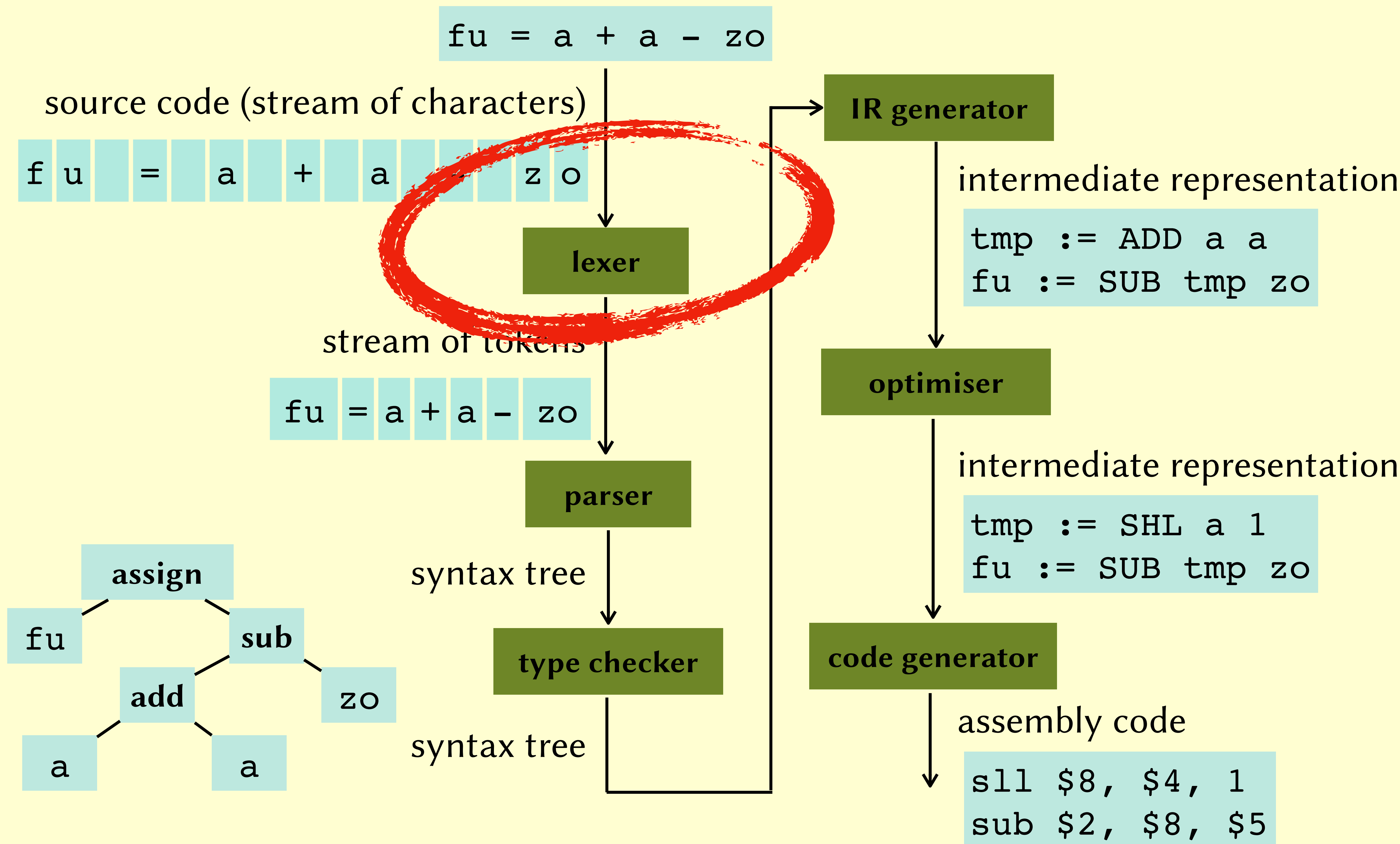
# Lecture 3: More lexing

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!

