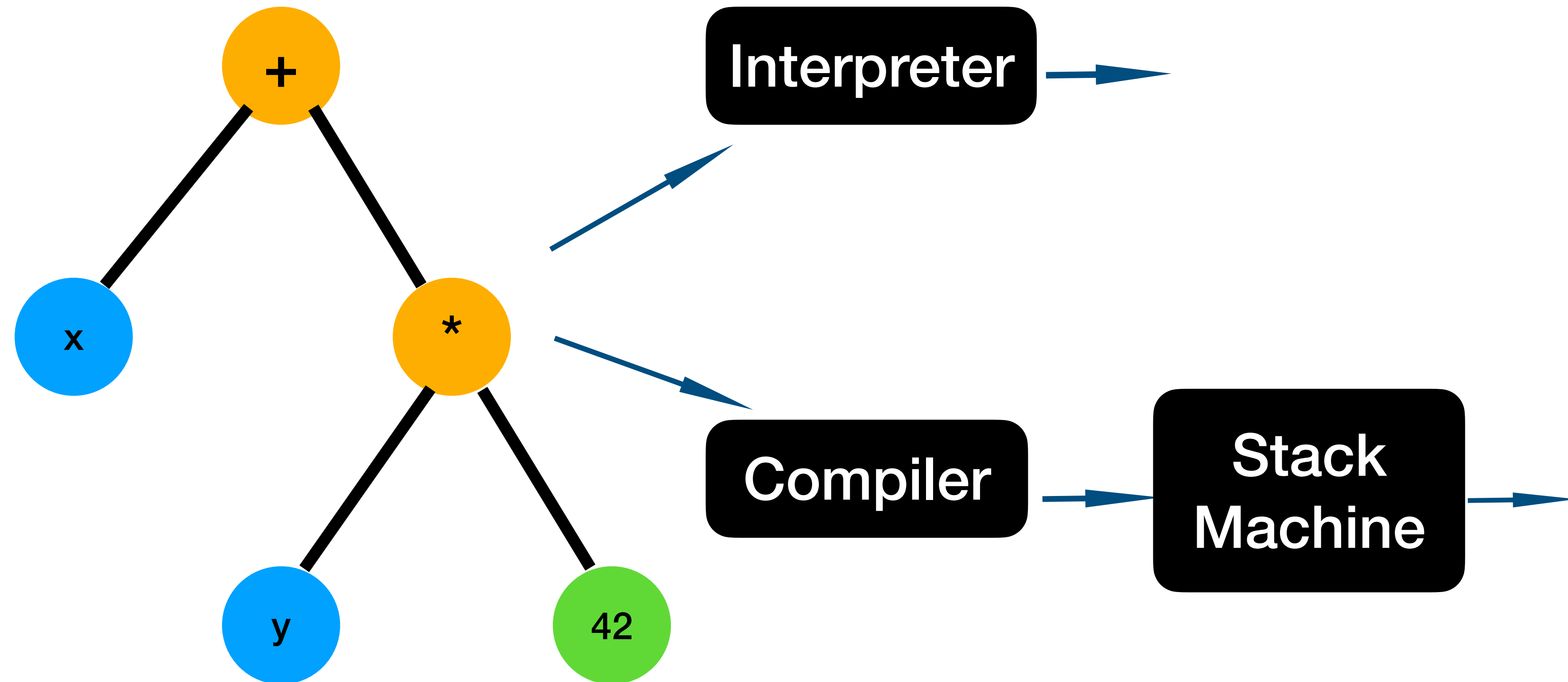
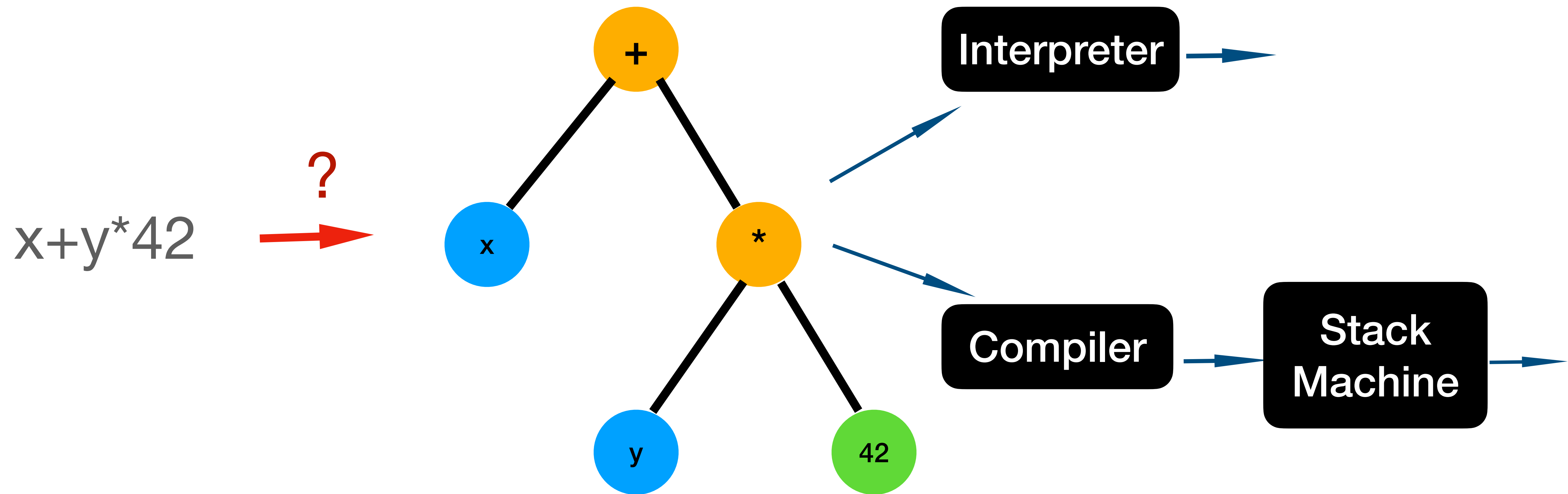


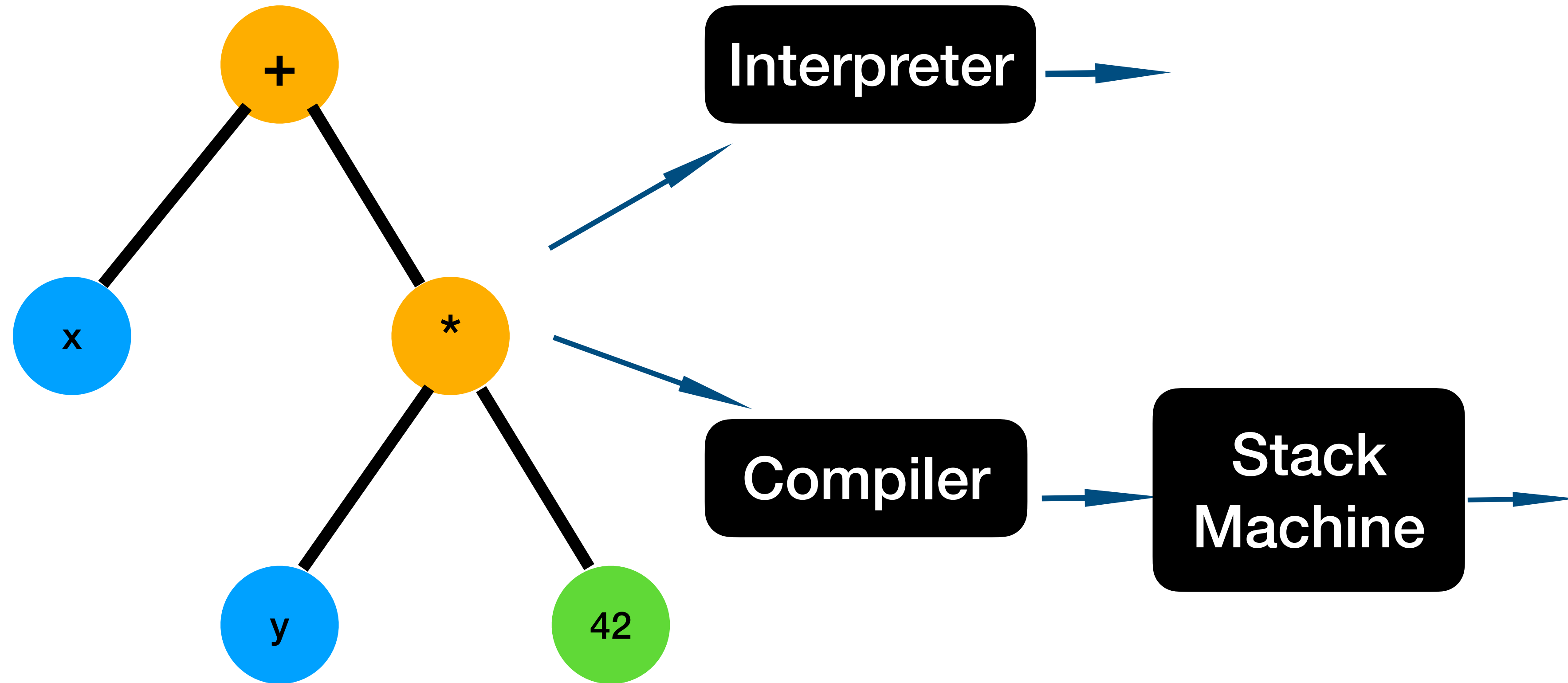
Previous lecture

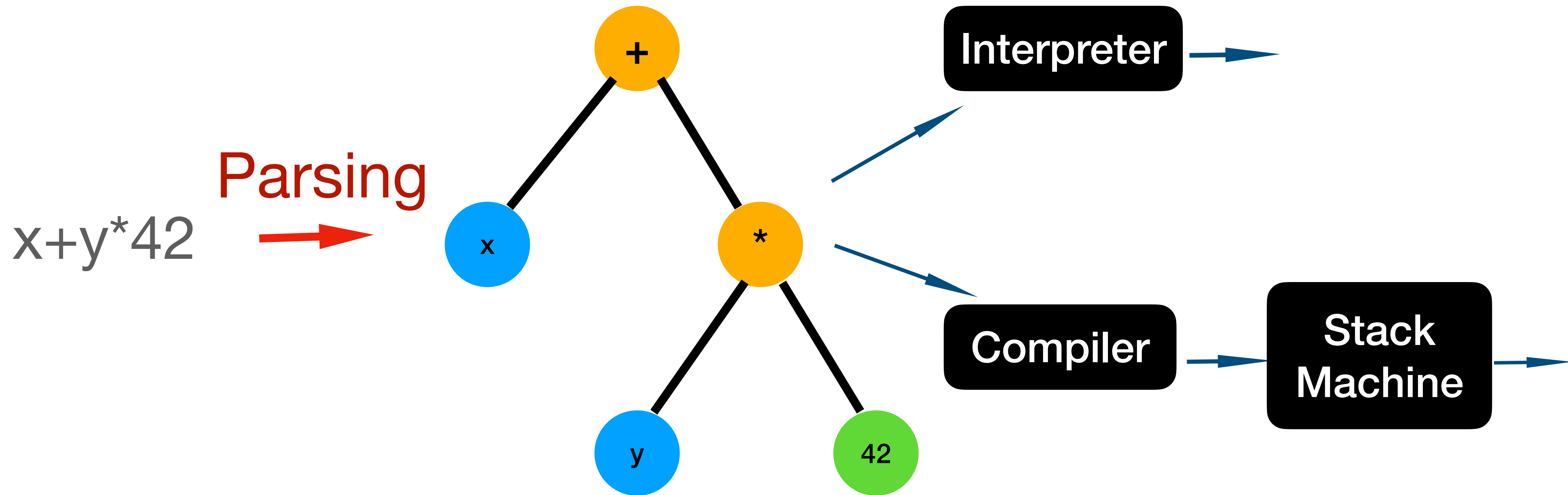




$$x + y^*42$$

Parsing



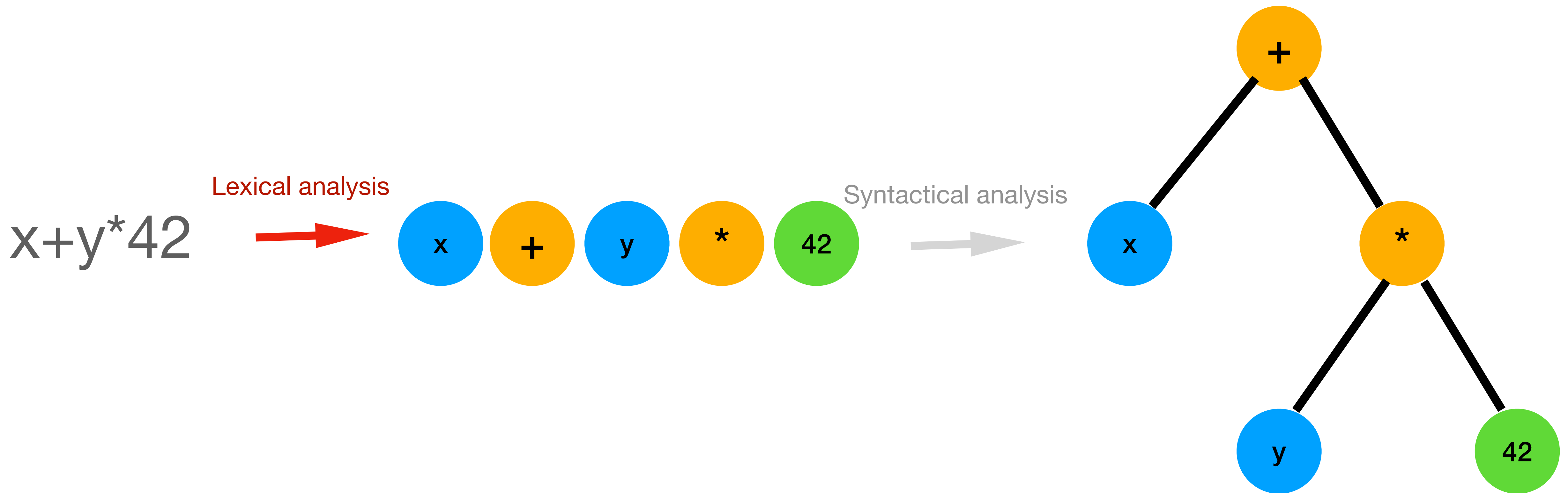


Parsing is about getting the structure

The structure matters

What is like parsing $0.1 + 0.2 + 0.3$?

