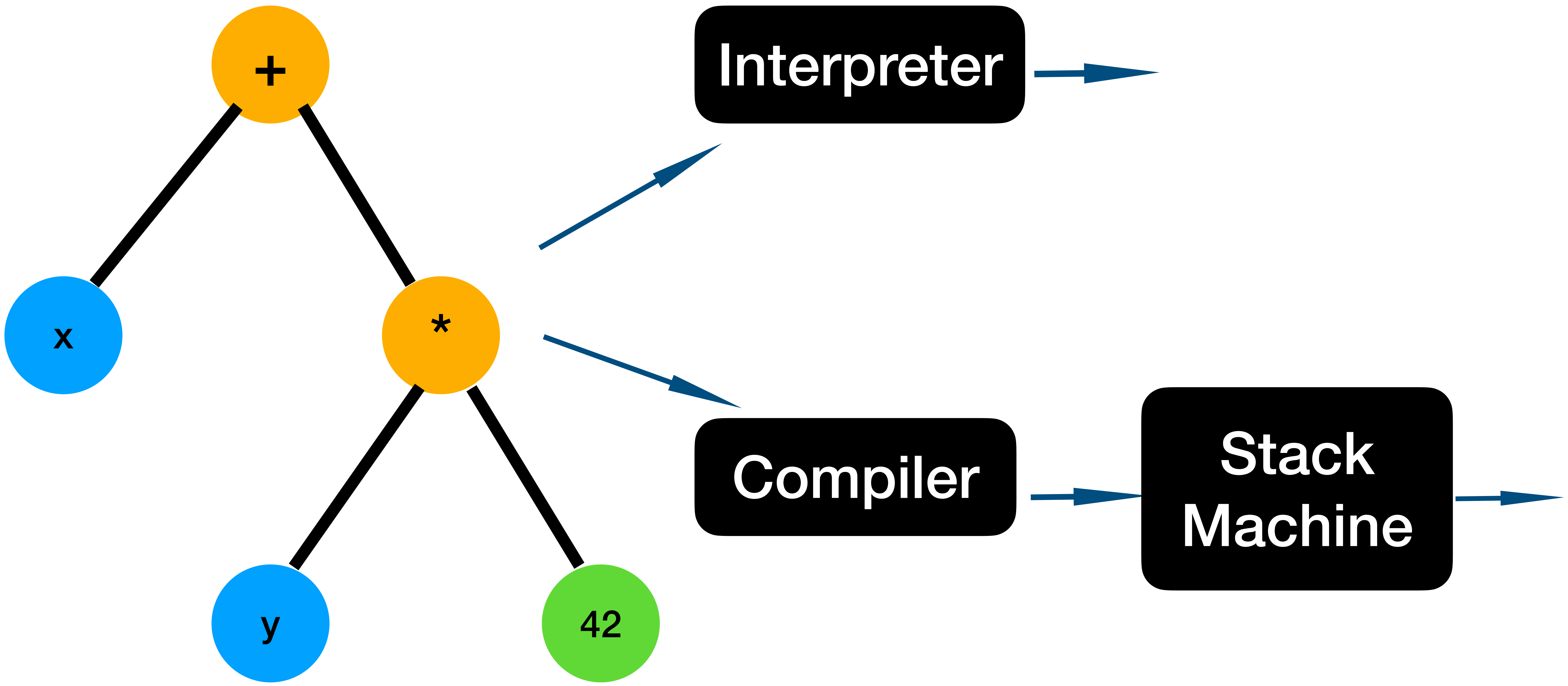
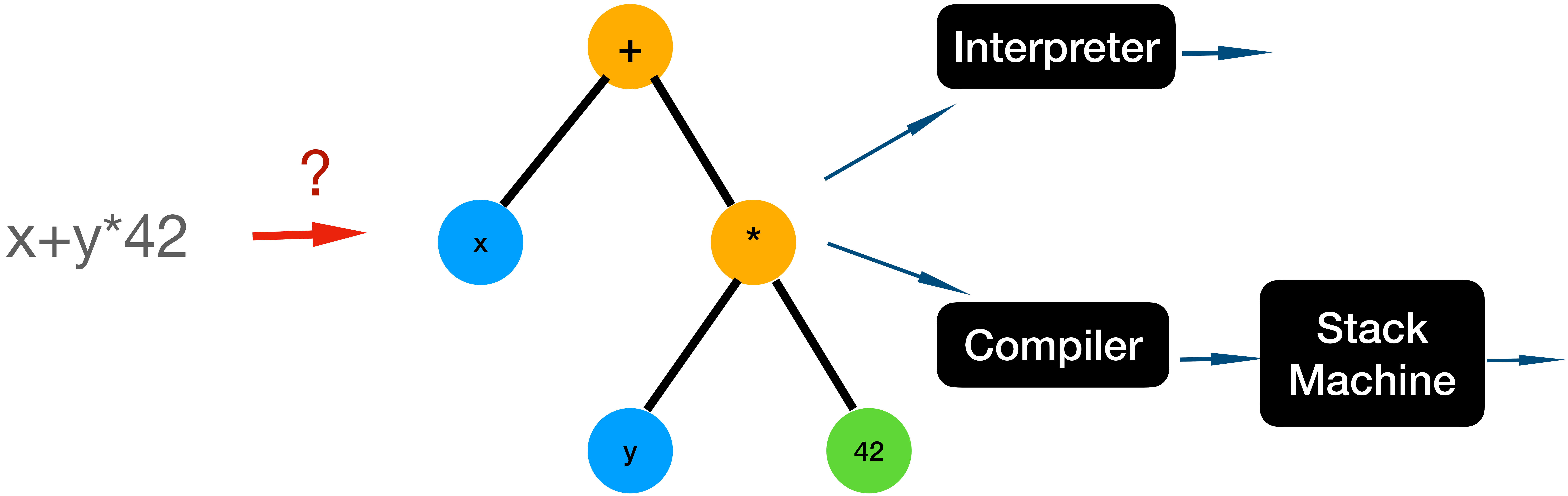
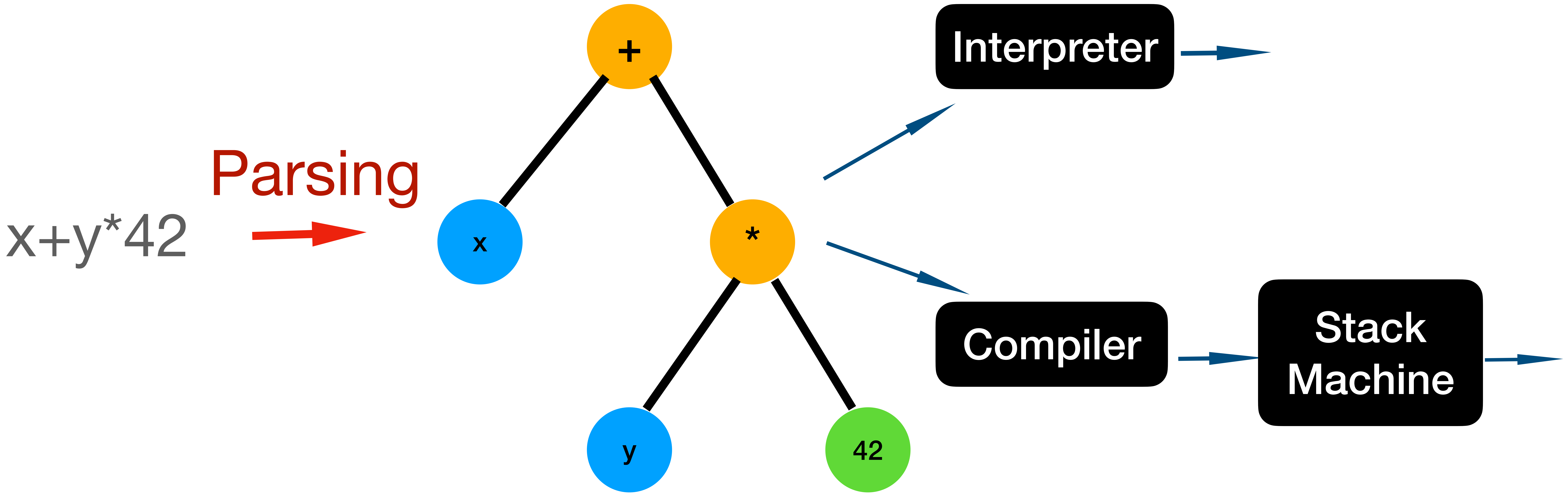
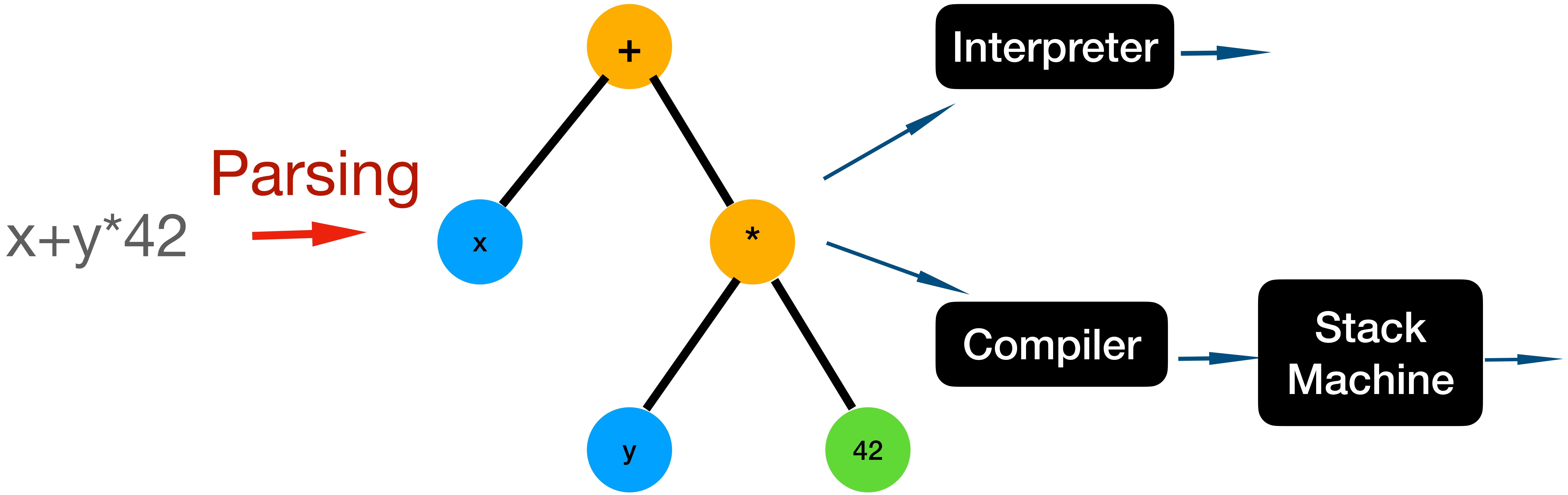
Previous lecture