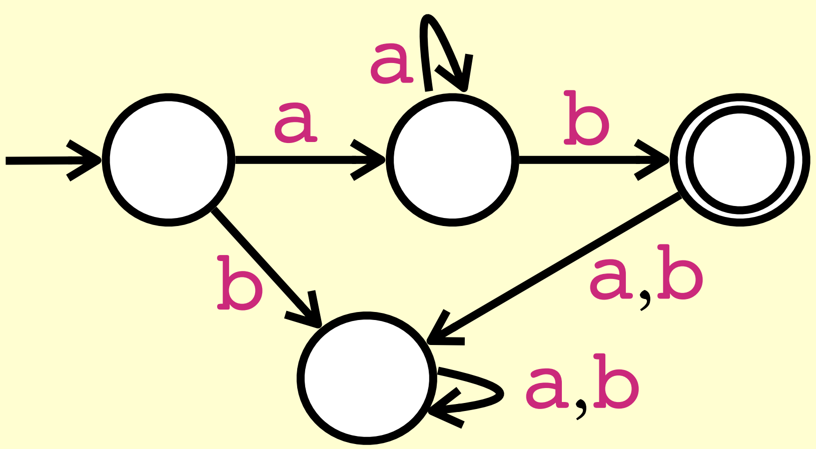
# Regexes and automata

- Does the regular expression $((\mathtt{a}+\mathtt{b})(\mathtt{ba})^*\mathtt{b}+\mathtt{a})^*$ 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:**

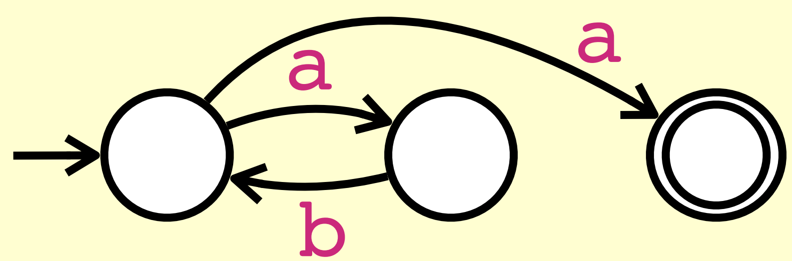
  $\mathtt{aa}^*\mathtt{b}$



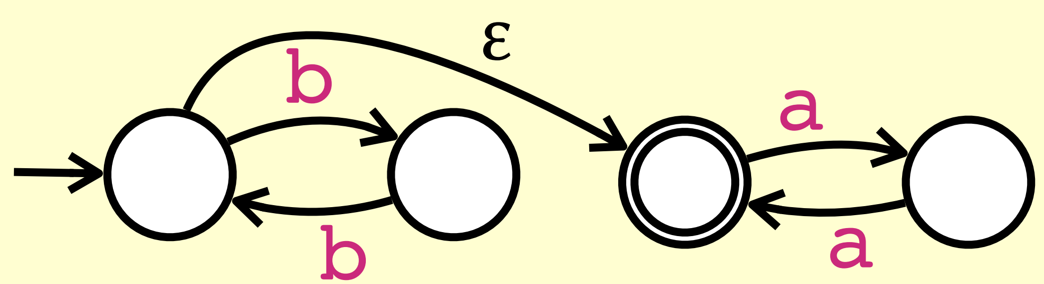

- If we can convert a regular expression into a finite automaton, we can use the automaton to check whether a word matches.