How to parse: A tale of two analyses



Parsing (1/3): Lexical Analysis

Zhoulai Fu

with a subset of slides taken, or inspired from those of last years

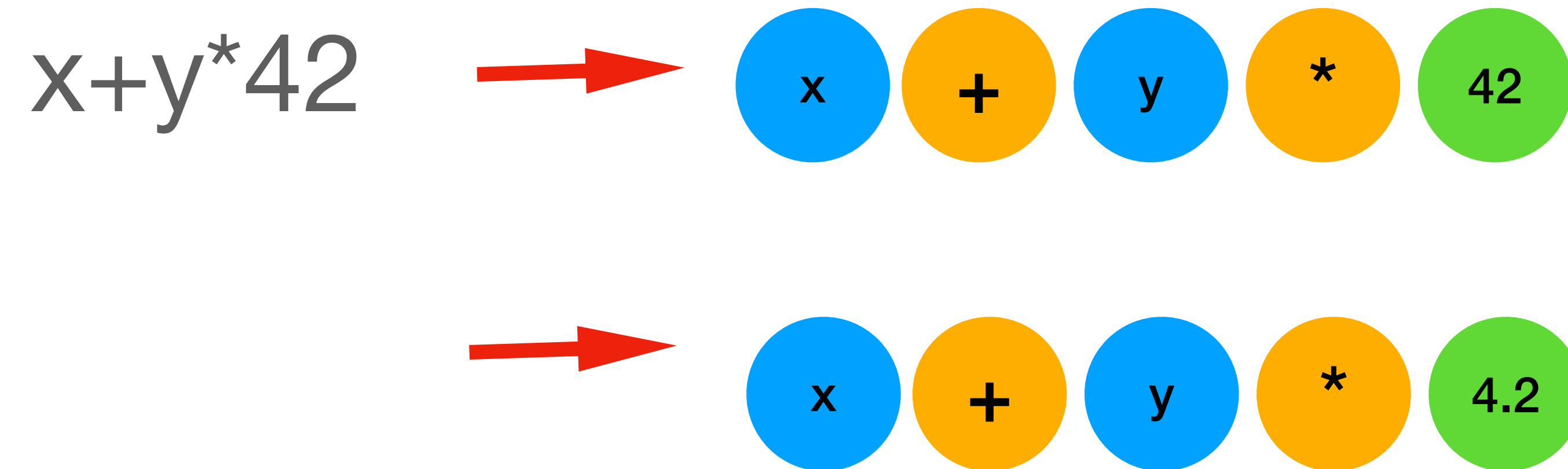
Parsing (1/3): Lexical Analysis

Program as data, autumn 2020

Zhoulai Fu, 01/09/2020

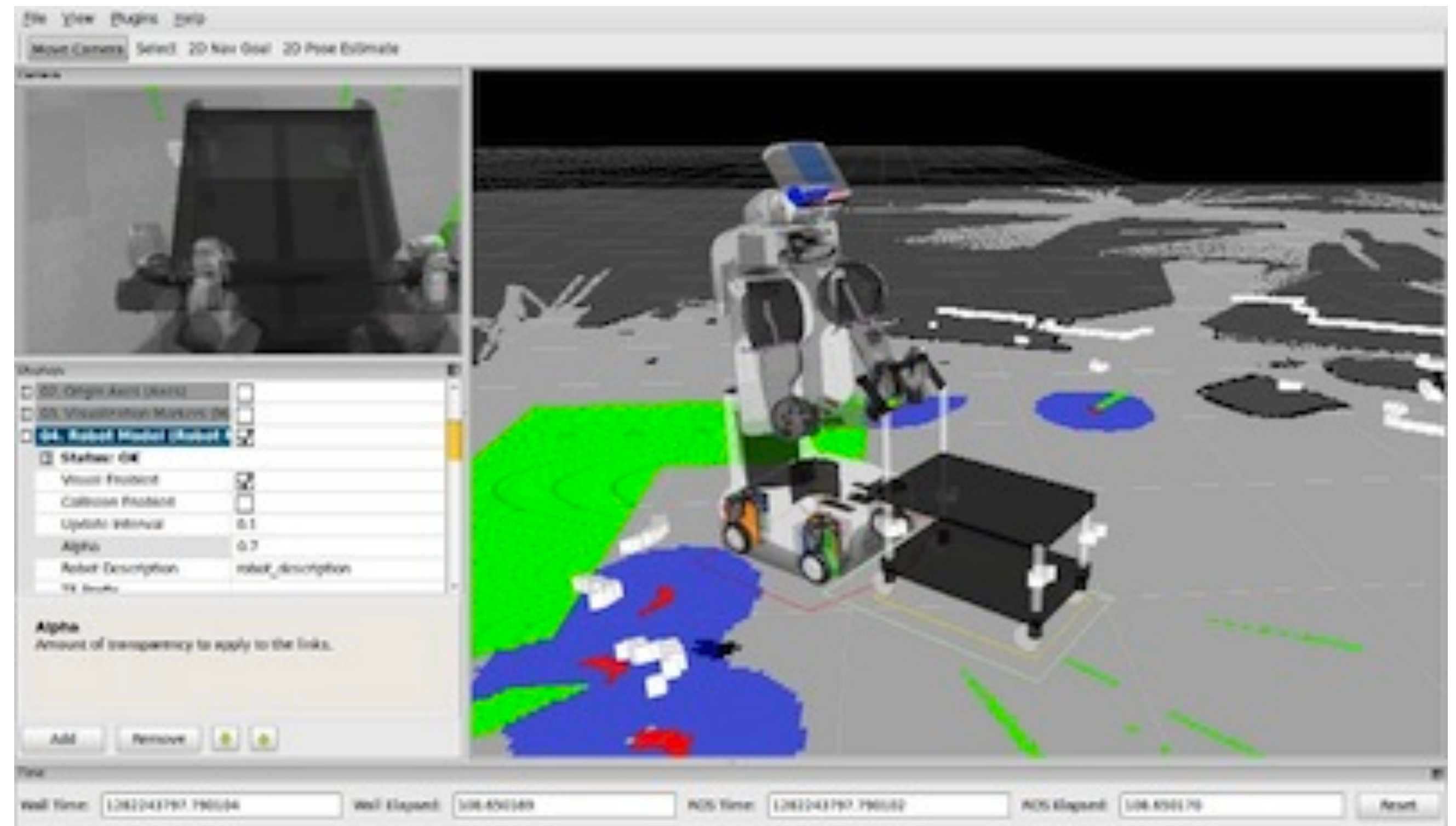
with some slides taken, or inspired from those of last years

Lexical analysis: LexBuffer to tokens, or manipulate the tokens



A research problem I worked on at ITU

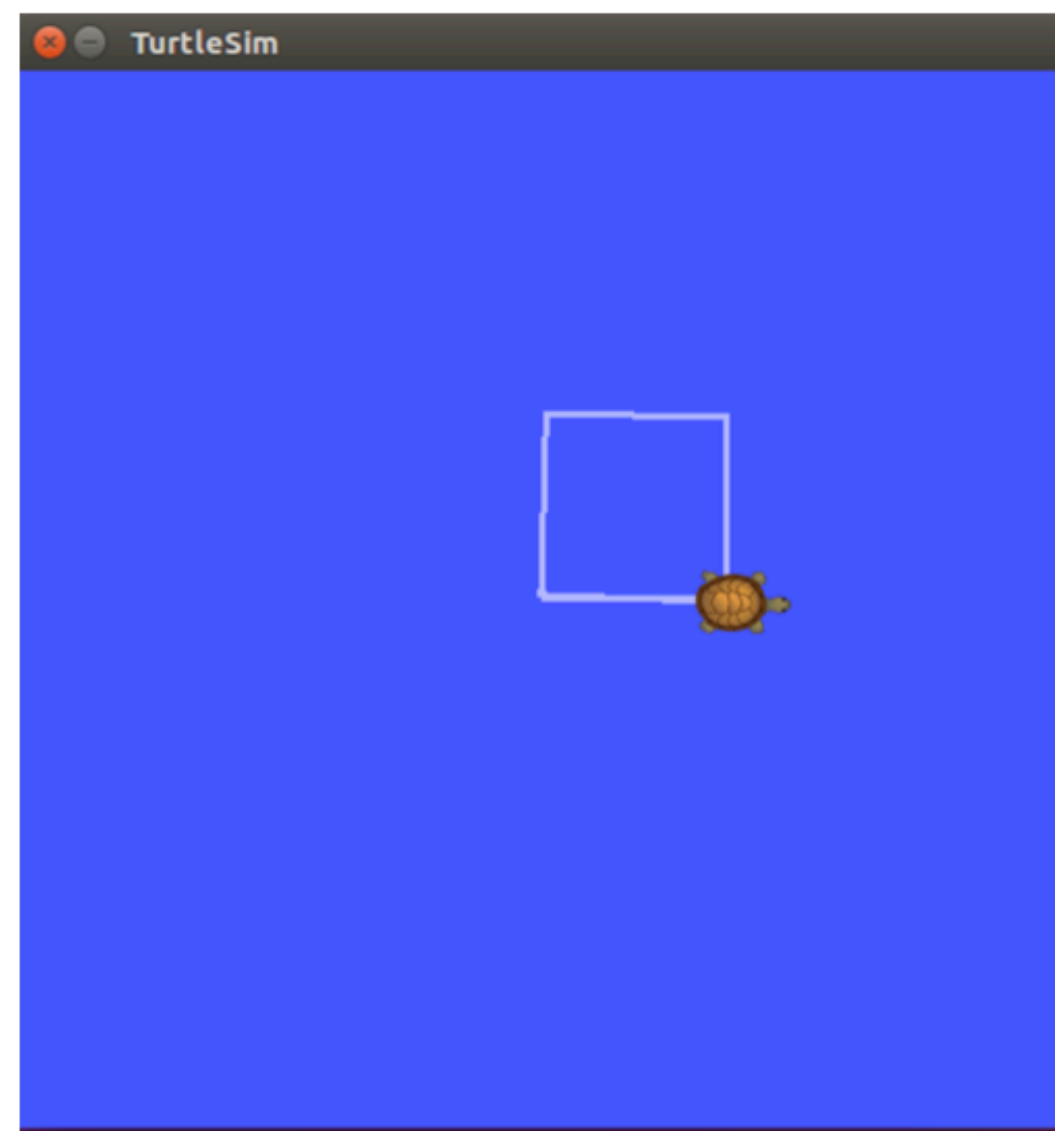
- Robot Operating System
- Automated Testing



