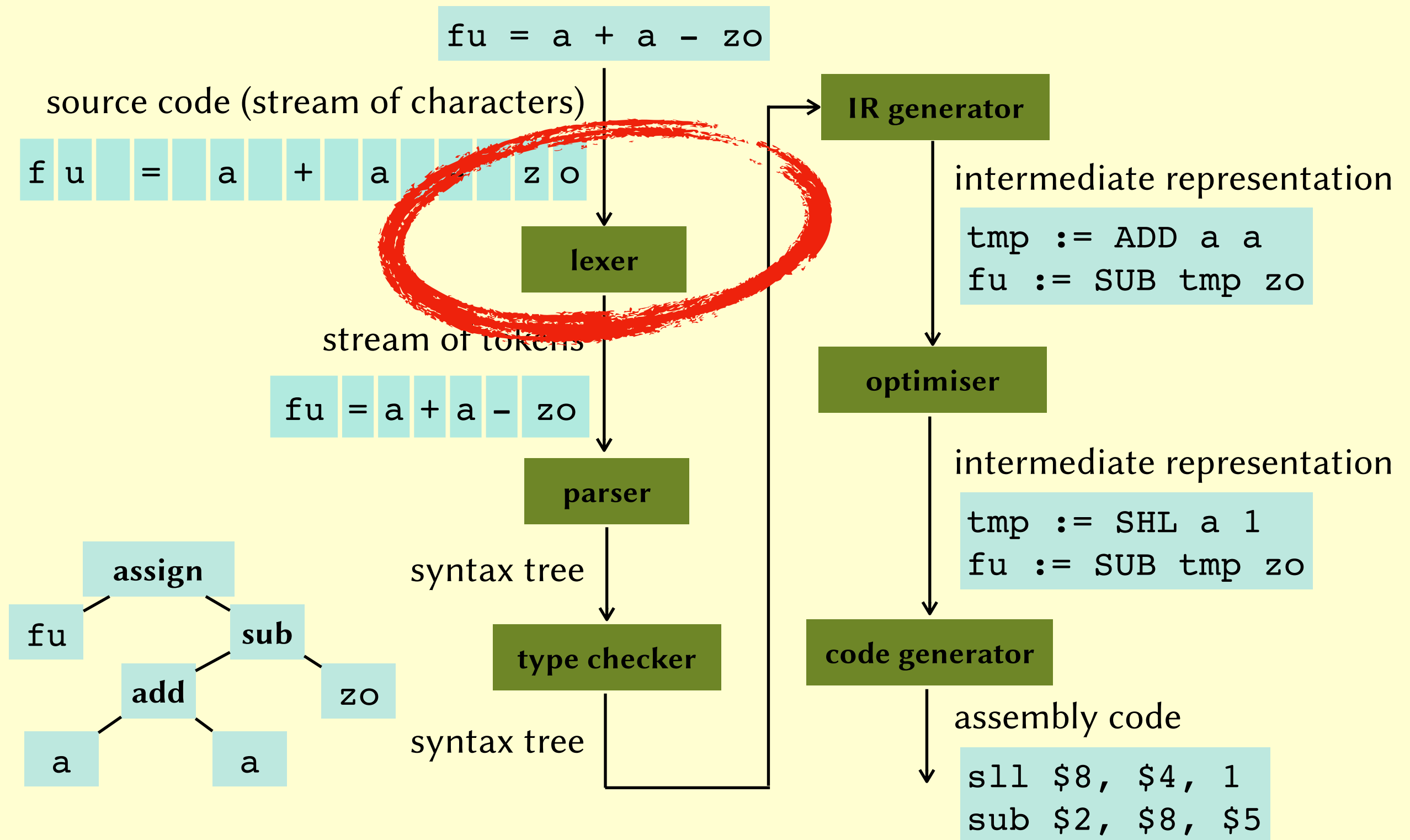


Lecture 3: More lexing

John Wickerson

Compilers

Anatomy of a compiler



What we know so far...

- Lexing is about recognising **tokens** in a stream of characters.
- The form of tokens can be defined using **regular expressions**.
- Given a regex, we can define the **language** that it accepts.
- **This lecture:** How can we tell whether a given word is accepted by a given regex?

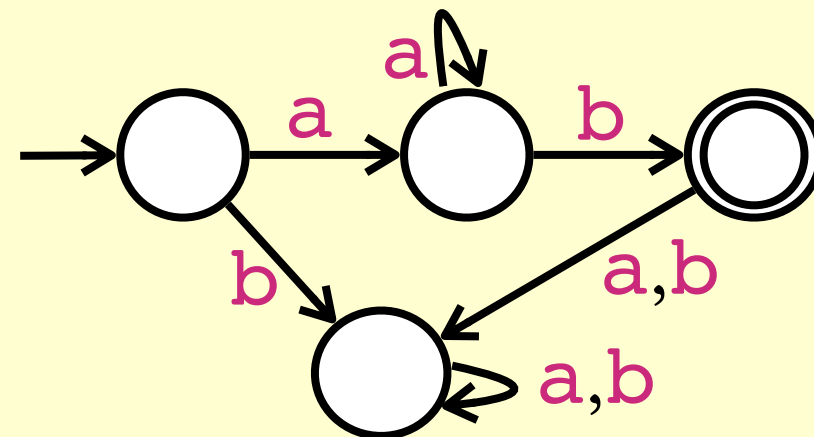
Quiz time!

Regexes and automata

- Does the regular expression $((a+b)(ba)^*b+a)^*$ accept the word **abababa**? How can we answer this question in general?
- A regular expression can be understood as a **finite automaton** (also called a state machine).

- **Example:**

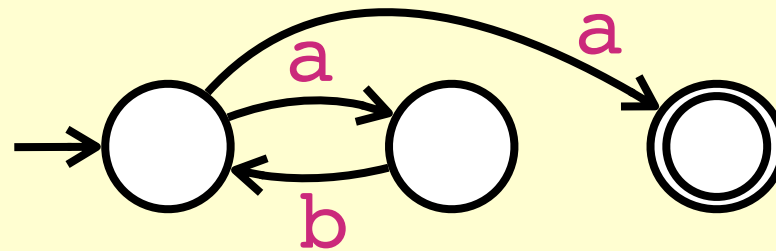
aa^*b



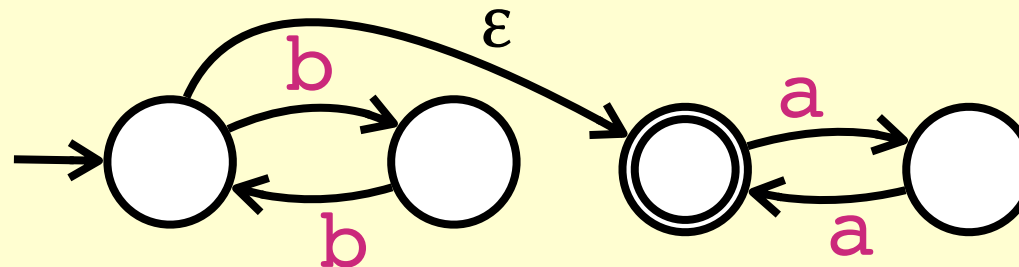
- If we can convert a regular expression into a finite automaton, we can use the automaton to check whether a word matches.

Regexes and automata

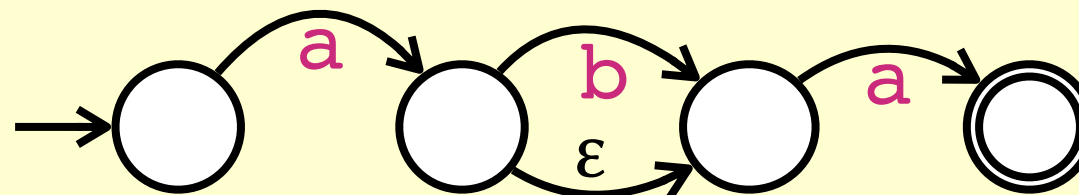
1. $(ab)^*a$



2. $(bb)^*(aa)^*$



3. $a(1+b)a$

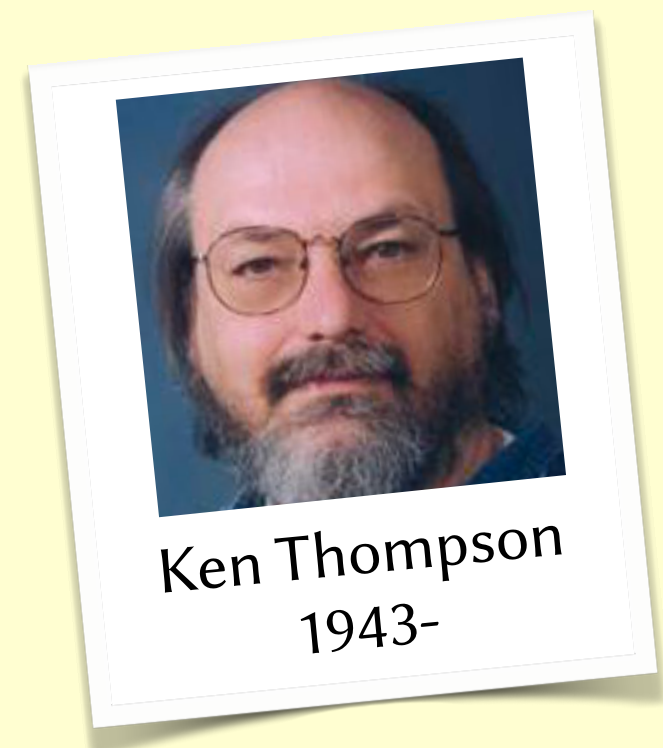


Finite automata

	Deterministic finite automaton (DFA)	Nondeterministic finite automaton (NFA)
Outgoing transitions from each state	Exactly one per symbol	Any number
ϵ -transitions	Not allowed	Allowed
Key advantage	Simpler to run	More concise

Regex \rightarrow NFA

- The following algorithm for converting a regular expression into an NFA is due to Thompson, creator of Unix.



Regex \rightarrow NFA

- $\text{NFA}(\mathbf{0}) = \rightarrow \bigcirc \quad \bigcirc\!\!\bigcirc$

- $\text{NFA}(\mathbf{1}) = \rightarrow \bigcirc \xrightarrow{\epsilon} \bigcirc\!\!\bigcirc$

- $\text{NFA}(\mathbf{c}) = \rightarrow \bigcirc \xrightarrow{\mathbf{c}} \bigcirc\!\!\bigcirc$

- $\text{NFA}(\mathbf{r+s}) =$

$\rightarrow \bigcirc$

\bigcirc

$\text{NFA}(\mathbf{r})$

$\bigcirc\!\!\bigcirc$

$\rightarrow \bigcirc$

\bigcirc

$\text{NFA}(\mathbf{s})$

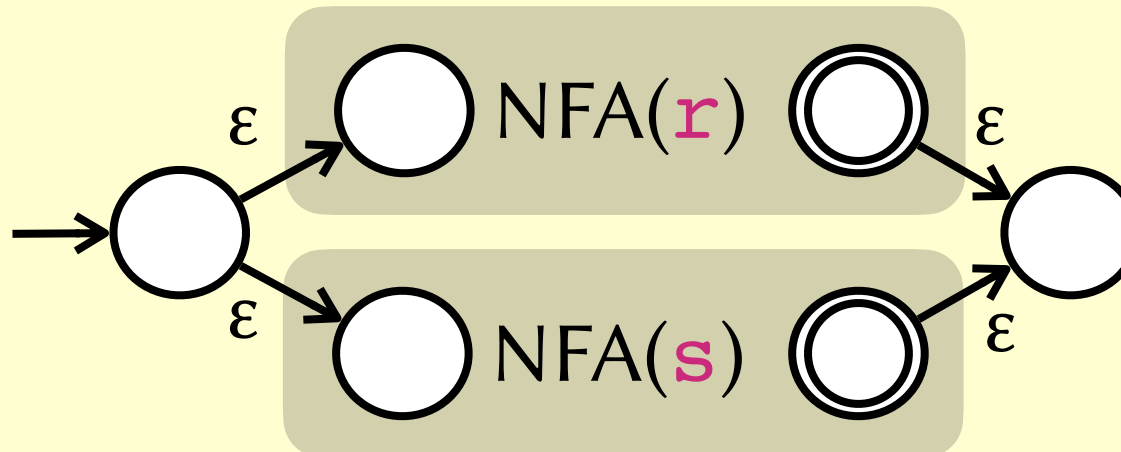
$\bigcirc\!\!\bigcirc$

Regex \rightarrow NFA

- $\text{NFA}(\mathbf{0}) = \rightarrow \bigcirc \quad \bigcirc\!\!\bigcirc$

- $\text{NFA}(\mathbf{1}) = \rightarrow \bigcirc \xrightarrow{\varepsilon} \bigcirc\!\!\bigcirc$

- $\text{NFA}(\mathbf{c}) = \rightarrow \bigcirc \xrightarrow{\mathbf{c}} \bigcirc\!\!\bigcirc$

- $\text{NFA}(\mathbf{r+s}) =$ The diagram illustrates the construction of an NFA for the regular expression $r+s$. It features a start state on the left with an incoming arrow. Two epsilon transitions (ε) lead from this start state to the start states of two sub-NFAs, $\text{NFA}(\mathbf{r})$ and $\text{NFA}(\mathbf{s})$. Each sub-NFA is enclosed in a light gray rounded rectangle and contains a start state and an accept state (double circle). The sub-NFA for \mathbf{r} has its start and accept states connected by an epsilon transition. Similarly, the sub-NFA for \mathbf{s} has its start and accept states connected by an epsilon transition. Finally, epsilon transitions (ε) lead from the accept states of both $\text{NFA}(\mathbf{r})$ and $\text{NFA}(\mathbf{s})$ to a single final state on the right, which has an incoming arrow.

Regex \rightarrow NFA

- $\text{NFA}(\mathbf{0}) = \rightarrow \bigcirc \quad \bigcirc\!\!\bigcirc$

- $\text{NFA}(\mathbf{1}) = \rightarrow \bigcirc \xrightarrow{\varepsilon} \bigcirc\!\!\bigcirc$

- $\text{NFA}(\mathbf{c}) = \rightarrow \bigcirc \xrightarrow{\mathbf{c}} \bigcirc\!\!\bigcirc$

- $\text{NFA}(\mathbf{r+s}) = \rightarrow \bigcirc \begin{array}{l} \xrightarrow{\varepsilon} \bigcirc \text{NFA}(\mathbf{r}) \bigcirc \xrightarrow{\varepsilon} \bigcirc\!\!\bigcirc \\ \xrightarrow{\varepsilon} \bigcirc \text{NFA}(\mathbf{s}) \bigcirc \xrightarrow{\varepsilon} \bigcirc\!\!\bigcirc \end{array}$

Regex \rightarrow NFA

- $\text{NFA}(\textcolor{violet}{r}\textcolor{violet}{s}) = \rightarrow \text{NFA}(\textcolor{violet}{r}) \text{NFA}(\textcolor{violet}{s})$

- $\text{NFA}(\textcolor{violet}{r} + \textcolor{violet}{s}) =$

The diagram illustrates the construction of an NFA for the union of two regular expressions, r and s . On the left, a single start state (a circle with an incoming arrow) branches into two paths. The top path is enclosed in a light gray box and represents the NFA for r , consisting of two states connected by an arrow labeled ϵ . The bottom path is also enclosed in a light gray box and represents the NFA for s , also consisting of two states connected by an arrow labeled ϵ . Both paths converge to a single final state (a double circle) on the right, with the transition from each path labeled ϵ .

Regex \rightarrow NFA

- $\text{NFA}(\mathbf{rs}) =$

- $\text{NFA}(\mathbf{r}^*) =$

- $\text{NFA}(\mathbf{r+s}) =$

Regex \rightarrow NFA

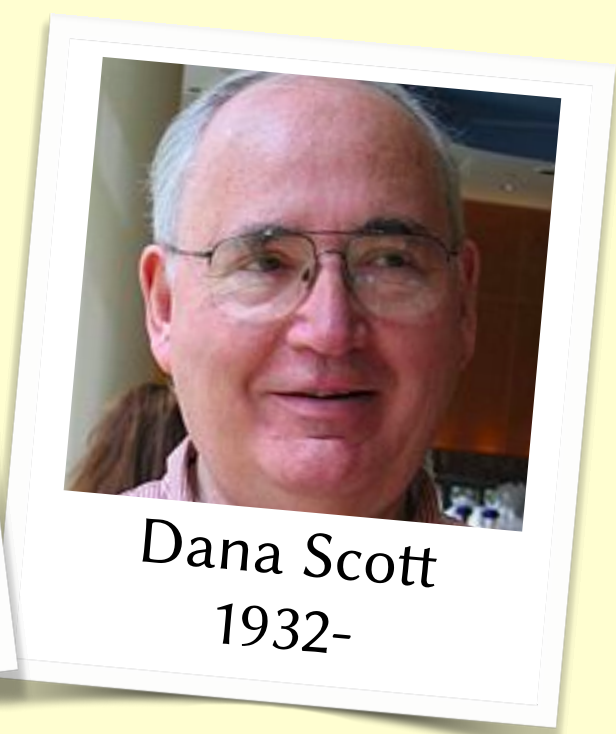
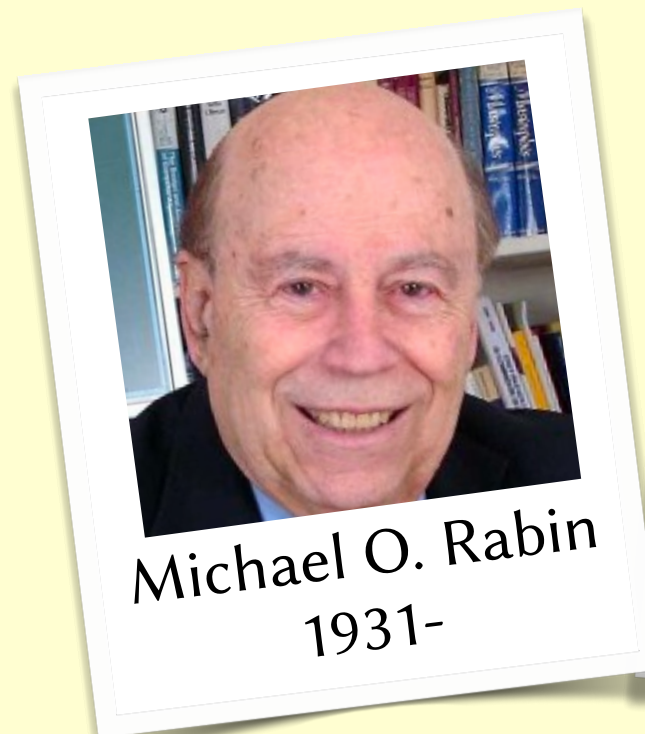
- $\text{NFA}(\mathbf{rs}) = \text{NFA}(\mathbf{r}) \xrightarrow{\epsilon} \text{NFA}(\mathbf{s})$

- $\text{NFA}(\mathbf{r}^*) = \text{NFA}(\mathbf{r}) \text{ with a loop}$

- $\text{NFA}(\mathbf{r+s}) = \text{NFA}(\mathbf{r}) \text{ OR } \text{NFA}(\mathbf{s})$

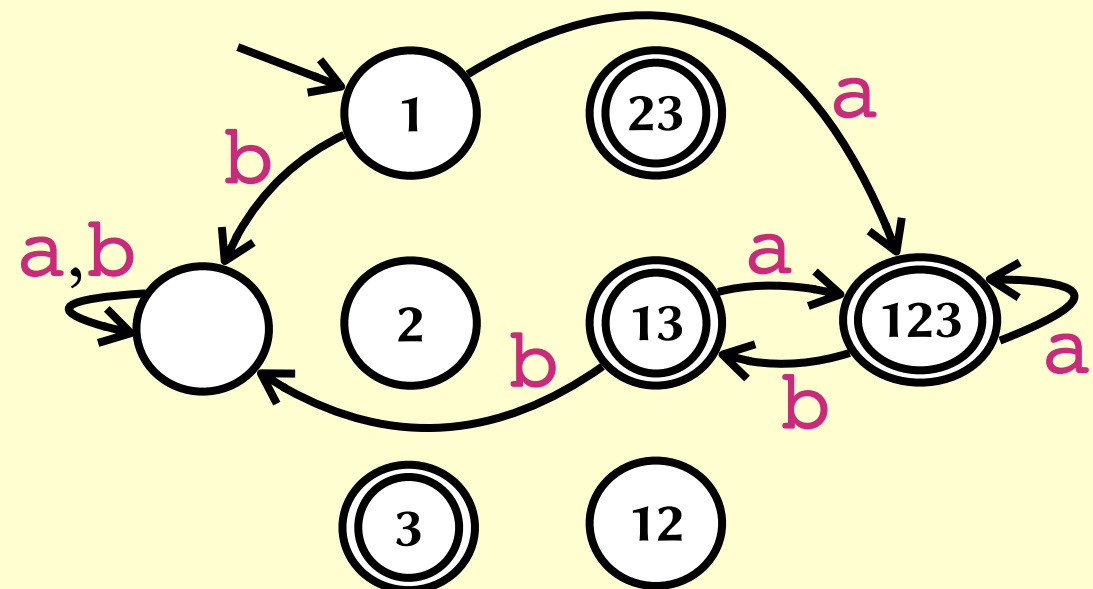
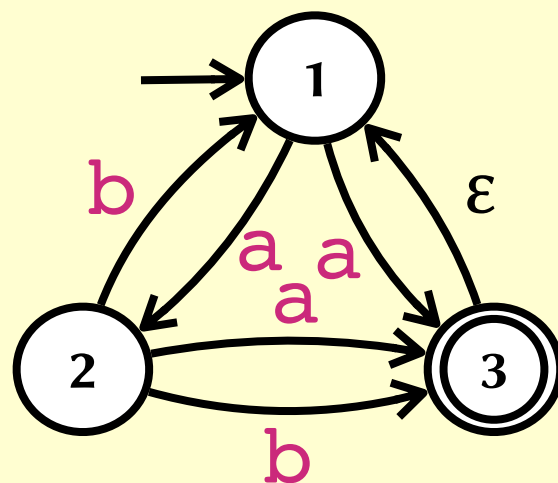
NFAs and DFAs

- These automata are non-deterministic, so not very helpful – we have to keep track of lots of states at once!
- However, we can convert a **non-deterministic** automaton into a **deterministic** one using the **subset construction**.