x+y*42
?
+
x
*
y
42
Interpreter
Compiler
Stack Machine


x+y*42
Parsing
+
x
*
y
42
Interpreter
Compiler
Stack
Machine



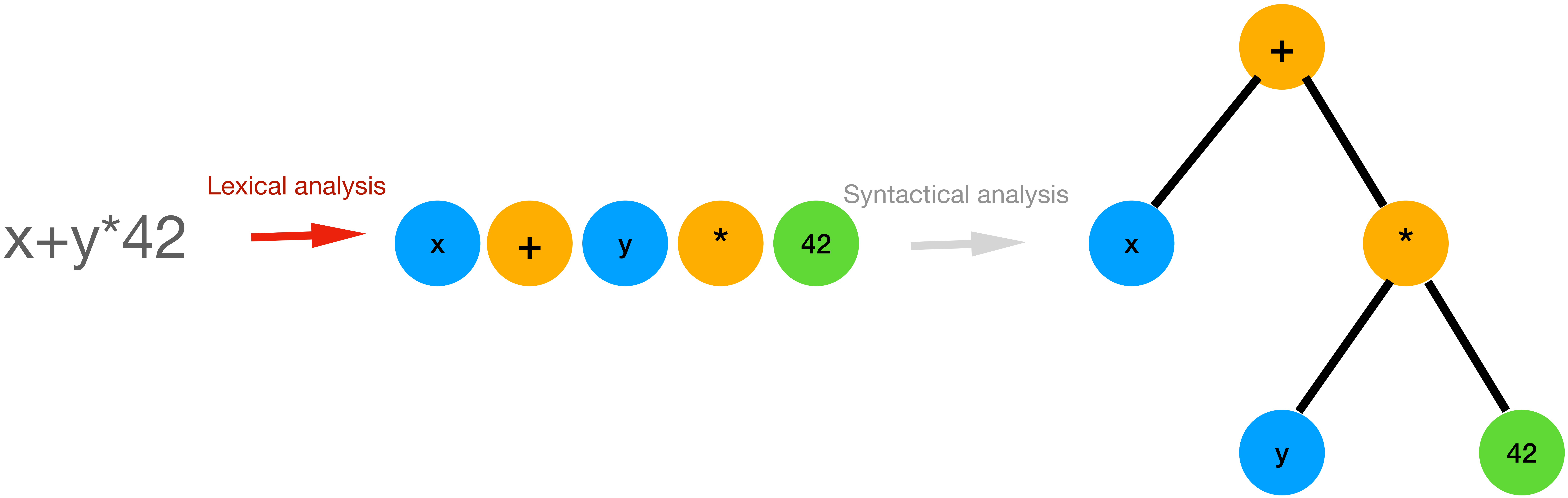
Parsing is about getting the structure

The structure matters

What is like parsing 0.1 + 0.2 + 0.3?