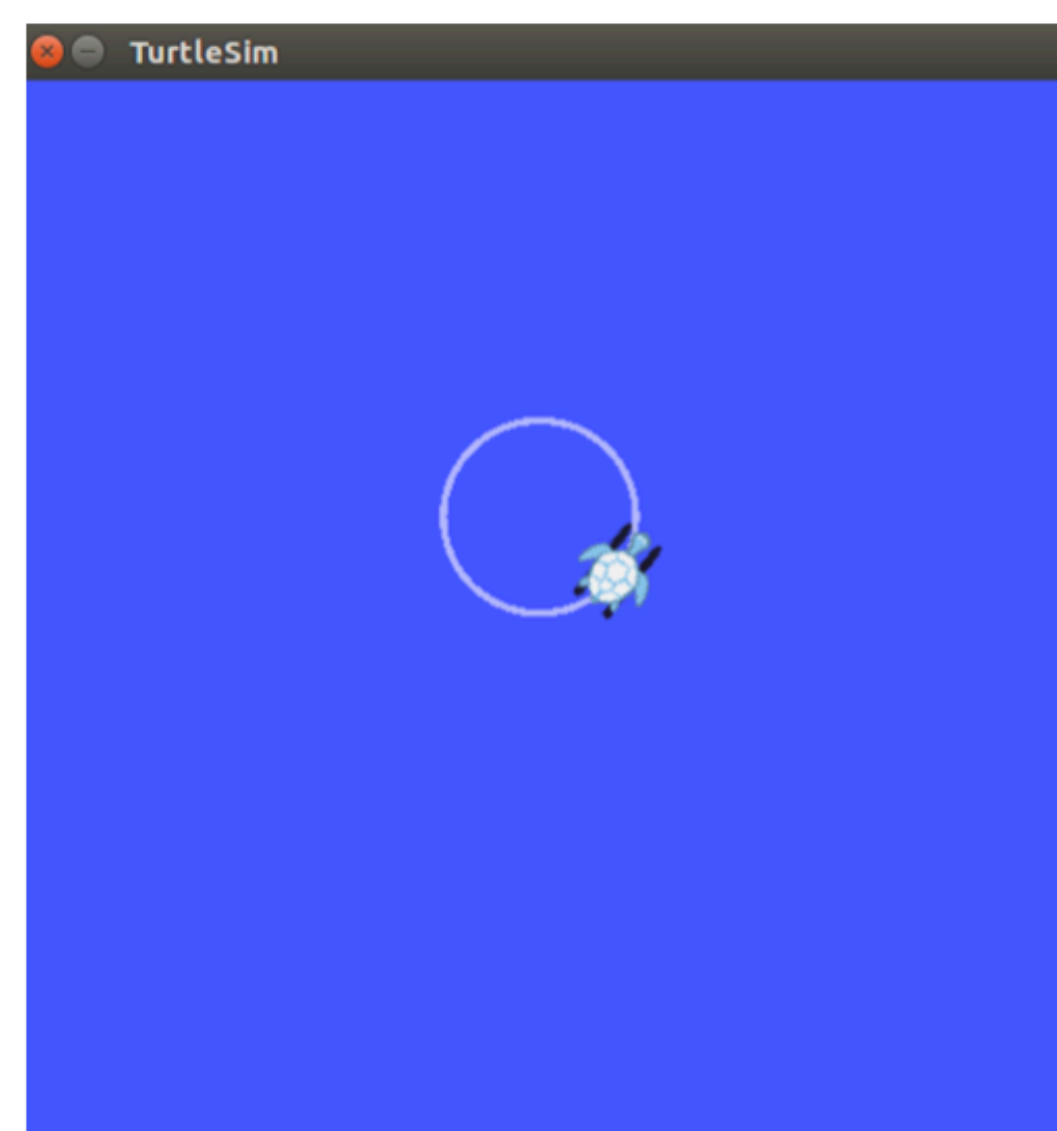
An idea due to lexical analysis

Demo

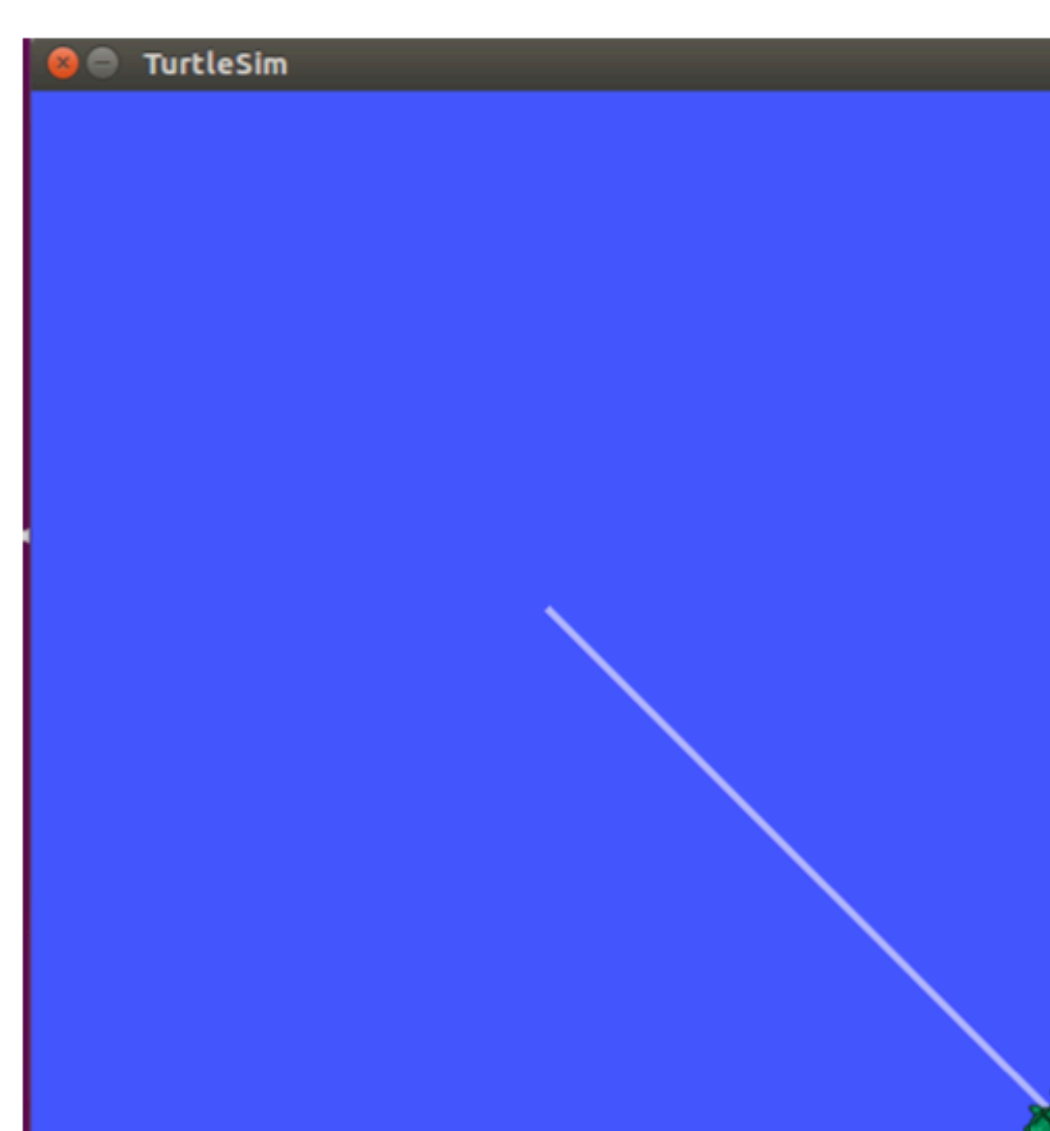
- Robot: Turn the wheel with 30 degree and a speed of 10 km/h
- Lexical analysis: Turn the wheel with degree = A, and a speed = B



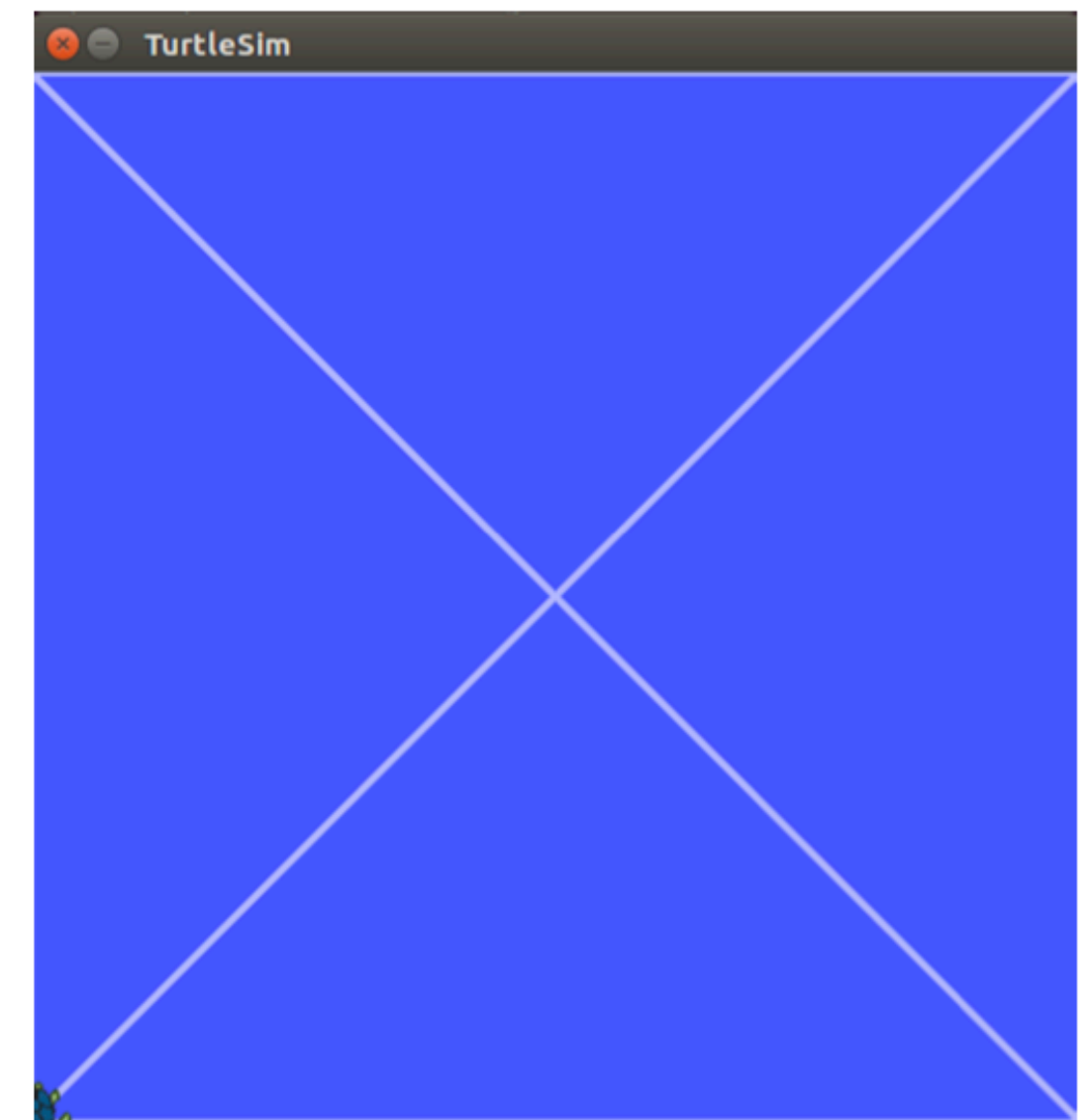
(a)



(b)



(c)



(d)

Menu in the following

- Write a lexer from scratch
- Languages and regular expressions
- Generate a lexer automatically

Write a lexer from scratch

- Problem to be solved: Recognize a valid positive integer
- 42 should be recognized
- 0, or 023 should not be recognized
- ==> The string starts with 1-9, followed by digits of 0-9
- ==> In other words, we can specify a lexer via a regular expression `[1-9][0-9]*`
- Demo with Linux command “grep”

Languages

- Danish; Python; C#; PHP; domain-specific languages
- An open question: How would you define a language?
- Or formulated in another way: What would we talk about when we talk about a language?

What we talk about when we talk about a language

- Does a word belong to the language => We talk about regular expressions
- Is a sentence grammatically correct => We will talk about context-free grammar next week
- What is the meaning of a sentence => We will talk about semantics later

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{ "a" \}$
ε	Empty string	$\{ "" \}$
$r_1 r_2$	r_1 followed by r_2	$\{ s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2) \}$
r^*	Zero or more r	$\{ s_1 \dots s_n \mid s_i \in \mathcal{L}(r), n \geq 0 \}$
$r_1 r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

Examples

ab^* represents $\{ "a", "ab", "abb", \dots \}$
 $(ab)^*$ represents $\{ "", "ab", "abab", \dots \}$
 $(a|b)^*$ represents $\{ "", "a", "b", "aa", "ab", "ba", \dots \}$

Exercise

What does $(a|b)c^*$ represent?

