Regular Expressions

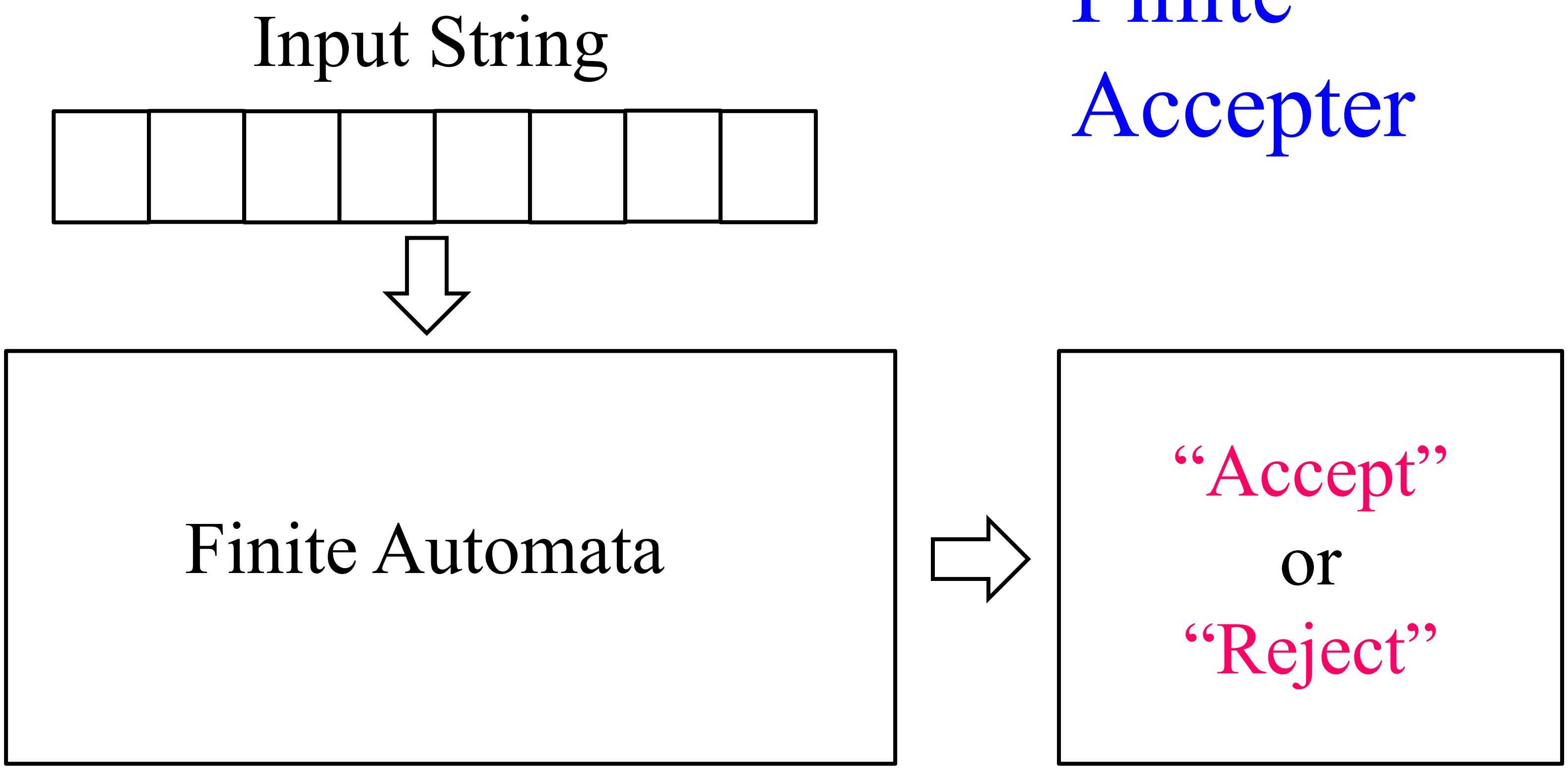
RegEx

Theory of Computation

CISC 603, Spring 2020, Daqing Yun

Automata Theory

Finite
Accepter



Finite Automata

- Definition: a *finite automaton* is a 5-tuple $M = (Q, \Sigma, q_0, A, \delta)$, where:
  - Q is a finite set of *states*
  - Σ is a finite *input alphabet*
  - $q_0 \in Q$ is the *initial state*
  - $A \subseteq Q$ is the set of *accepting* states
  - $\delta : Q \times \Sigma \rightarrow Q$ is the *transition* function
- From state q the machine will move to state $\delta(q, \sigma)$ if it receives input symbol σ

Regular Expressions

- *Regular expressions* provide a language for writing textual *patterns* against which strings may be matched
- Examples
  - **`hello`**, which only matches the string “hello”
  - **`hello|goodbye`**, which matches the strings “hello” and “goodbye”
  - **`(hello)*`**, which matches the strings “hello”, “hellohello”, “hellohellohello”, and so on, as well as the *empty* string

Regular Languages and Regular Expressions

- Many simple languages can be expressed by a *formula* involving languages containing a single string of length 1 and the operations of union, concatenation, and repetition
- Examples
  - Strings ending in **`aa`**: **`{a,b}*{aa}`**
  - Strings containing **`ab`** or **`bba`**: **`{a,b}*{ab,bba}{a,b}*`**
- These are called *regular* languages