Demo

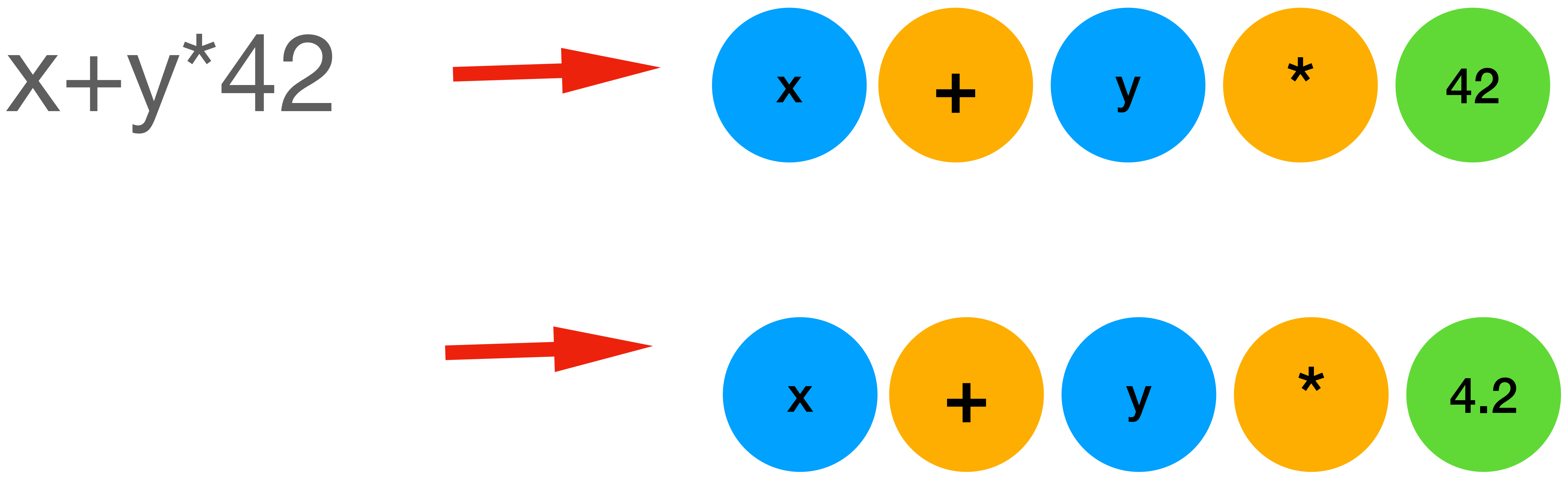
How to parse: A tale of two analyses



Parsing (1/3): Lexical Analysis

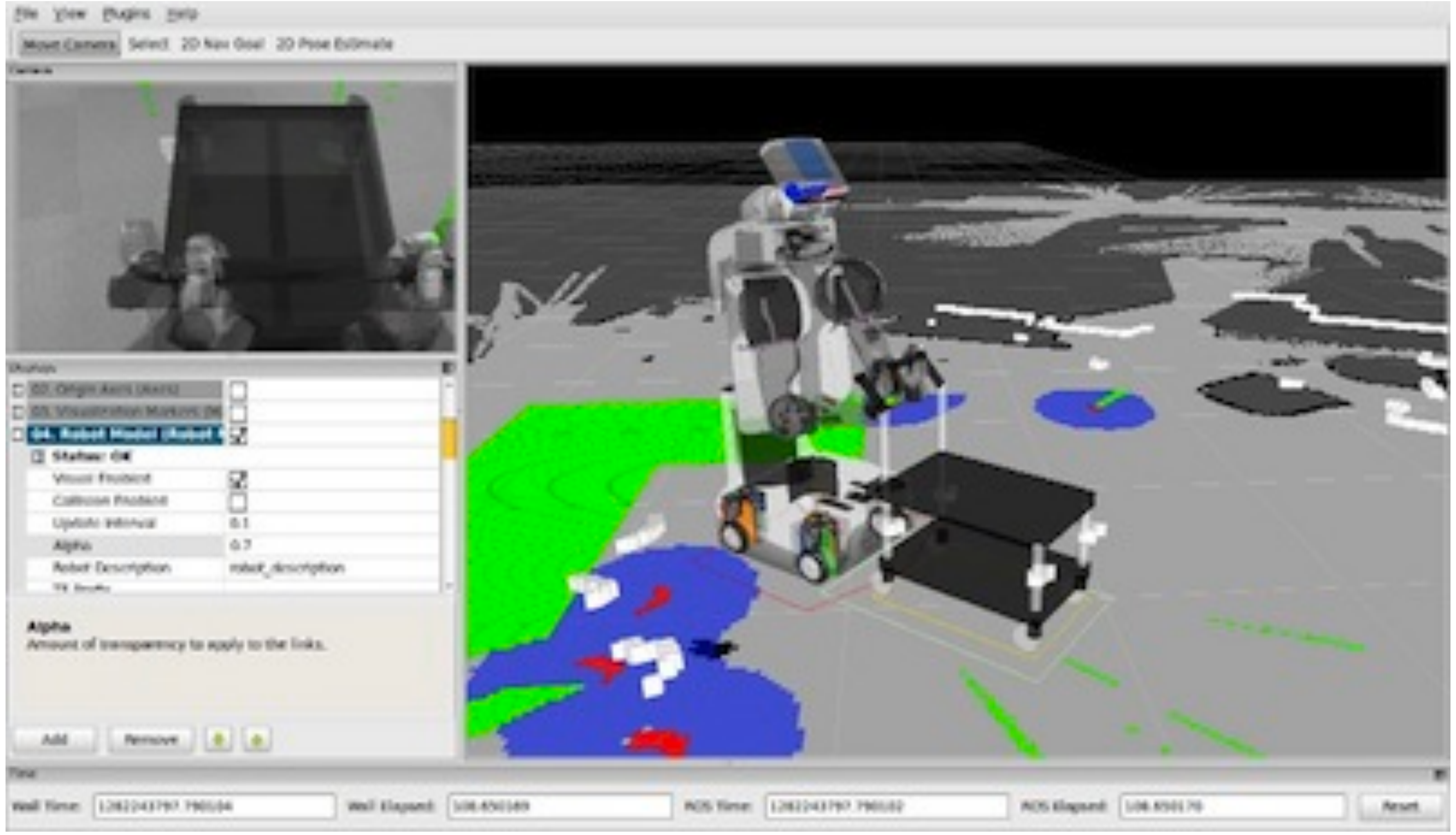
Program as Data, Autumn 2020

Lexical analysis: LexBuffer to tokens, or manipulate the tokens



A research problem I worked on at ITU

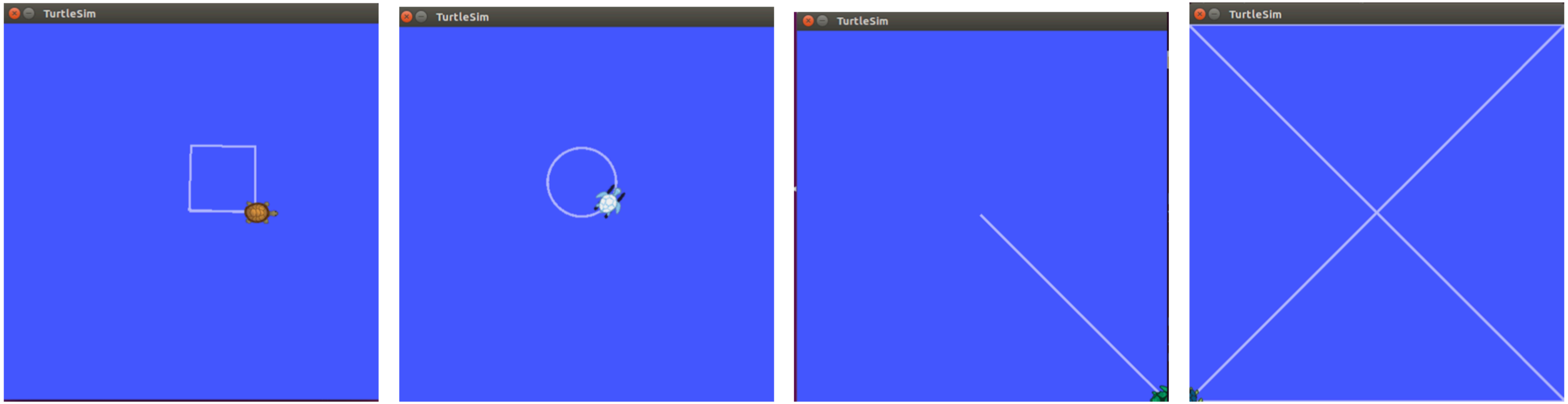
- Robot Operating System
- Automated Testing



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 $\mathcal{L}(r)$
$\texttt{a}$	Character $\texttt{a}$	$\{"\texttt{a}"\}$
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	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$
$r_1 \vert r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

Examples

$\texttt{ab}^*$ represents $\{"\texttt{a}", "\texttt{ab}", "\texttt{abb}", \ldots\}$

$(\texttt{ab})^*$ represents $\{"", "\texttt{ab}", "\texttt{abab}", \ldots\}$

$(\texttt{a}|\texttt{b})^*$ represents $\{"", "\texttt{a}", "\texttt{b}", "\texttt{aa}", "\texttt{ab}", "\texttt{ba}", \ldots\}$

Exercise

What does $(\texttt{a}|\texttt{b})\texttt{c}^*$ represent?

Common extensions of regular expressions

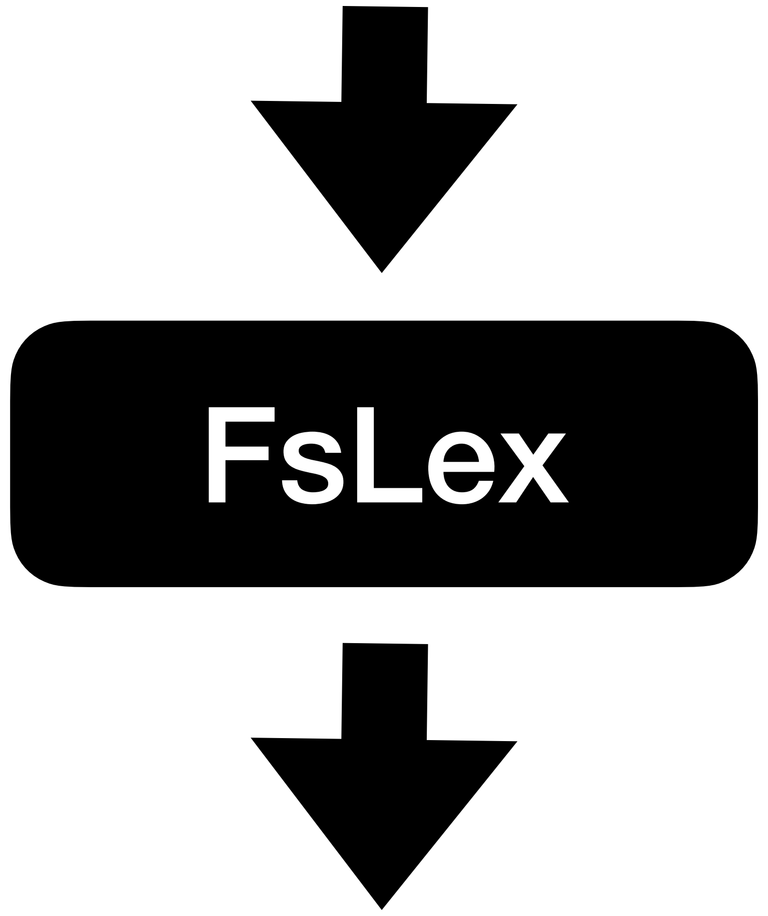
Abbrev.	Meaning	Expansion
`[aeiuo]`	Set	`a|e|i|o|u`
`[0-9]`	Range	`0|1|...|8|9`
`[0-9a-Z]`	Ranges	`0|1|...|8|9|a|b|...|y|z`
$r?$	Zero or one r	$r \mid \varepsilon$
r^+	One or more r	$r\,r^*$