# Regexes and automata

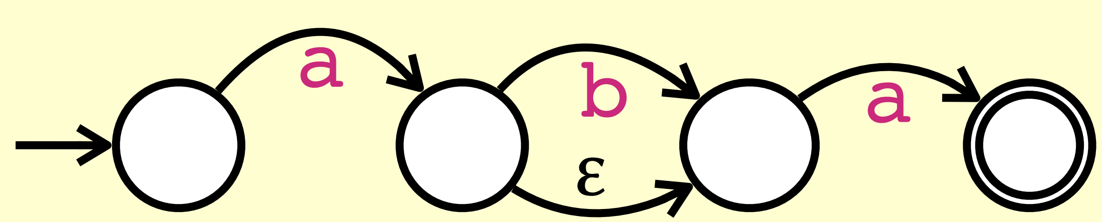
1. (ab)*a




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 |

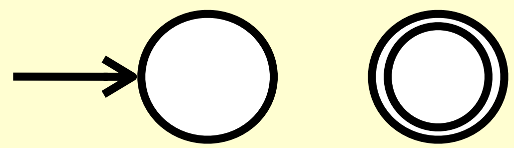
# Regex $\longrightarrow$ NFA

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

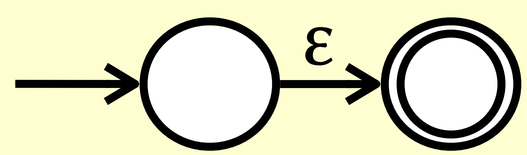
Ken Thompson
1943-

# Regex → NFA

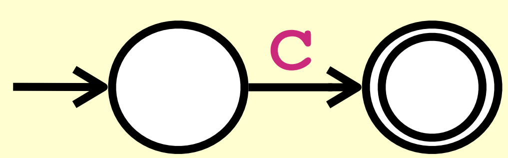
- NFA(**0**) =



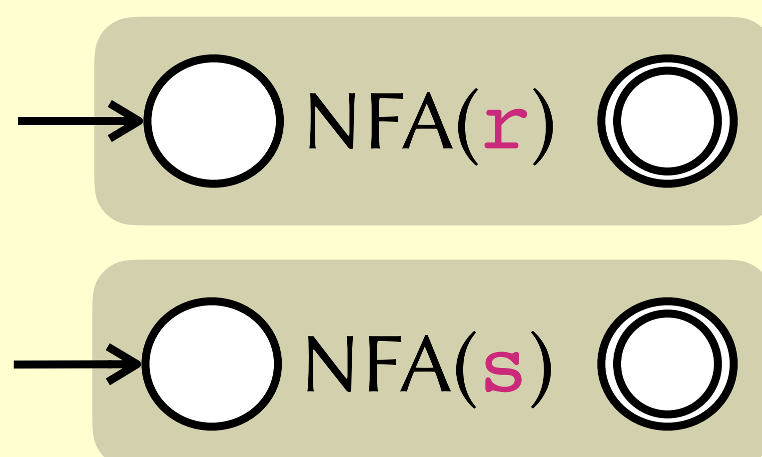
- NFA(**1**) =



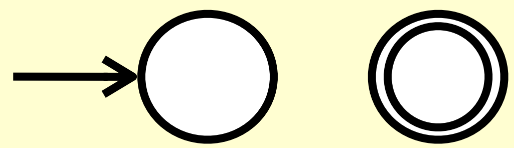
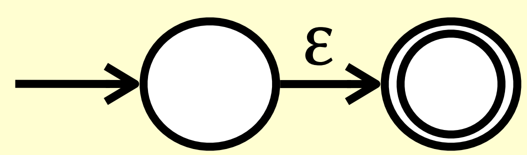

- NFA(`c`) =

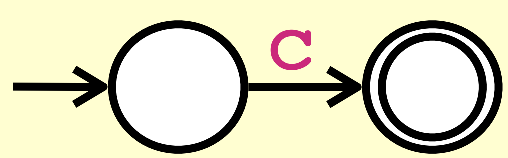


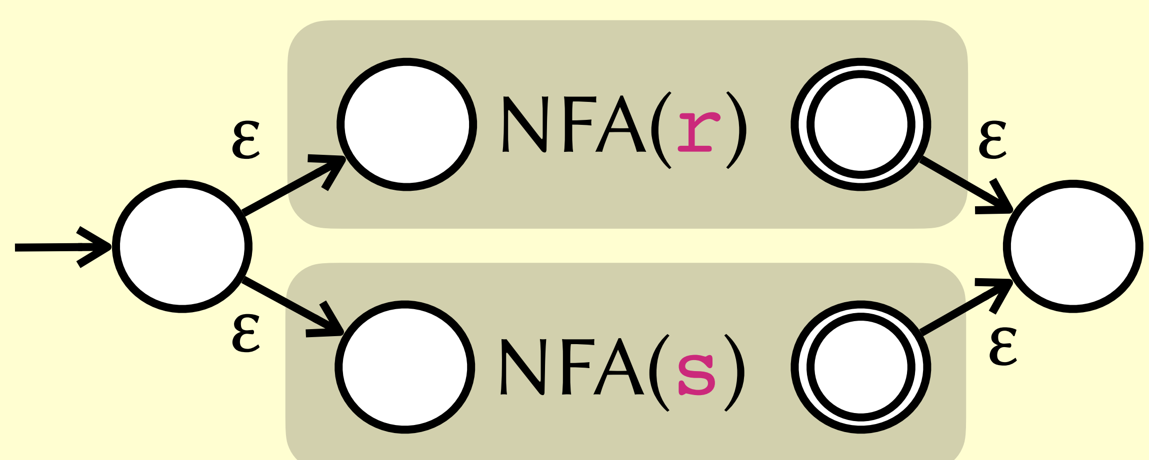

- NFA(`r`+`s`) =



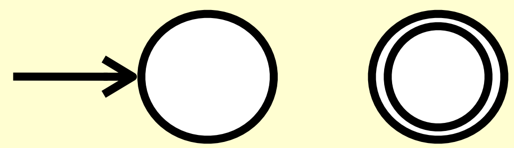
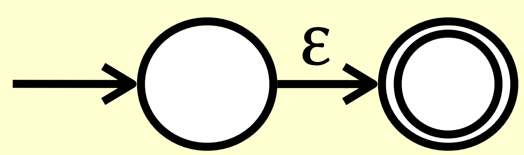
# Regex → NFA