Regular Languages and Regular Expressions

- Definition: If Σ is an alphabet, the set R of regular languages over Σ is defined as follows:
  - The language $\varnothing$ is an element of R, and for every $\sigma \in \Sigma$, the language $\{\sigma\}$ is in R
  - For every two languages L_1 and L_2 in R, the three languages $L_1 \cup L_2$, $L_1 L_2$, and $L_1{}^*$ are elements of R
- Examples:
  - $\{\Lambda\}$, because $\varnothing^* = \{\Lambda\}$
  - $\{a, b\}^*\{aa\} = (\{a\} \cup \{b\})^* (\{a\}\{a\})$

Regular Languages and Regular Expressions

- A *regular expression* for a language is a slightly more user-friendly formula
  - Parentheses replace curly braces, and are used only when needed, and the union symbol is replaced by +

Regular Language	**Regular Expression**
$\varnothing$	$\varnothing$
$\{\Lambda\}$	Λ
$\{a,b\}^*$	$(a+b)^*$
$\{aab\}^*\{a,ab\}$	$(aab)^*(a+ab)$

Regular Languages and Regular Expressions

- Two regular expressions are equal if the languages they describe are equal
- For example,
  - **`(a*b*)*=(a+b)*`**
  - **`(a+b)*ab(a+b)*+b*a*=(a+b)*`**

Question

- Given a regular expression and a string, how do we write a program to decide whether the string matches that expression?
- Many programming languages, e.g., perl, ruby, etc., already have regular expression support, but how does that support work?
- How should we implement regular expressions if the language did not already have the support?

Nondeterministic Finite Automata

- Finite automata are perfectly suited to the job
- Any regular expression can be converted into an equivalent NFA
- Every string matched by the regular expression is accepted by the NFA, and vice versa
- Match a string by feeding it to a simulation of corresponding NFA

Nondeterministic Finite Automata

- Definition: A *nondeterministic finite automaton* (NFA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$, where:
  - Q is a finite set of states,
  - Σ is a finite input alphabet
  - $q_0 \in Q$ is the initial state
  - $A \subseteq Q$ is the set of accepting states
  - $\delta : Q \times (\Sigma \cup \{\Lambda\}) \rightarrow 2^Q$ is the transition function.

  (The values of δ are not single states, but *sets* of states)
- For every element q of Q and every element σ of $\Sigma \cup \{\Lambda\}$, we interpret $\delta(q, \sigma)$ as the *set of states* to which the NFA can move from state q on input σ

Syntax

What do we mean by regular expression?

- Two kinds of extremely simple regular expression:
  - An empty regular expression
    - This matches the empty string and nothing else
  - A regular expression containing a single, literal character
    - For example, **`a`** and **`b`** are regular expressions that match only the strings **`a`** and **`b`** respectively

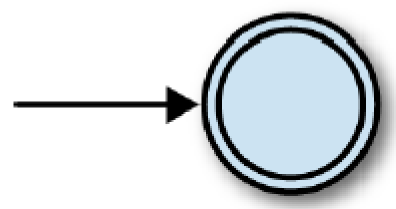
Syntax
What do we mean by regular expression?

- Three ways to combine and build more complex expressions:
  - **Concatenate** two patterns
    - **`a`** and **`b`** to get **`ab`**, which only matches the string **`ab`**
  - **Choose** between two patterns
    - By joining them with the | operator (disjunction)
    - **`a`** or **`b`** to get **`a|b`**, which matches the strings **`a`** and **`b`**
  - **Repeat** a pattern zero or more times
    - By suffixing it with the **`*`** operator
    - **`a`** to get **`a*`**, which matches the strings **`a`**, **`aa`**, **`aaa`**, and so on, as well as the *empty* string (i.e., zero repetitions)

Semantics

How to convert a RegEx syntax into an NFA?

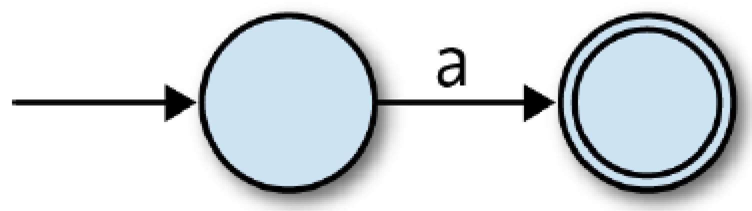
- The easiest class to convert is *empty*



Semantics

How to convert a RegEx syntax into an NFA?