- `[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
```

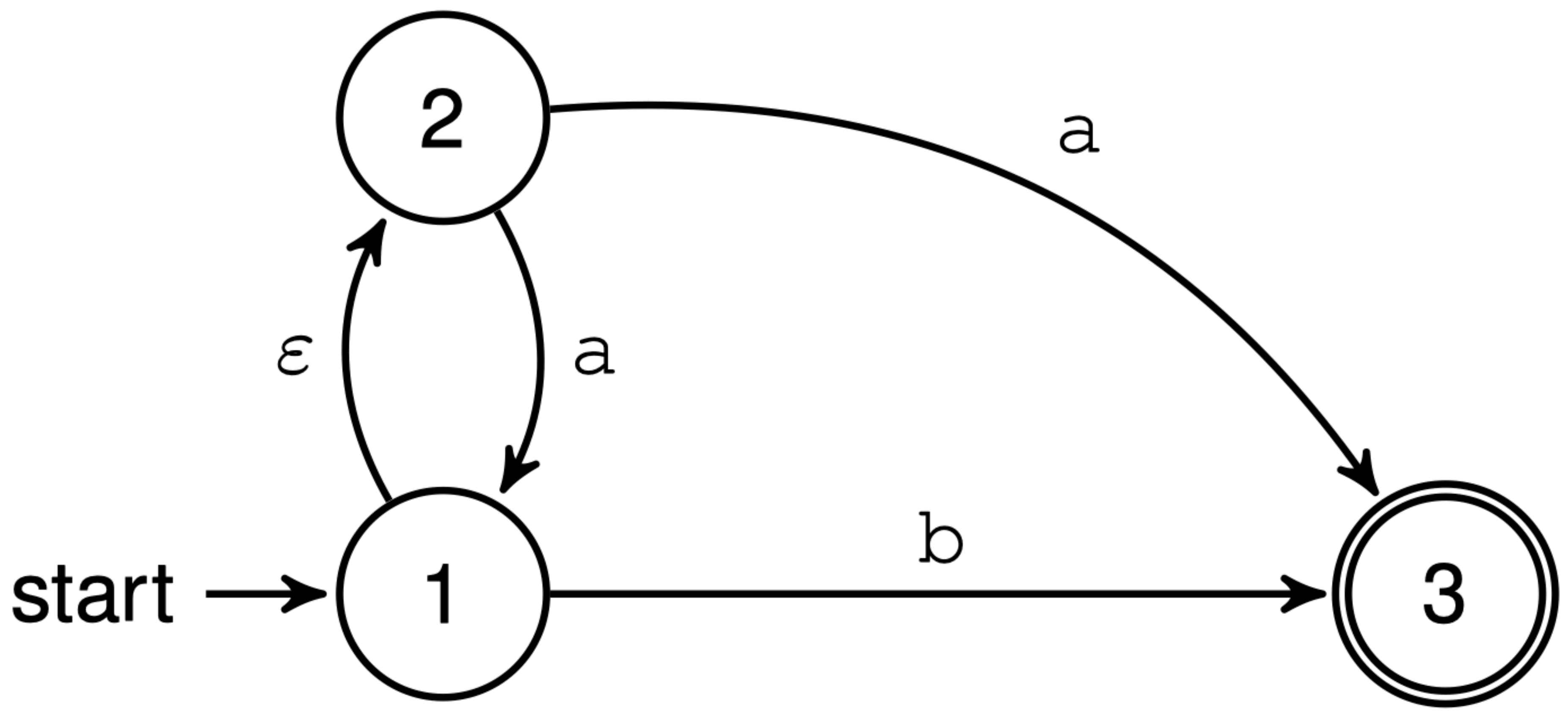
- The regular expression specified a single digit
- “LexBuffer.LexemeString lexbuf” is a utility of FsLex that transforms the recognized regular expression to a string
- In this way, Tokenize will be a function that maps a LexBuffer to a string

Demo

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

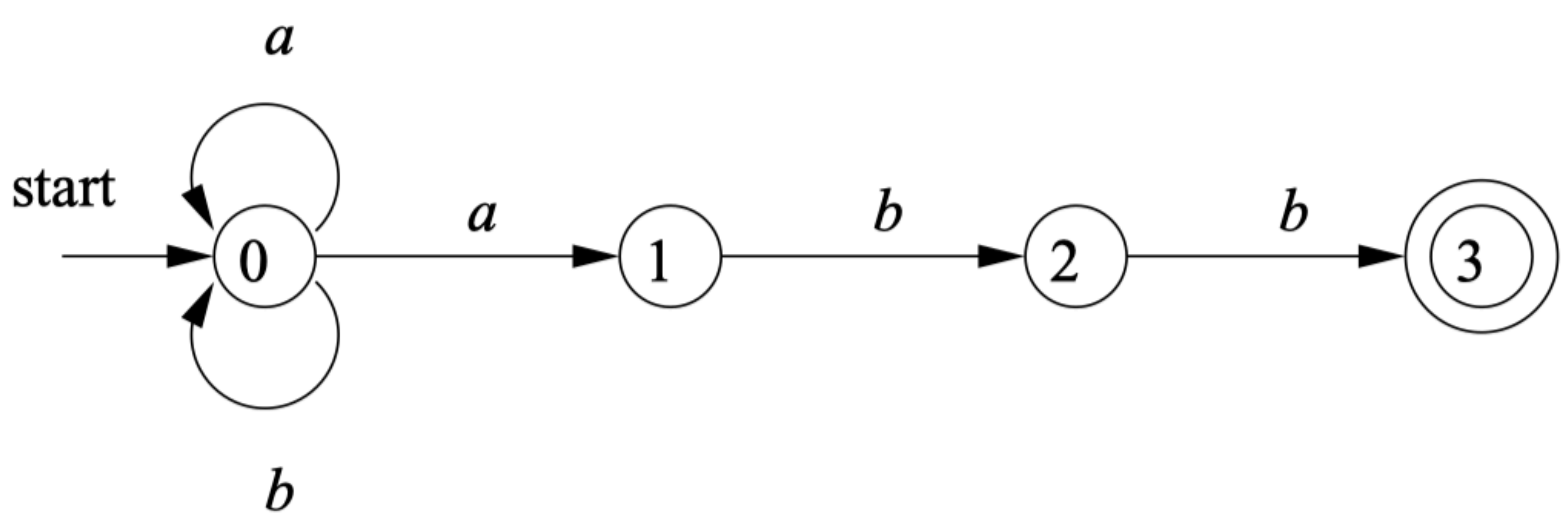
(a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ϵ, the empty string, is a possible label.

(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.