Common extensions of regular expressions

Abbrev.	Meaning	Expansion
<code>[aeiou]</code>	Set	<code>a e i o u</code>
<code>[0-9]</code>	Range	<code>0 1 ... 8 9</code>
<code>[0-9a-z]</code>	Ranges	<code>0 1 ... 8 9 a b ... y z</code>
<code>r?</code>	Zero or one <i>r</i>	<code>r ε</code>
<code>r⁺</code>	One or more <i>r</i>	<code>r r*</code>

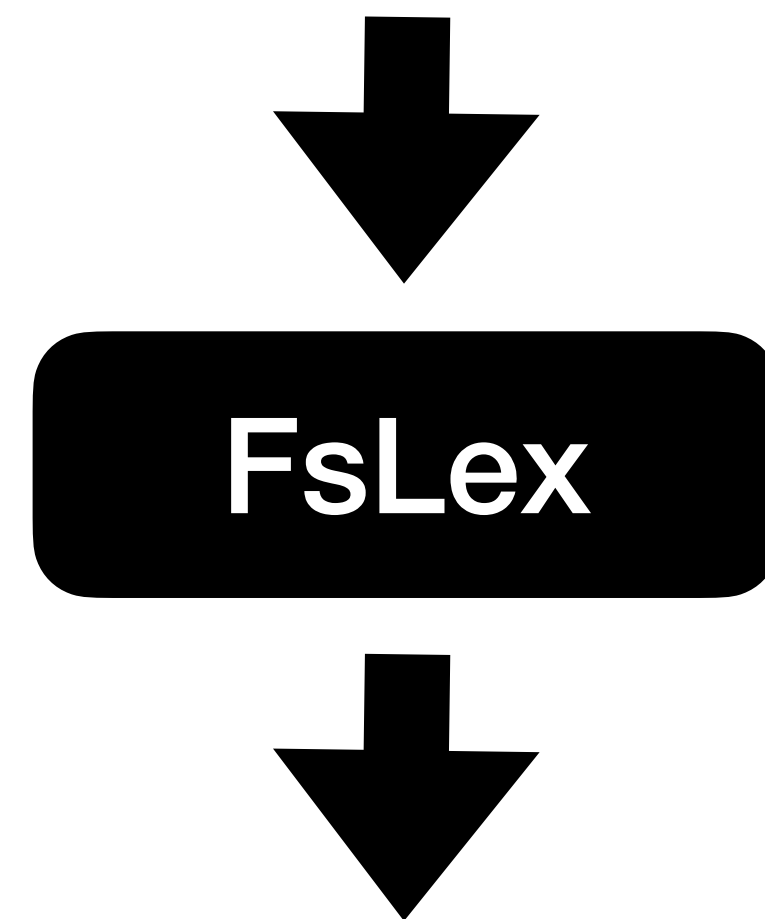
- `[hc]?at` matches "at", "hat", and "cat".
- `[hc]+at` matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on, but not "at".

Generate a lexer

- Suppose we have a set of regular expressions
- Each regular expression represents a kind of token
- Each regular expression is associated with an F# expression, called its semantic value
- Then, we should specify a lexer by defining the association that maps regular expressions to semantic values
- FsLex is a tool that, given the association, generates a lexer automatically

Lexer specification (.fsl)

Regular expressions -> Semantic values



Lexer (.fs)

LexemeBuffer-> Semantic values

A minimal lexer specification

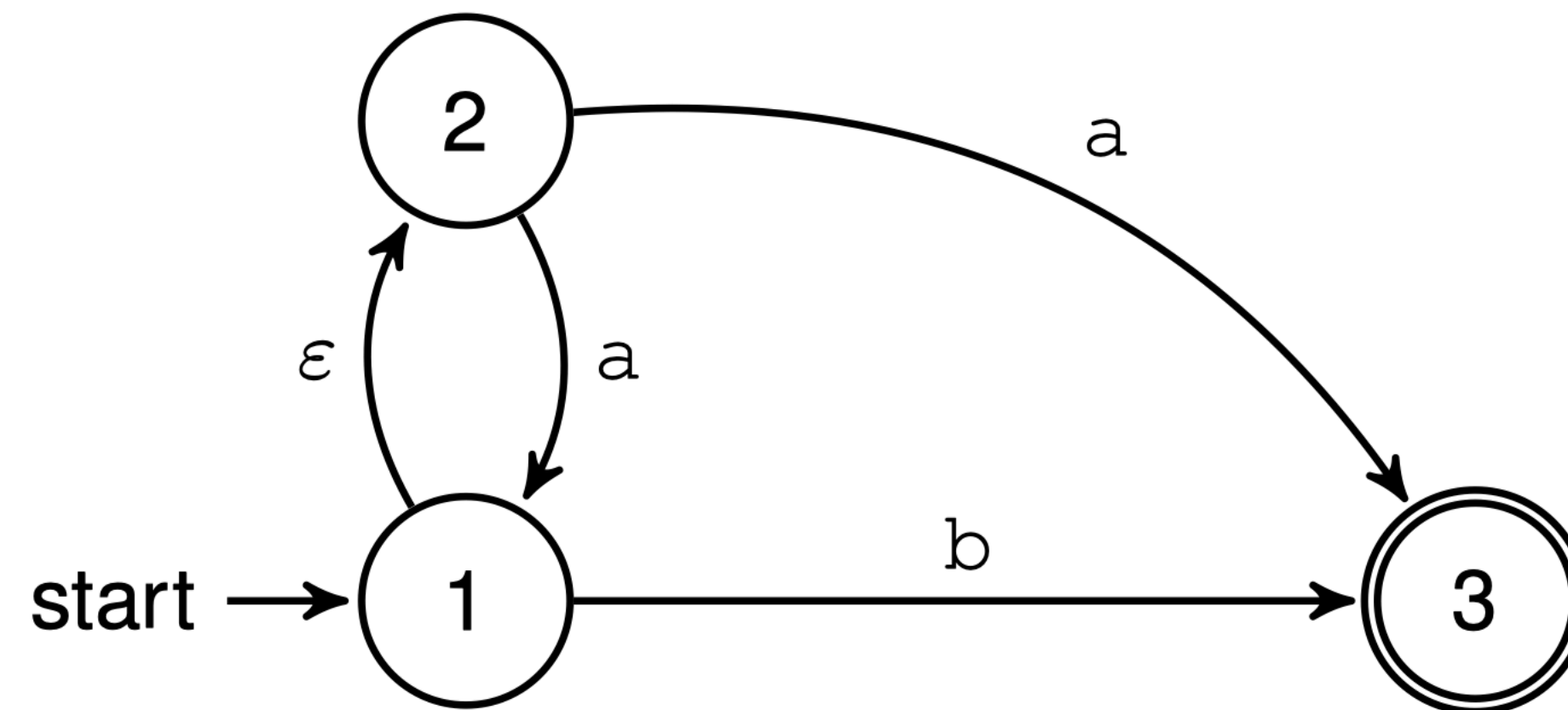
```
rule Tokenize = parse
  | ['0'-'9']      { LexBuffer<char>.LexemeString lexbuf }
  | _              { failwith "Lexer error: illegal symbol"
```

- The regular expression specified a single digit
- “LexBuffer.LexemeString lexbuf” is a utility of FsLex that transforms the recognized regular expression to a string
- In this way, Tokenize will be a function that maps a LexBuffer to a string

**Behind FsLex, regular expressions
are transformed to automata**

Finite automata

- A finite automaton, FA, is a graph of states (nodes) and labelled transitions (edges)



- An FA accepts string s if there is a path from start to an accept state such that the labels make up s
- Epsilon (ϵ) does not contribute to the string
- This automaton is nondeterministic (NFA)
- It accepts string b
- Does it accept a or aa or ab or aba ?

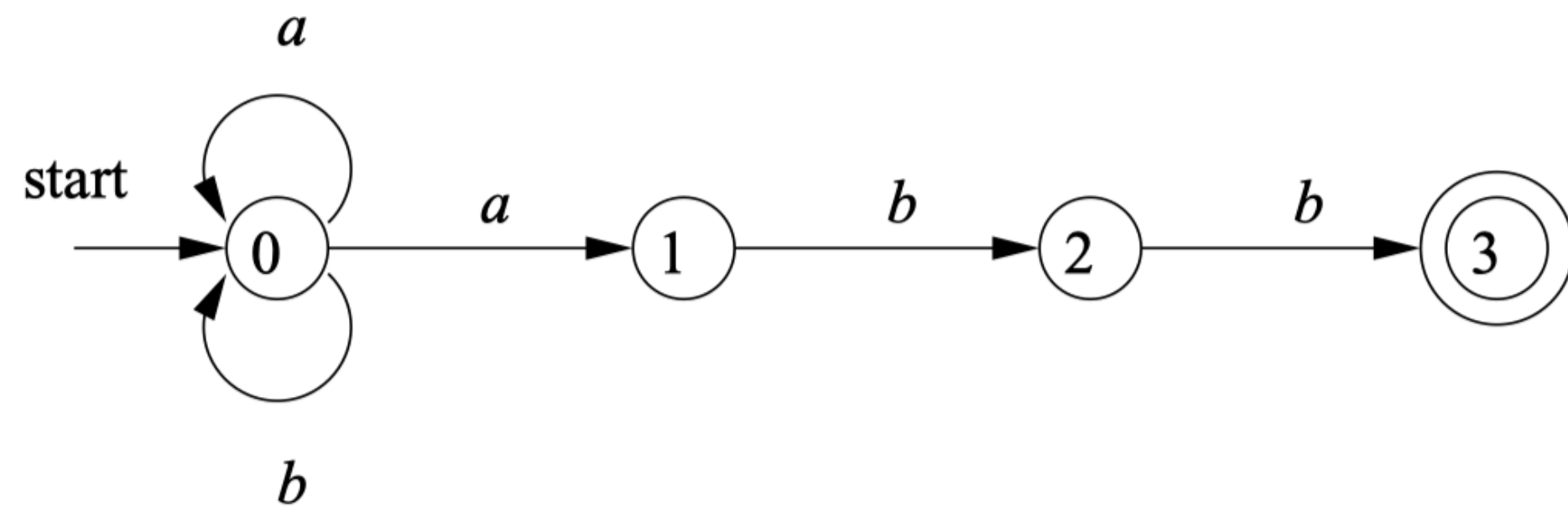
Finite automata come in two flavors

- (a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ϵ , the empty string, is a possible label.
- (b) *Deterministic finite automata* (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

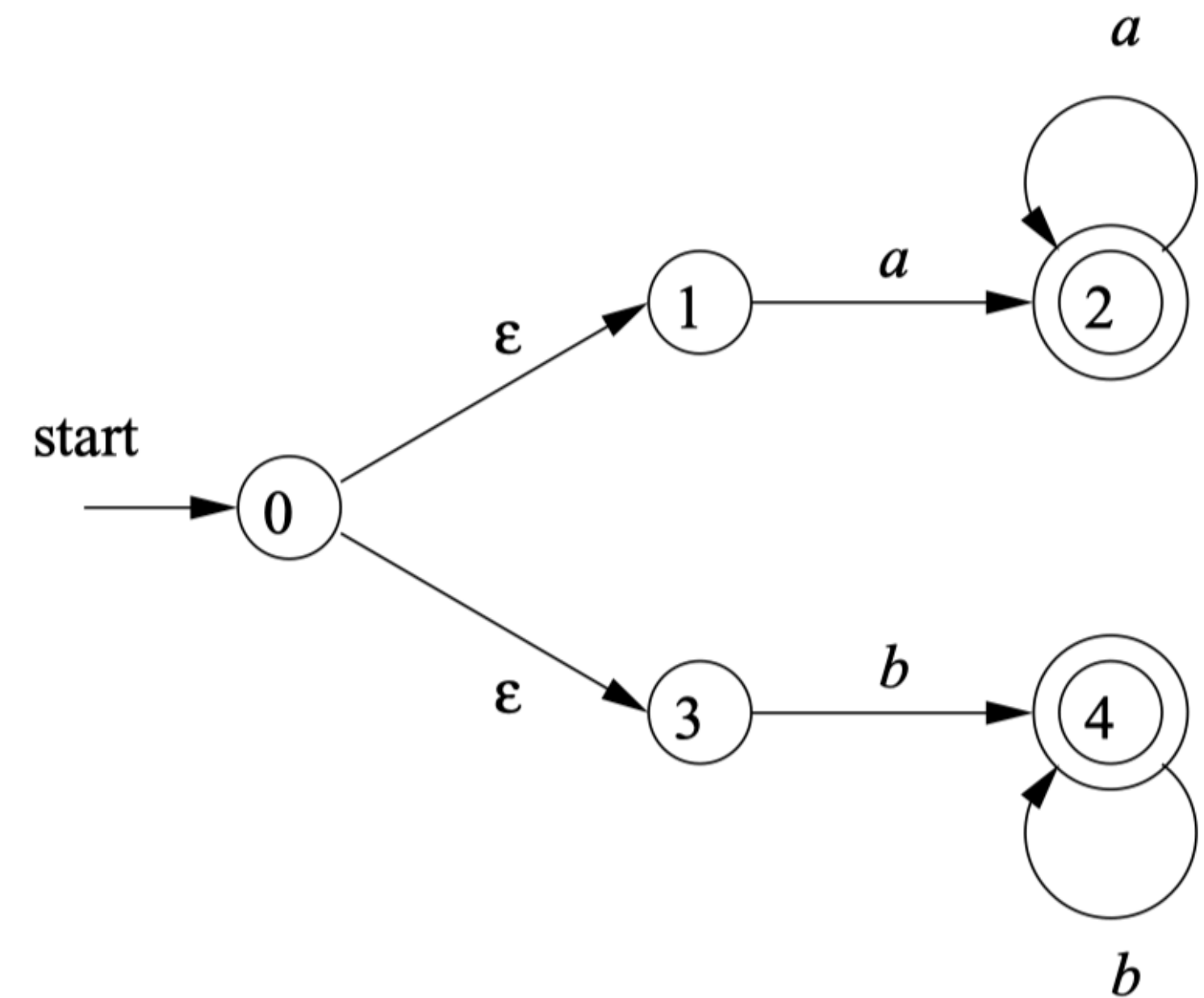
An NFA consists of:

1. A finite set of states S .
2. A set of input symbols Σ , the *input alphabet*. We assume that ϵ , which stands for the empty string, is never a member of Σ .
3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.
4. A state s_0 from S that is distinguished as the *start state* (or *initial state*).
5. A set of states F , a subset of S , that is distinguished as the *accepting states* (or *final states*).

Examples of NFA



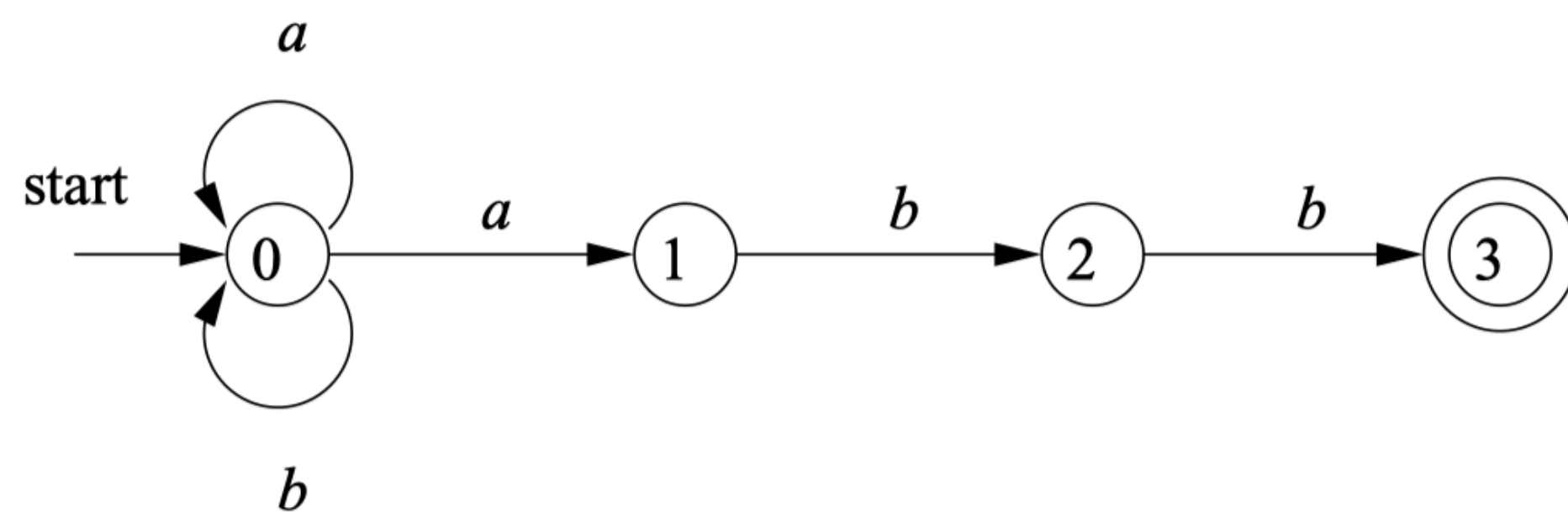
NFA accepting $(a|b)^*abb$



NFA accepting $aa^*|bb^*$

Transition function in NFA

A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.



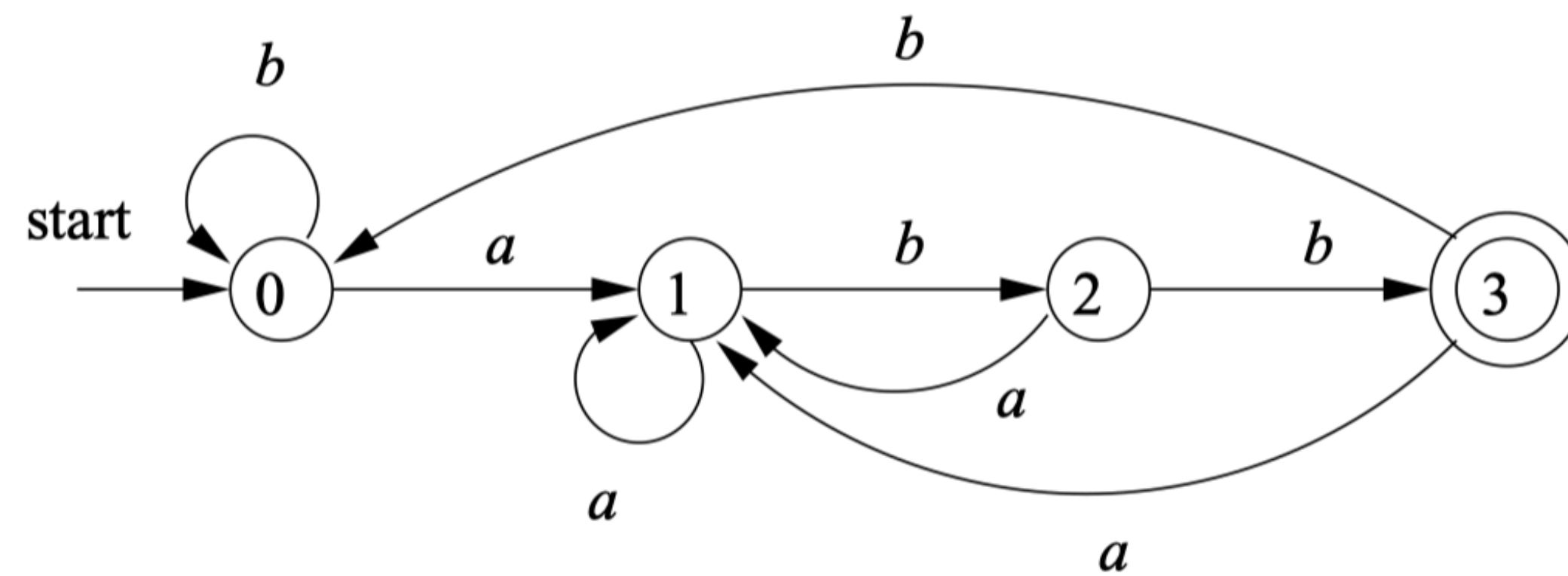
STATE	a	b	ϵ
0	$\{0, 1\}$	$\{0\}$	\emptyset
1	\emptyset	$\{2\}$	\emptyset
2	\emptyset	$\{3\}$	\emptyset
3	\emptyset	\emptyset	\emptyset

NFA accepting $(a|b)^*abb$

A DFA consists of a special case of an NFA where:

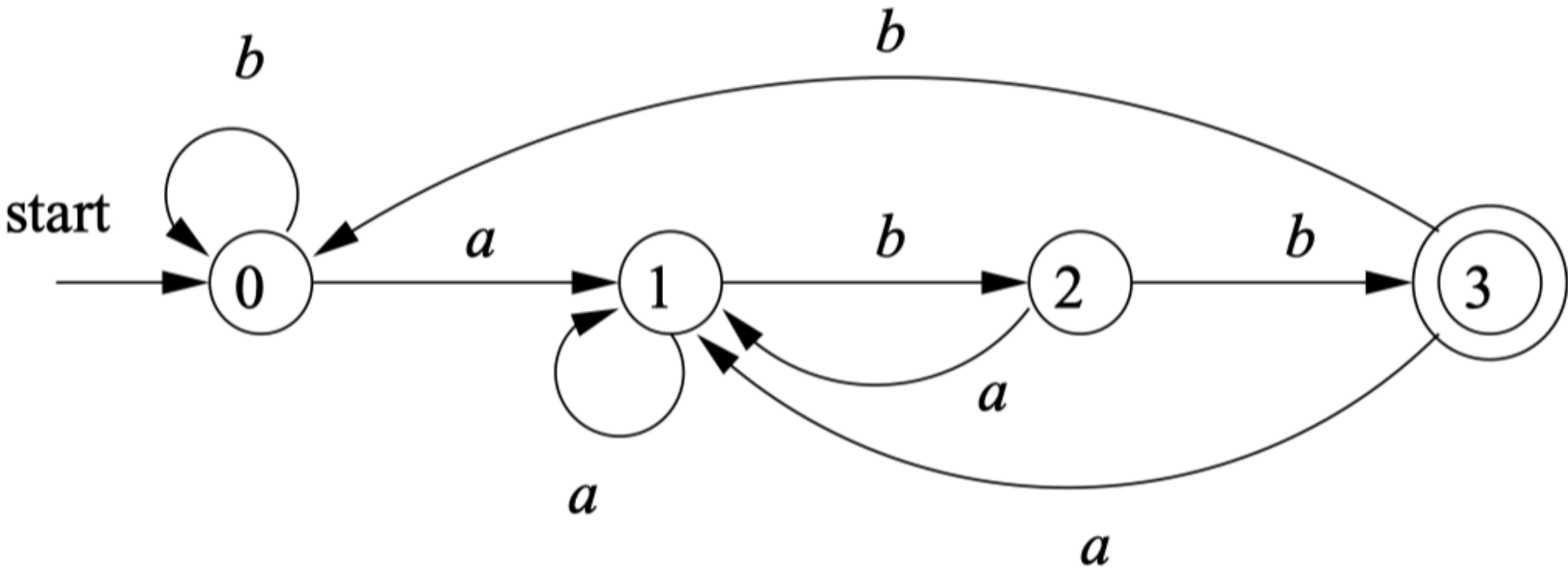
1. There are no moves on input ϵ , and
2. For each state s and input symbol a , there is exactly one edge out of s labeled a .