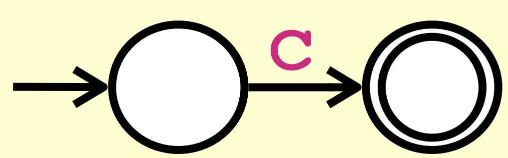
- NFA(**0**) = 
- NFA(**1**) = 

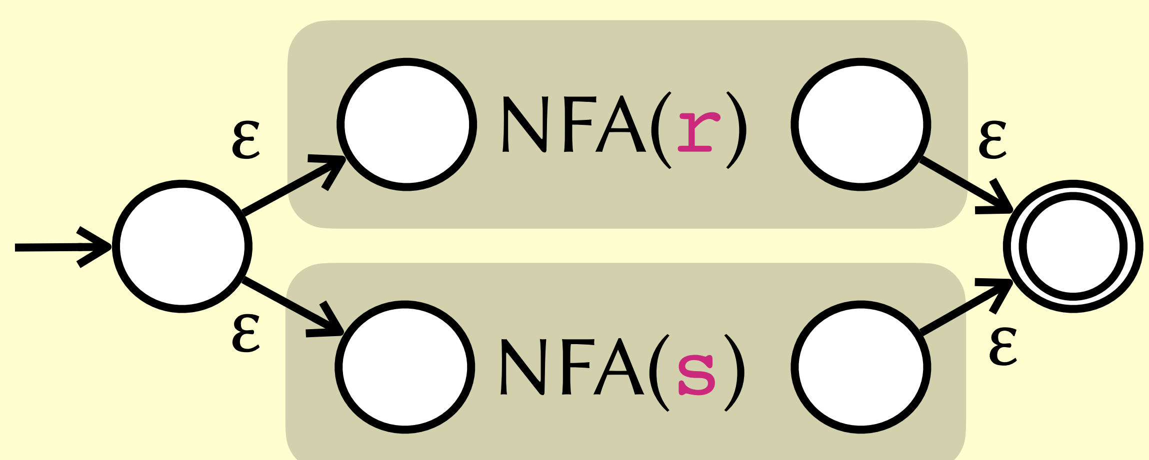
- NFA(`c`) = 

- NFA(`r`+`s`) = 

# Regex → NFA

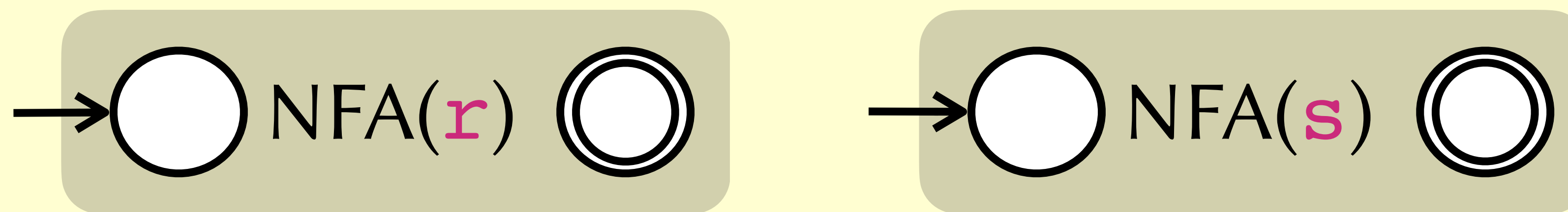
- NFA(**0**) = 
- NFA(**1**) = 

- NFA(`c`) = 

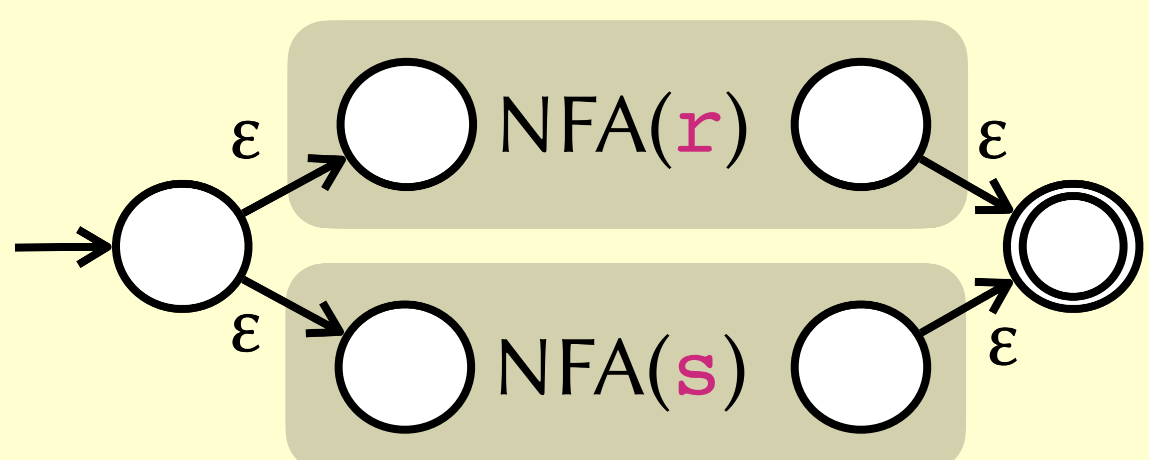
- NFA(`r`+`s`) = 