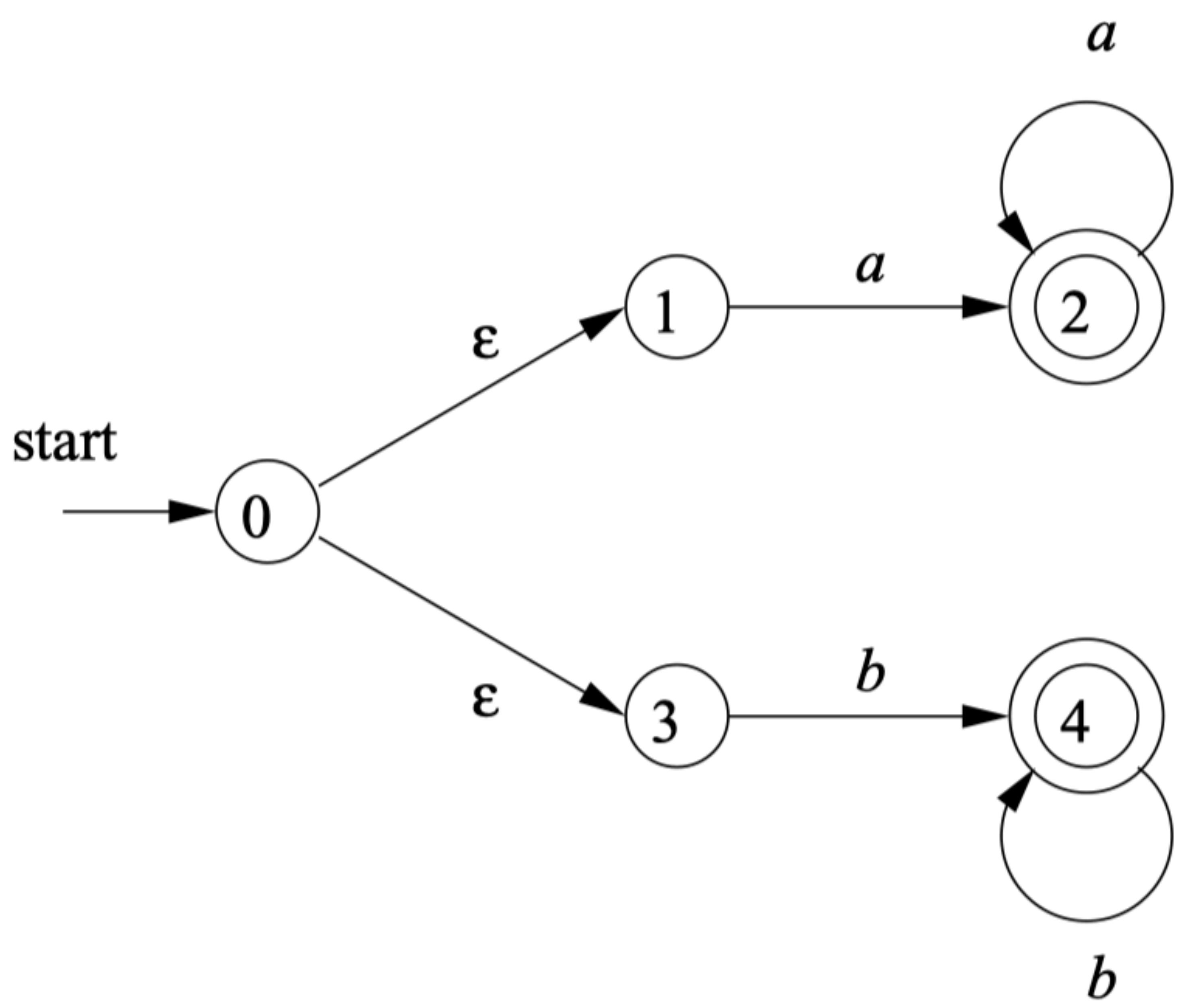
An NFA consists of:

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



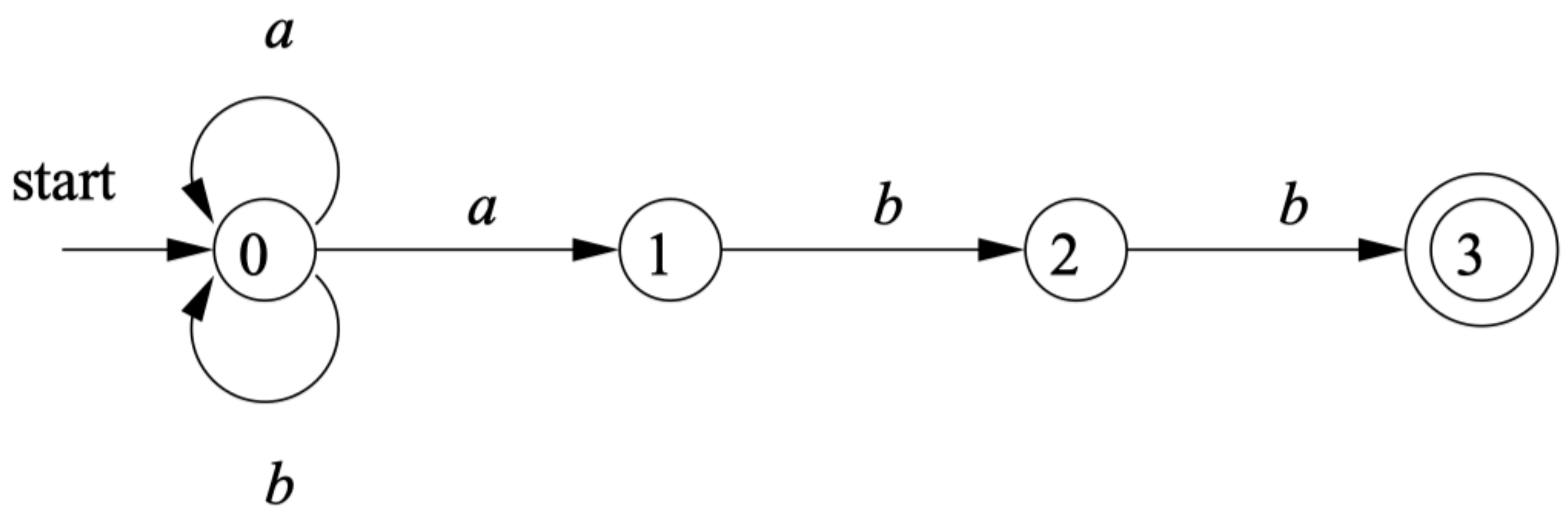
NFA accepting (a|b)*abb



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*.



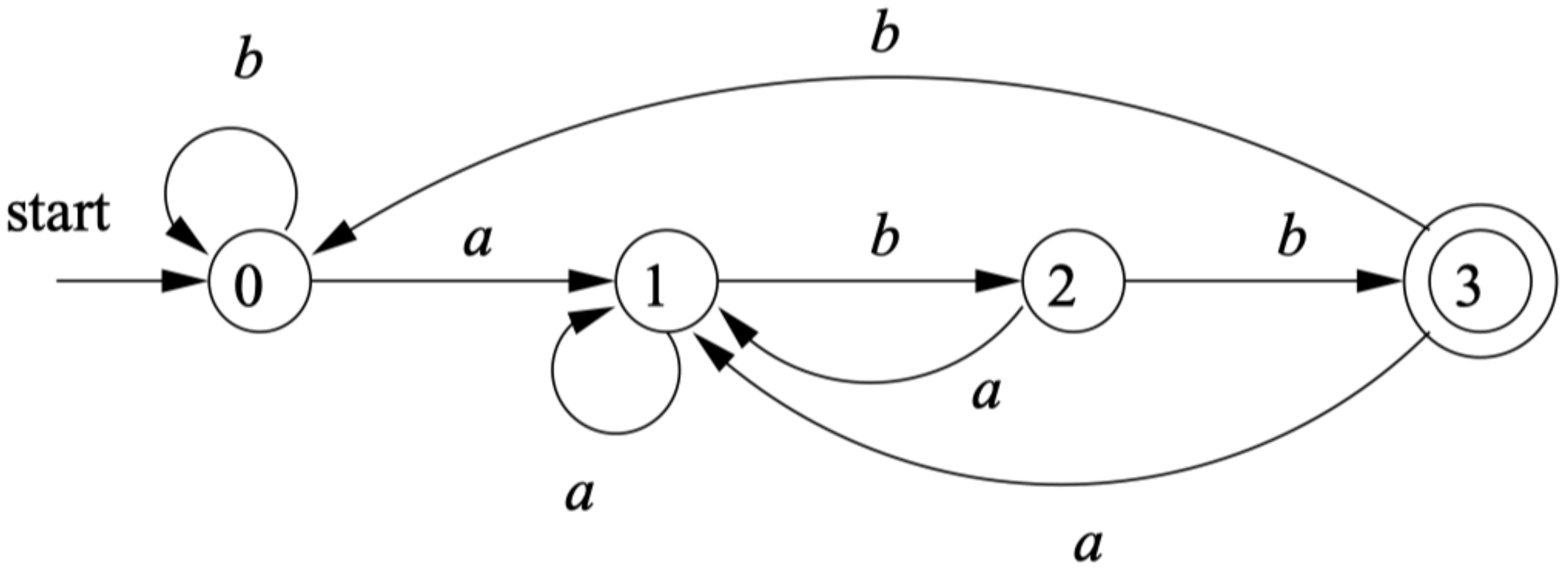
STATE	a	b	ϵ
0	$\{0, 1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

NFA accepting (a|b)*abb

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

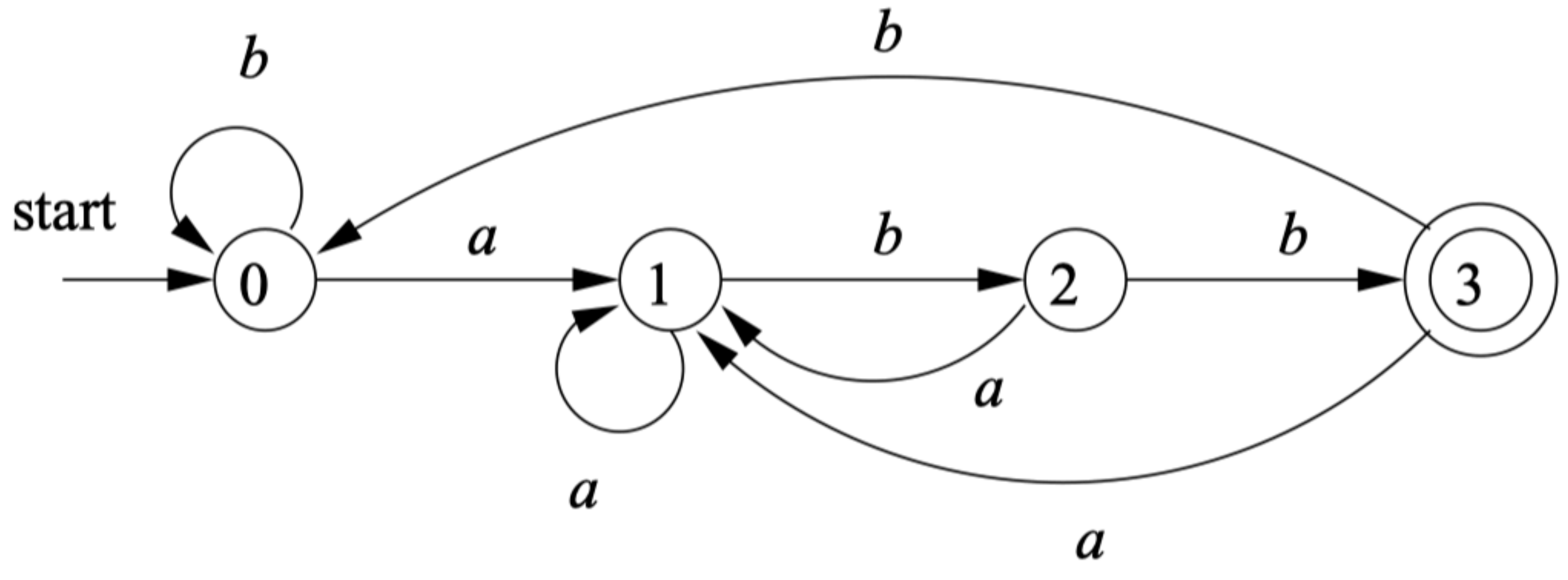
1. There are no moves on input ϵ, and
2. For each state s and input symbol a, there is exactly one edge out of s labeled a.

An example of DFA



