, we can find at least one way of ending to q2.



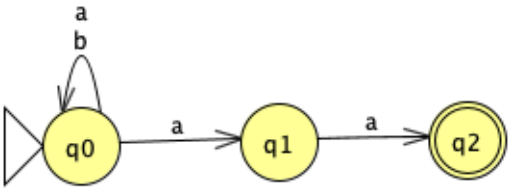
25

Consider the following NFA and the string aaaaa.
what states could we be in after reading the first a?
- q_0 or q_1
- what states would we be in after reading the second a?
  - we could start in either q_0 *or* q_1
    - if we were in q_0, we'd end up in q_0 or q_1
    - if we were in q_1, we'd end up in q_2
  - therefore, after reading two a's we could end up in any of: q_0 or q_1 or q_2
- the third a?
  - we could be in q_0 or q_1 or q_2
    - if we were in q_0 -> q_0 or q_1
    - if we were in q_1 -> q_2
    - if we were in q_2 -> reject
      - does this matter?
      - No. We only need to find *one* path through the state transitions that ends in an accepting state
- the fourth a?
  - q_o or q_1 or q2

- the fifth a?
  - q_o or q_1 or q2
  - since q2 is *an* option, then there's a set of transitions between the states that gets us to an accepting state

NFA Example 7

- Consider the following NFA.
- Is ababaa in the language?
- Yes, we can find at least one way of ending to q2.

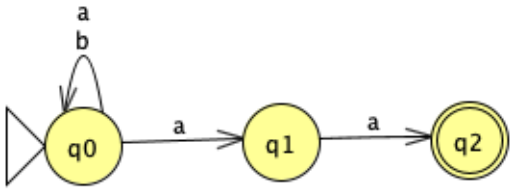


26

What about ababaaa?
what states would we be in after reading the first a?
- q_0 or q_1
- second b?
- q_0
- third letter, a?
- q_0 or q_1
- fourth letter, b?
- q_0
- fifth letter, a?
- q_0 or q_1
- sixth letter, a?
- q_0 or q_1 or q_2
- since q_2 is *an* option, then there's a set of transitions between the states that gets us to an accepting state

NFA Example 7

- Consider the following NFA.
- Is ababa in the language?
- No way to get to a final state.



27

What about ababa

- a:
  - q_0 or q_1
- b:
  - q_0
- a:
  - q_0 or q_1
- b:
  - q_0
- a:
  - q_0 or q_1
  - reject: no way to get to an accepting state

Every NFA can be converted to a DFA

- if we have an NFA M1, then we can construct a DFA M2 such that M1 and M2 accept the same set of words.
- While NFAs can have fewer states, the transitions are more complicated than DFAs.

28

We said that NFAs are generalizations of DFAs. The cool thing is that we can take an NFA and convert it into a DFA so that they accept the same set of words. Keep in mind that NFAs tend to have fewer states but transitions are more complicated than DFAs

Constructing a DFA from an NFA

- Each state in the DFA is a subset of the states in the NFA.
- The start state of the DFA is the set containing only the start state of the NFA.
- The final state of the DFA is all of its states which have the final state of the NFA as elements.
- Transitions are more complicated as the output state of any DFA state for any given input is the *union* of output states of all of the NFA states of that DFA state.
- How many DFA states can we have at most?
  - $2^n - 1$ where n is the number of NFA states.
  - Think of each DFA state like a n bit number
    - 1 if it represents the original NFA state, 0 otherwise.
    - Can't have all zeros, so $2^n - 1$

29

There is an algorithm on how we can construct a DFA from an NFA. Each state in the DFA is a subset of the states in the NFA (that is, we might have more states in the DFA). The start state is the same. The final state for the DFA will be all the sattes which have the final state of the NFA. Transitions are more complicated as the output state of any DFA state for any given input is the *union* of output states of all of the NFA states of that DFA state. If you think about it, if we started with n NFA states, we would have at most 2^n-1 states in the corresponding DFA. You could reason about this if you think of each DFA state like an n bit number where it is 1 if it represents the original NFA and 0 otherwise. Since we can't have all 0s we end up with 1 fewer state, i.e. 2^n-1.

Constructing a DFA from an NFA

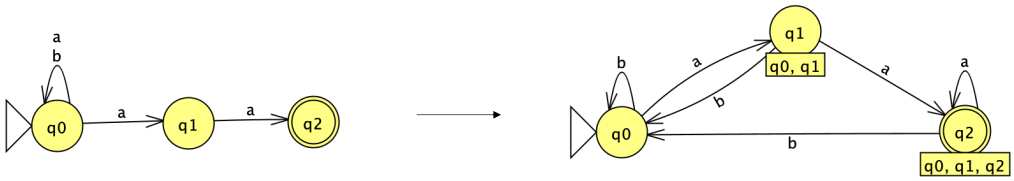
- Create the start state (q_0) and add it to queue.
  - In a queue, the first in is the first out (FIFO, the opposite idea to LIFO in stacks).
- As long as queue isn't empty:
  - Remove front state s from queue
  - $s_{new} = []$
  - for each letter l in the alphabet
    - if any "old state", q_i, in s has a transition $q_i : l \rightarrow q_j$
      - add q_j to s_{new}
      - if s_{new} doesn't exist already, create state s_{new} and add s_{new} to queue q
  - if any states don't have transitions for all letters in the alphabet, create a "sink" state that transitions to itself on all letters and have states transition to here for any remaining alphabet letters