NFA \rightarrow DFA

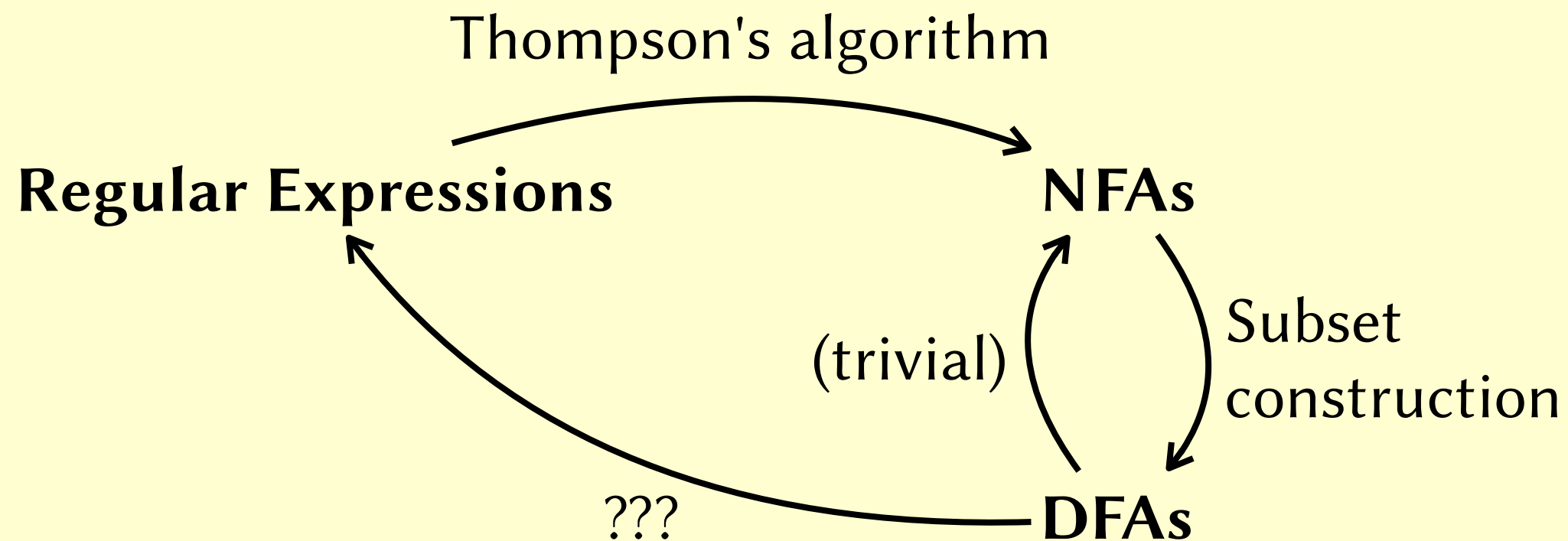
- To simulate a DFA: keep track of **the** state you are in.
- To simulate an NFA: keep track of **all** the states you are in.
- So: convert an NFA with states $\{s_1, s_2, \dots\}$ into a DFA whose states are **all subsets** of $\{s_1, s_2, \dots\}$.
- **Example.**



Summary

- Lexing is about recognising **tokens** in a stream of characters.
- The form of tokens can be defined using **regular expressions**.
- Given a regex, we can define the **language** that it accepts.
- We can turn any regex into an equivalent **NFA**.
- We can turn any NFA into an equivalent **DFA**.
- By simulating this DFA, we can quickly check whether a given word matches the regex.

Something to ponder



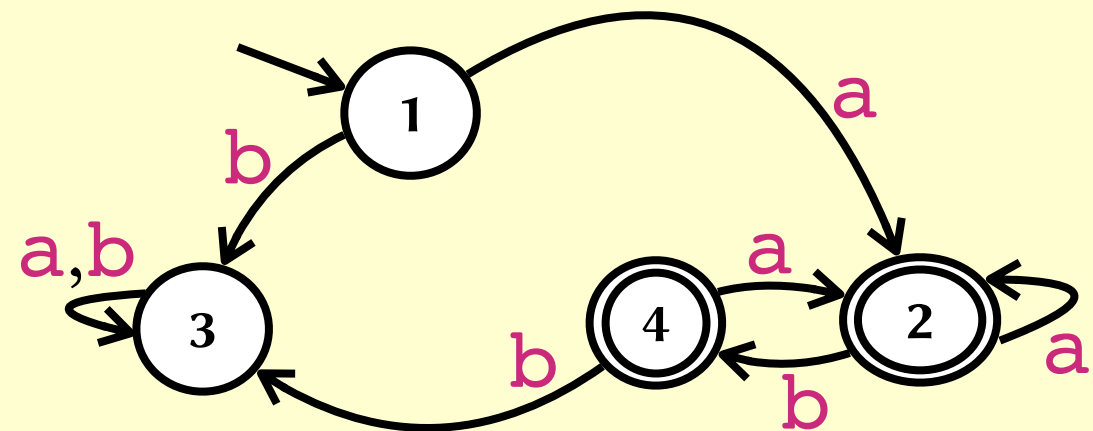
DFA \rightarrow Regex

$$R_1 = aR_2 + bR_3$$

$$R_2 = aR_2 + bR_4 + 1$$

$$R_3 = aR_3 + bR_3$$

$$R_4 = aR_2 + bR_3 + 1$$

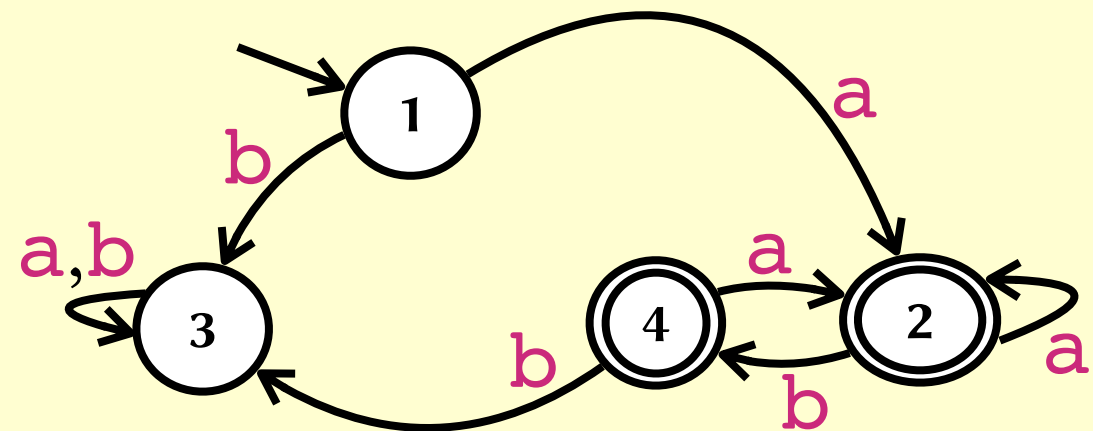


DFA \rightarrow Regex

$$R_1 = aR_2 + bR_3$$

$$R_2 = aR_2 + b(aR_2 + bR_3 + 1) + 1$$

$$R_3 = aR_3 + bR_3$$

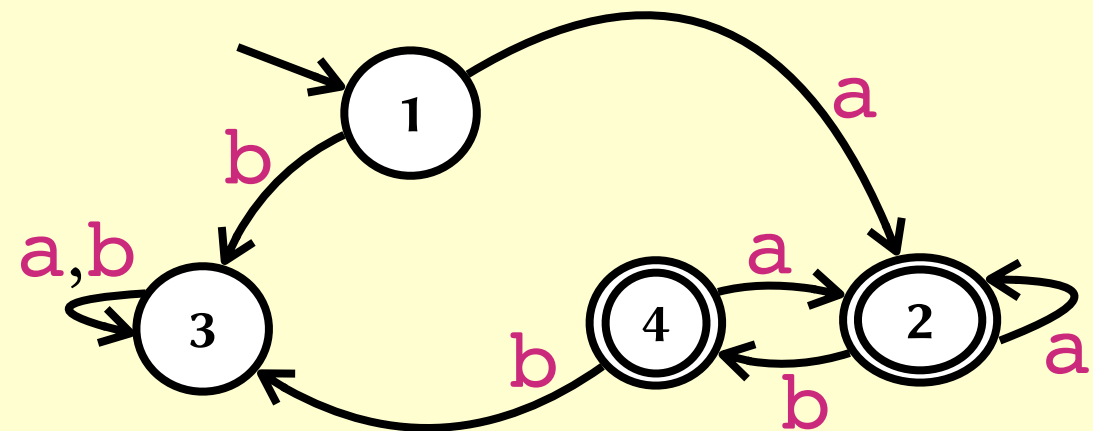


DFA \rightarrow Regex

$$R_1 = aR_2 + bR_3$$

$$R_2 = aR_2 + baR_2 + bbR_3 + b1 + 1$$

$$R_3 = aR_3 + bR_3$$

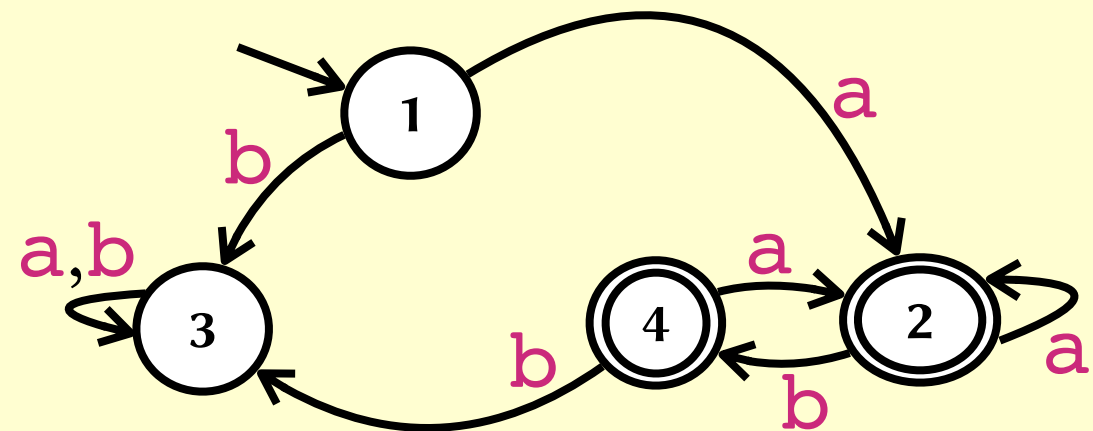


DFA \rightarrow Regex

$$R_1 = aR_2 + bR_3$$

$$R_2 = aR_2 + baR_2 + bbR_3 + b + 1$$

$$R_3 = aR_3 + bR_3$$

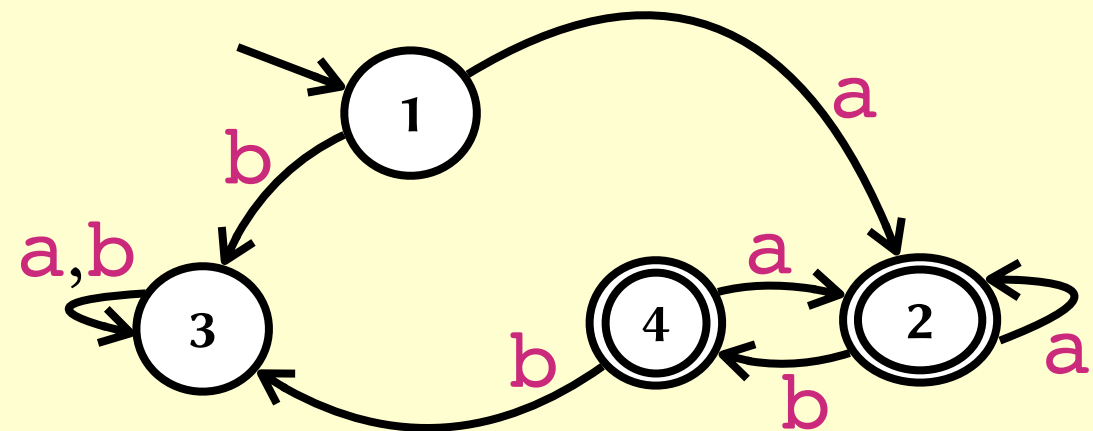


DFA \rightarrow Regex

$$R_1 = aR_2 + bR_3$$

$$R_2 = (a + ba)R_2 + bbR_3 + b + 1$$

$$R_3 = aR_3 + bR_3$$

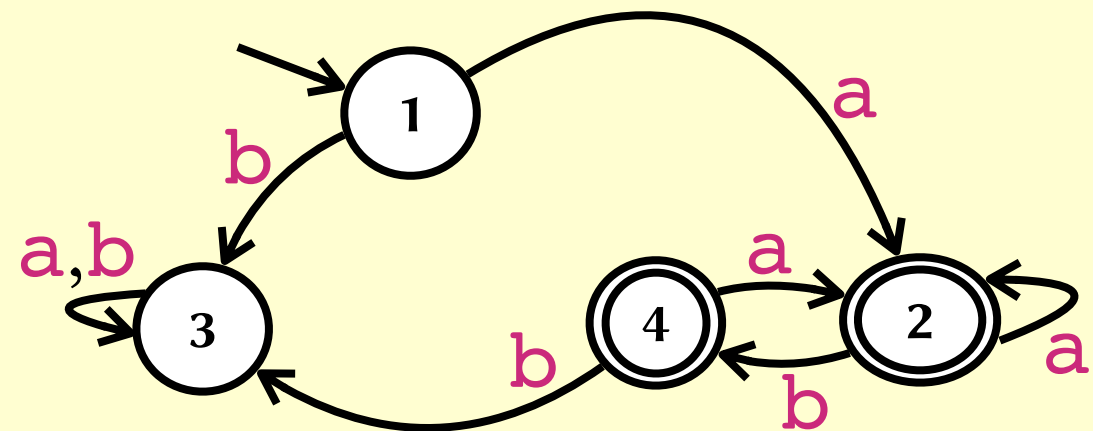


DFA \rightarrow Regex

$$R_1 = aR_2 + bR_3$$

$$R_2 = (a + ba)R_2 + bbR_3 + b + 1$$

$$R_3 = (a + b)R_3$$

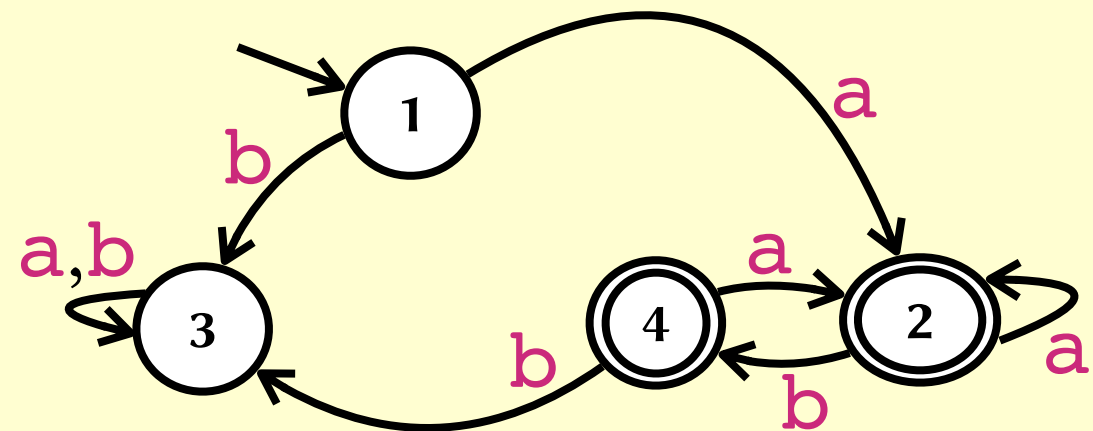


DFA \rightarrow Regex

$$R_1 = \textcolor{violet}{a}R_2 + \textcolor{violet}{b}R_3$$

$$R_2 = (\textcolor{violet}{a} + \textcolor{violet}{ba})R_2 + \textcolor{violet}{bb}R_3 + \textcolor{violet}{b} + \mathbf{1}$$

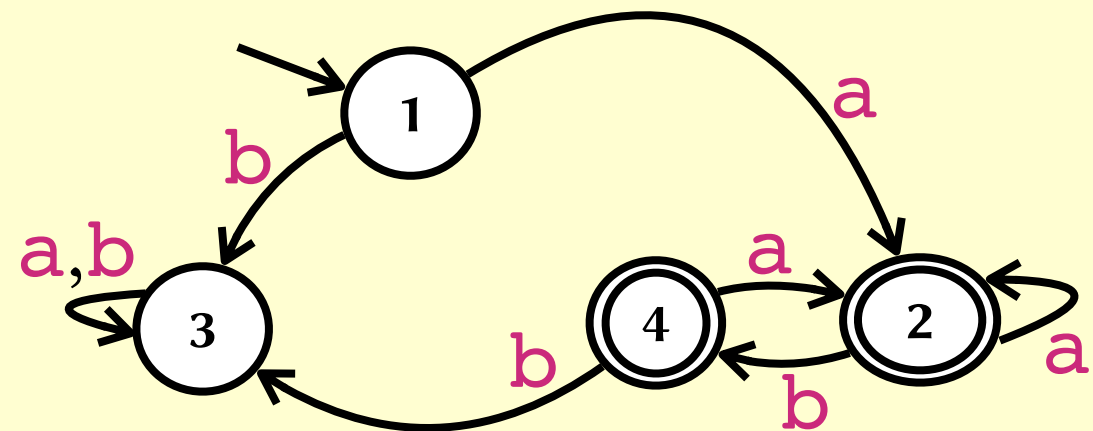
$$R_3 = \mathbf{0}$$



DFA \rightarrow Regex

$$R_1 = aR_2 + b0$$

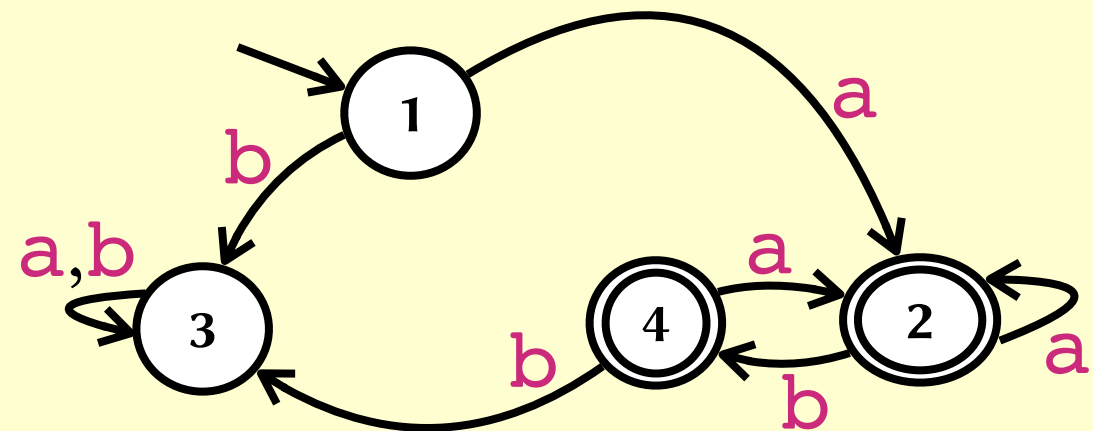
$$R_2 = (a + ba)R_2 + bb0 + b + 1$$



DFA \rightarrow Regex

$$R_1 = aR_2$$

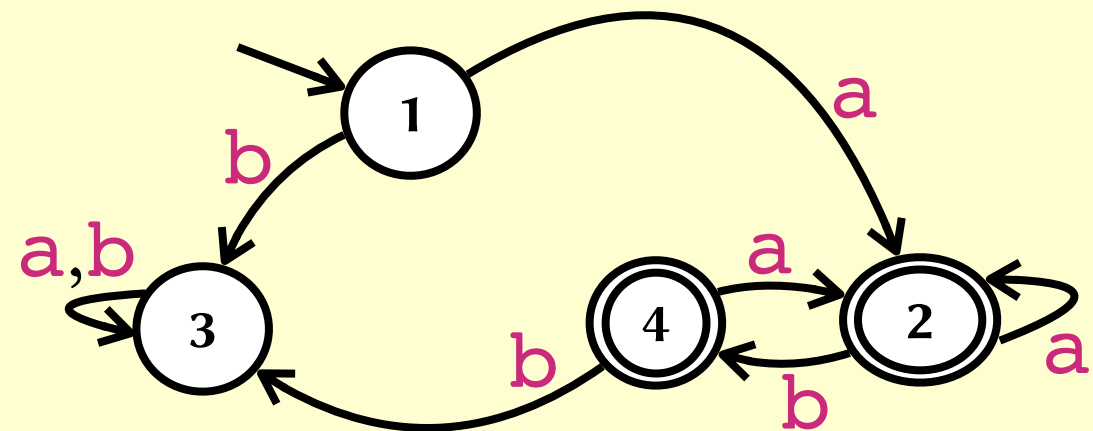
$$R_2 = (a + ba)R_2 + b + 1$$



DFA \rightarrow Regex

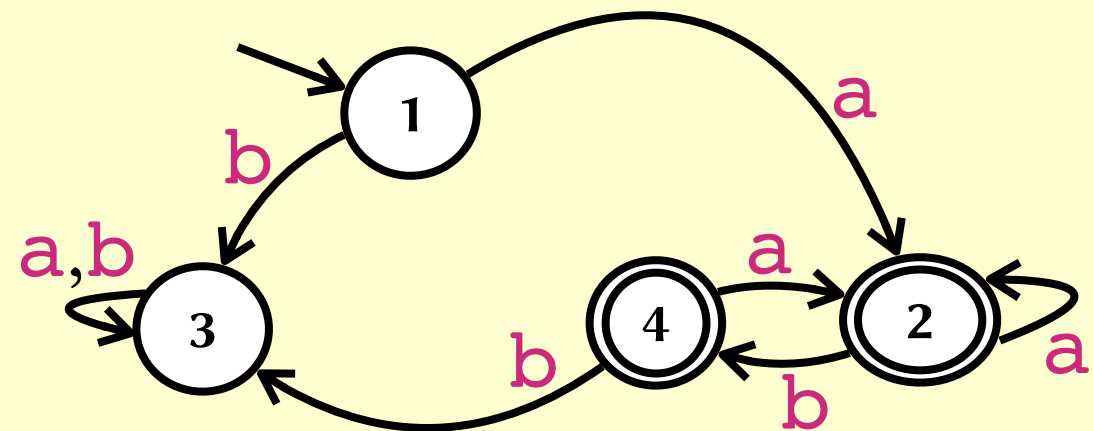
$$R_1 = \textcolor{violet}{a}R_2$$

$$R_2 = (\textcolor{violet}{a} + \textcolor{violet}{b}\textcolor{violet}{a})^*(\textcolor{violet}{b} + \mathbf{1})$$