DFA accepting (a|b)*abb

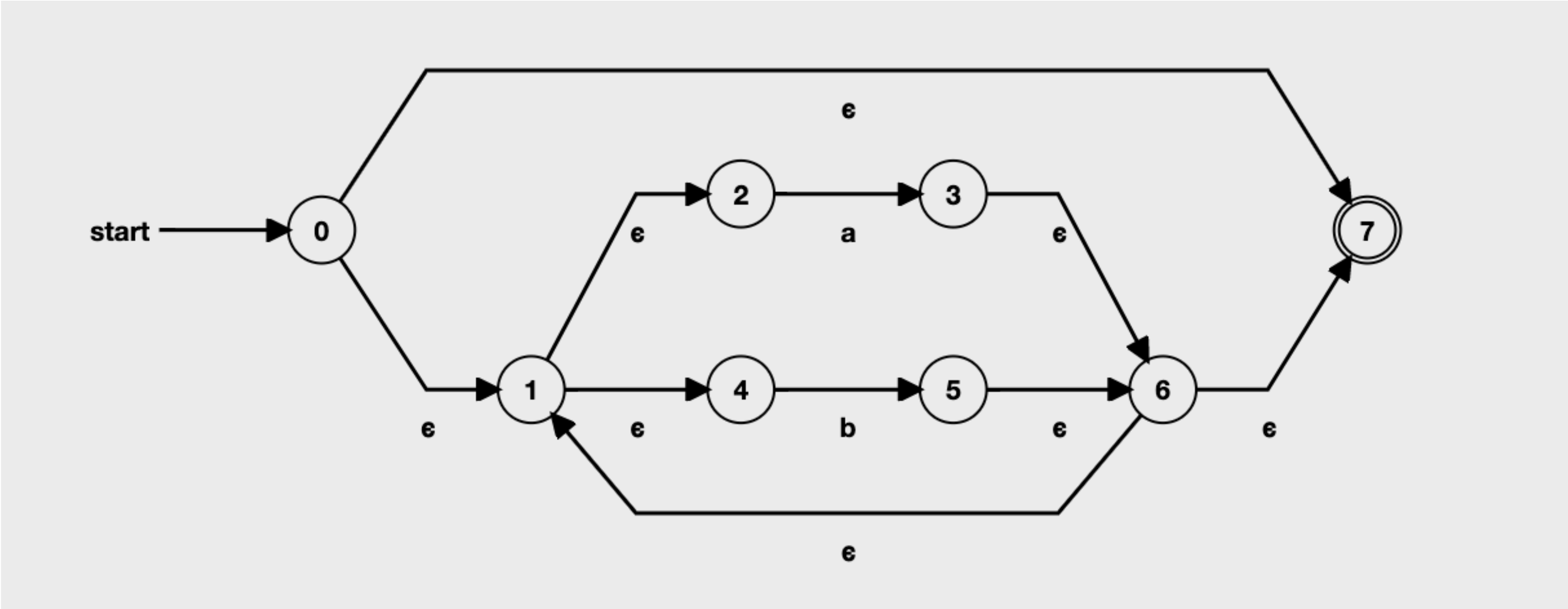
Transition function of a DFA

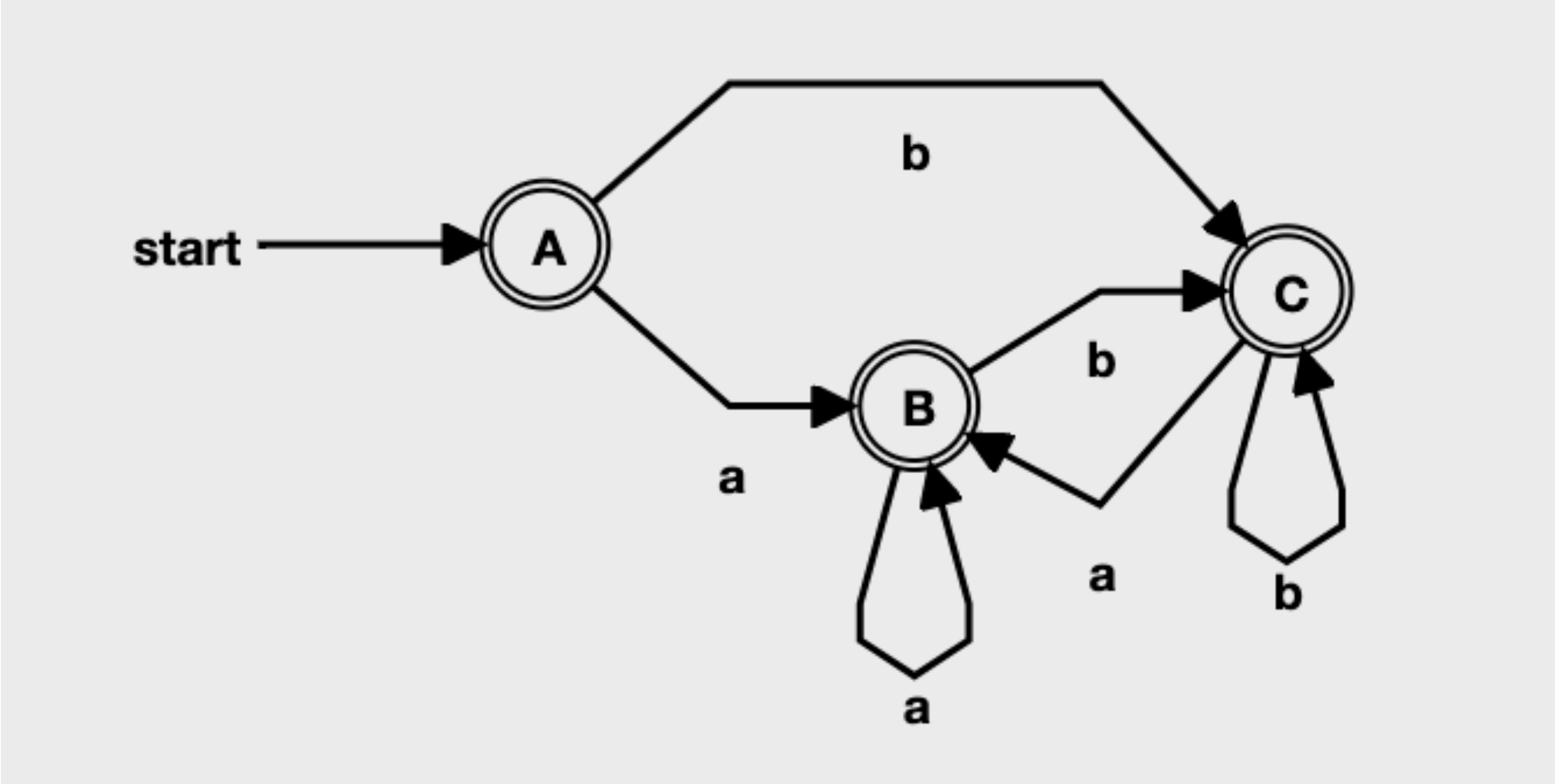


DFA accepting (a|b)*abb

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

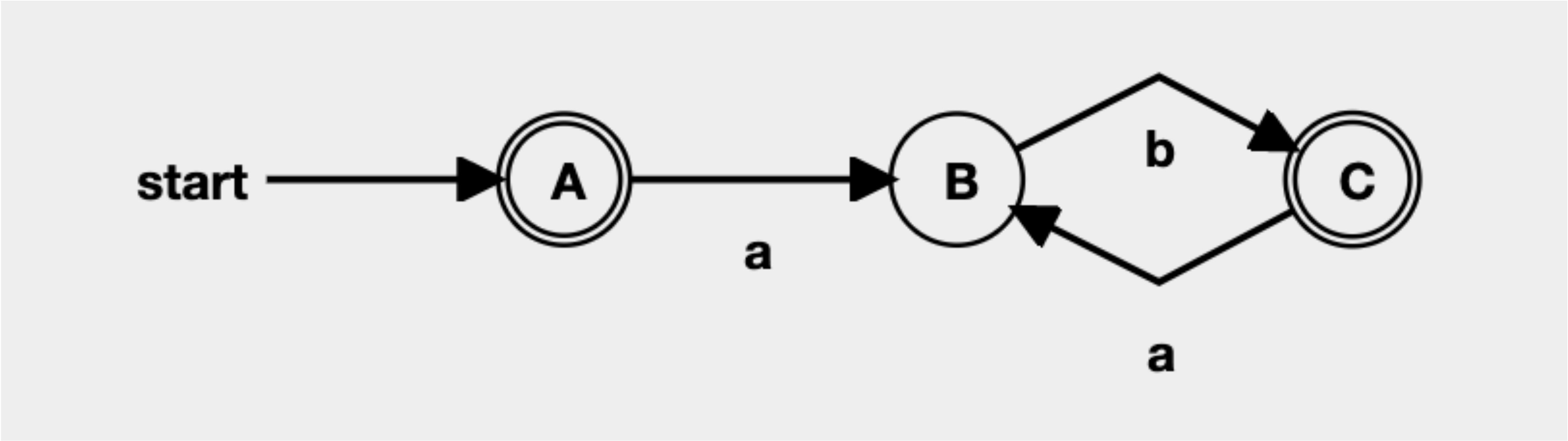
Quiz: NFA or DFA

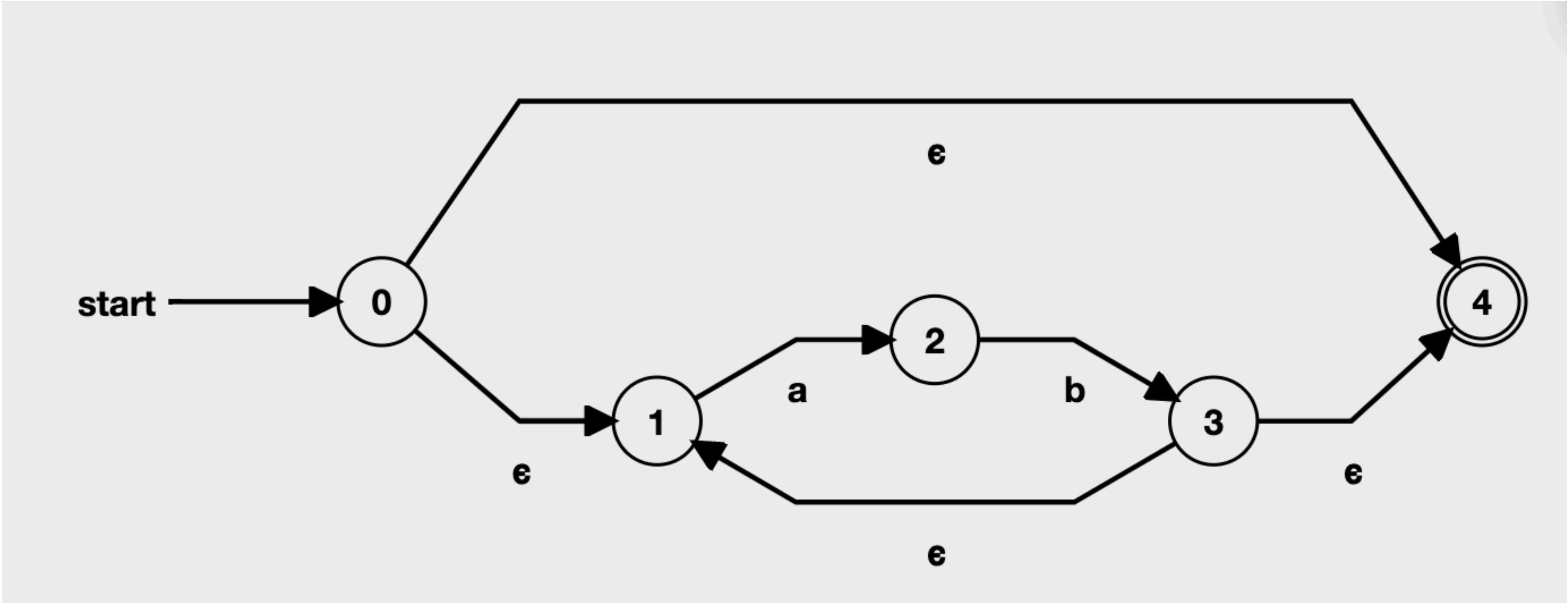




Both recognize (a|b)*

Quiz: NFA or DFA

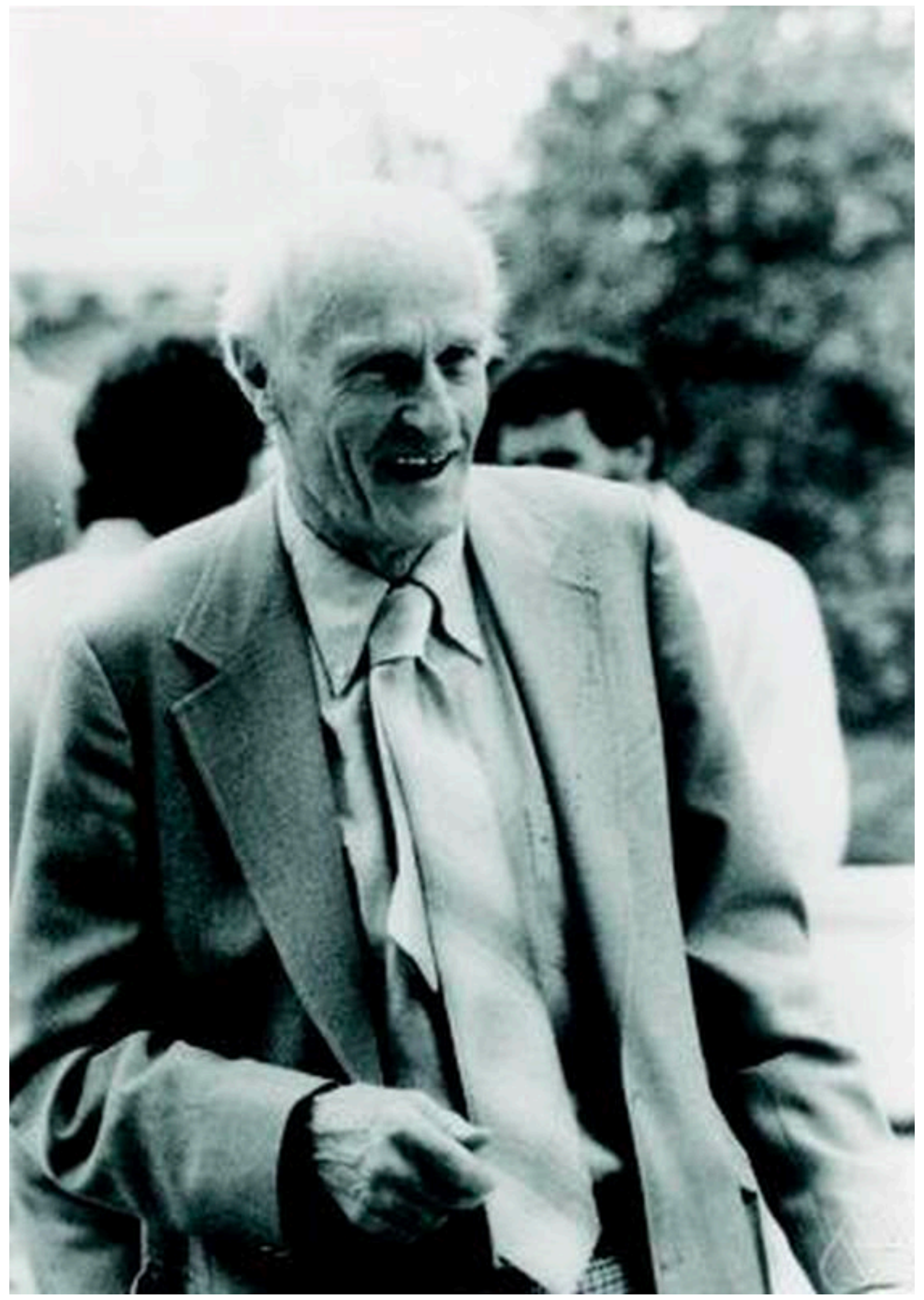




Both recognize (ab)*

Kleene’s theorem

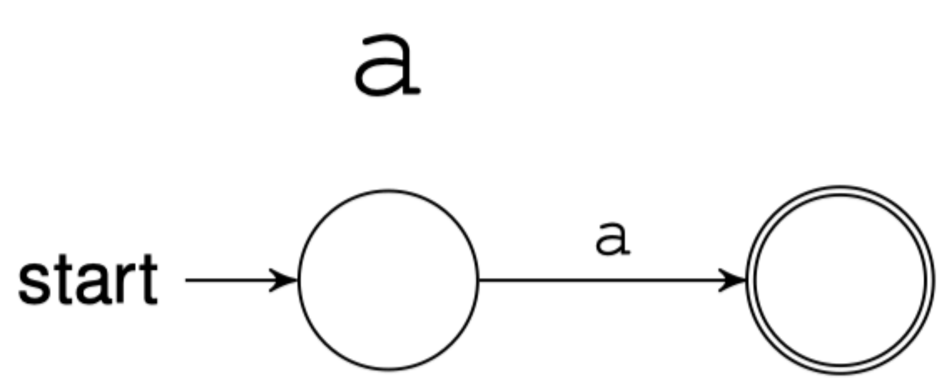
- Theorem: The set of regular languages, the set of NFA-recognizable languages, and the set of DFA-recognizable languages are all the same.