30

We will employ the idea of a queue which is the opposite of that of stack, that is the first one to get in is the first one that gets served (FIFO). We start by adding the start state to the queue. As long as the queue is not empty we repeat the following: Remove the front state from the queue. Define a new set of states that you will populate as follows. For each letter of the alphabet if any old state has a transition, we add that state to the state set which we add to the queue. if any states don't have transitions for all letters in the alphabet, create a "sink" state that transitions to itself on all letters and have states transition to here for any remaining alphabet letters

Convert example 4 from NFA to DFA

- Accepts strings of a's that end in 2 a's
- (a|b)*aa

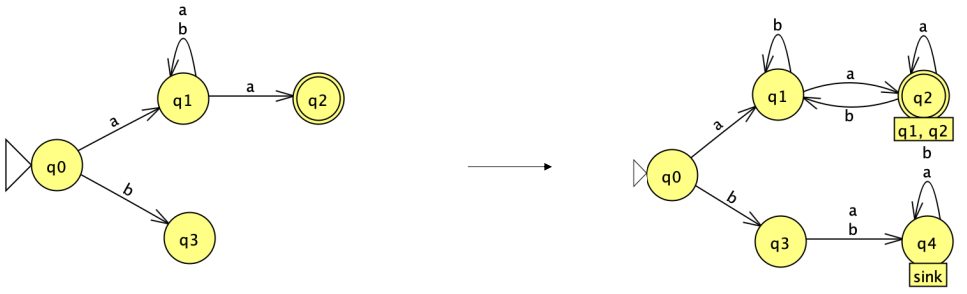


31

Here's an example of a conversion of an NFA that accepts strings of as that end in 2as to a DFA. We start with q0 and note that in the NFA, if we read a b we stay in q0. But if we read an a, we could go either to q0 or q1. So we add a new state with the union of these two. If we were in either q0 or q1, we go with b to q0, But if we were to read a, we go to q0, q1, or q2 which are shown as a new state. In the newly created q2 (that corresponds to q0,q1,q2 from the original), If we read a b, we go to q0. But if we read an a, we go to either q0, q1, or q2 that is we stay where we are

Convert example 2 from NFA to DFA

- Accepts strings of a's that start and end with a
- a(a|b)*a



32

Here's another example of converting an NFA that accepts strings of as that start and end with a. q0 is again our start state. If we read a b we go to q3. If we are at q3 and read either a or b, there's nowhere to go so we mark this by introducing a sink state where we get trapped no matter how many as or bs read. Going back to q0, if we read a, we go to q1. From q1, if read b we stay put. But if we read a we either stay in q1 or go to q2. That requires us to introduce a new state that corresponds to {q1, q2} From q1 or q2, reading b takes us back to q1 but reading a leaves us where we are.

Regular languages

- A **regular language** is a language that can be described by a DFA.
- A regular language is also any language that can be described by a regular expression!

33

Off to introducing some new terminology. A **regular language** is a language that can be described by a DFA (or an NFA, remember, they're equivalent). A regular language is also any language that can be described by a regular expression!

Regular languages

- Here is an example of a non-regular language
- 0^n 1^n for any n, that is the language of some number of zeros followed by the *same* number of 1s.
- Can you come up with a regular expression for this language?
  - seems hard, since there's no tool for us to count
- Can you come up with a DFA or NFA for this language?
  - would have to have 2^(n+1) states and states are the only way we can count.
- only problem is that n isn't finite!
  - Consider any DFA that recognizes strings of 0^n 1^n for some fixed n.
  - It won't recognize string 0^(n+1) 1^(n+1)
- Basic idea behind pumping lemma (more in CS101)

34

Here is an example of a non-regular language:
0^n 1^n for any n, that is the language of some number of zeros followed by the *same* number of 1s. Since we don't have a way of remembering n we cannot come up with such an automaton. This idea is central to theory of computation and if you continue with cs, you will encounter it in cs101 with the pumping lemma. In case you are interesting, the pumping lemma for regular languages is a lemma that describes an essential property of all regular languages. Informally, it says that all sufficiently long strings in a regular language may be pumped—that is, have a middle section of the string repeated an arbitrary number of times—to produce a new string that is also part of the language. The pumping lemma is useful for proving that a specific language is not a regular language, by showing that the language does not have the property.

Practice Time

- Which of these languages are regular?
  - Language consisting of repetitions of either 01 or 10
  - Language of strings with an equal number of 0s and 1s
  - Language over a's and b's of alternating a's starting with an a, i.e. every other character in the string is an a
  - Language of all words that are palindromes, i.e. read the same forward and reversed

35

Let's practice by thinking which of these languages are regular

Answer

- Which of these languages are regular?
  - Language consisting of repetitions of either 01 or 10
    - Regular: (01|10)*
  - Language of strings with an equal number of 0s and 1s
    - Not regular
  - Language over a's and b's of alternating a's starting with an a, i.e. every other character in the string is an a
    - Regular: a((a|b)a)*|a((a|b)a)*(a|b)
  - Language of all words that are palindromes, i.e. read the same forward and reversed
    - Not regular

36

Did you get this?

JFLAP examples:

- NFA examples

37

If you want to practice with the JFLAP examples we saw today, check the link above