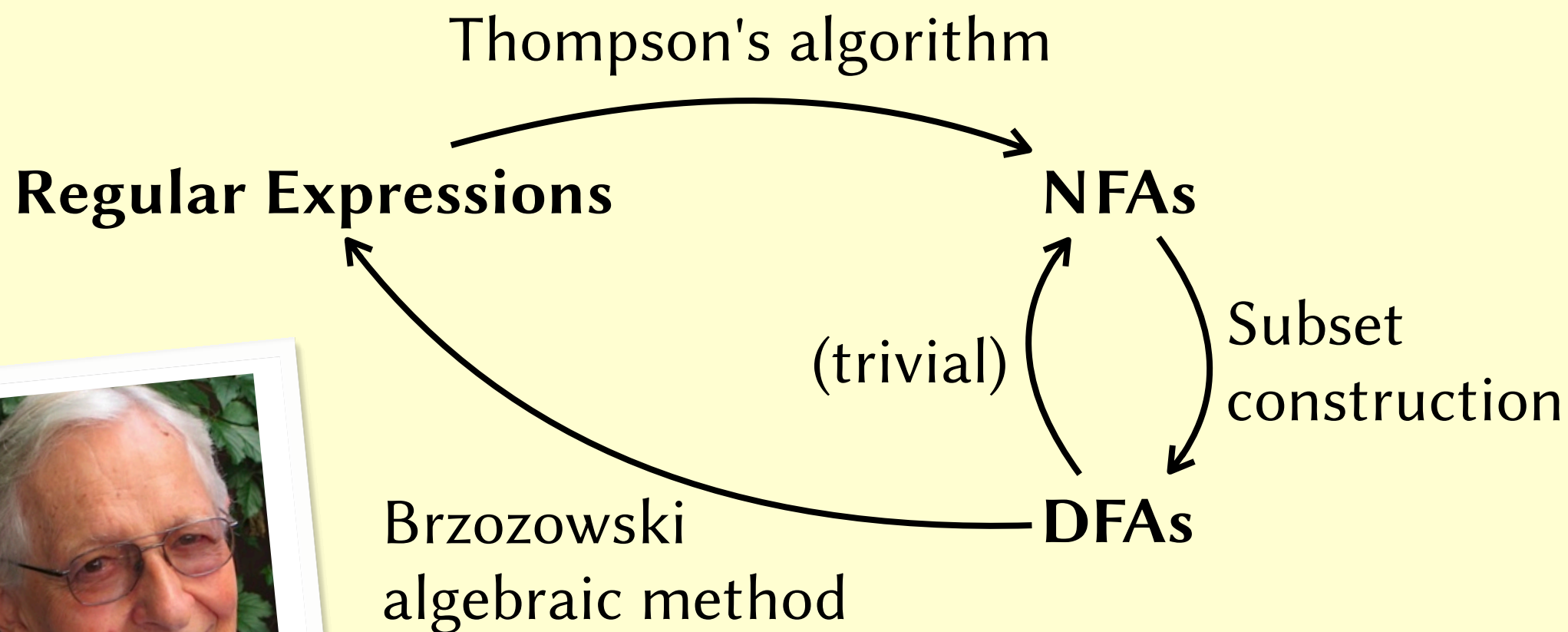


DFA \rightarrow Regex

$$R_1 = a(a + ba)^*(b + 1)$$



Something to ponder

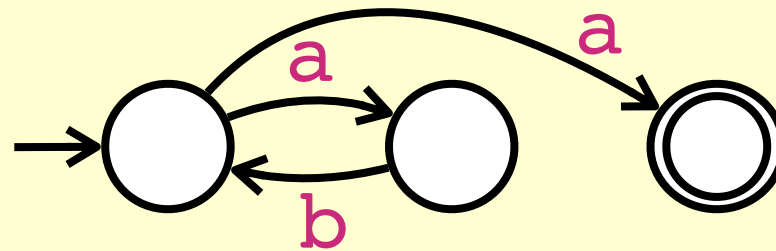


John Brzozowski
1935-2019

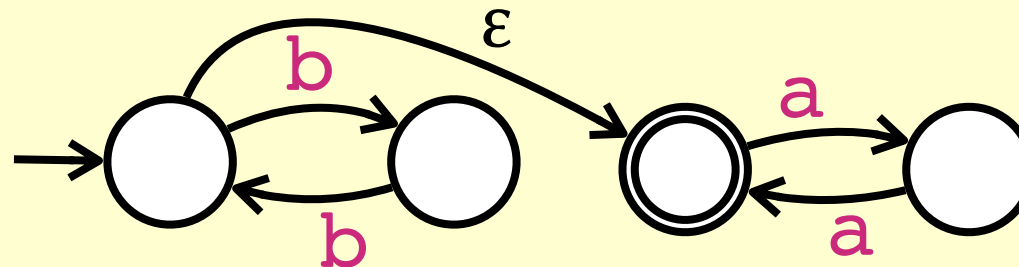
What about Lex?

- So far: decide whether some regex matches some input.
- Lex must decide which regex matches how much input.
- So: run multiple DFAs in parallel, and choose the one that can consume the most input characters.