- Proof: RE -> NFA -> DFA -> RE

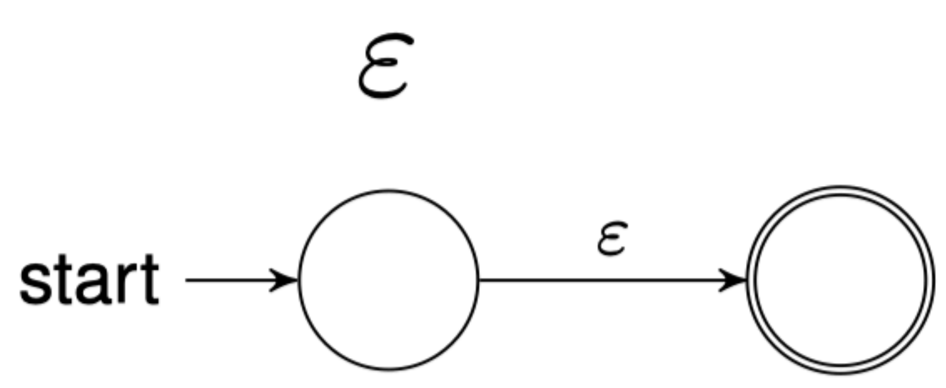


How to prove RE -> NFA

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

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



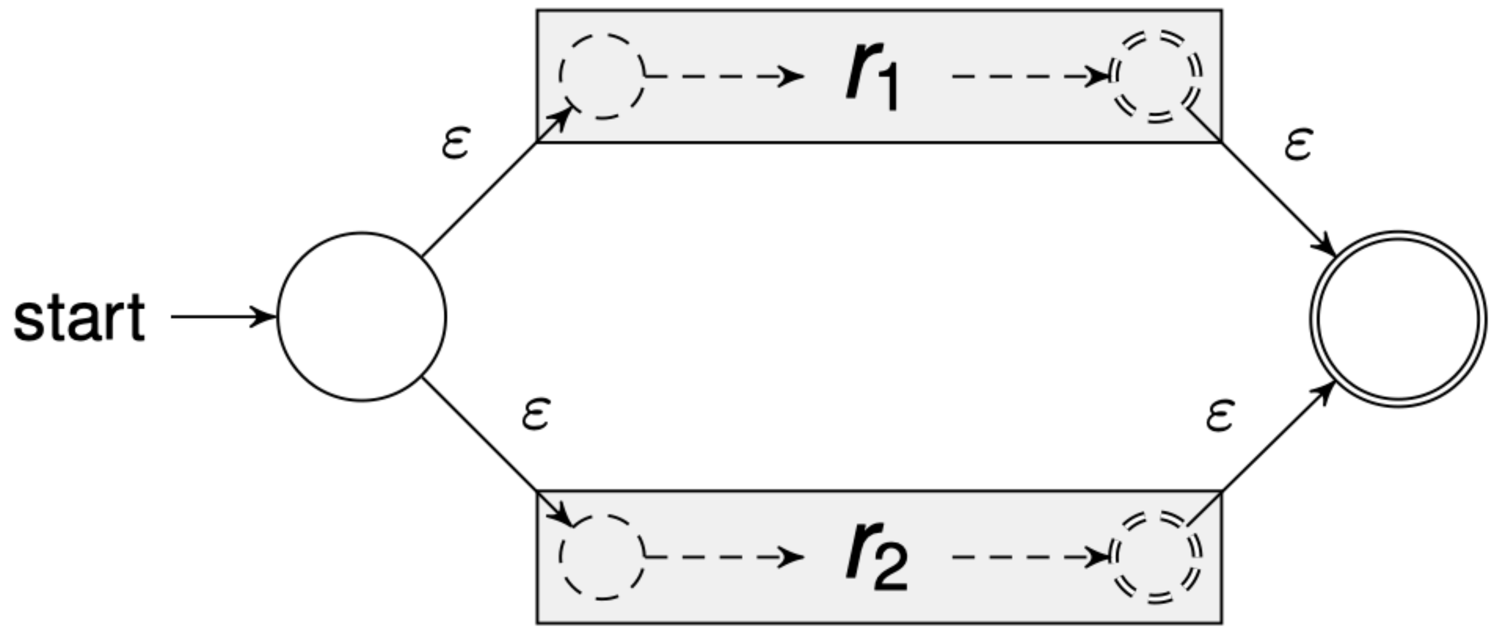


How to prove RE -> NFA

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

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

$$r_1 | r_2$$

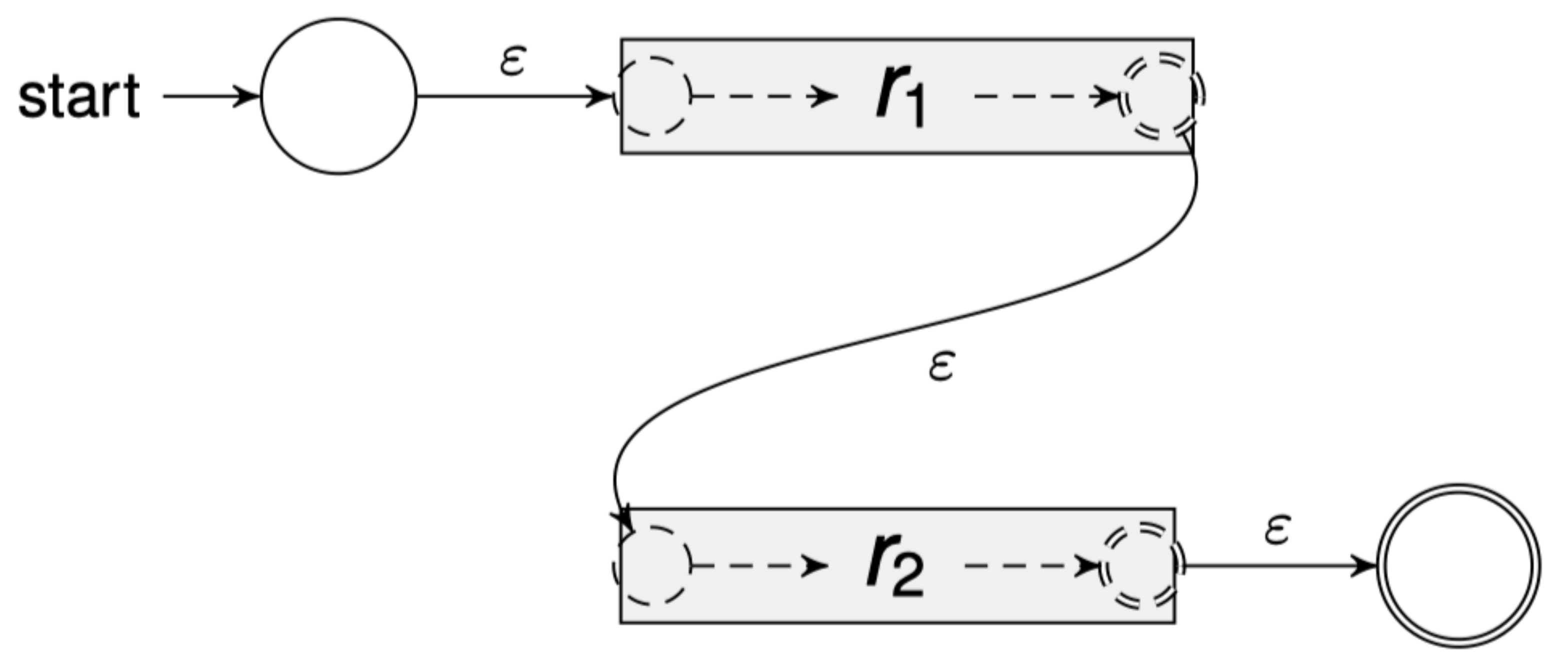


How to prove RE -> NFA

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

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

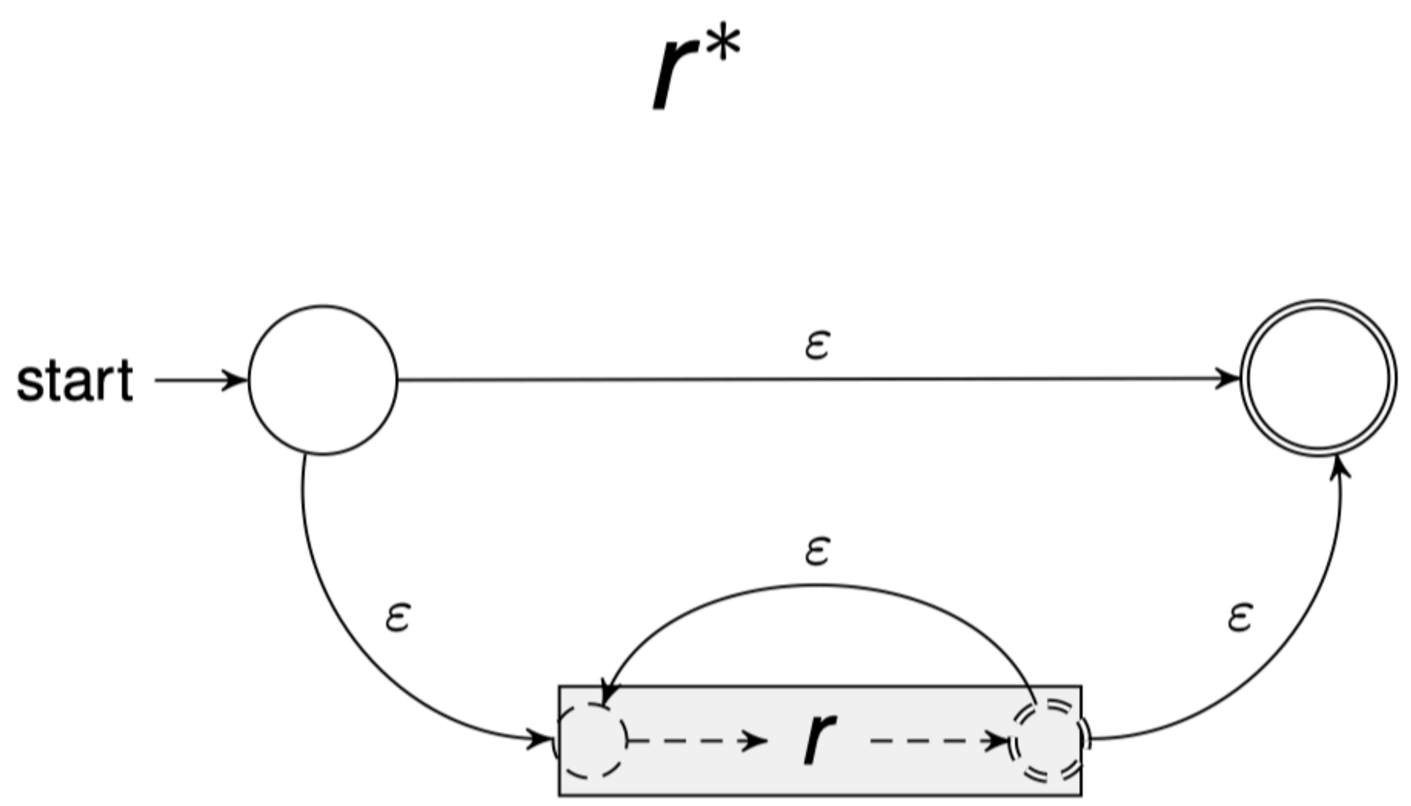