An example of DFA



DFA accepting $(a|b)^*abb$

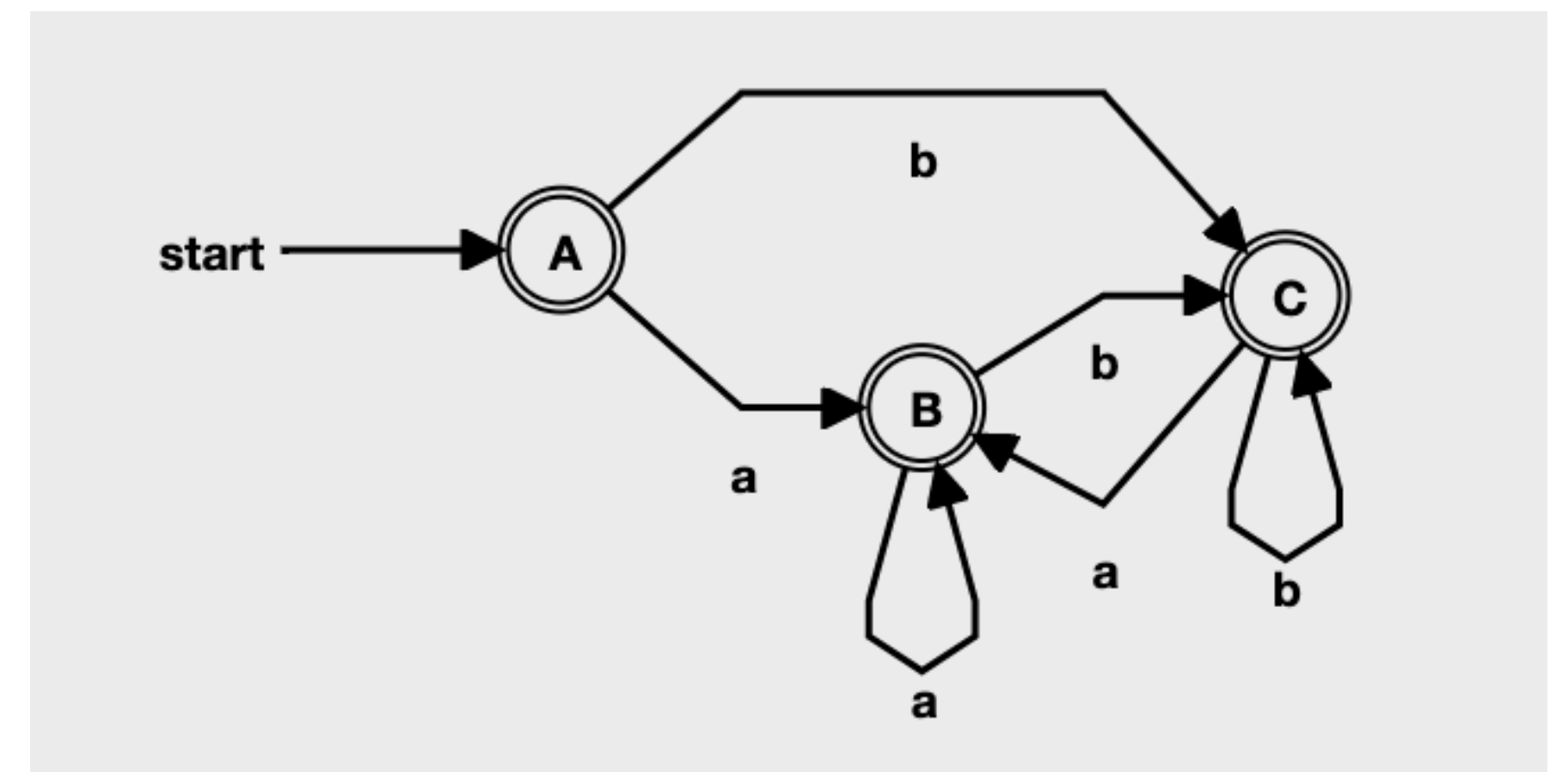
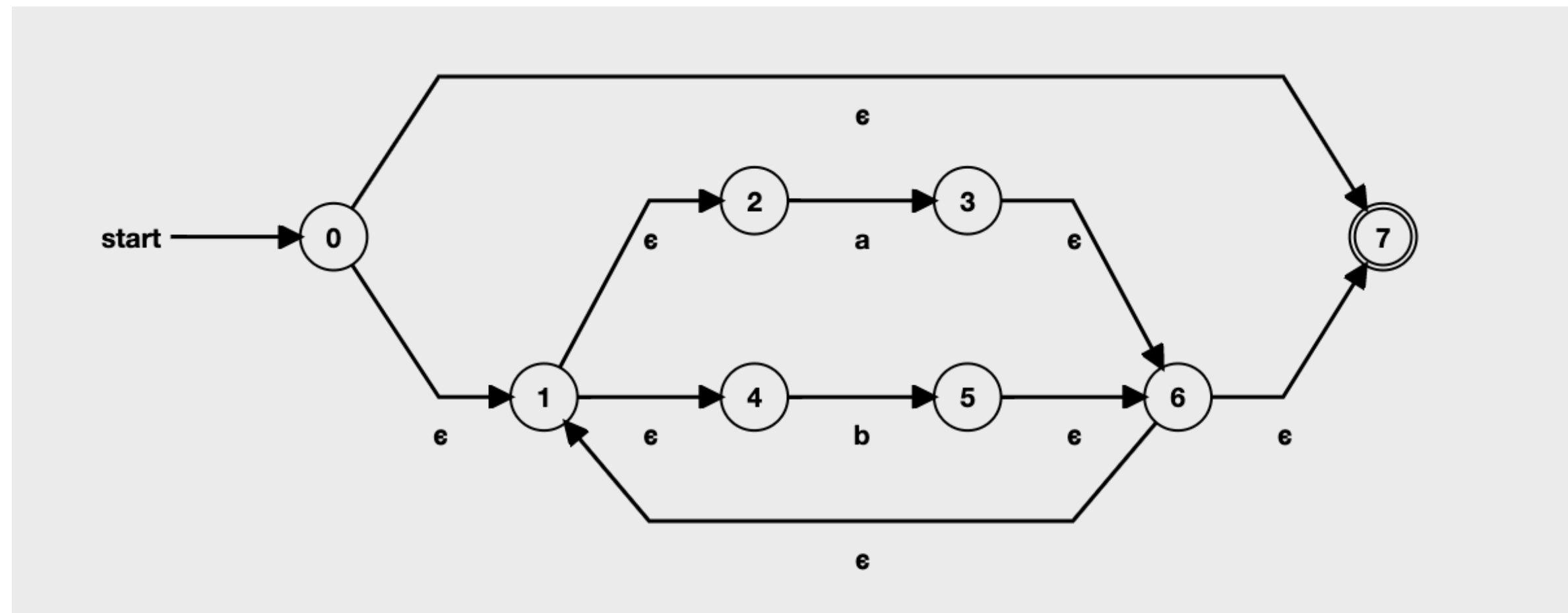
Transition function of a DFA



DFA accepting $(a|b)^*abb$

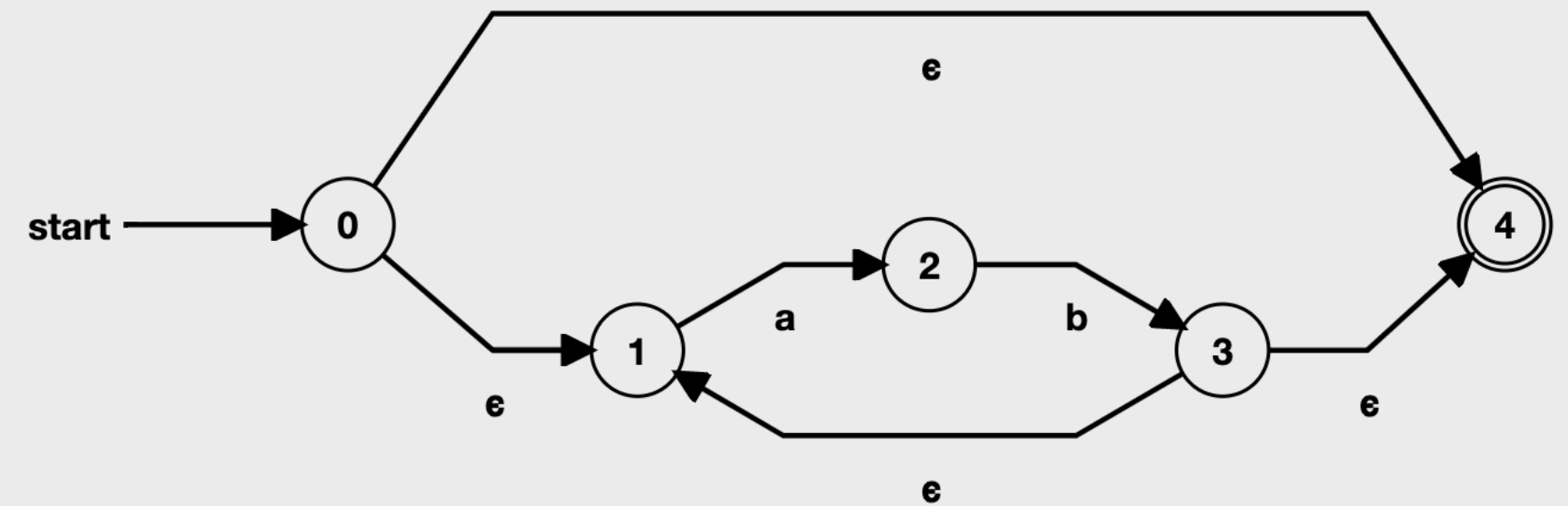
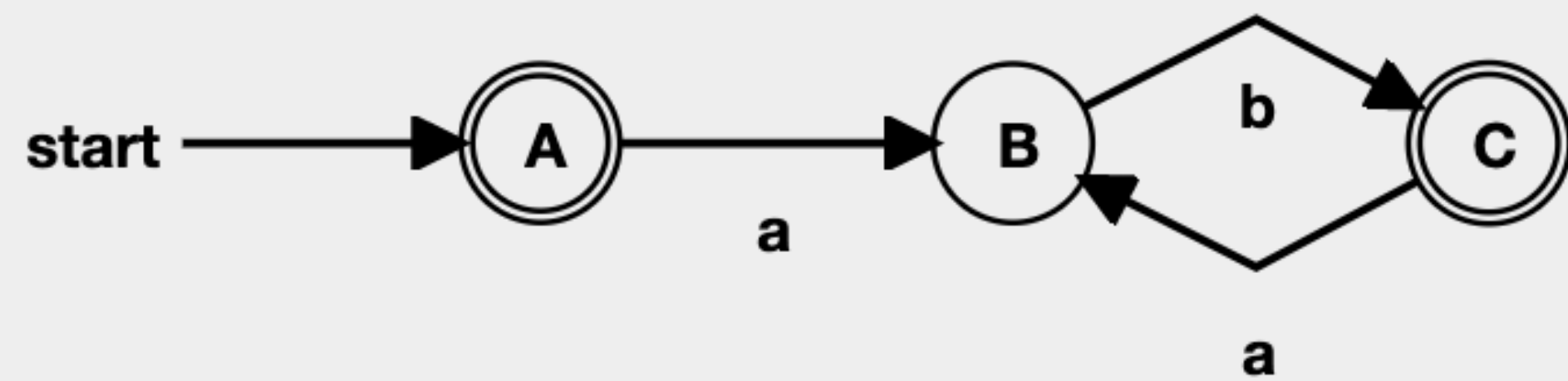
	a	b
0	1	0
1	1	2
2	1	3
3	1	0