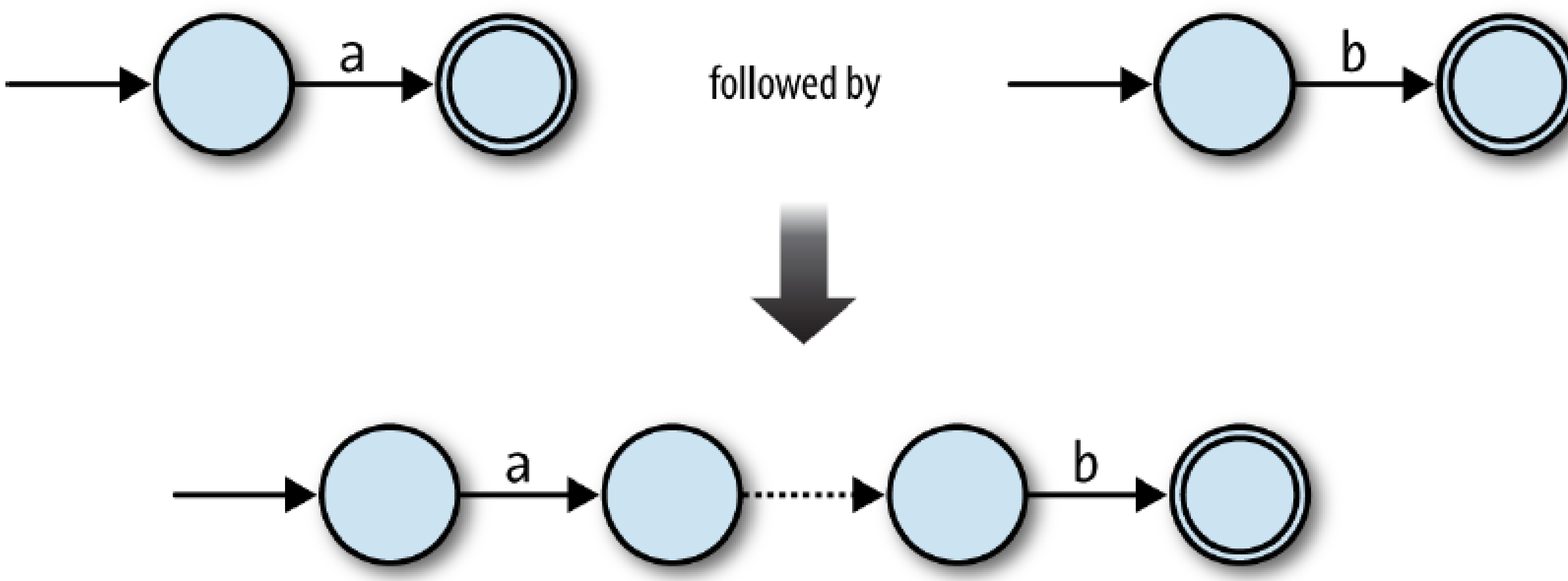
- Literal, single-character pattern



Semantics

How to convert a RegEx syntax into an NFA?

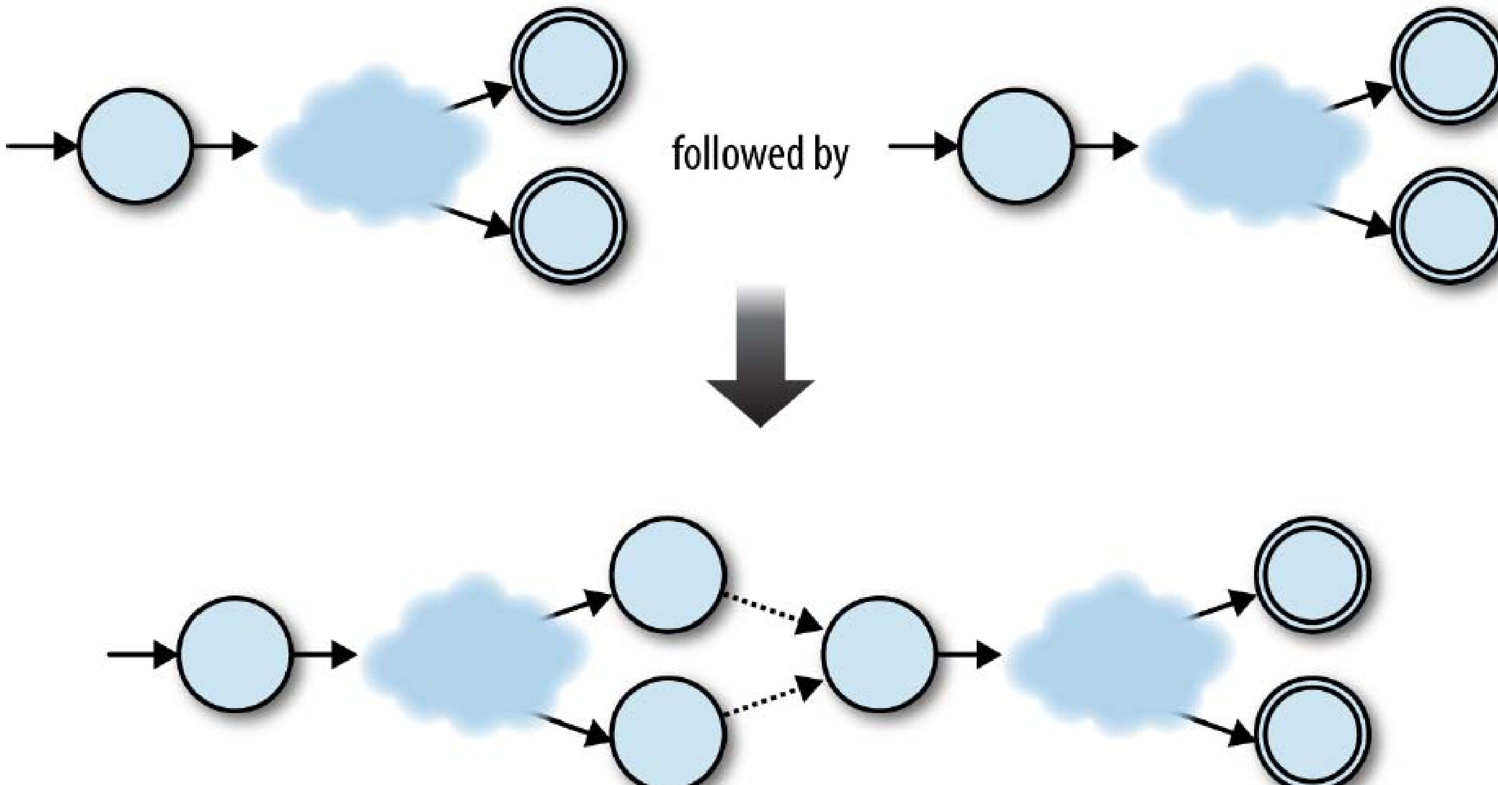
- Concatenate



Semantics

How to convert a RegEx syntax into an NFA?