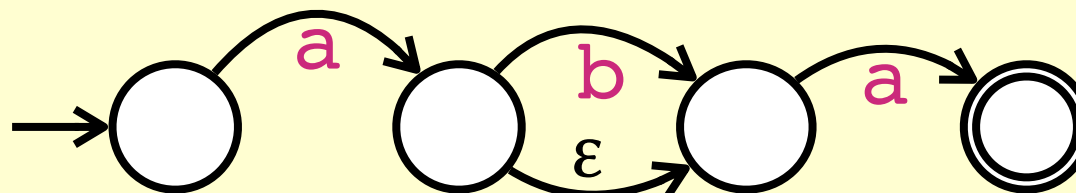
1. $(ab)^*a$



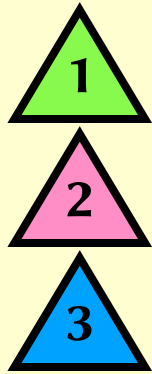
2. $(bb)^*(aa)^*$



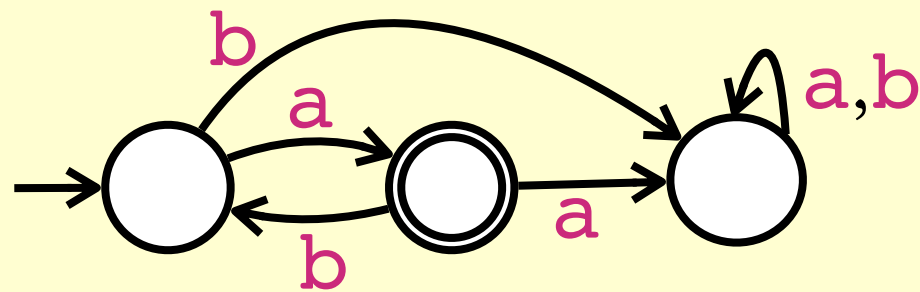
3. $a(1+b)a$



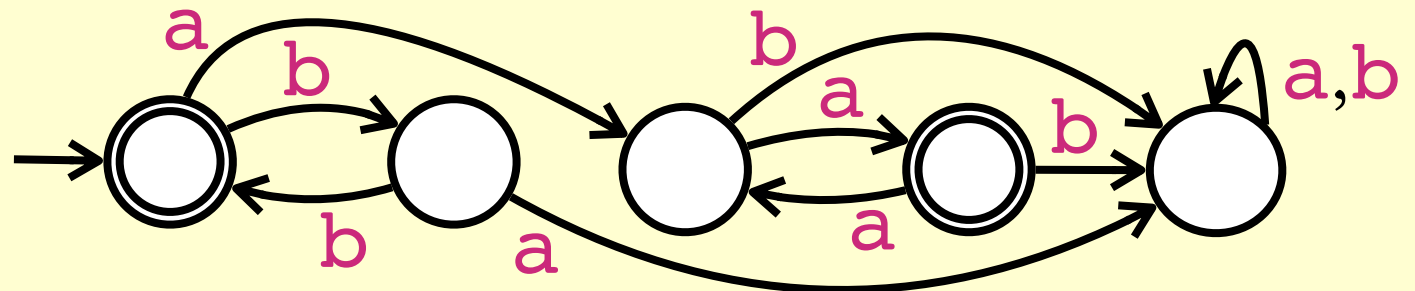
a b a b a b a \$



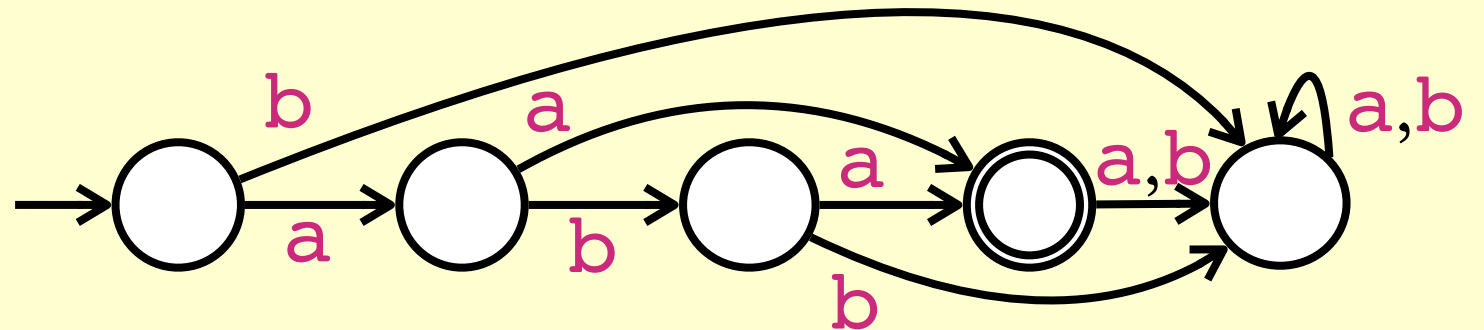
1. $(ab)^*a$



2. $(bb)^*(aa)^*$



3. $a(1+b)a$





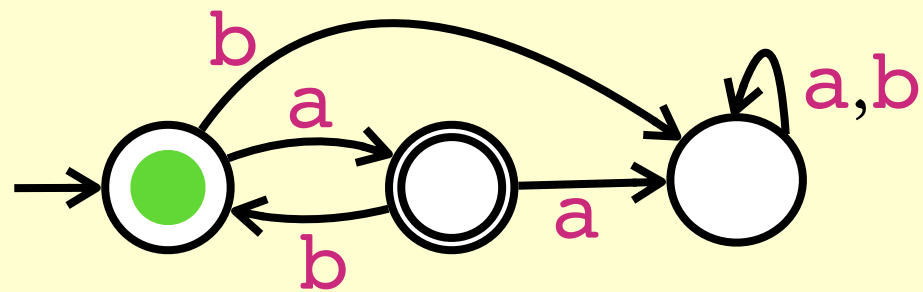
a b a b a b a \$

1

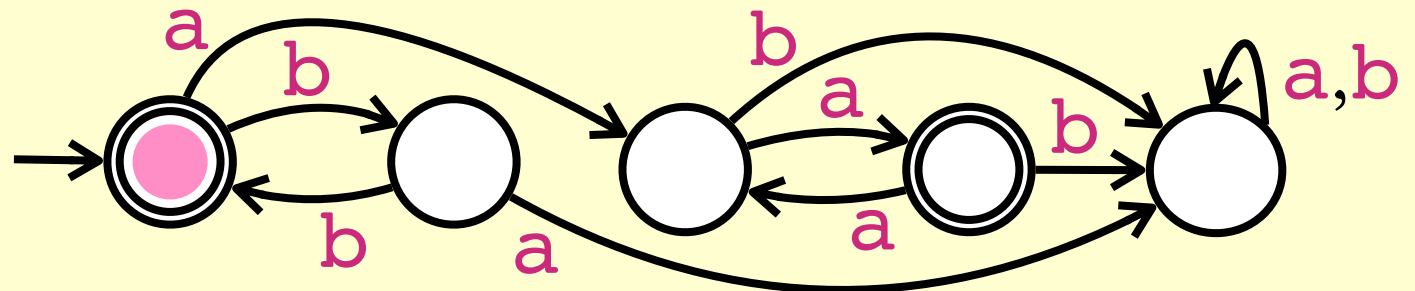
2

3

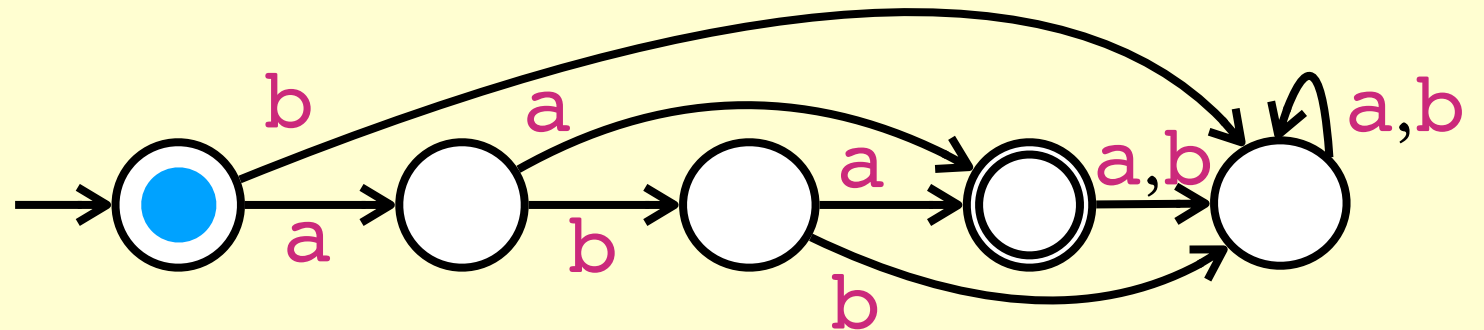
1. $(ab)^*a$



2. $(bb)^*(aa)^*$



3. $a(1+b)a$





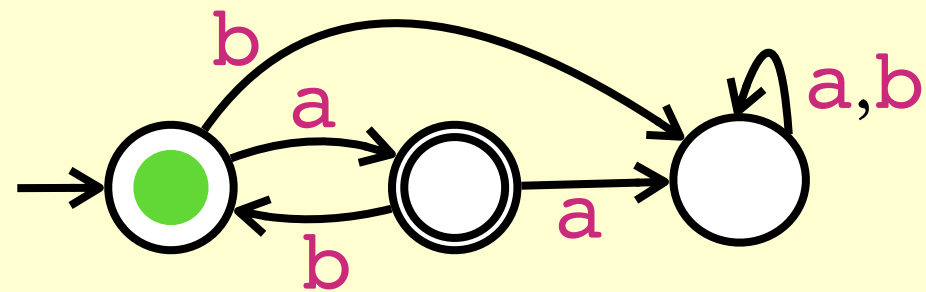
a b a b a b a \$

1

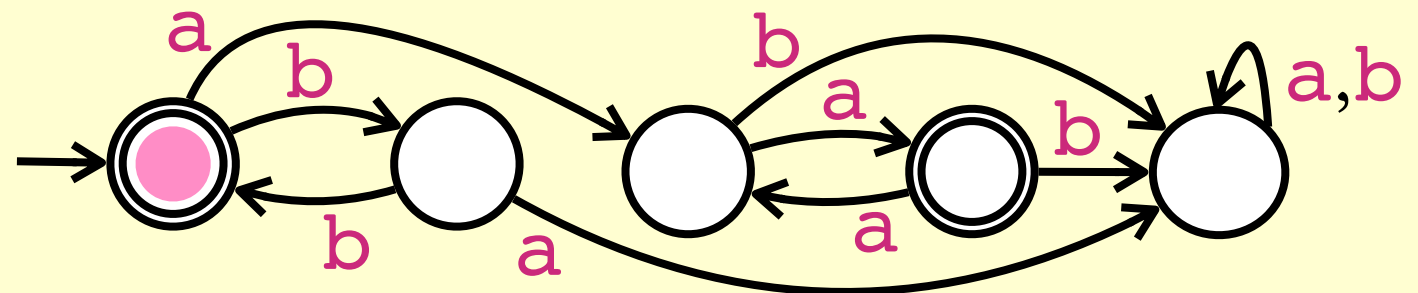
2

3

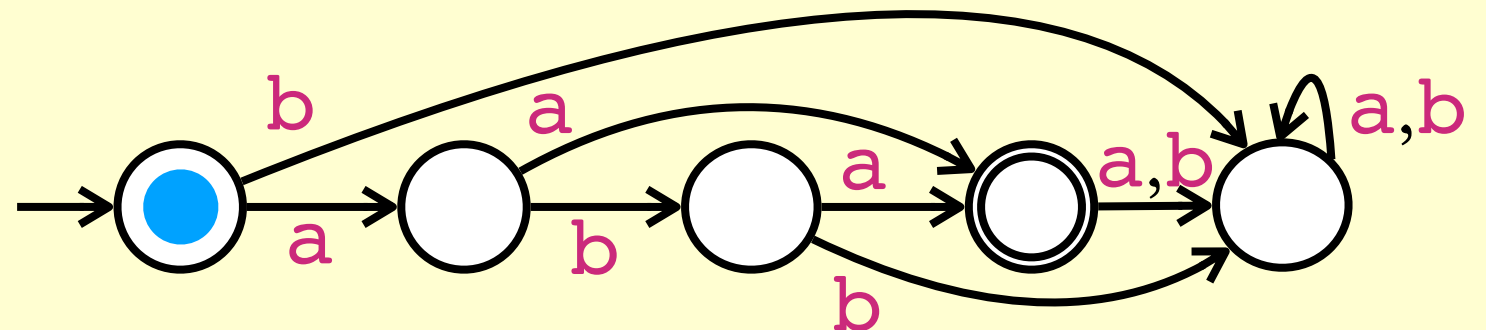
1. $(ab)^*a$

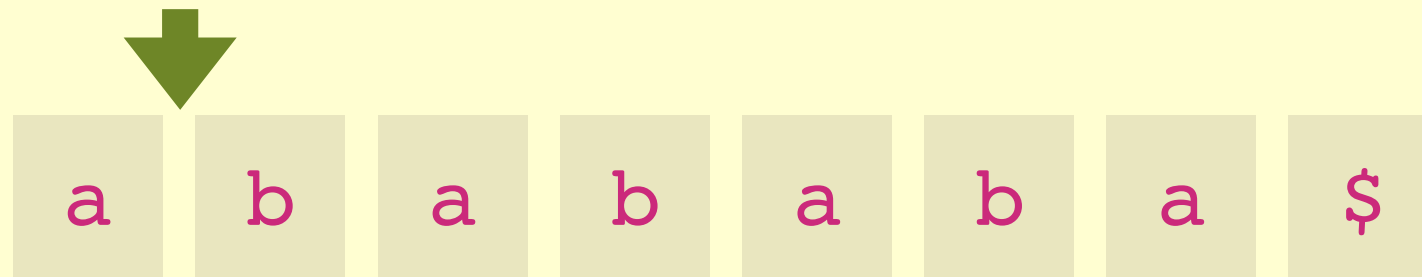


2. $(bb)^*(aa)^*$



3. $a(1+b)a$

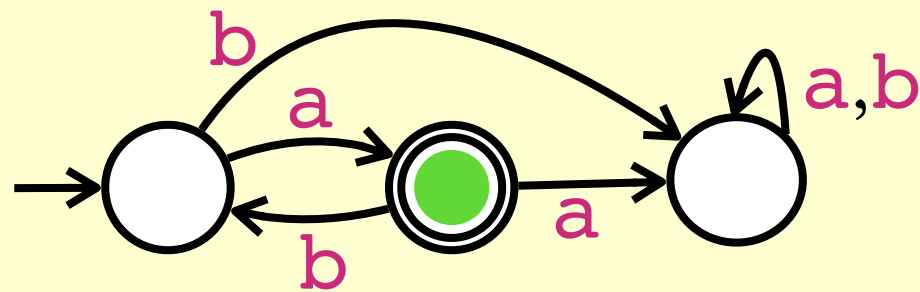
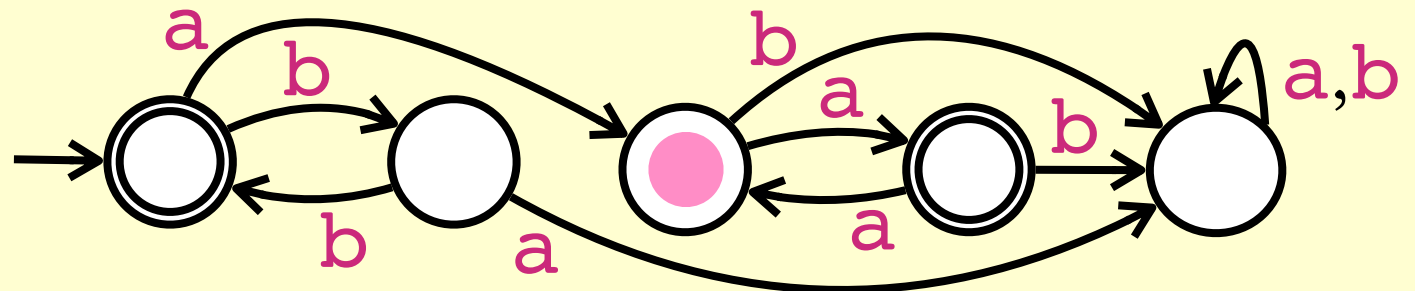
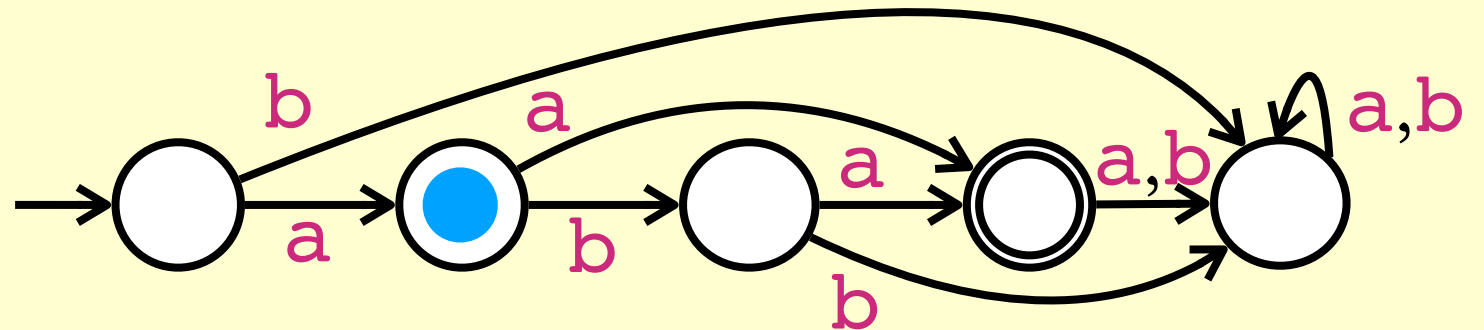


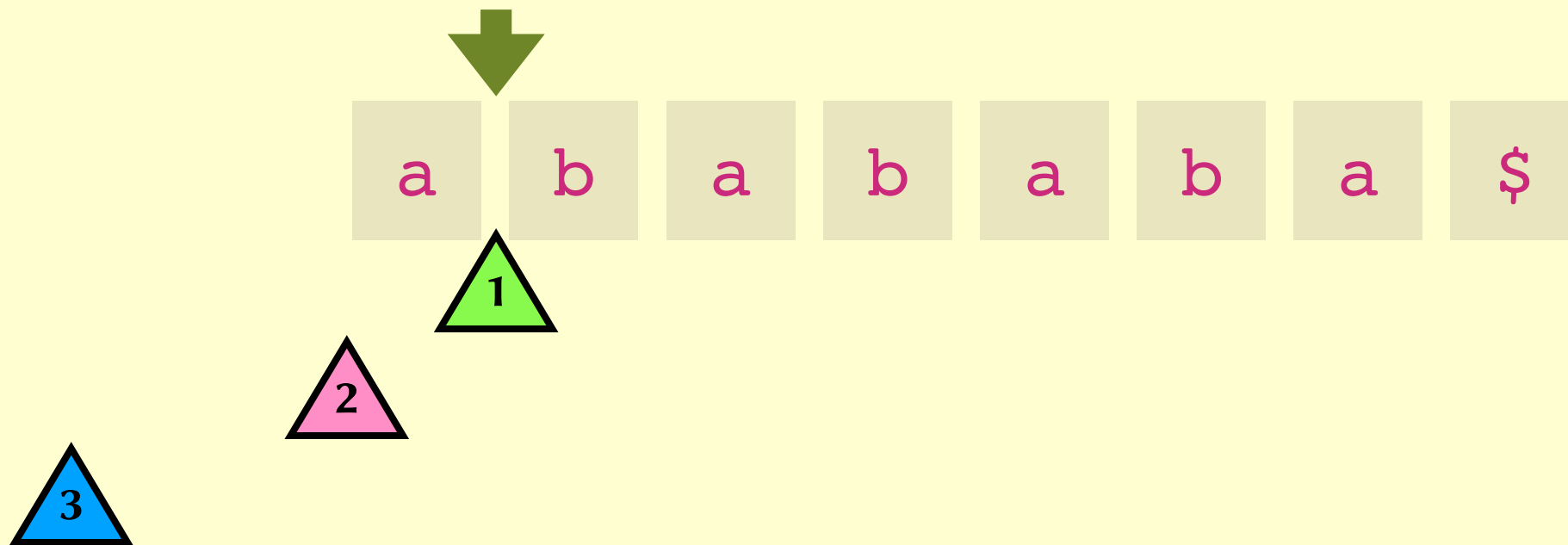


1

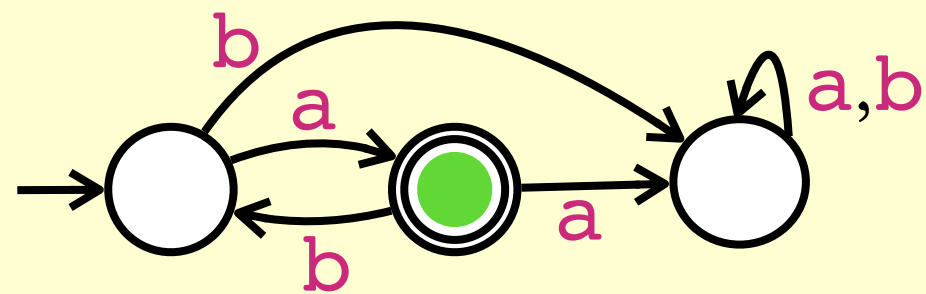
2

3

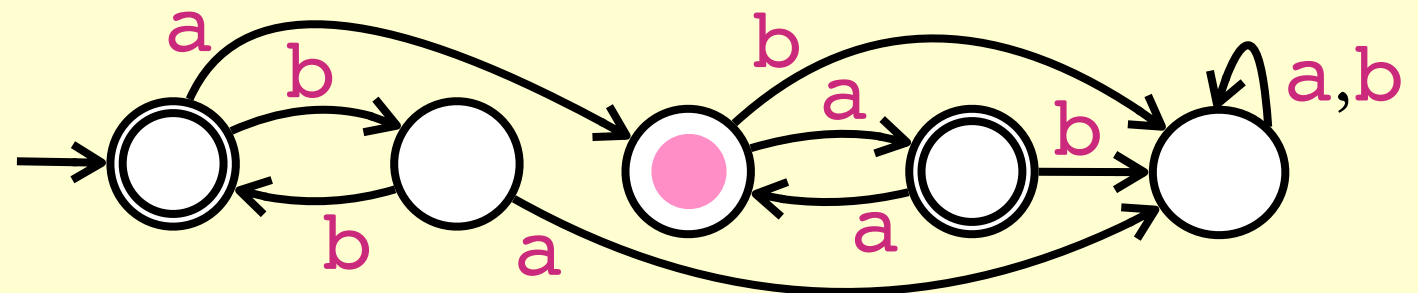
1. $(ab)^*a$ 2. $(bb)^*(aa)^*$ 3. $a(1+b)a$ 