Quiz: NFA or DFA



Both recognize $(a|b)^*$

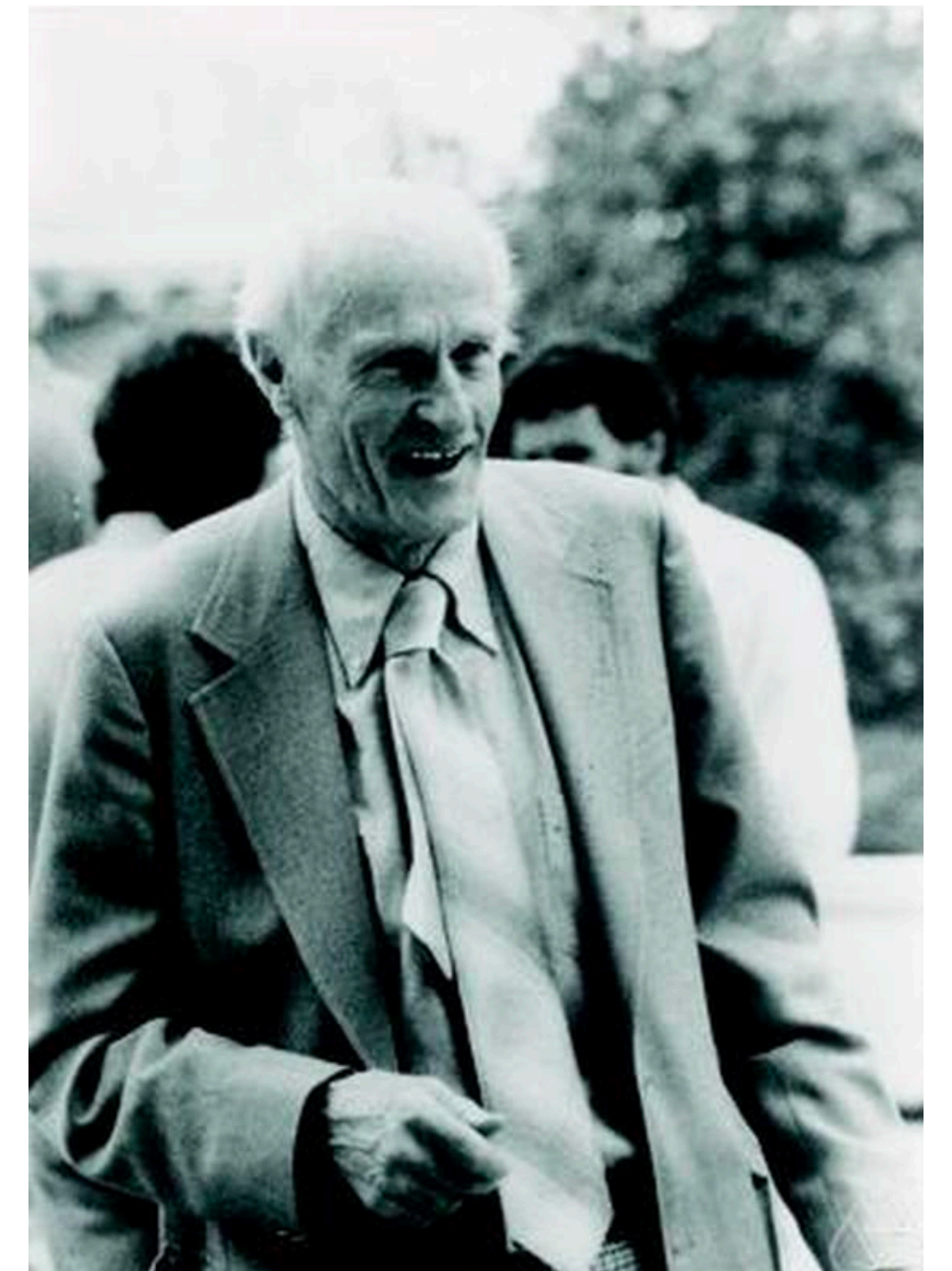
Quiz: NFA or DFA



Both recognize $(ab)^*$

Kleene's theorem

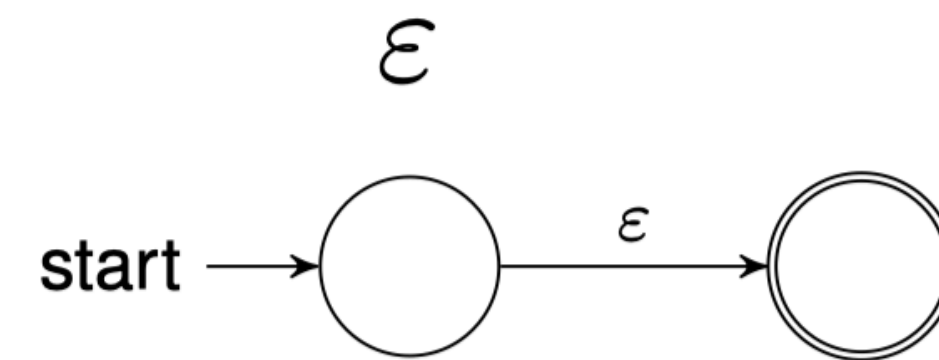
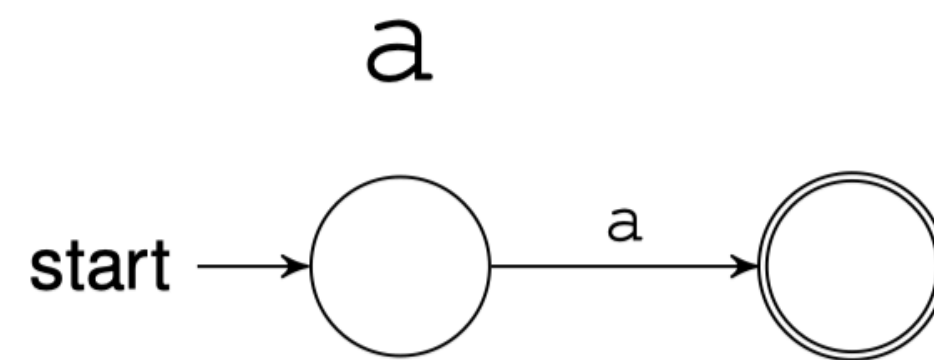
- Theorem: The set of regular languages, the set of NFA-recognizable languages, and the set of DFA-recognizable languages are all the same.
- Proof: $RE \rightarrow NFA \rightarrow DFA \rightarrow RE$



How to prove RE \rightarrow NFA

Proof objective: Each regular expression can be represented by an NFA

Proof strategy: Build NFA recursively by the case of the regular expression.

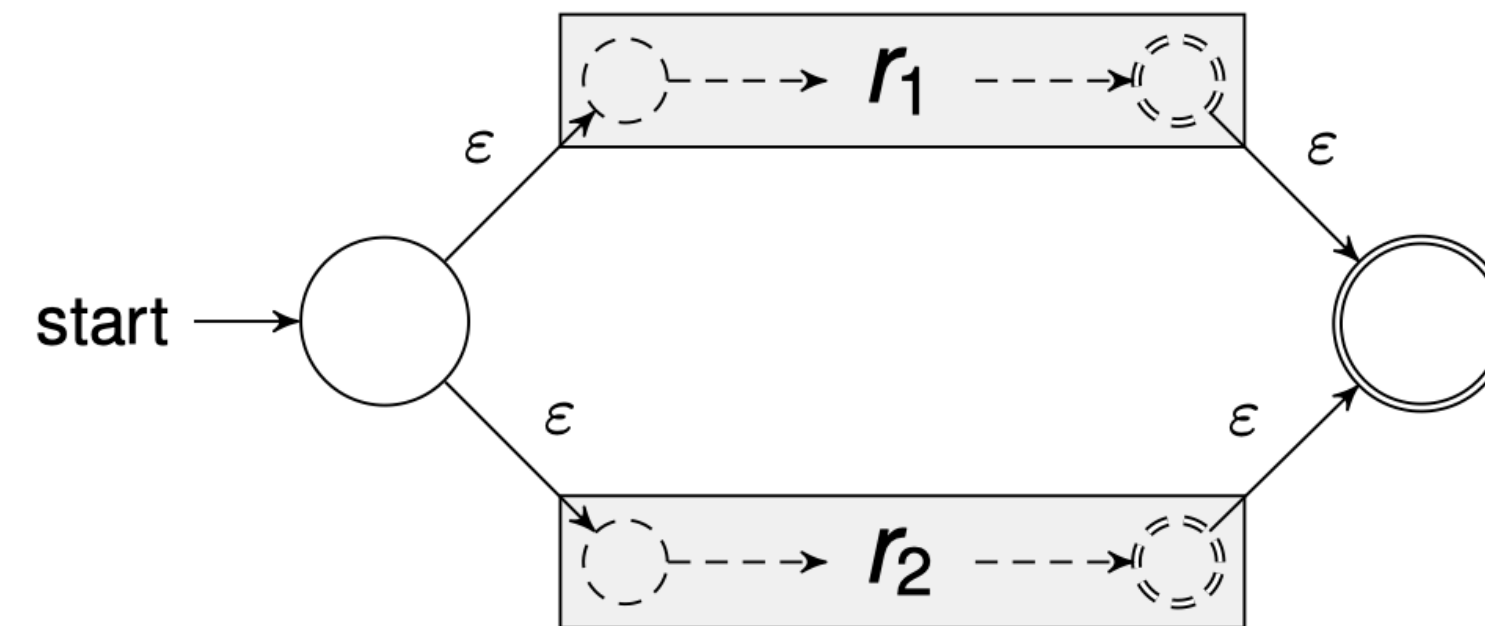


How to prove RE \rightarrow NFA

Proof objective: Each regular expression can be represented by an NFA

Proof strategy: Build NFA recursively by the case of the regular expression.