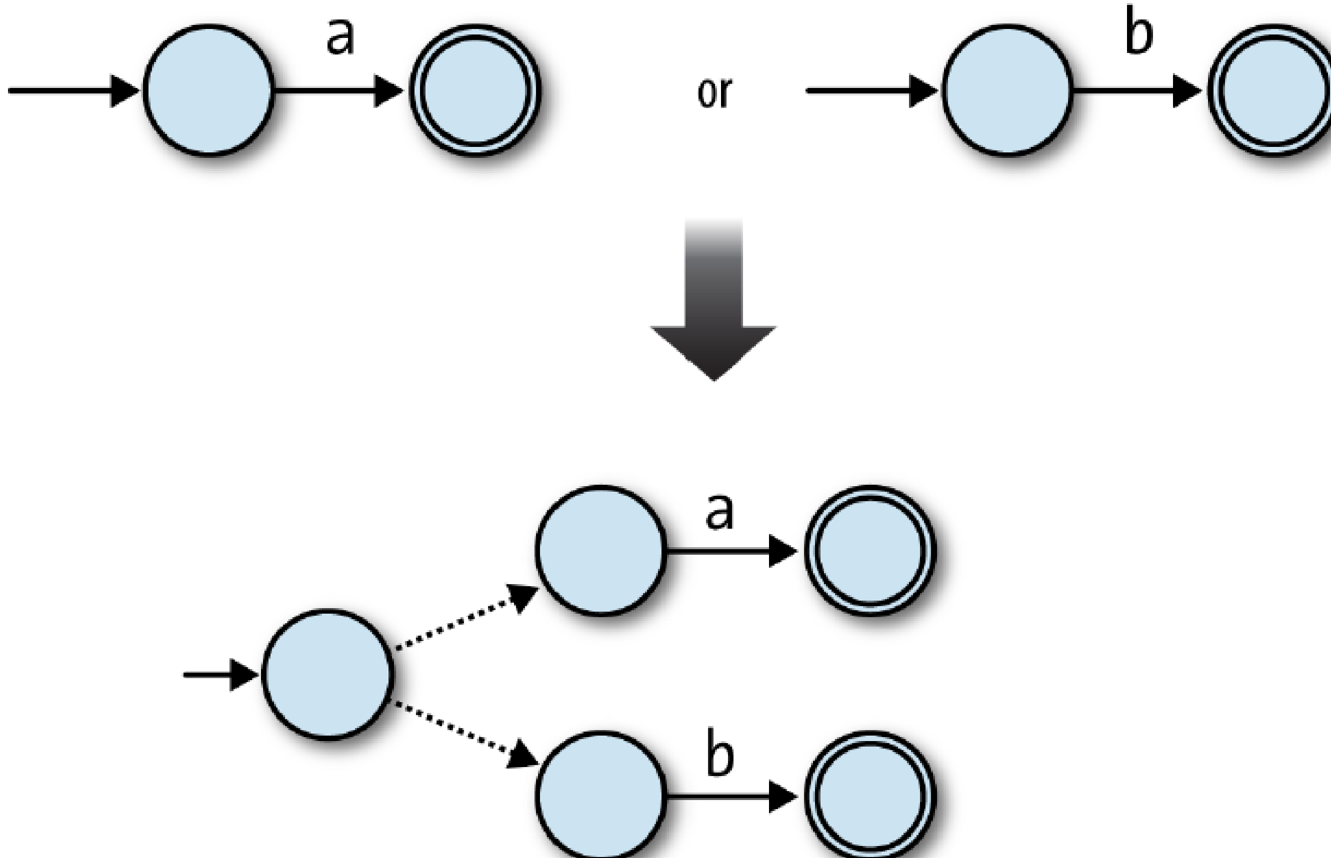
- Concatenate



Semantics

How to convert a RegEx syntax into an NFA?