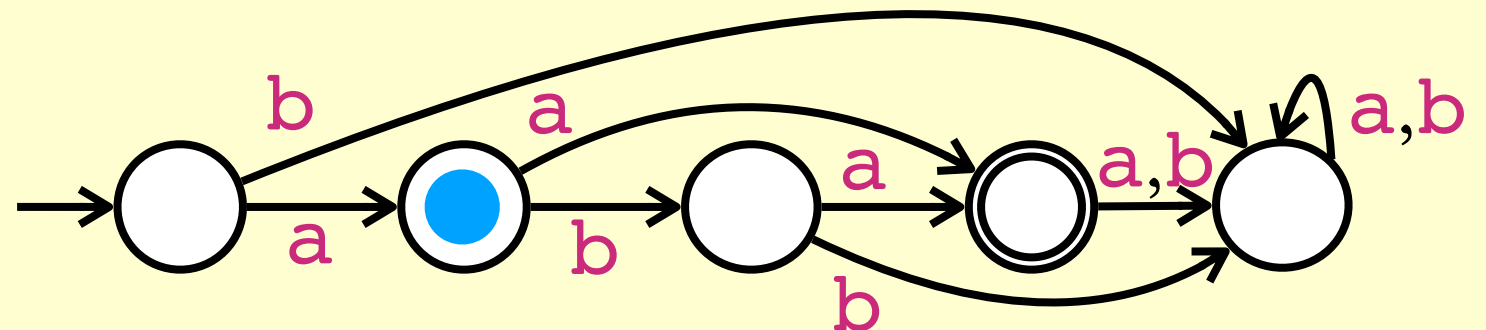
1. $(ab)^*a$

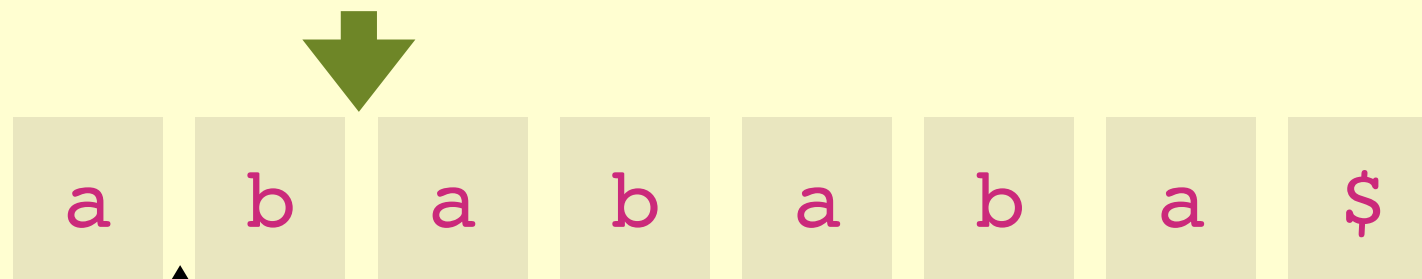


2. $(bb)^*(aa)^*$

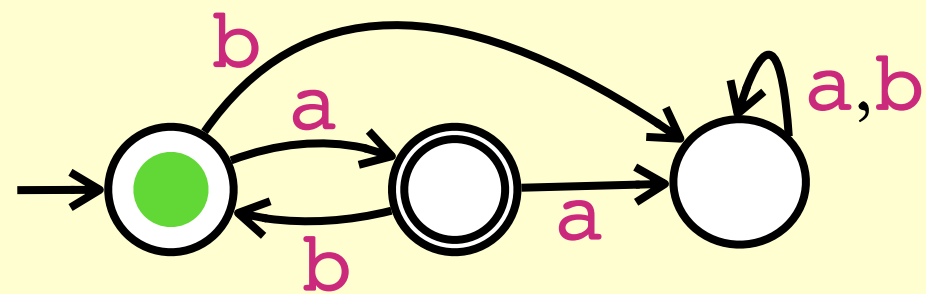


3. $a(1+b)a$

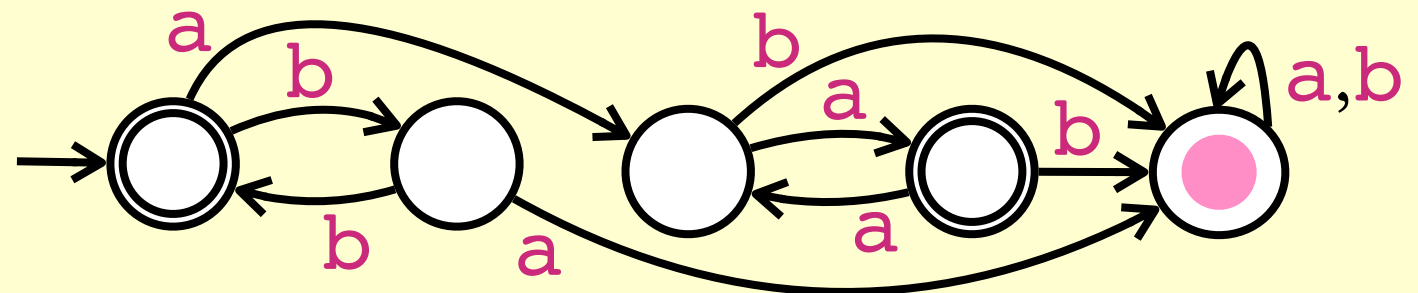




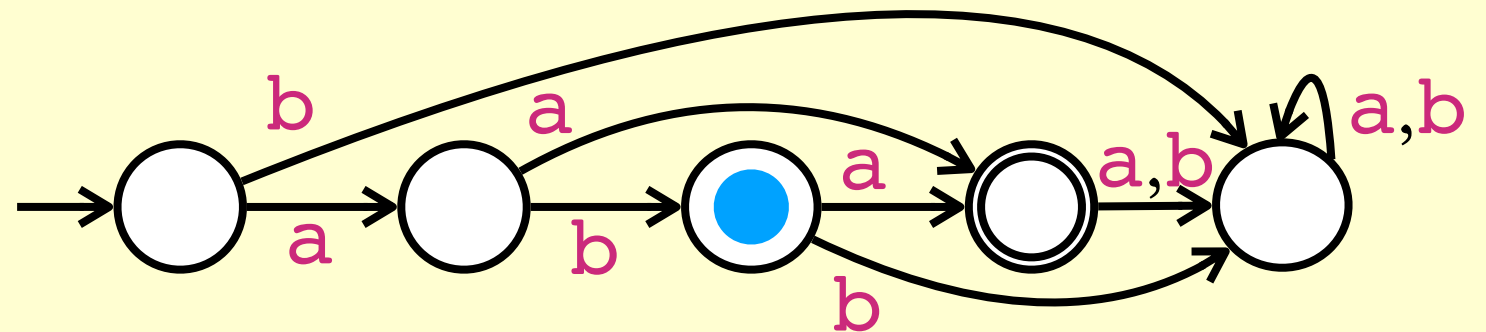
1. $(ab)^*a$

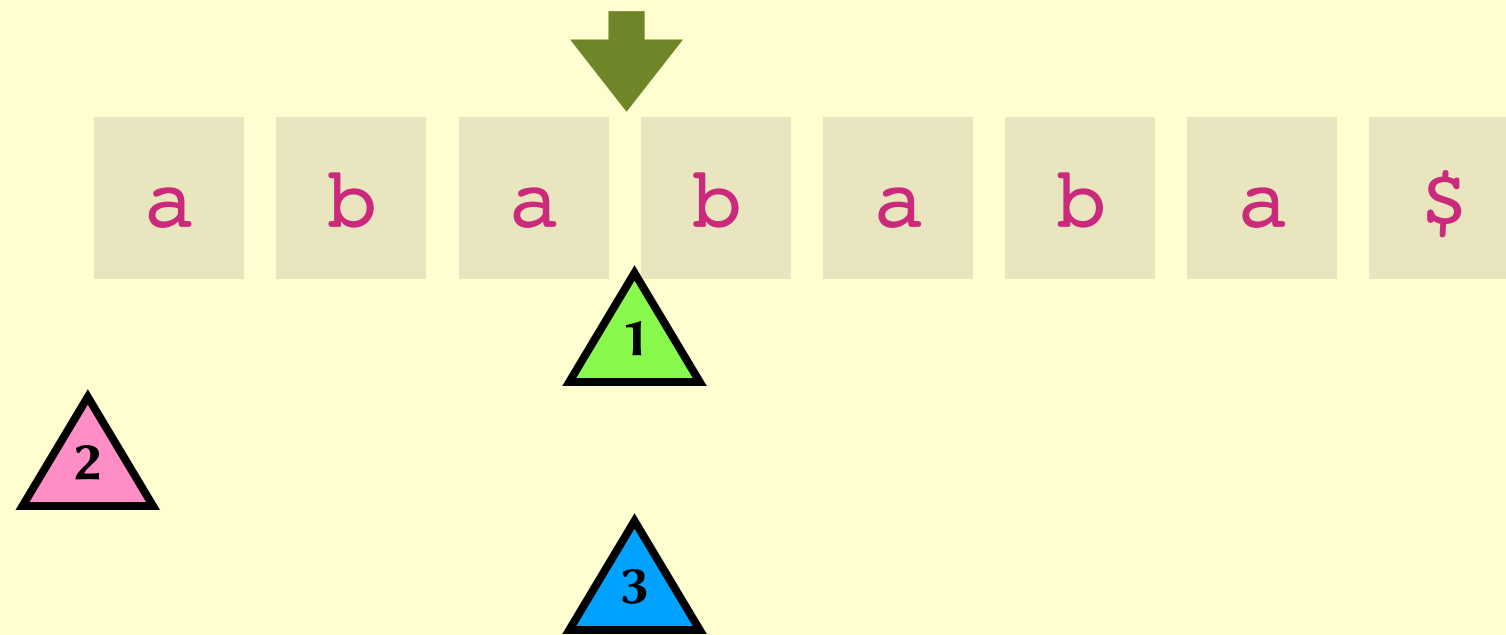


2. $(bb)^*(aa)^*$

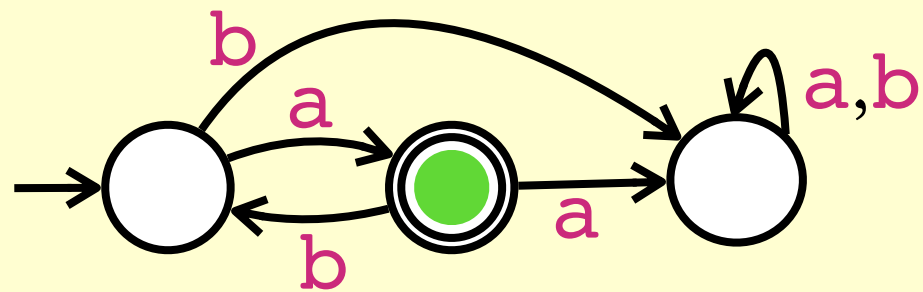


3. $a(1+b)a$

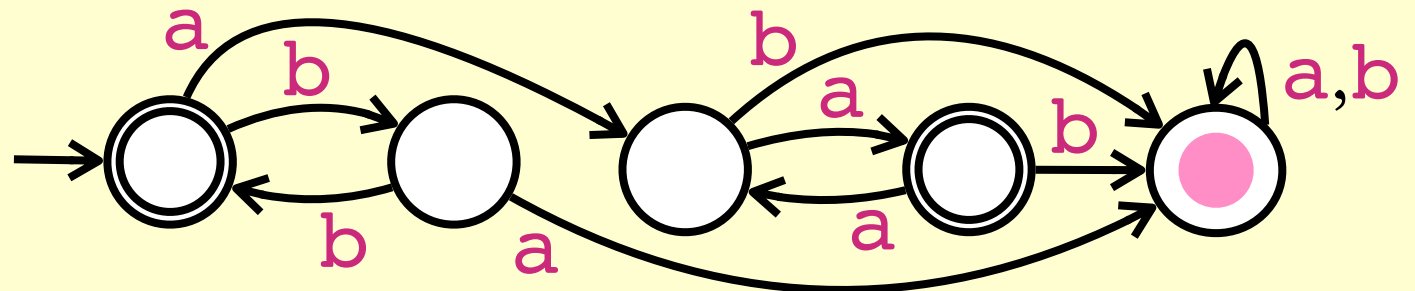




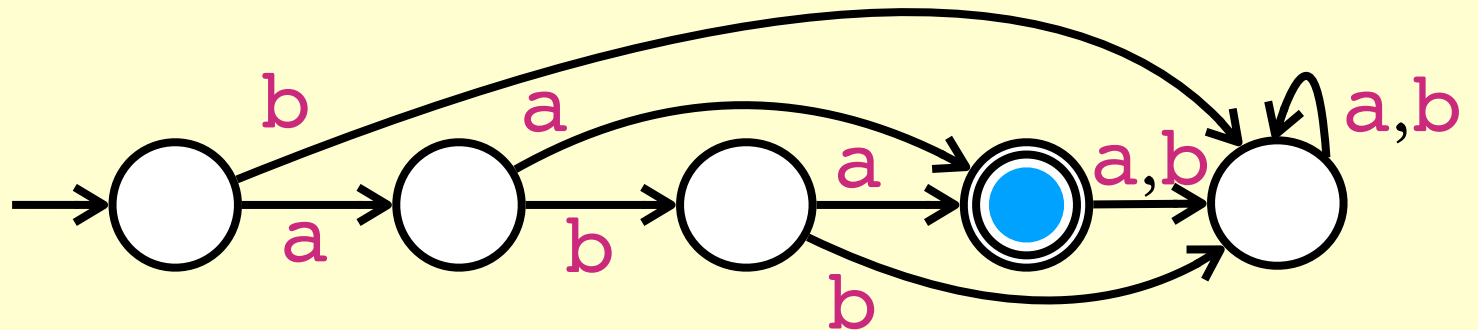
1. $(ab)^*a$

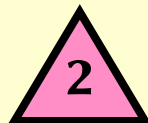
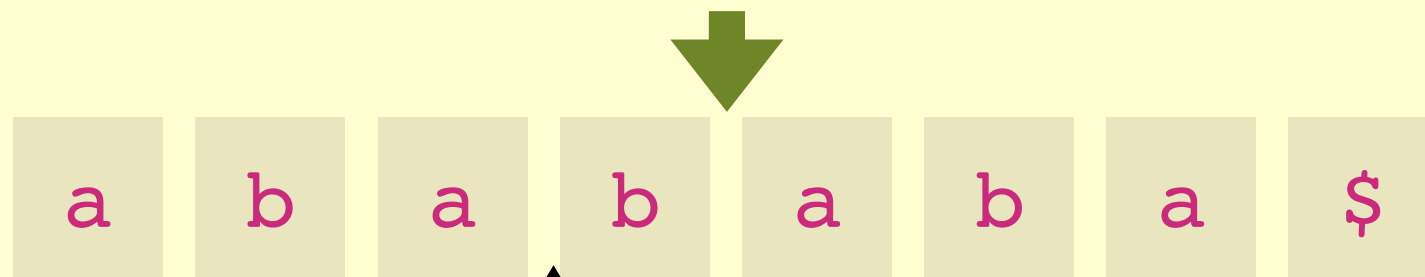


2. $(bb)^*(aa)^*$

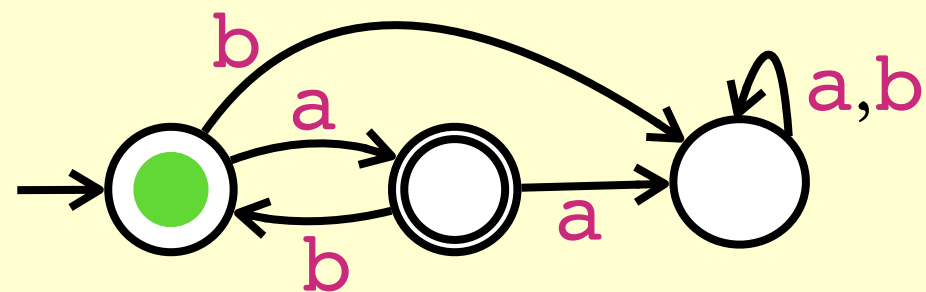


3. $a(1+b)a$

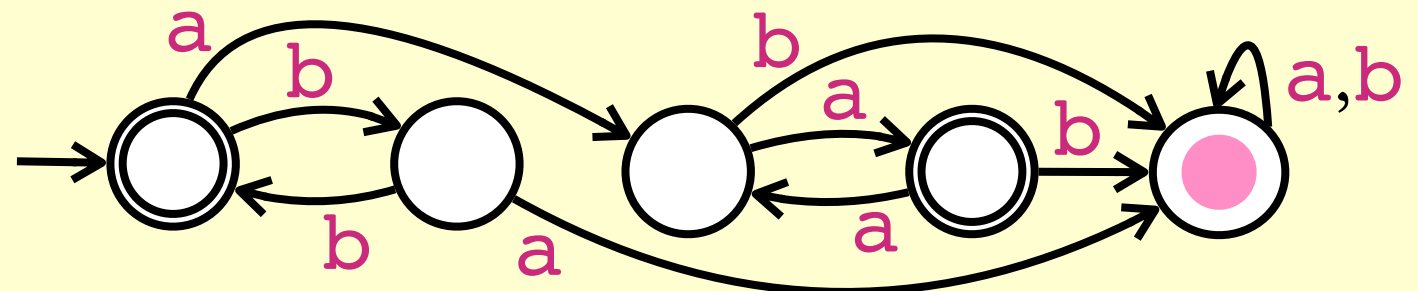




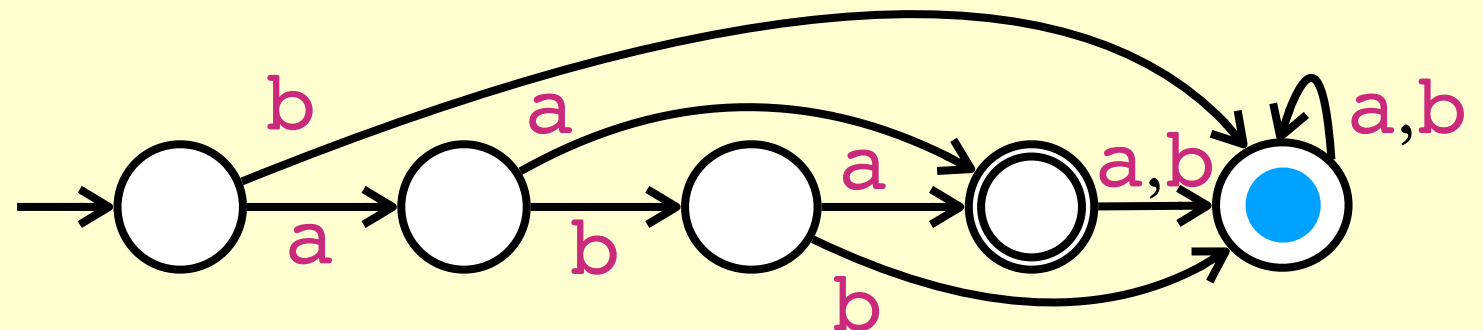
1. $(ab)^*a$

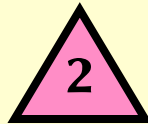
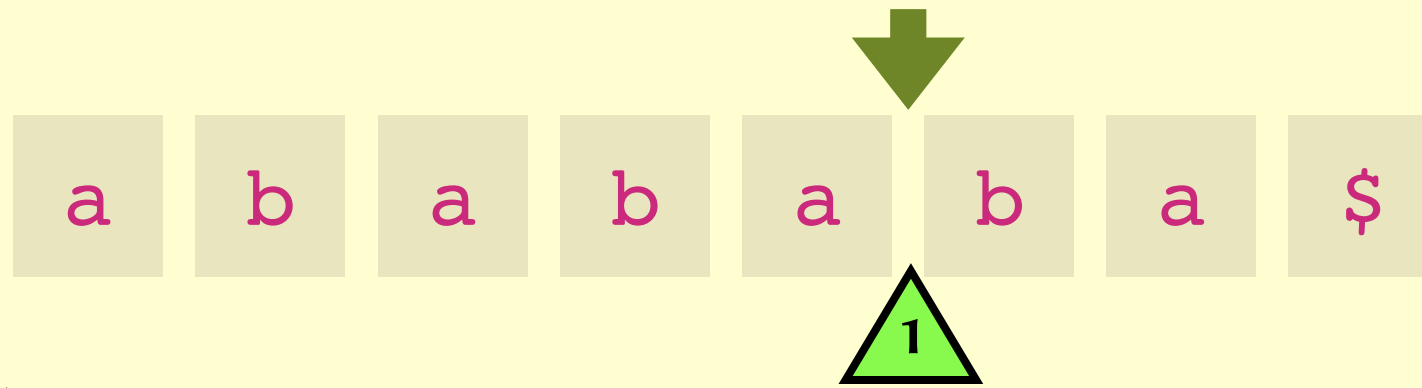


2. $(bb)^*(aa)^*$

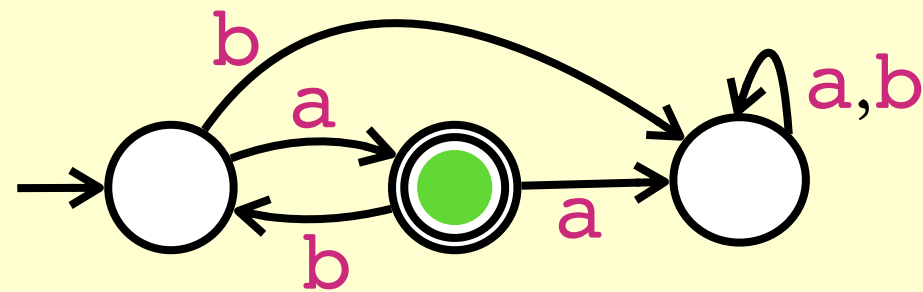


3. $a(1+b)a$

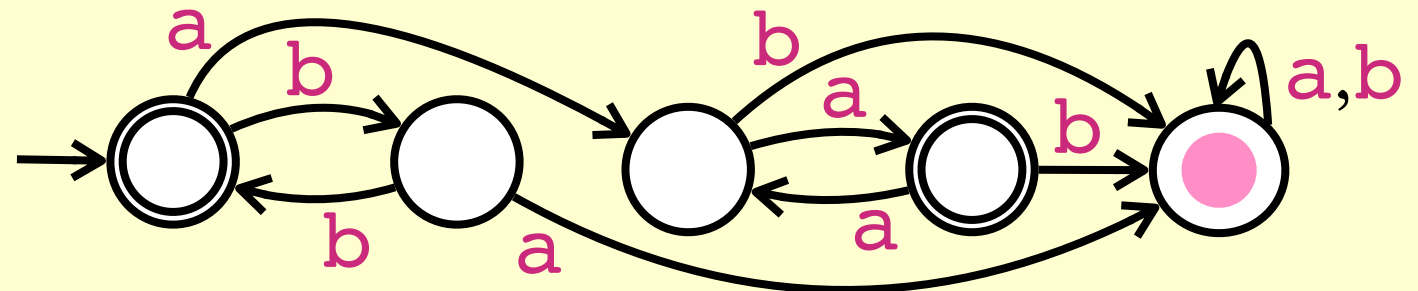




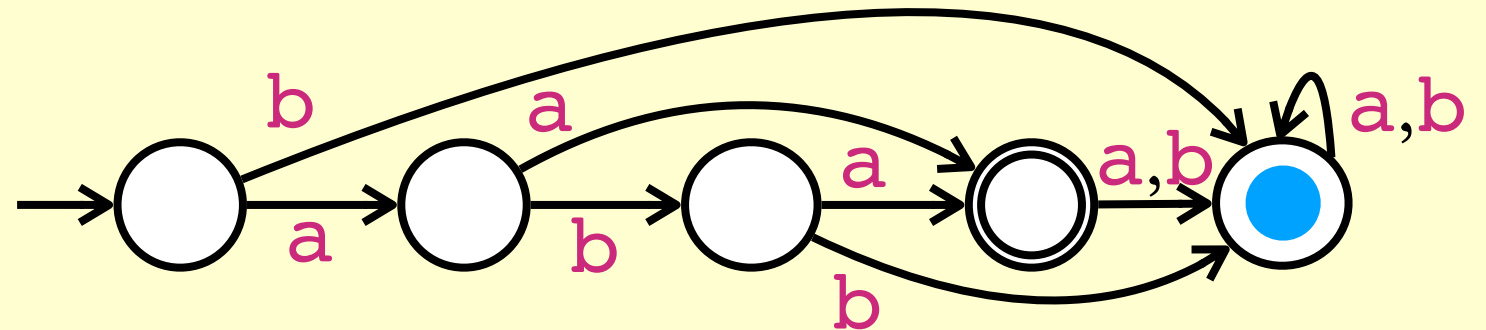
1. $(ab)^*a$

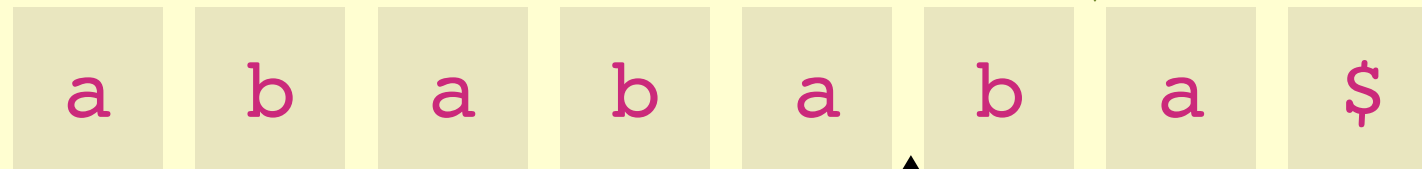


2. $(bb)^*(aa)^*$

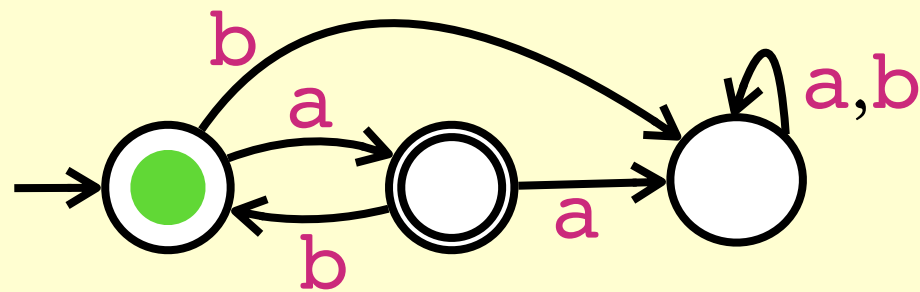


3. $a(1+b)a$

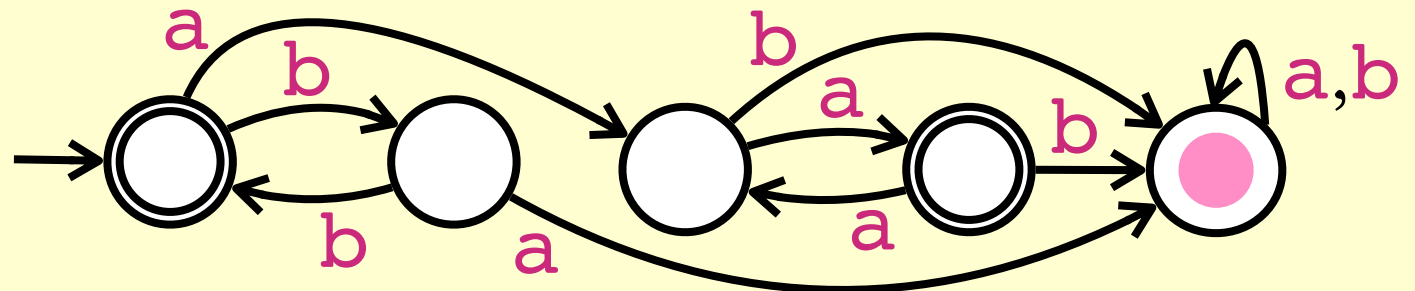




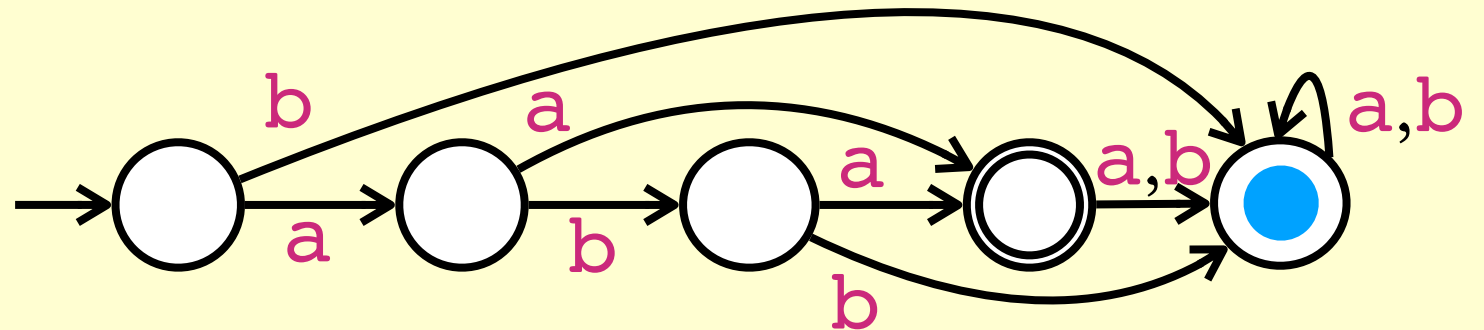
1. $(ab)^*a$

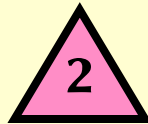
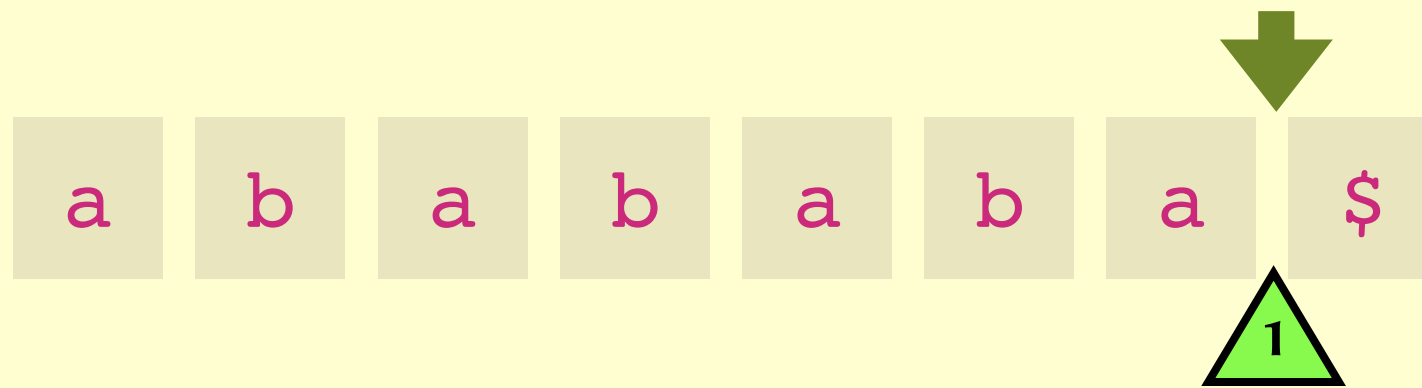