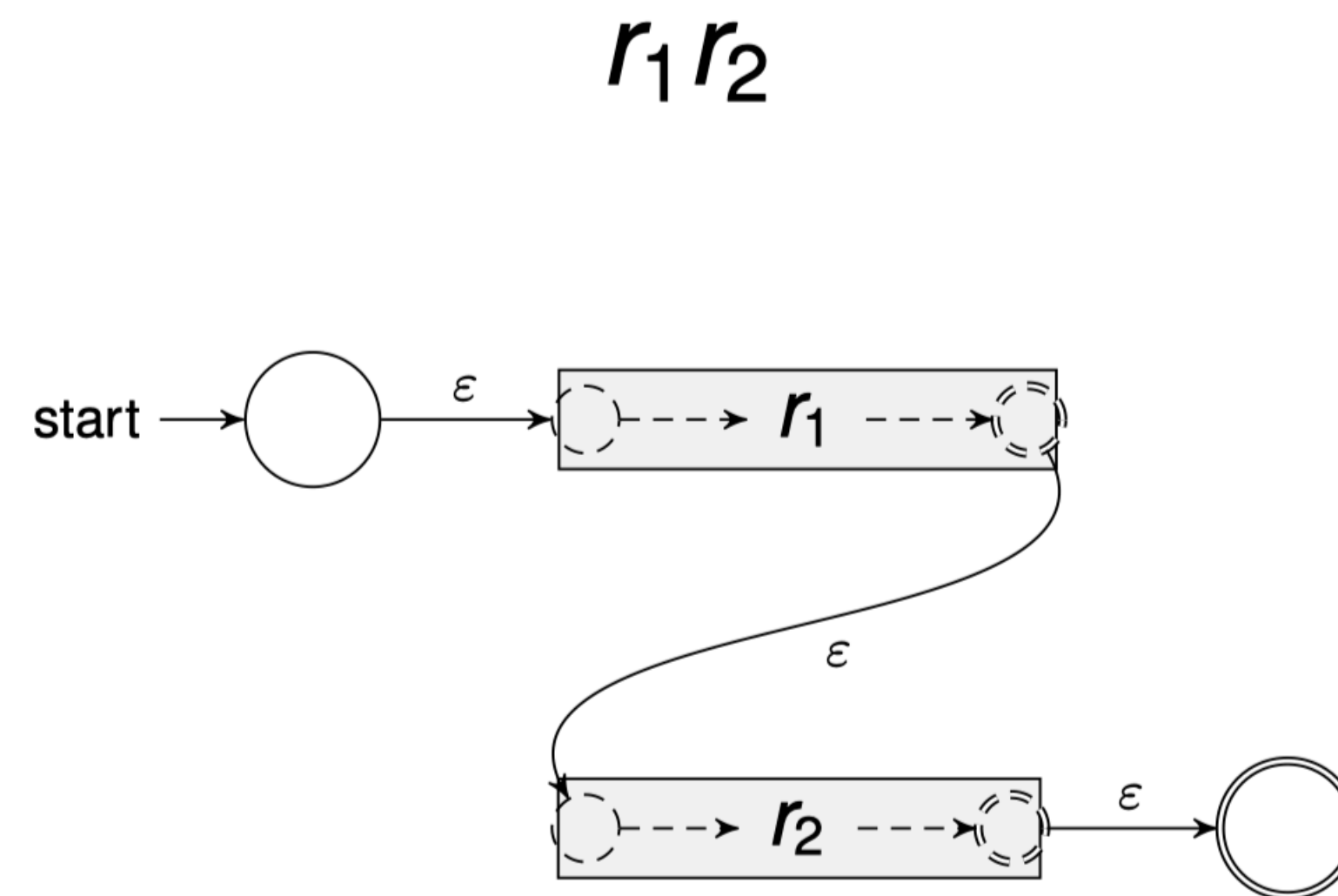
$$r_1 | r_2$$



How to prove RE \rightarrow NFA

Proof objective: Each regular expression can be represented by an NFA

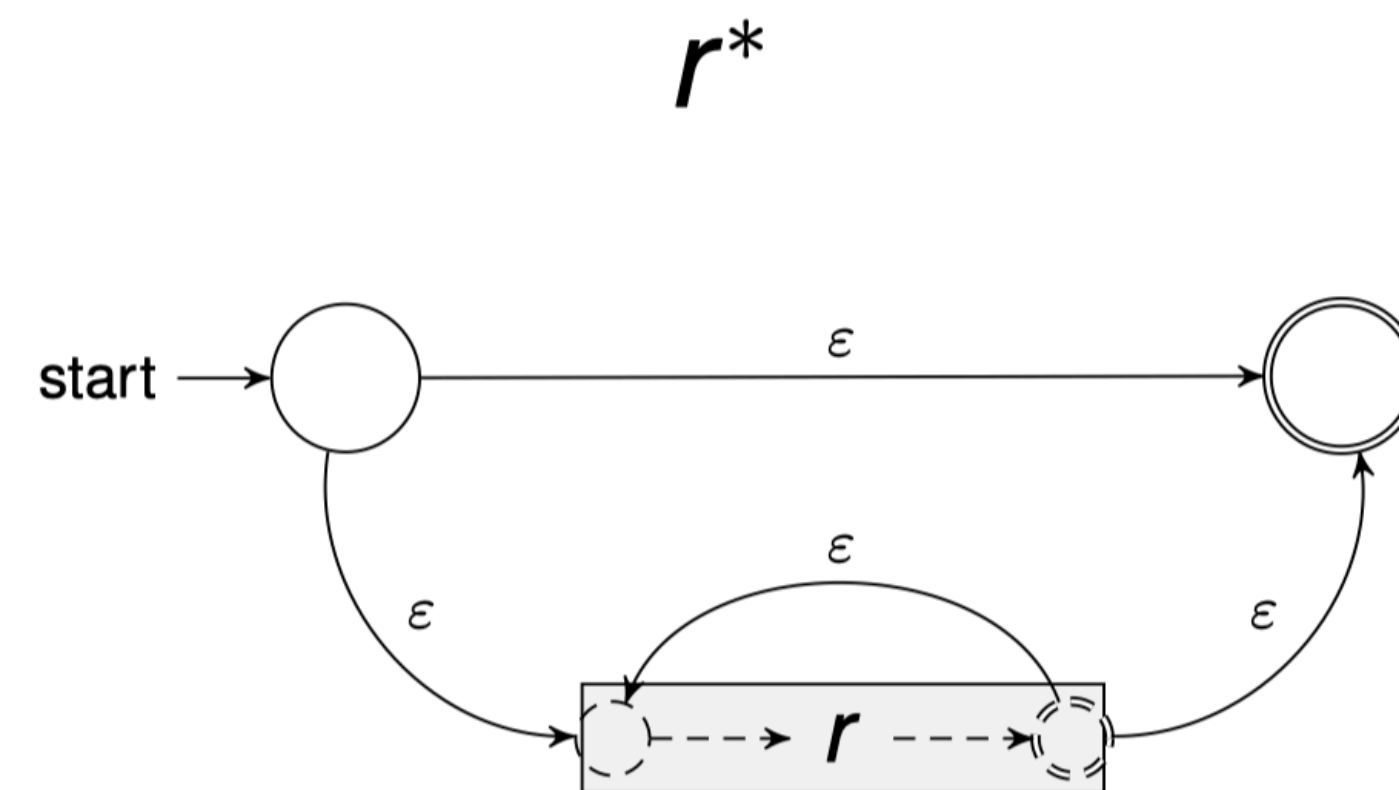
Proof strategy: Build NFA recursively by the case of the regular expression.



How to prove RE \rightarrow NFA

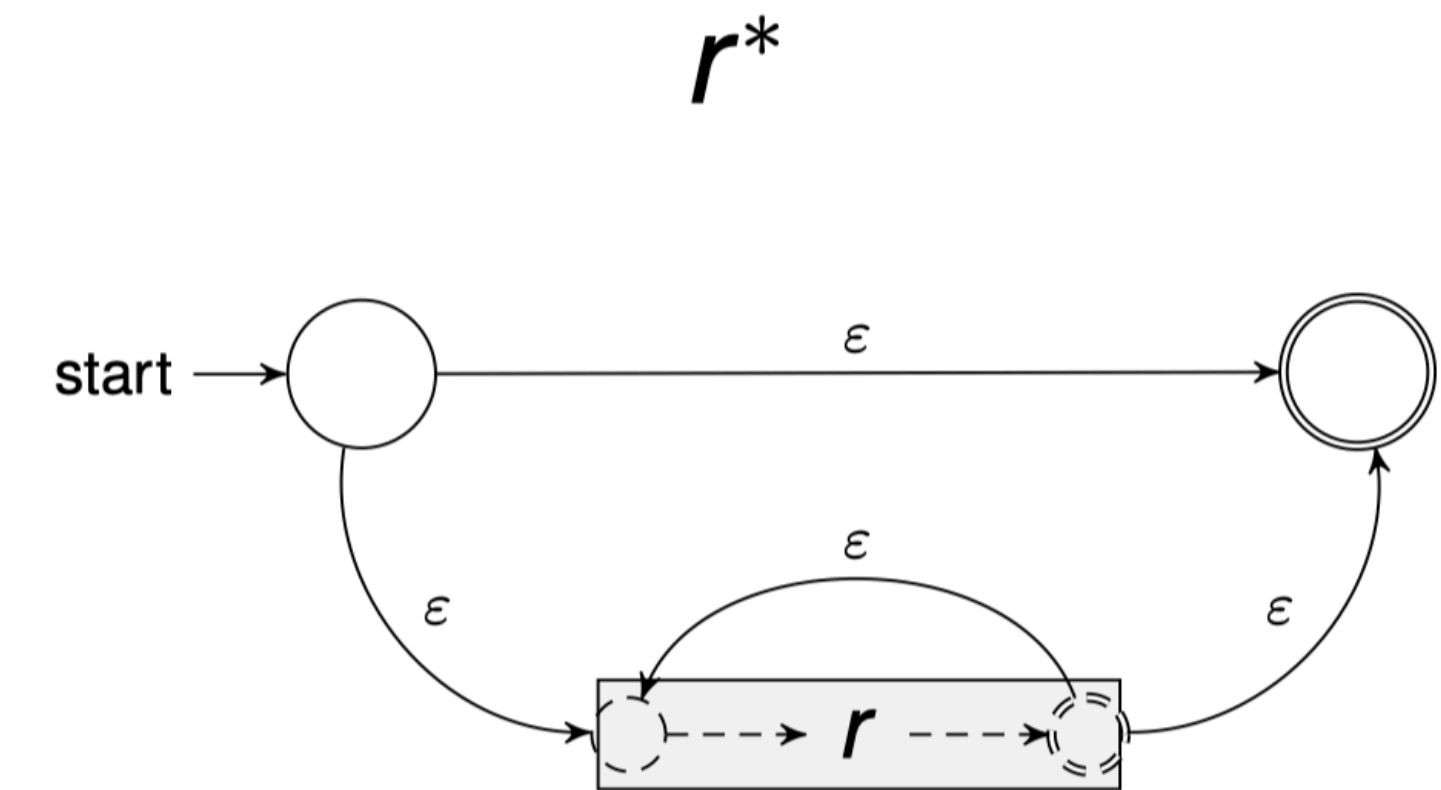
Proof objective: Each regular expression can be represented by an NFA

Proof strategy: Build NFA recursively by the case of the regular expression.



Quiz

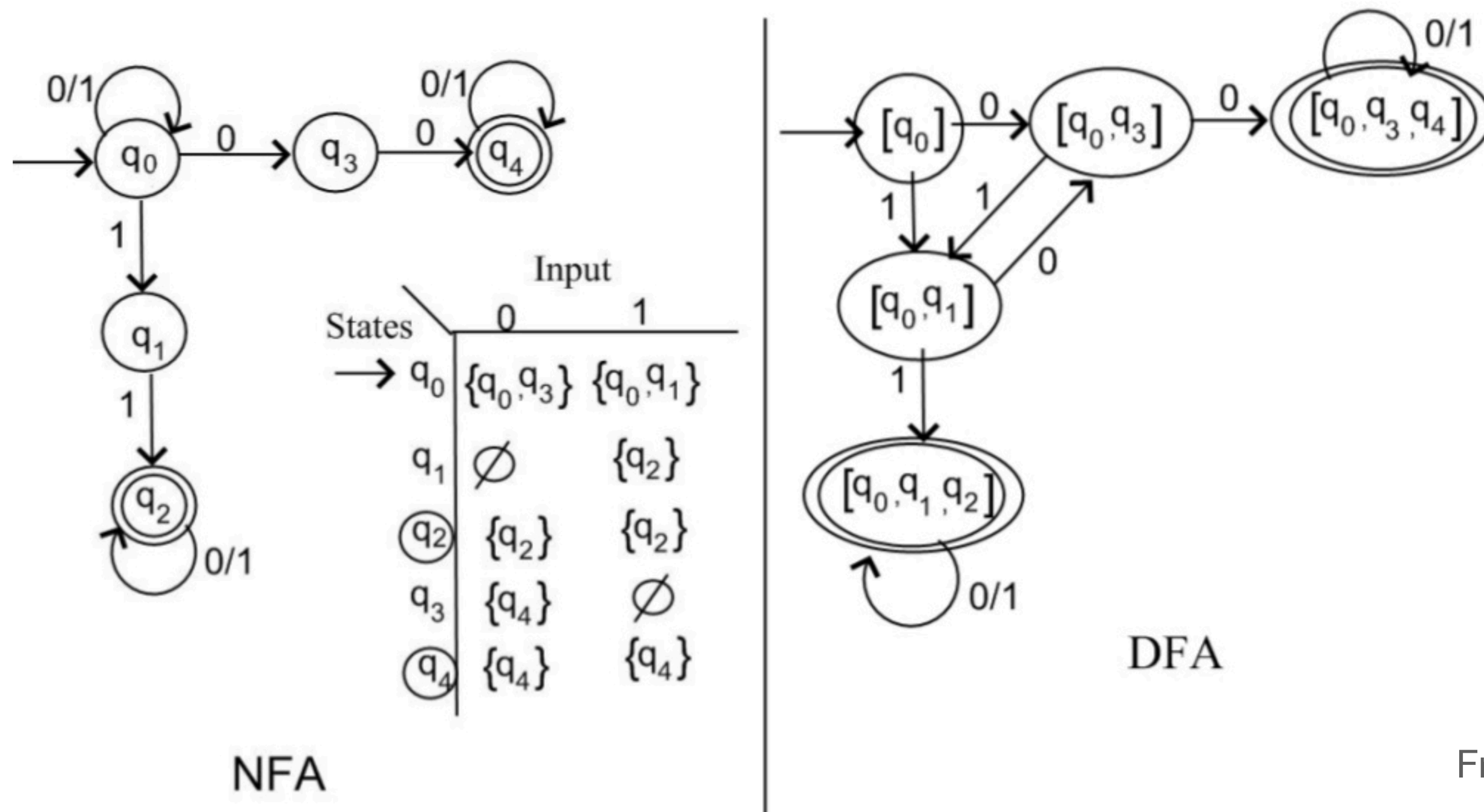
- *Make NFA for $(ab)^*$*
- *Make NFA for $(a|b)^*$*



How to prove NFA \rightarrow DFA

Proof objective: Each NFA can be represented by an DFA

Proof idea: each state of the constructed DFA corresponds to a set of NFA states. Construct the DFA so that its transition function summarizes that of the NFA.



From <https://www.slideshare.net/rsivashankari/nfa-todfa-60168395>

Example: RE $(ab)^*$ \rightarrow NFA \rightarrow DFA

- Let Q denote the states of the constructed DFA
- Q is initialized as the starting state of the NFA
- For each state in Q , find the possible set of states for each input symbol on NFA (assuming epsilon-closure). Add the set of states into Q .
- Final states will be all states containing the final states of of NFA (epsilon-closure is assumed)

Summary: How FsLex works behind the scene

- For every regular expression r , there exists a deterministic finite automaton that recognizes precisely the strings described by r .

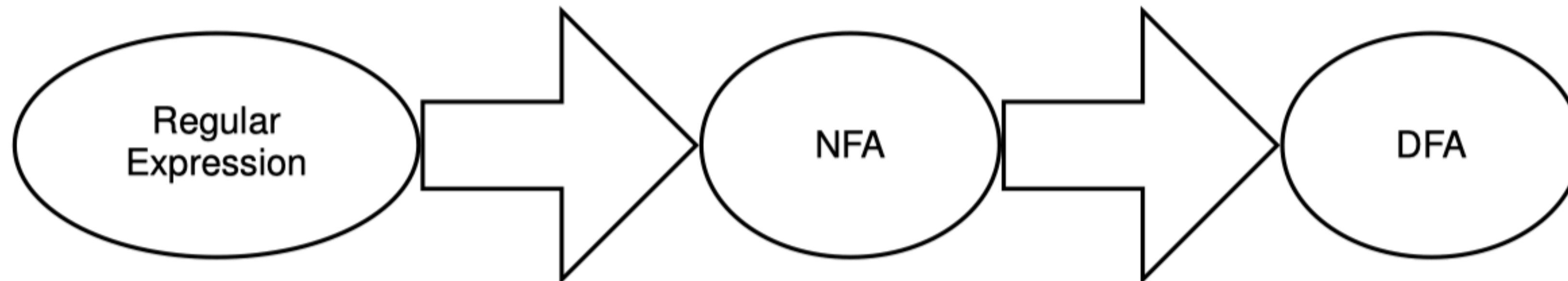
(The converse is also true.)

- Construction:

Regular expression

\Rightarrow *Nondeterministic finite automaton (NFA)*

\Rightarrow *Deterministic finite automaton (DFA)*



- Results in an efficient way of determining whether a given string is described by a regular expression

Conclusions

- Parsing is about understanding the structure
- Lexical analysis gets tokens from a buffer
- A lexer can be generated by FsLex, via specifying semantic values of regular expressions
- FsLex usually makes this conversion behind the scene: Regular expressions
-> NFA -> DFA
- Kleene's theorem

Thank you!