$r_1 r_2$



How to prove RE -> NFA

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

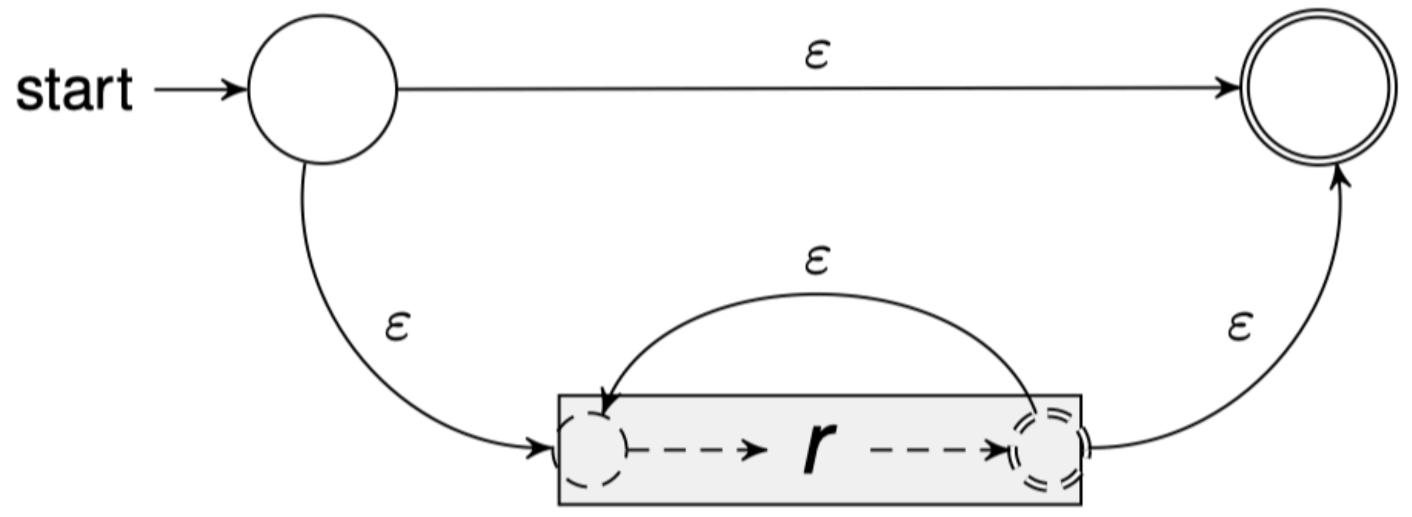
Proof strategy: Build NFA recursively by the case of the regular expression.



Quiz

- *Make NFA for* `(ab)*`
- *Make NFA for* `(a|b)*`

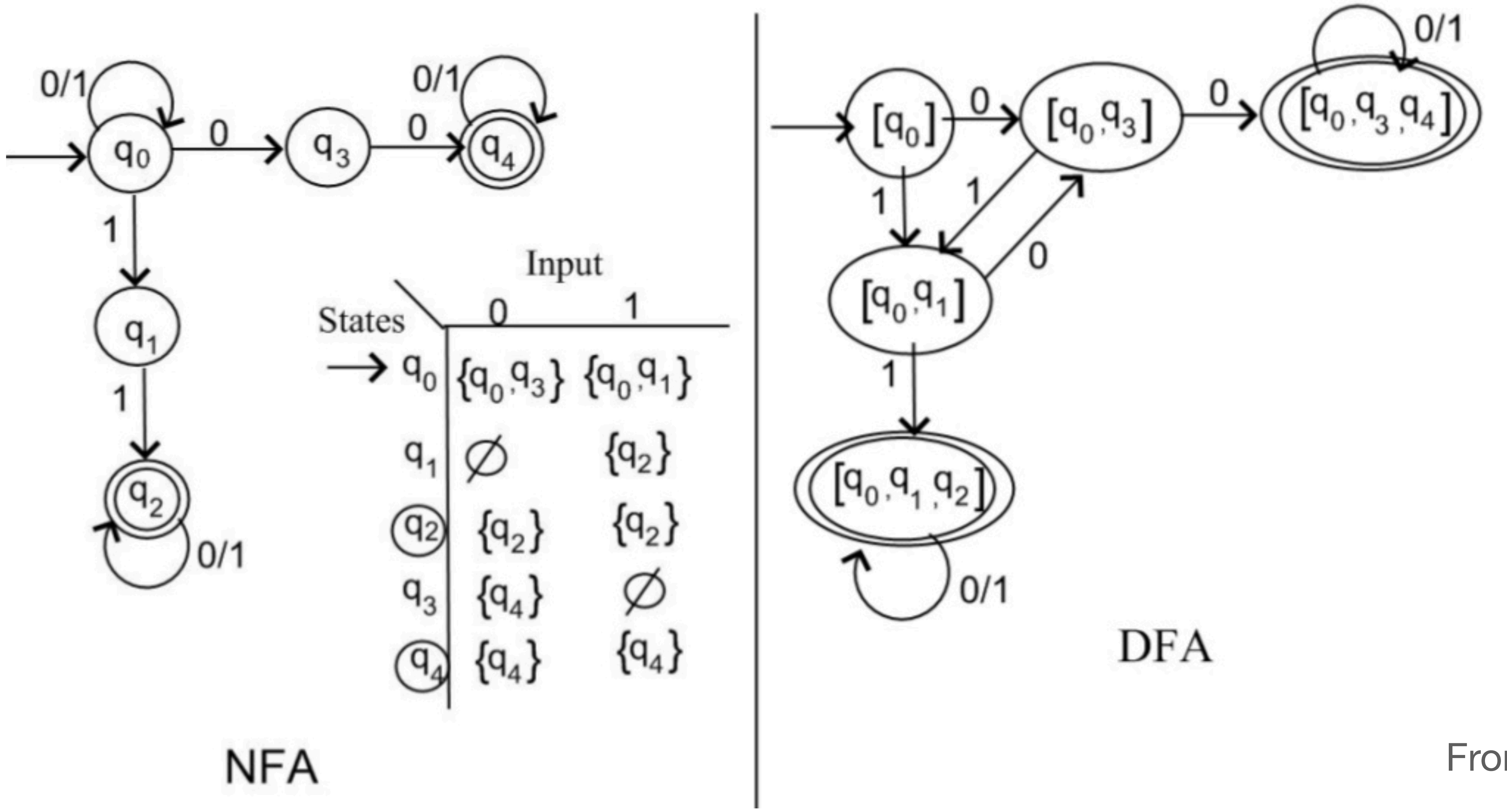
*r**



How to prove NFA -> 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)* -> NFA -> 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)

Blackboard D

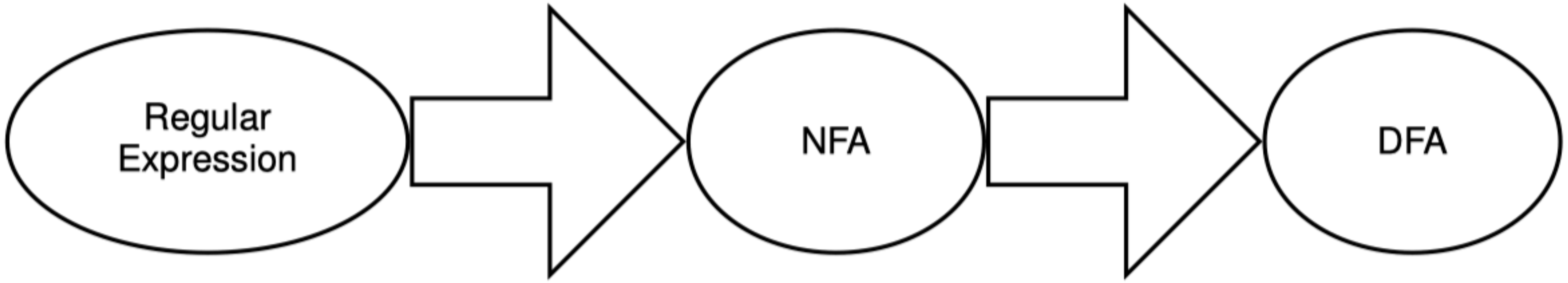
Summary: How FsLex works behind the scene

- 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!