# Regex → NFA

- NFA(rs) =

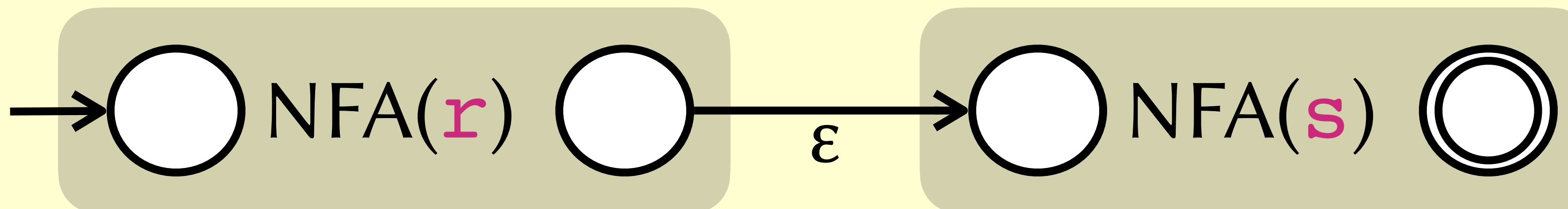



- NFA(r+s) =



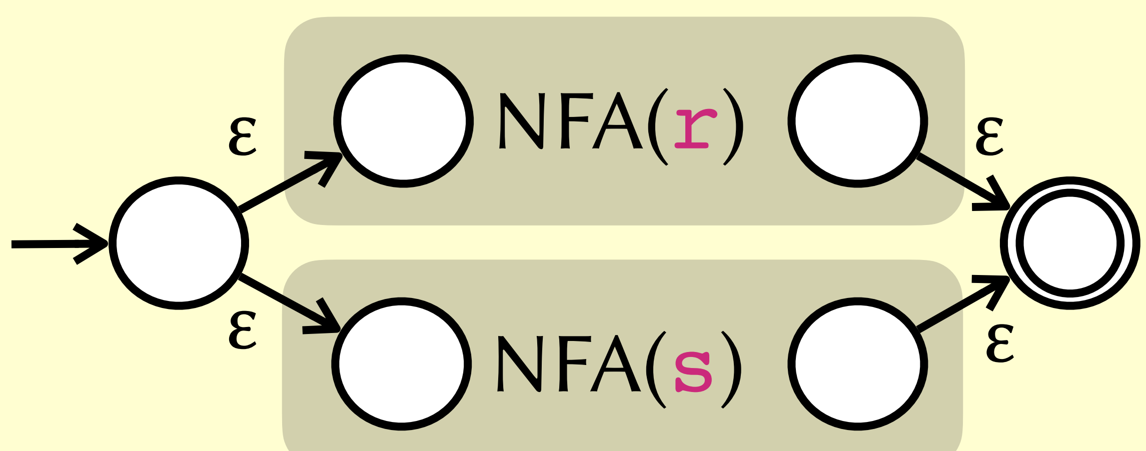
# Regex → NFA

- NFA(`rs`) =



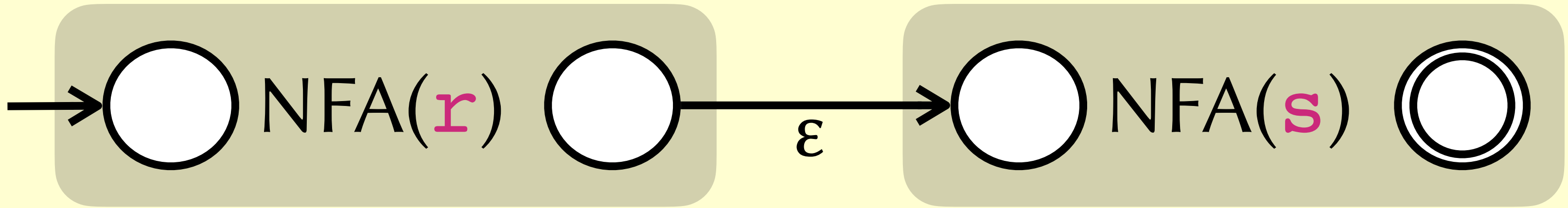

- NFA(`r`*) =


- NFA(`r`+`s`) =

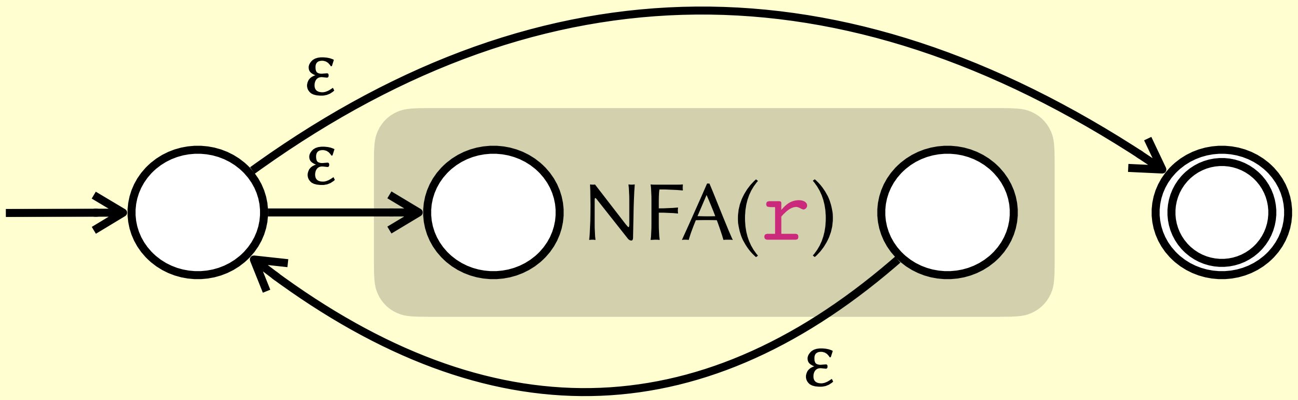


# Regex → NFA

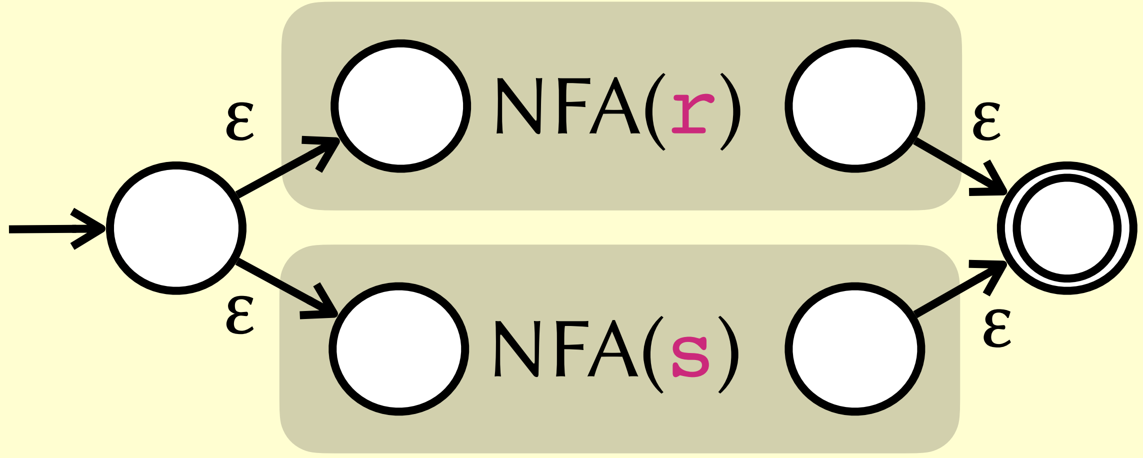