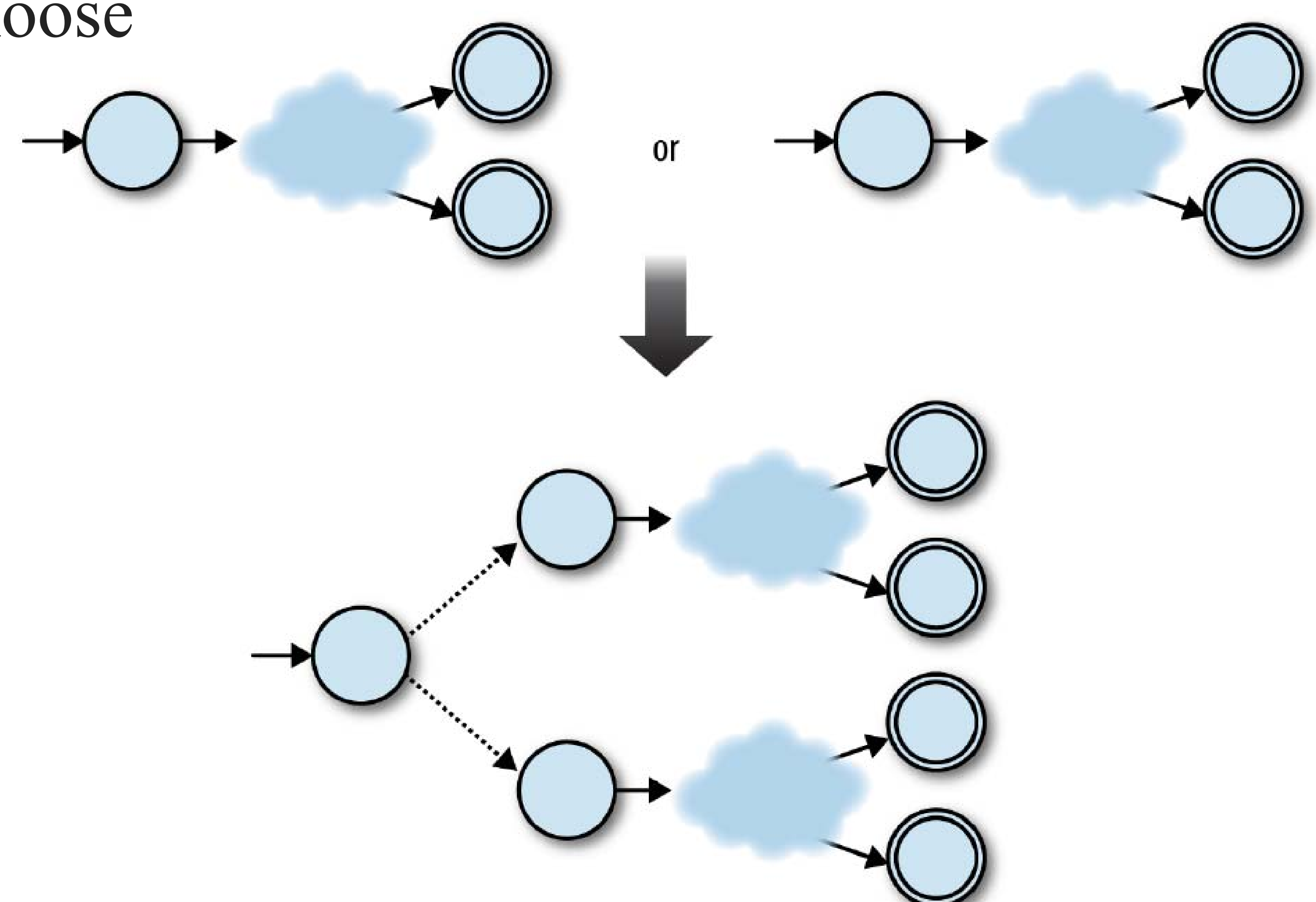
- Choose



Semantics

How to convert a RegEx syntax into an NFA?

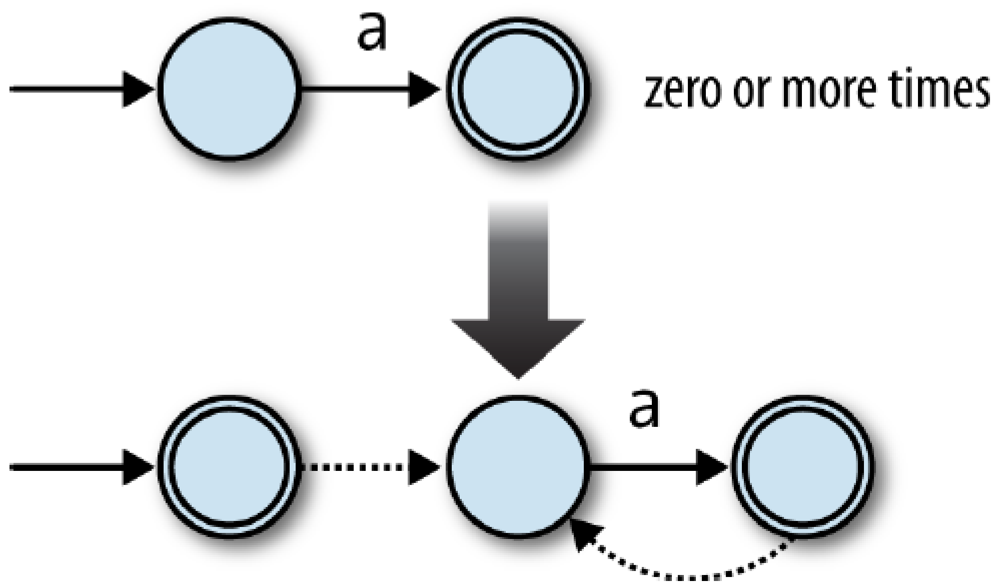
- Choose



Semantics

How to convert a RegEx syntax into an NFA?