2. $(bb)^*(aa)^*$

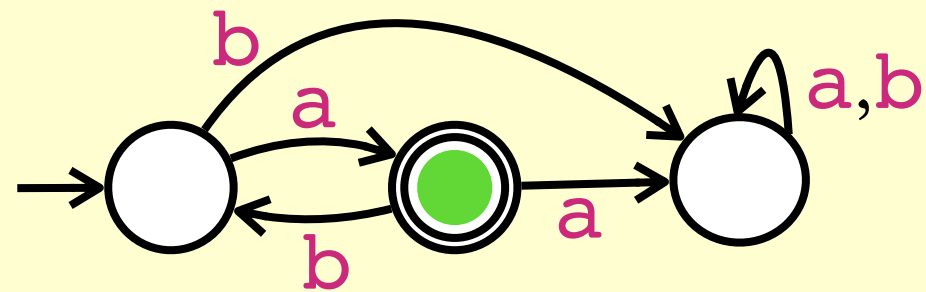


3. $a(1+b)a$

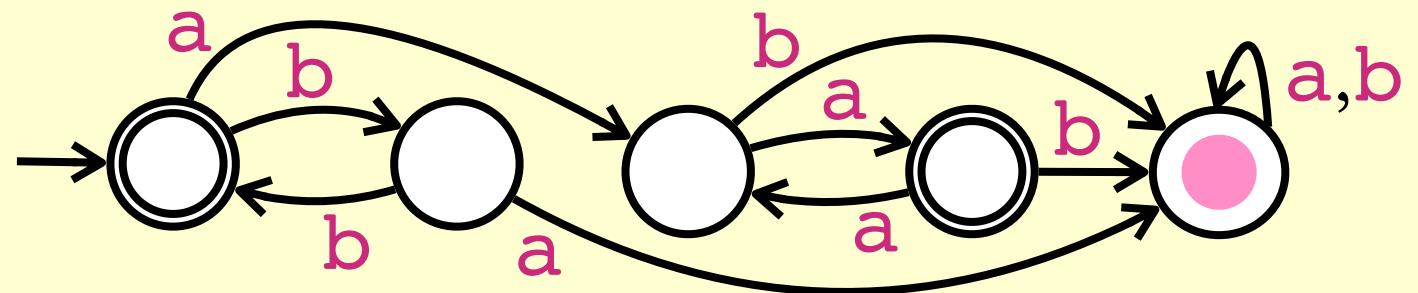




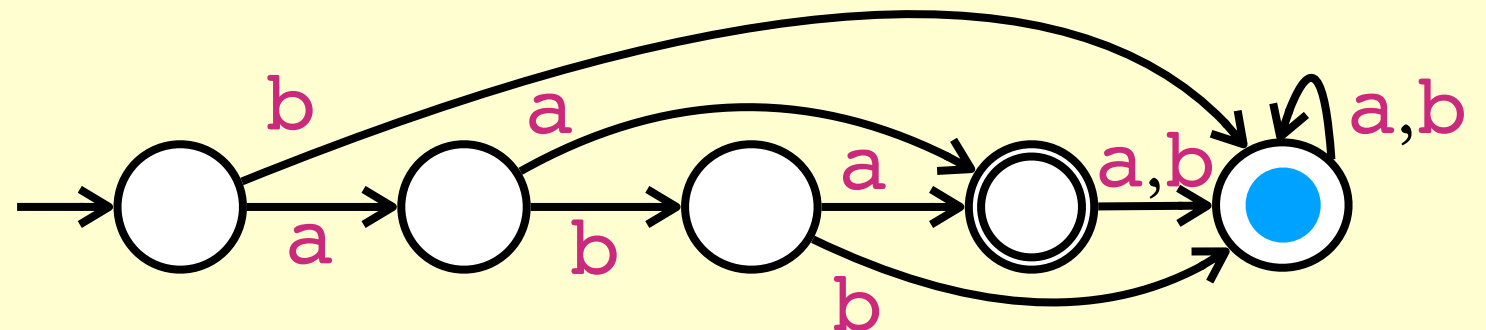
1. $(ab)^*a$



2. $(bb)^*(aa)^*$

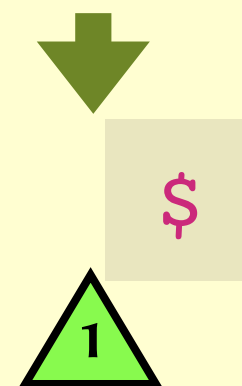


3. $a(1+b)a$

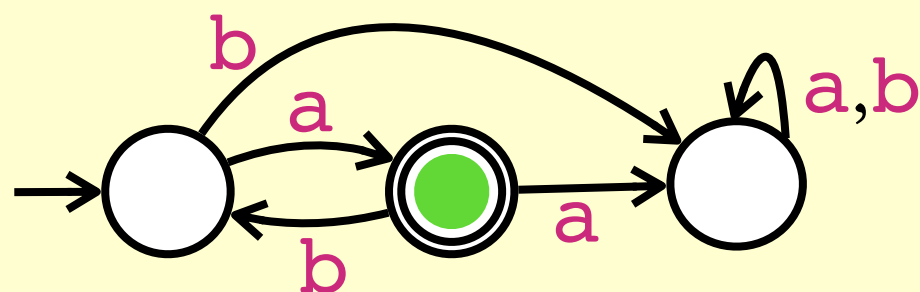


2

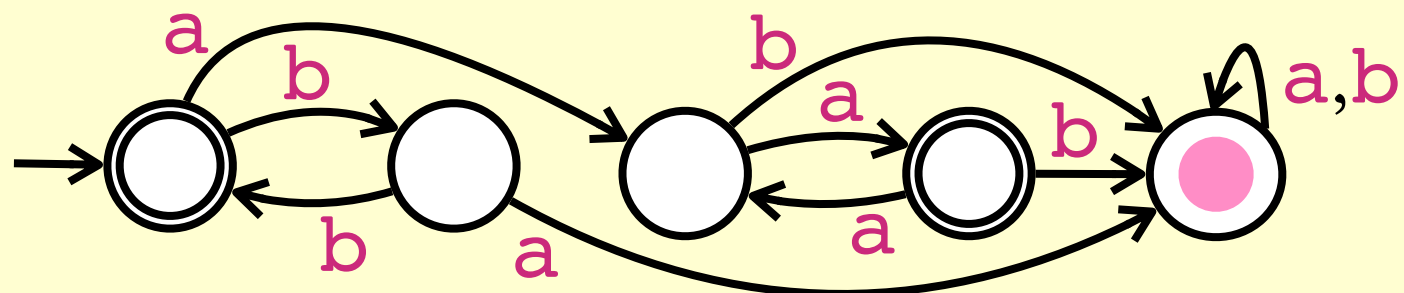
3



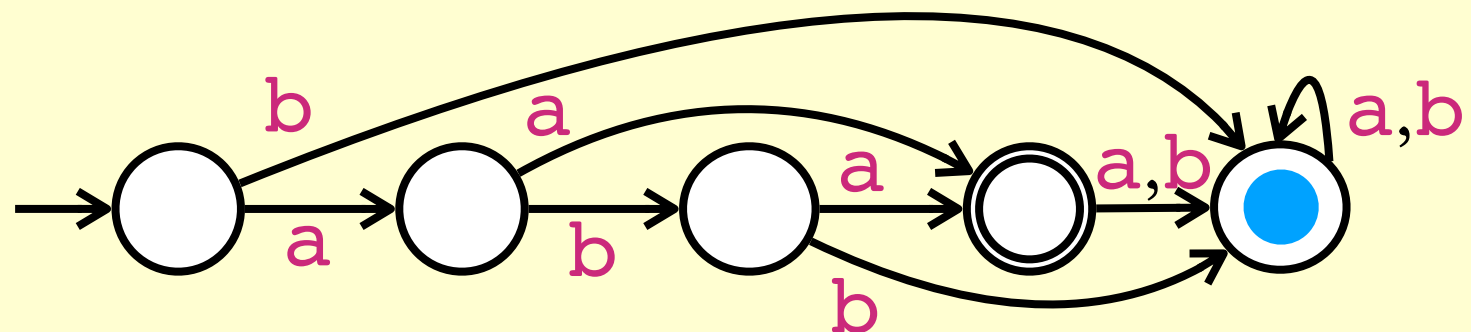
1. $(ab)^*a$



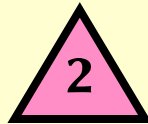
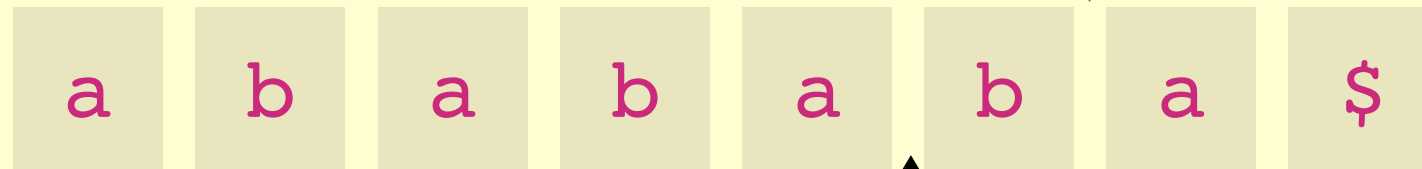
2. $(bb)^*(aa)^*$



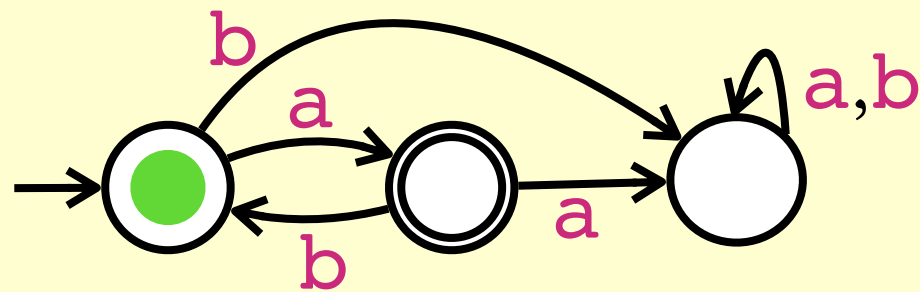
3. $a(1+b)a$



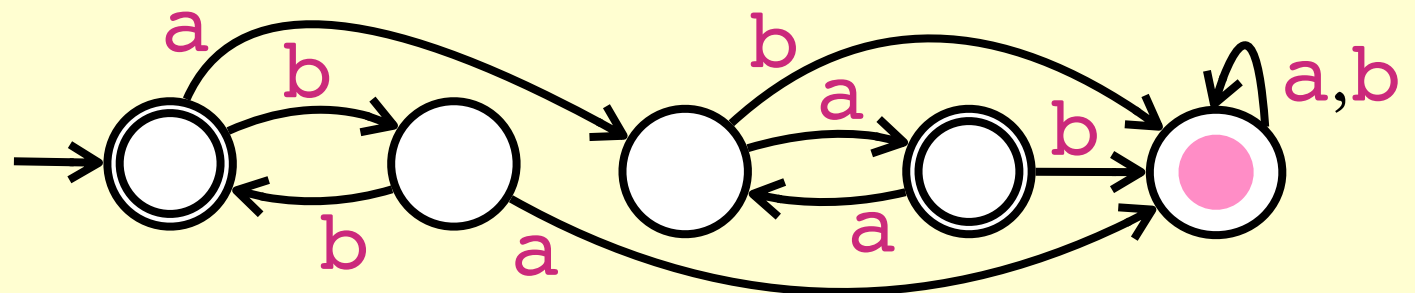
a b a b a b a



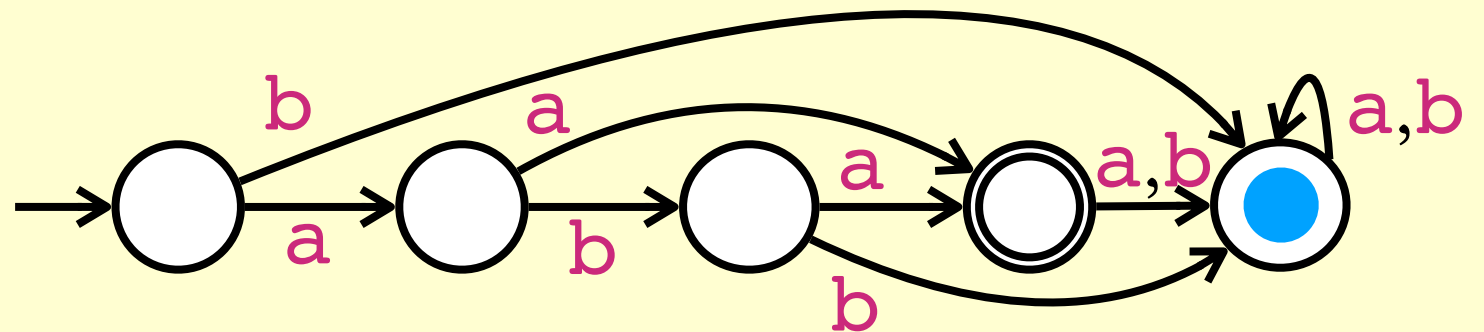
1. $(ab)^*a$

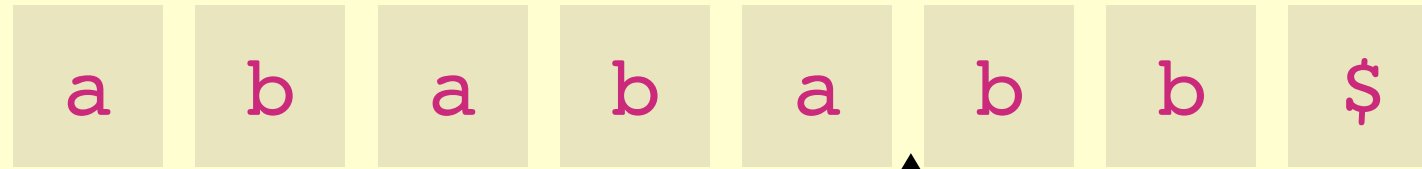


2. $(bb)^*(aa)^*$

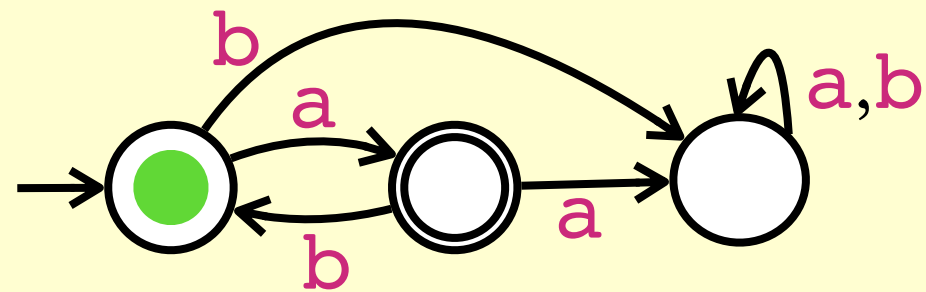


3. $a(1+b)a$

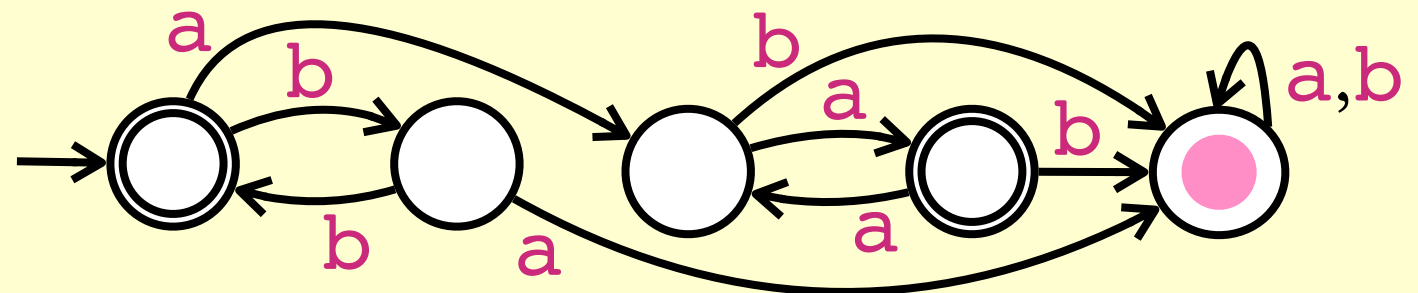




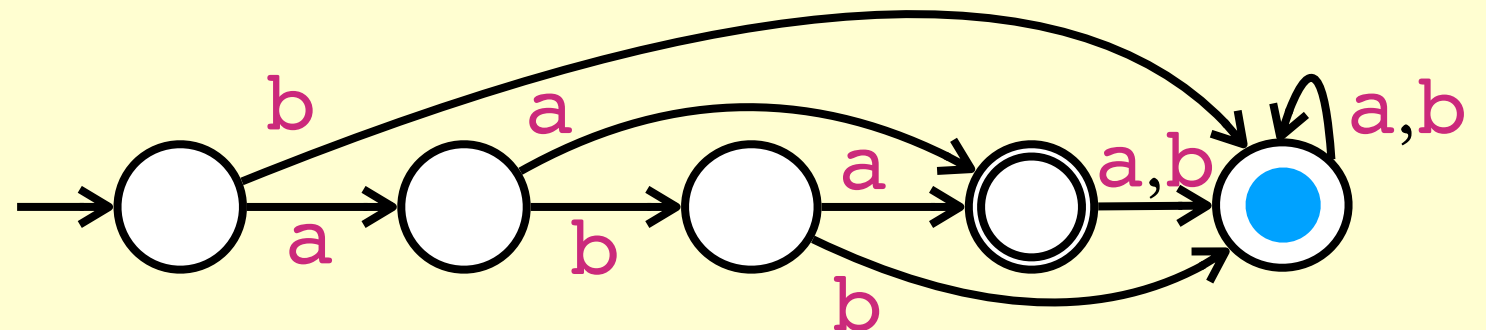
1. $(ab)^*a$

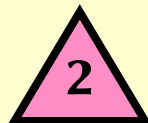
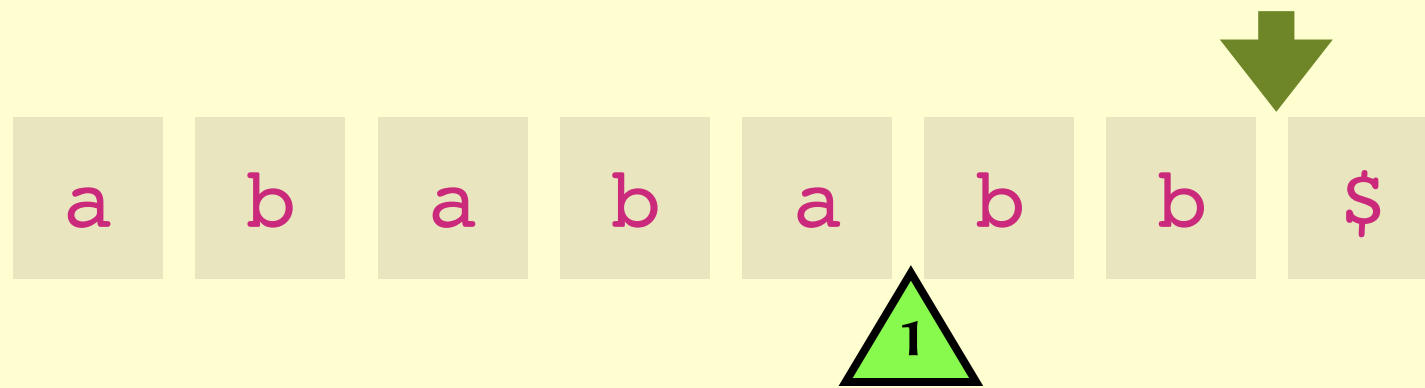