- NFA(`rs`) =




- NFA(`r`*) =

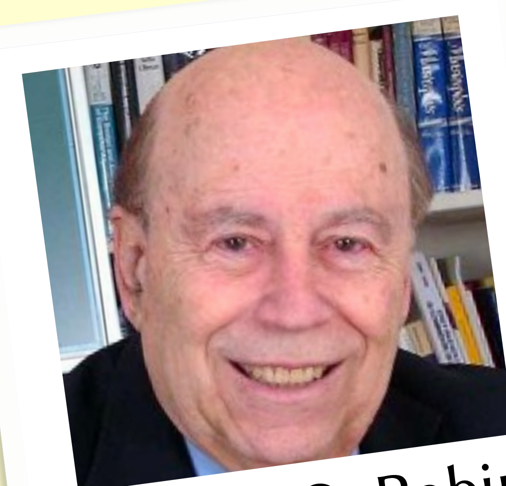



- NFA(`r`+`s`) =



# NFAs and DFAs

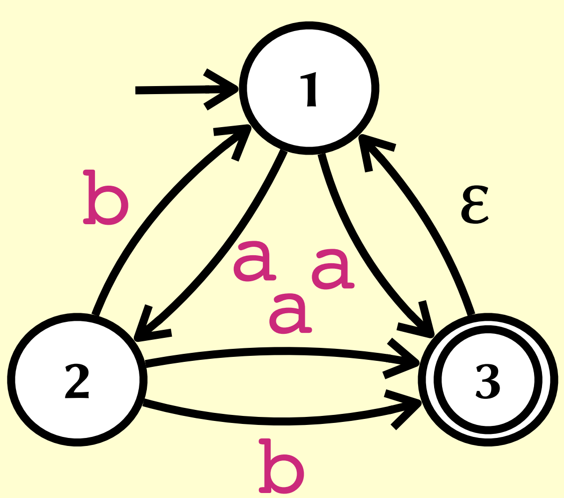
- These automata are non-deterministic, so not very helpful – we have to keep track of lots of states at once!
- However, we can convert a **non-deterministic** automaton