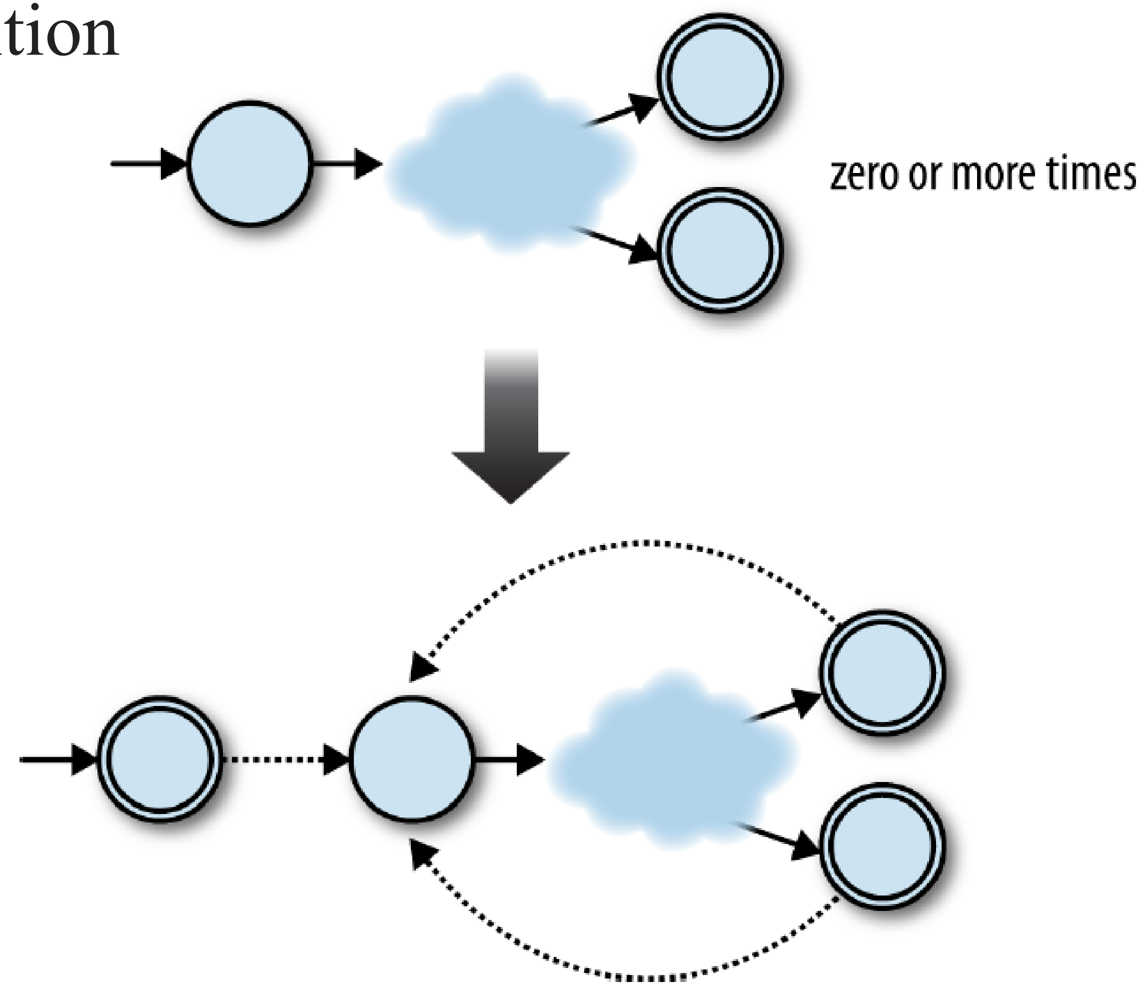
- Repetition



Semantics

How to convert a RegEx syntax into an NFA?

- Repetition



Equivalence

- Deterministic state machine and added more features
  - Nondeterminism
  - Free moves
- Do they let us do anything that we cannot do with a standard DFA?
- It is possible to convert any nondeterministic finite automaton into a deterministic one that accepts exactly the same strings

Equivalence

- Consider a particular DFA whose behavior we want to simulate:
  - Before the machine has read any input, it is in state 1
  - The machine reads the character **`a`**, and now it is in state 2
  - The machine reads the character **`b`**, and now it is in state 3
  - There is no more input, and state 3 is an accept state, so the string **`ab`** has been accepted
- The simulation, which is a program, say written in C/C++, running on a real computer, is recreating the behavior of the DFA. Every time the "imaginary" DFA changes state, so does the simulation

Equivalence

- Both the DFA and the simulation are deterministic
- Their states match up exactly
  - When the DFA is in state 2, the simulation is in a state that means “the DFA is in state 2”
  - In the simulation, this *simulation state* is effectively the value of the DFA instance’s “current state” attribute

Equivalence

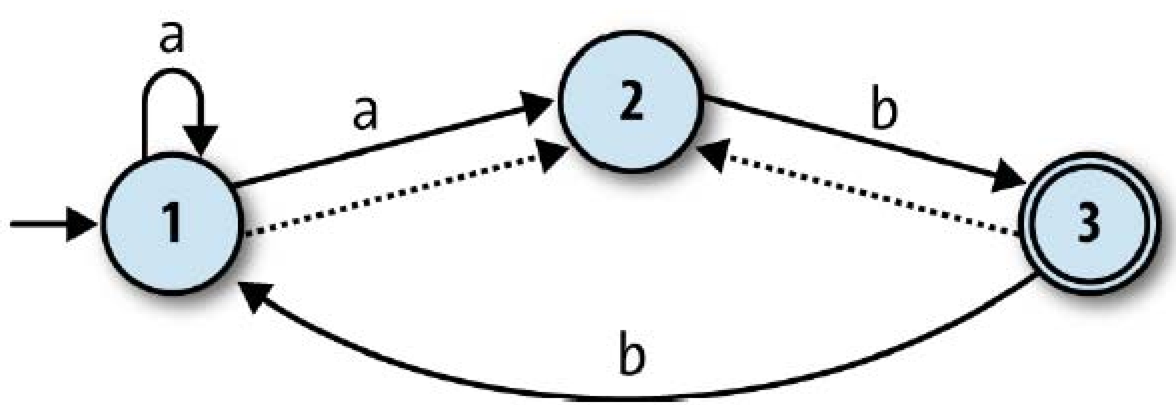
- The simulation of a hypothetical NFA reading some characters does not look hugely different:
  - Before the machine has read any input, it is possible for it to be in either state 1 or state 3
  - The machine reads the character **`c`**, and now it is possible for it to be in one of states 1, 3, or 4
  - The machine reads the character **`d`**, and now it is possible for it to be in either state 2 or state 5
  - There is no more input, and state 5 is an accept state, so the string **`cd`** has been accepted