2. $(bb)^*(aa)^*$

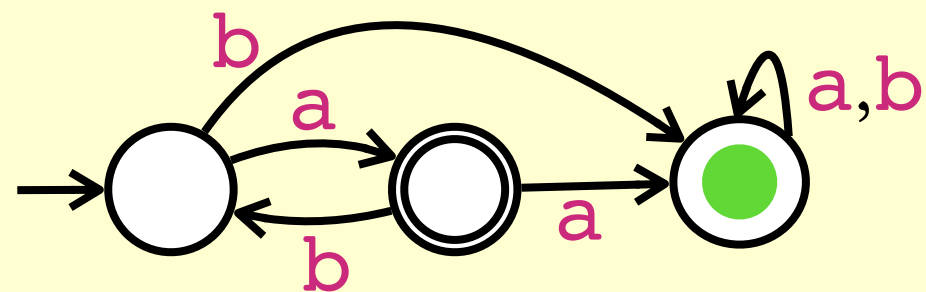


3. $a(1+b)a$

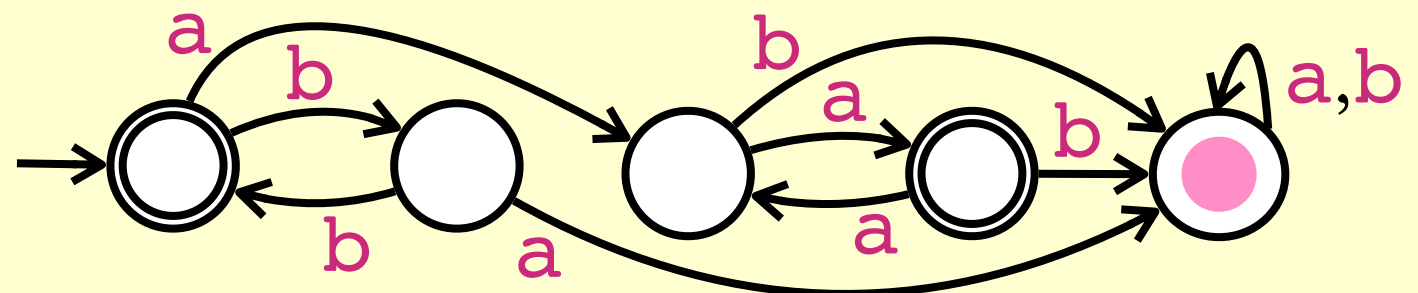




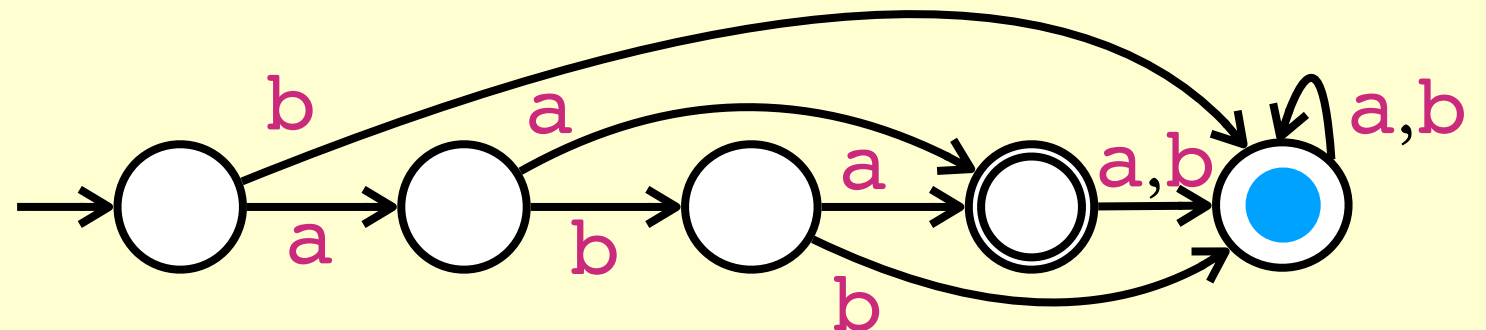
1. $(ab)^*a$

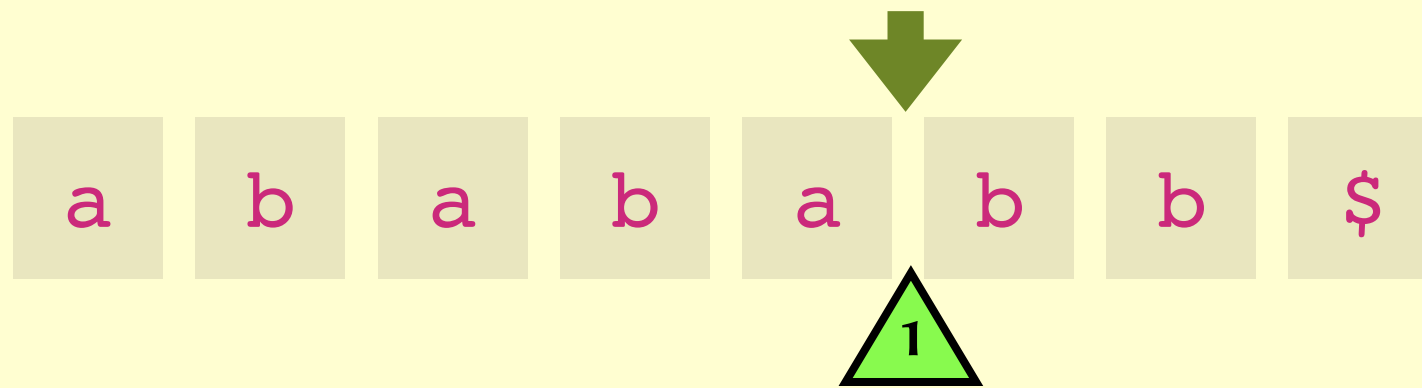


2. $(bb)^*(aa)^*$

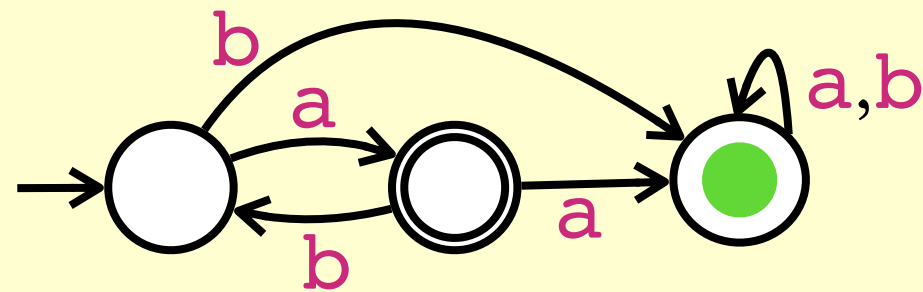


3. $a(1+b)a$

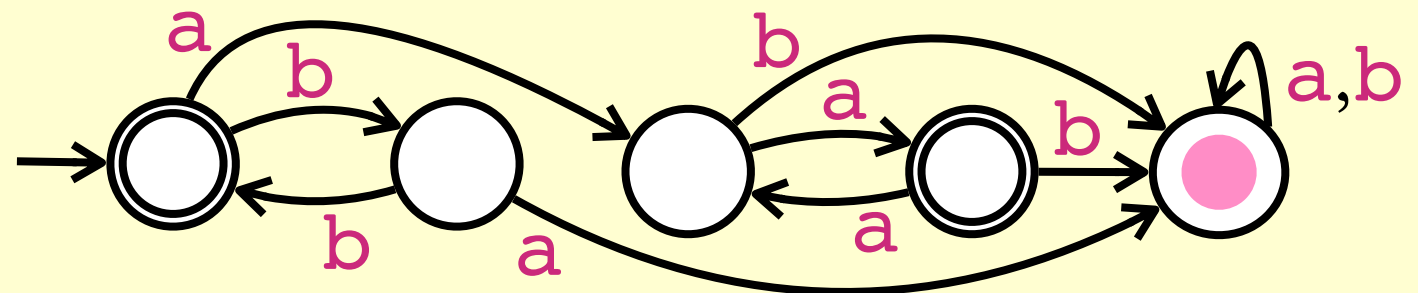




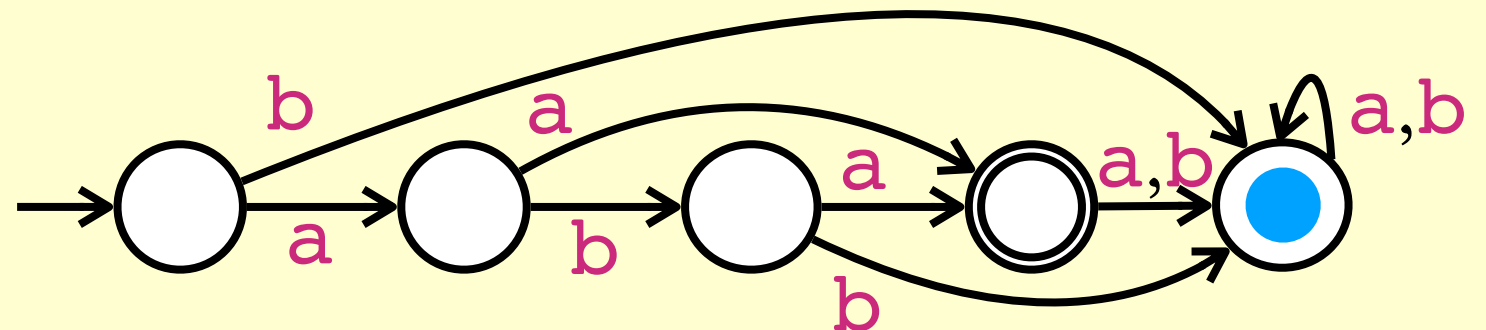
1. $(ab)^*a$



2. $(bb)^*(aa)^*$



3. $a(1+b)a$



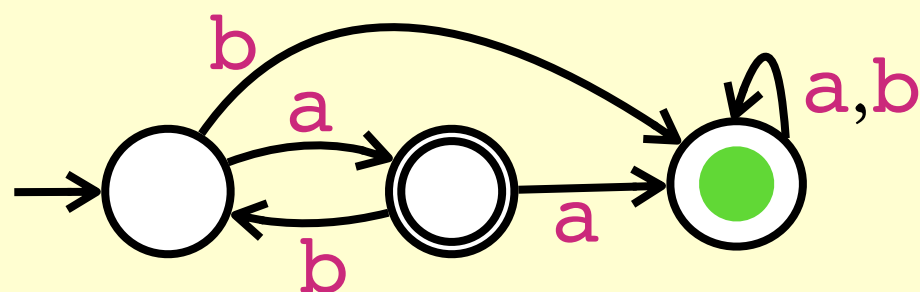
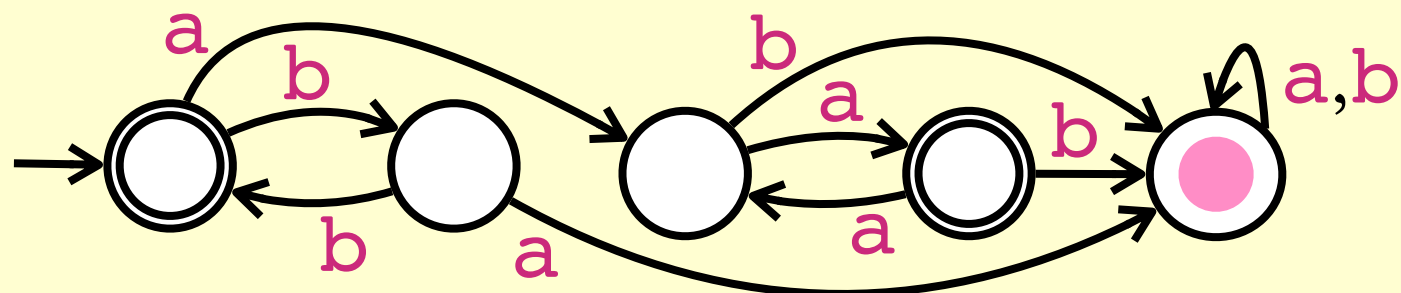
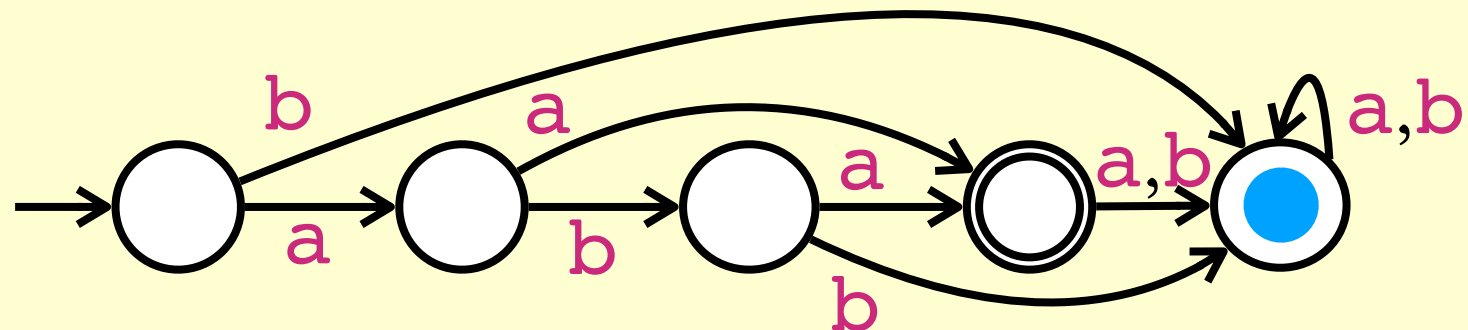
2