into a **deterministic** one using the **subset construction**.

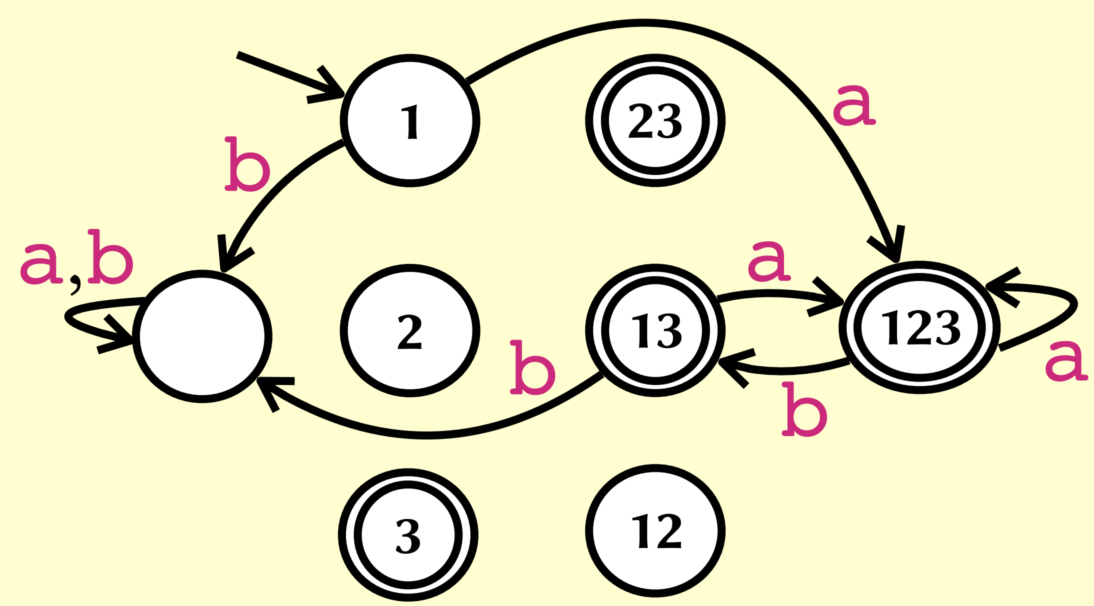

Michael O. Rabin
1931-


Dana Scott
1932-

# NFA → 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 { $s_1$, $s_2$, ... } into a DFA whose states are **all subsets** of { $s_1$, $s_2$, ... }.
- **Example.**