Equivalence

- The difference: the DFA moves from one current state to another, whereas the NFA moves from one current *set of possible states* to another
- We can always construct a DFA whose job is to simulate a particular NFA
  - DFA states – sets of possible states of the NFA
  - The rules between DFA states – the ways in which the NFA can move between its sets of possible states
  - The resulting DFA can completely simulate the behavior of the NFA

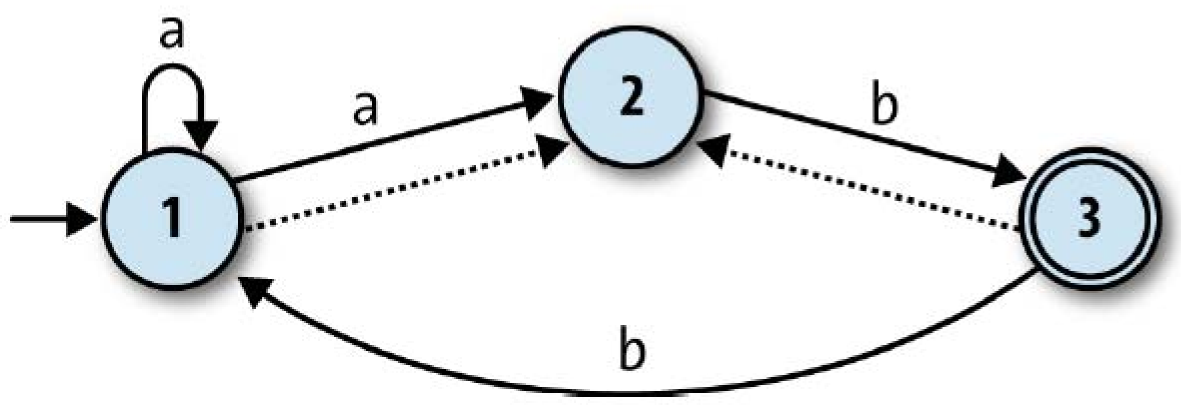
Example

- It is possible for this NFA to be in state 1 or state 2 before it has read any input (state 1 is the start state, and state 2 is reachable via a free move), so the simulation will begin in a state we can call “1 or 2”



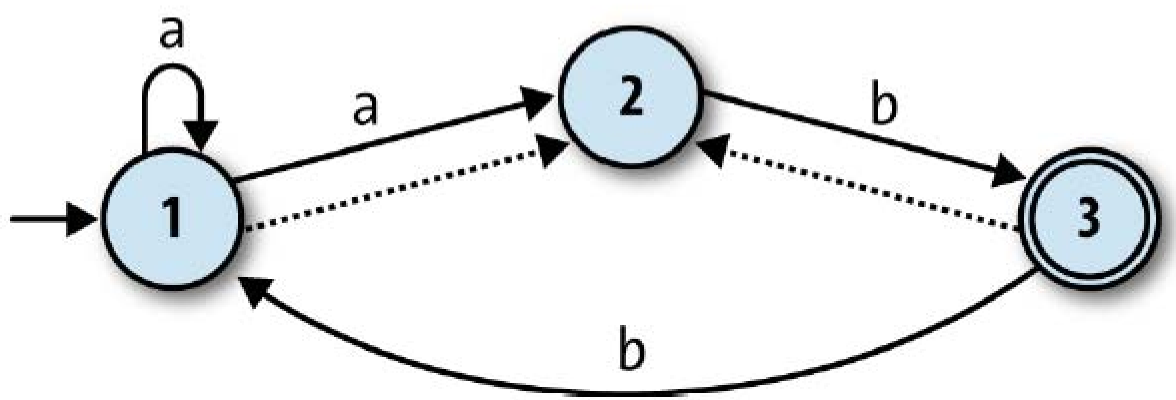
Example

- If it reads an **`a`**, it will remain in state "1 or 2":
  - When the NFA's in state 1 it can read an a and either follow the rule that keeps it in state 1 or the rule that takes it into state 2
  - From state 2, it has no way of reading an a at all



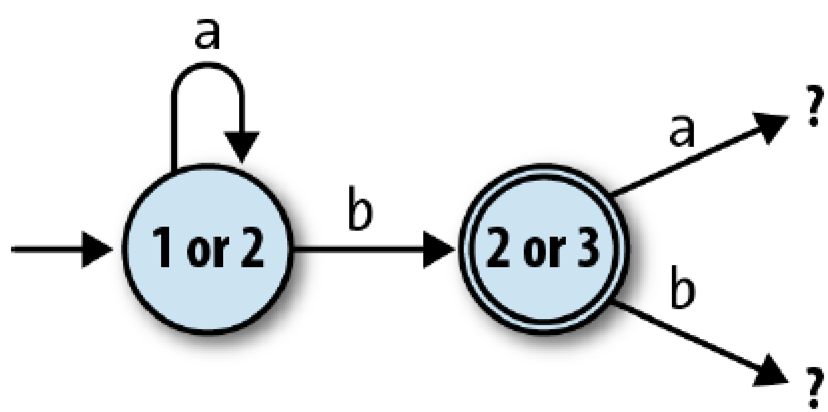
Example

- If it reads a **`b`**, it is possible for the NFA to end up in state 2 or state 3
  - From state 1, it cannot read a **`b`**, but from state 2, it can move into state 3 and potentially take a free move back into state 2 – the simulation moves into a state called "2 or 3" when the input is **`b`**