3

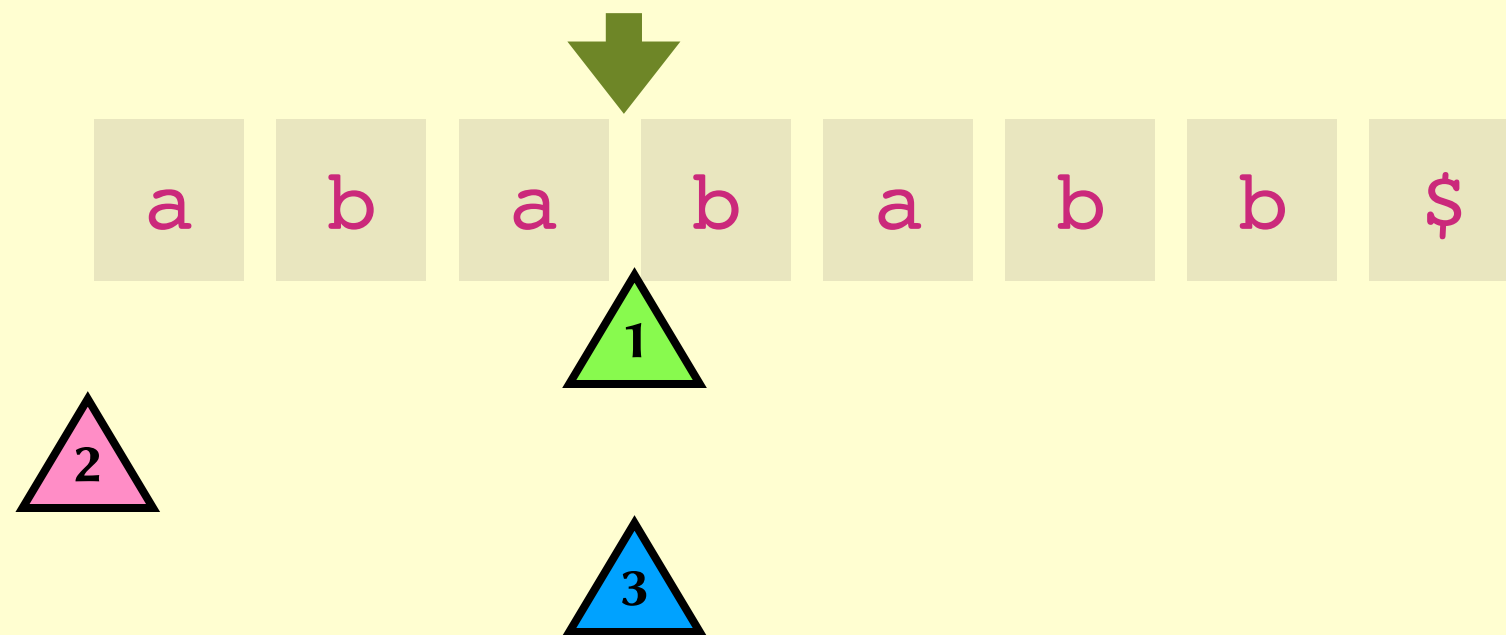


b b \$

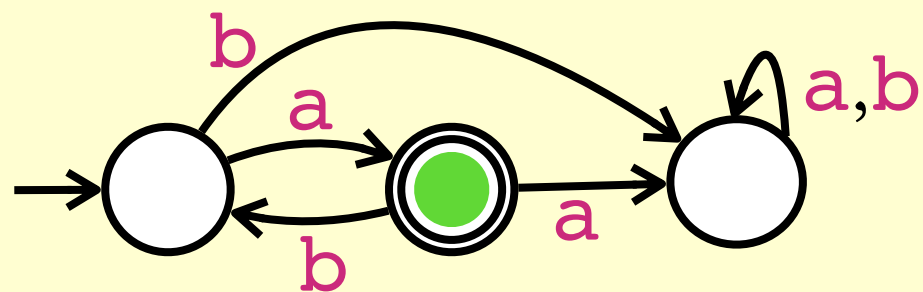
1

1. $(ab)^*a$ 2. $(bb)^*(aa)^*$ 3. $a(1+b)a$ 

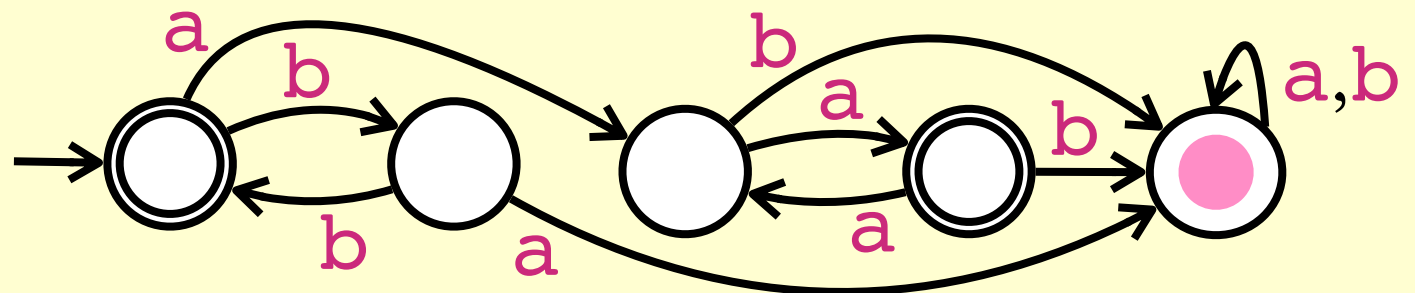
a b a b a



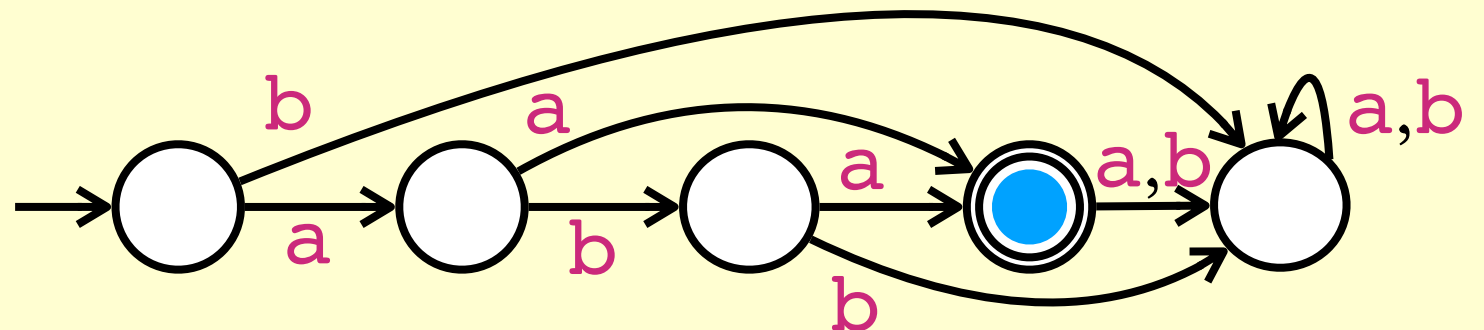
1. $(ab)^*a$

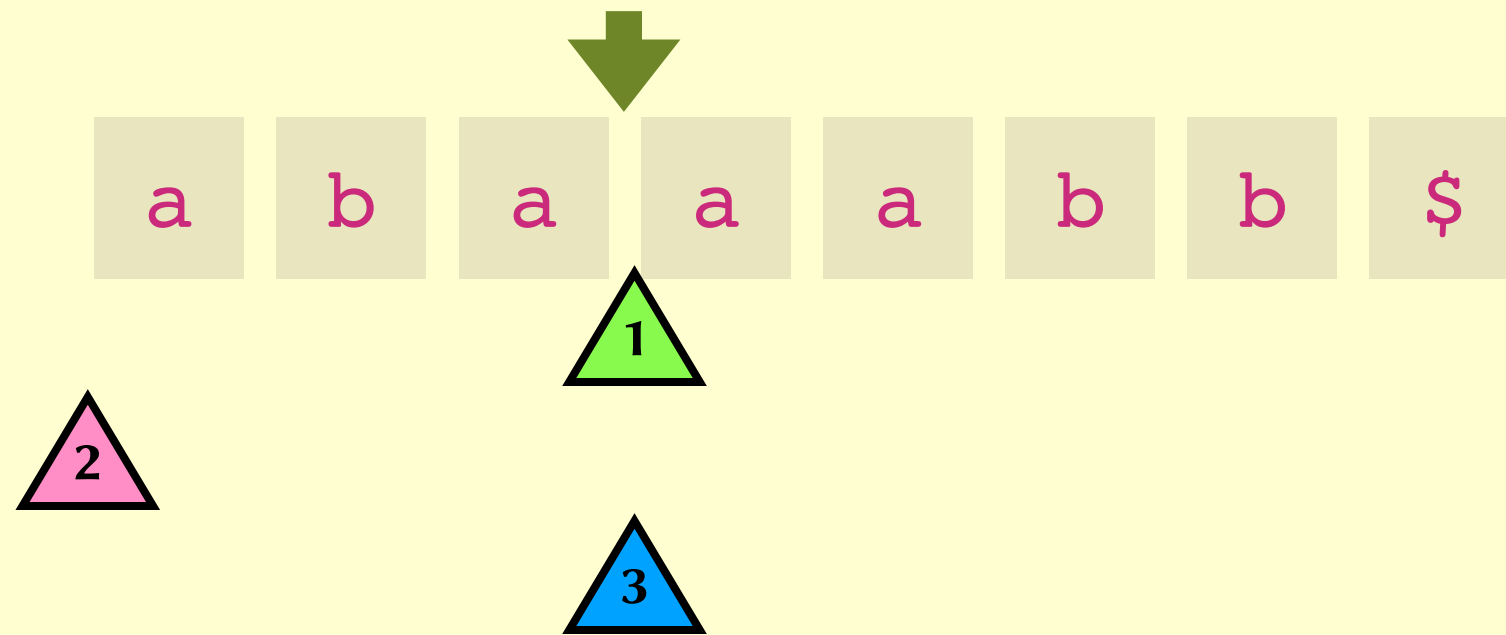


2. $(bb)^*(aa)^*$

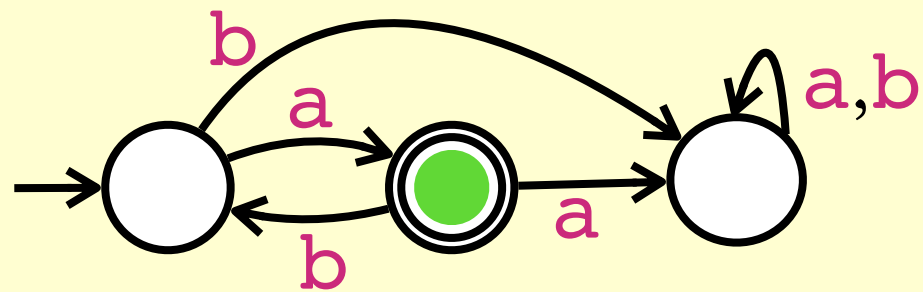


3. $a(1+b)a$

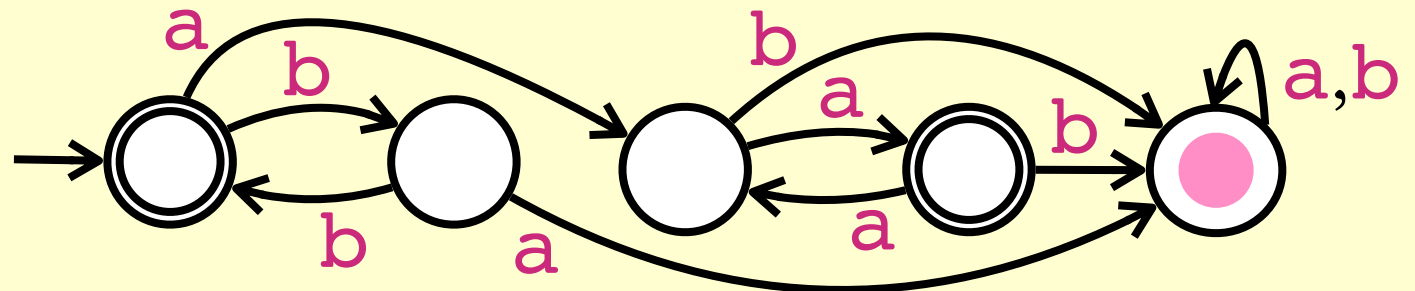




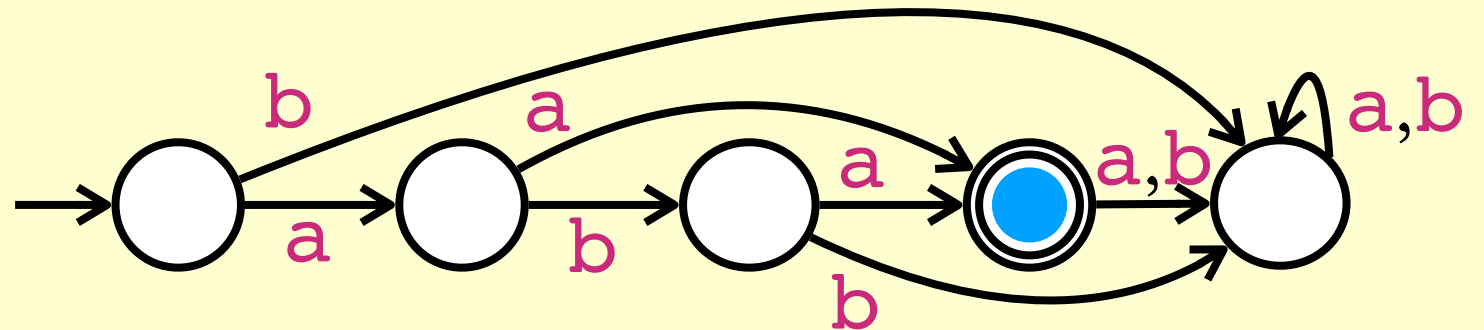
1. $(ab)^*a$

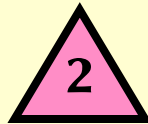
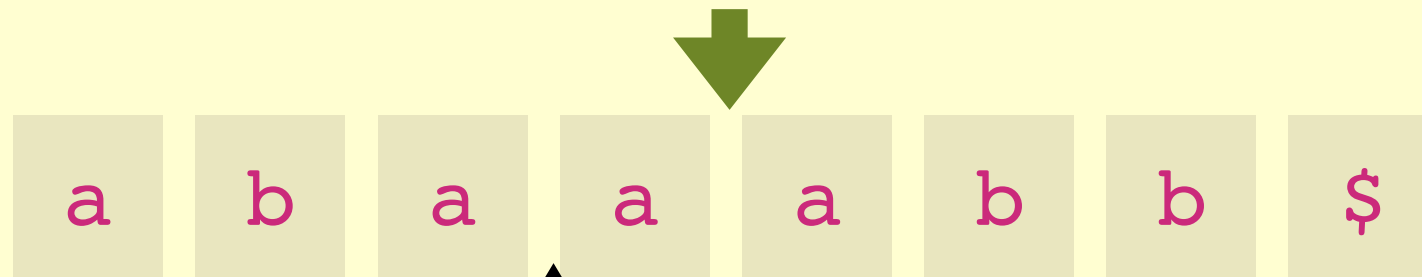


2. $(bb)^*(aa)^*$

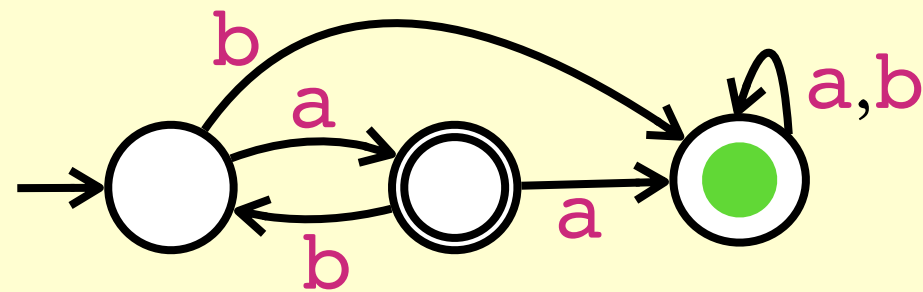


3. $a(1+b)a$

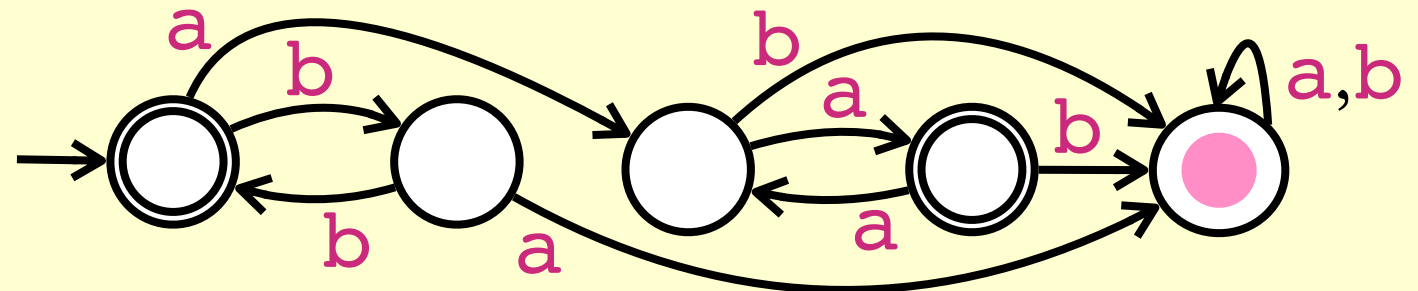




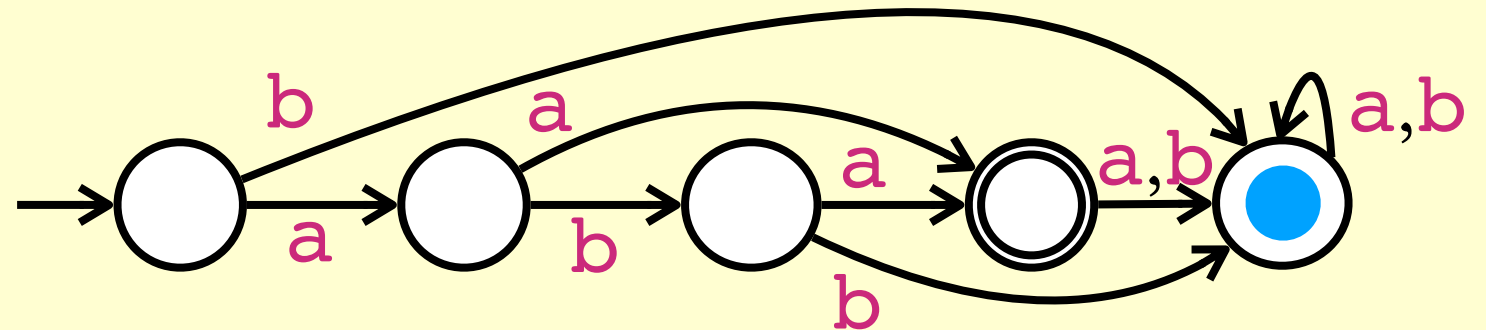
1. $(ab)^*a$

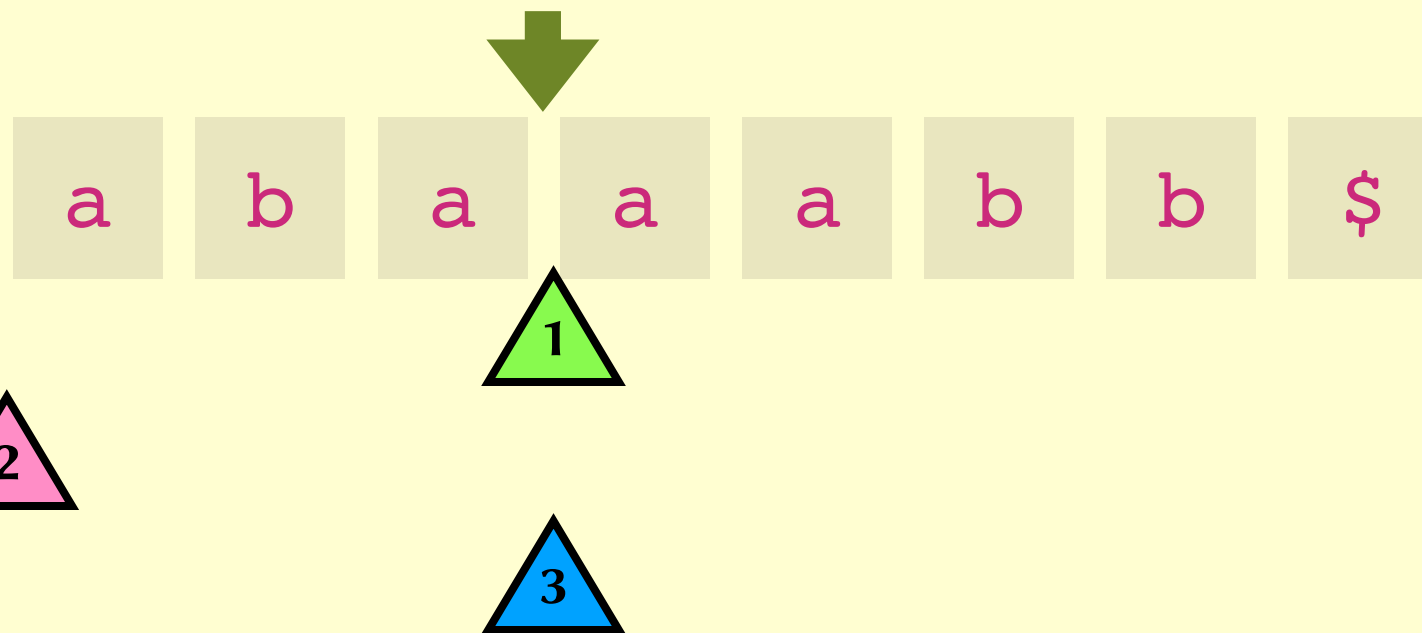


2. $(bb)^*(aa)^*$

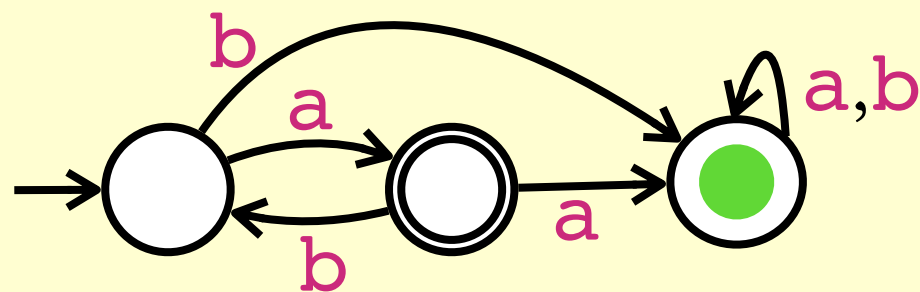


3. $a(1+b)a$

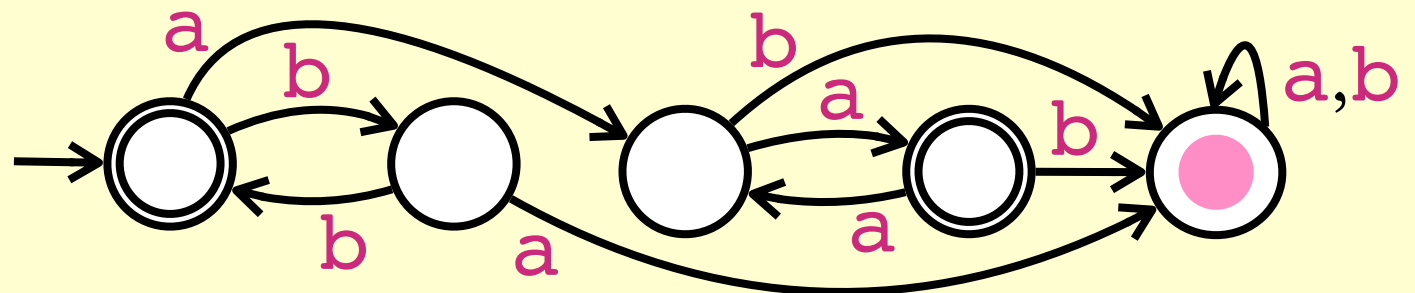




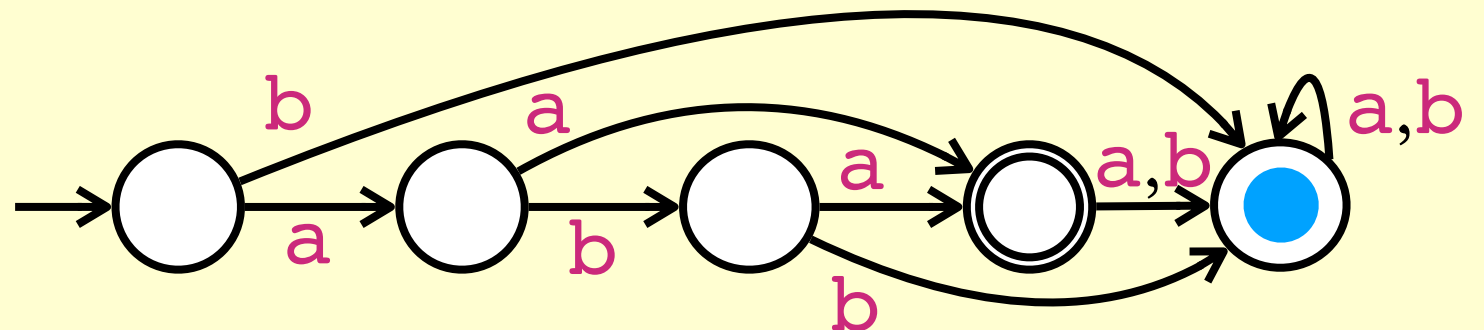
1. $(ab)^*a$

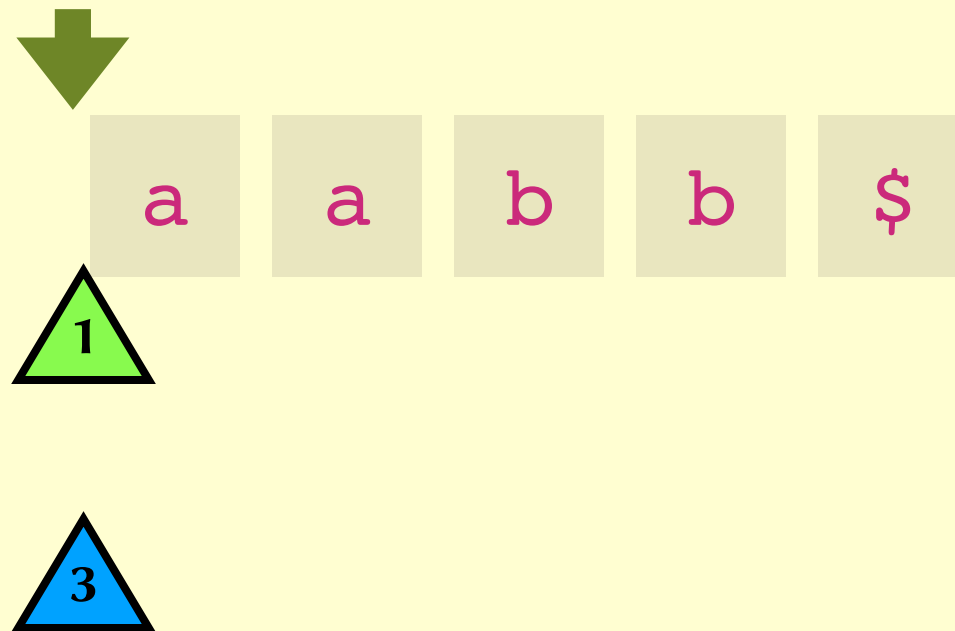


2. $(bb)^*(aa)^*$

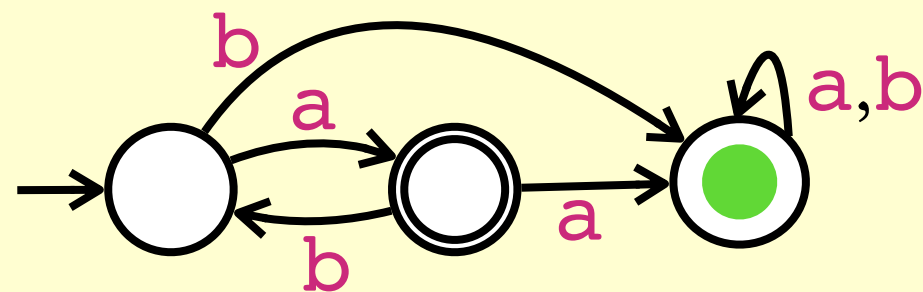


3. $a(1+b)a$

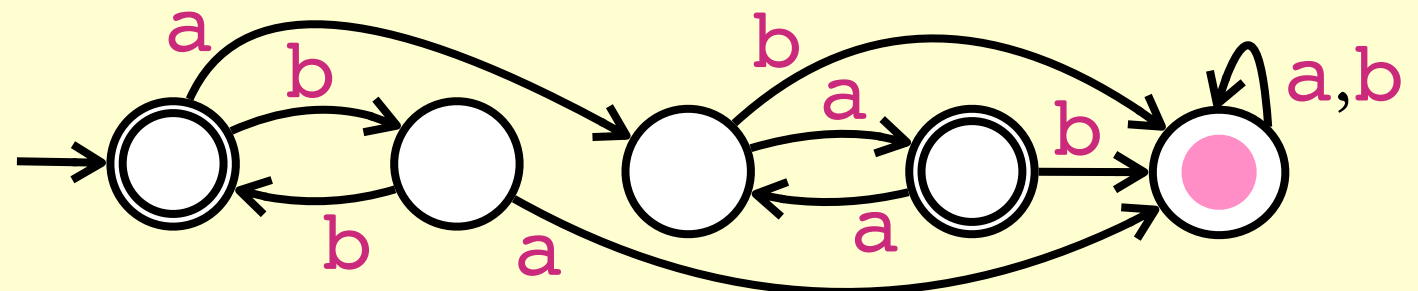




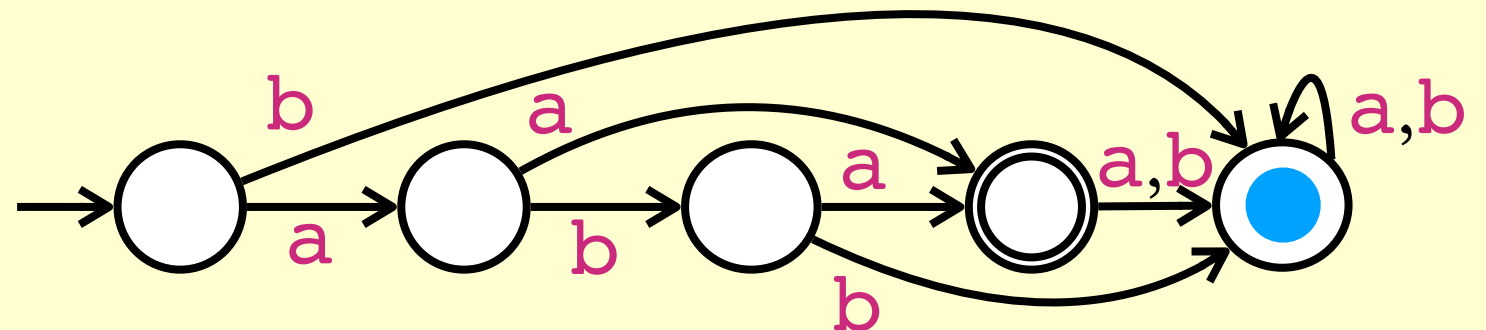
1. $(ab)^*a$



2. $(bb)^*(aa)^*$



3. $a(1+b)a$



a b a

What's next?

- Tomorrow's lecture will be on the topic of **parsing**.