Example

- Construct a state machine for that simulation:



Note that "2 or 3" is an accept state for the simulation, because state 3 is an accept state for the NFA

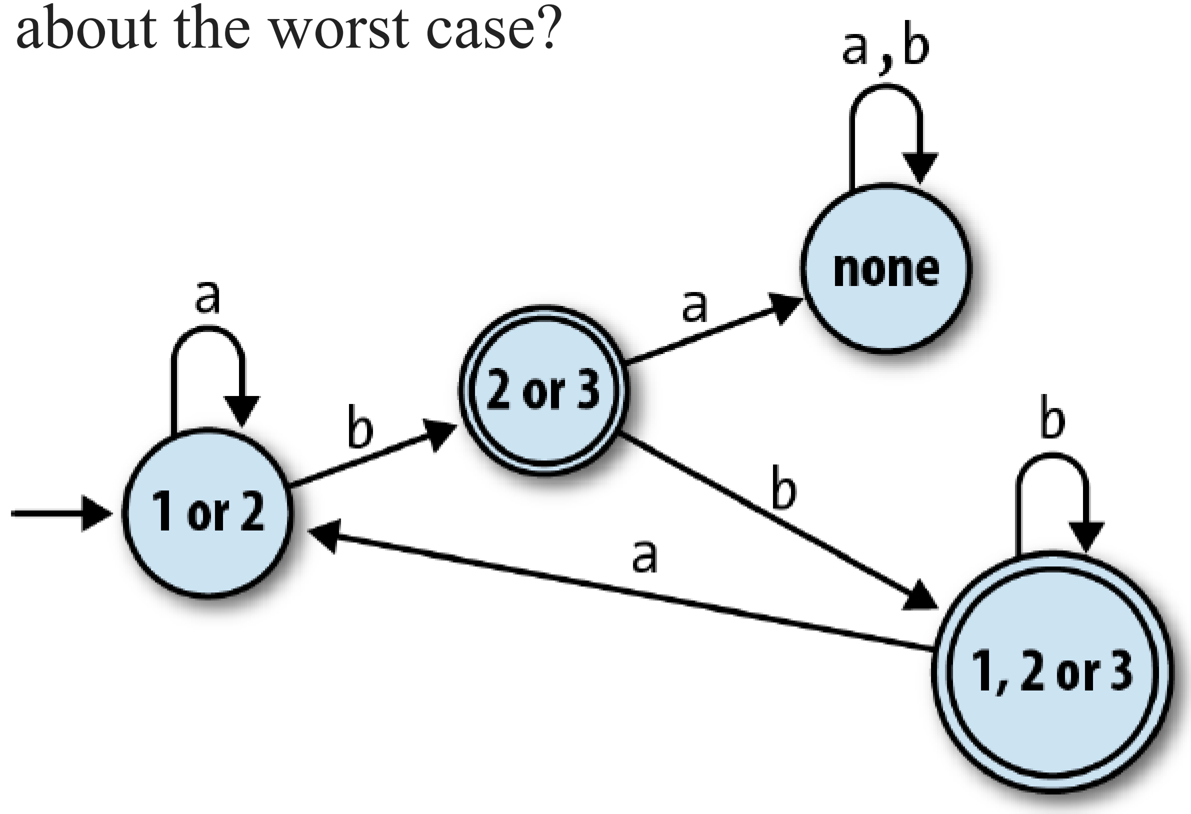
Example

- There are only four distinct combinations of states

If the NFA is in state(s)…	and reads the character…	it can end up in state(s)…
1 or 2	a	1 or 2
	b	2 or 3
2 or 3	a	none
	b	1, 2, or 3
None	a	none
	b	none
1, 2, or 3	a	1 or 2
	b	1, 2, or 3

Example

- This DFA only have one more state than the NFA
- Could produce fewer states for some NFAs
- What about the worst case?



Equivalence

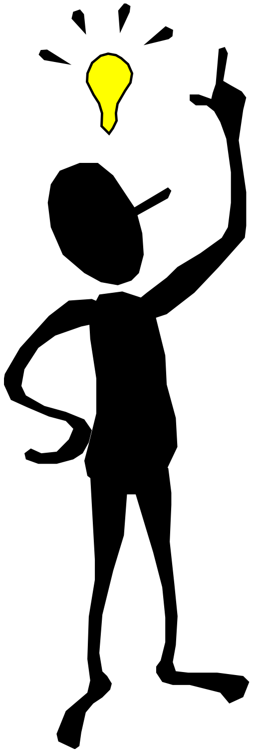
- Adding extra features of NFAs will not let us do anything that we cannot do with a DFA
- Nondeterminism and free moves are just convenient repackaging of what DFA can already do
- A DFA is easier to simulate than an NFA – a regular expression implementation can convert a pattern into first an NFA and then a DFA

Kleene’s Theorem

- By using aforementioned constructions, we can create for every regular expression an NFA that accepts the corresponding language
- Theorem: For every finite automaton M=(Q, Σ, q_0, A, δ), the language $L(M)$ is regular

Parsing

- We almost built a complete (albeit basic) regular expression implementation
- We need a *parser* for pattern syntax: it would be much more convenient if we could just write **`(a(|b))*`** instead of building the abstract syntax tree manually with **repeat**, **choose**, and **concatenate**
- Language grammar → next topic



Thanks ! ☺

Questions ?