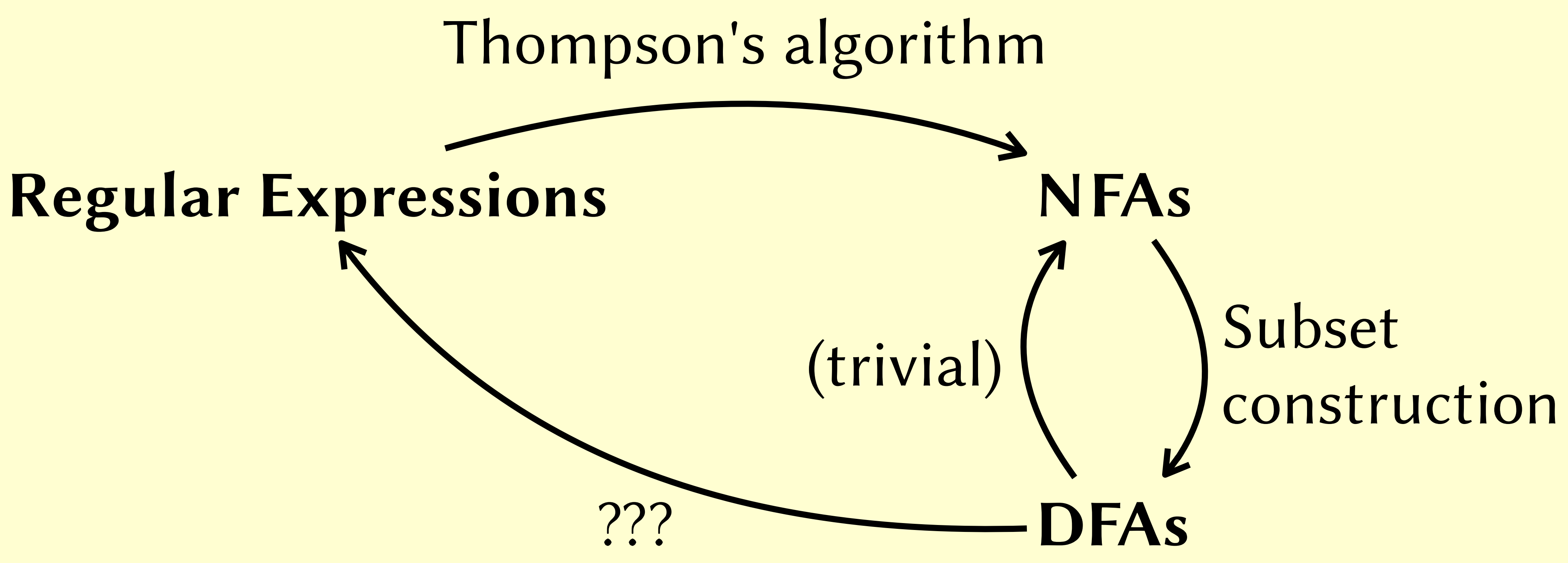


# Summary

- 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



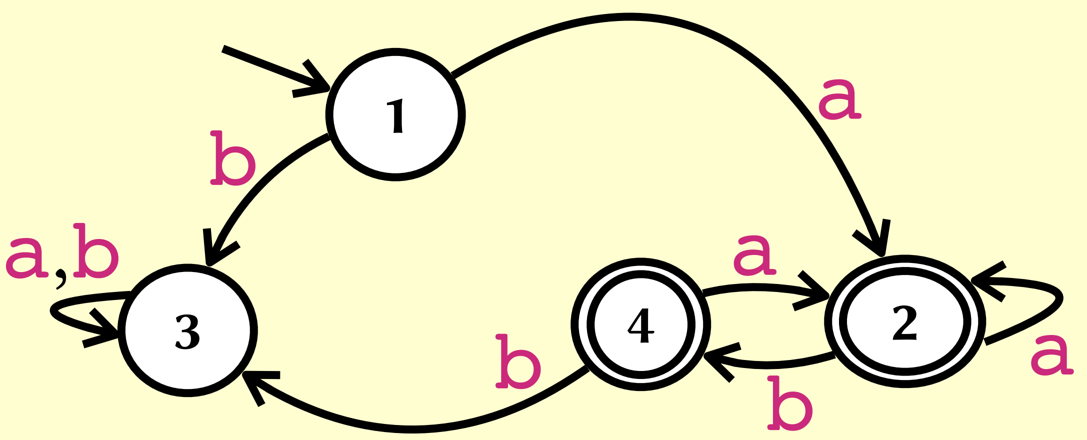
# DFA → Regex

$$R_1 = \mathtt{a}R_2 + \mathtt{b}R_3$$

$$R_2 = \mathtt{a}R_2 + \mathtt{b}R_4 + \mathbf{1}$$

$$R_3 = \mathtt{a}R_3 + \mathtt{b}R_3$$

$$R_4 = \mathtt{a}R_2 + \mathtt{b}R_3 + \mathbf{1}$$

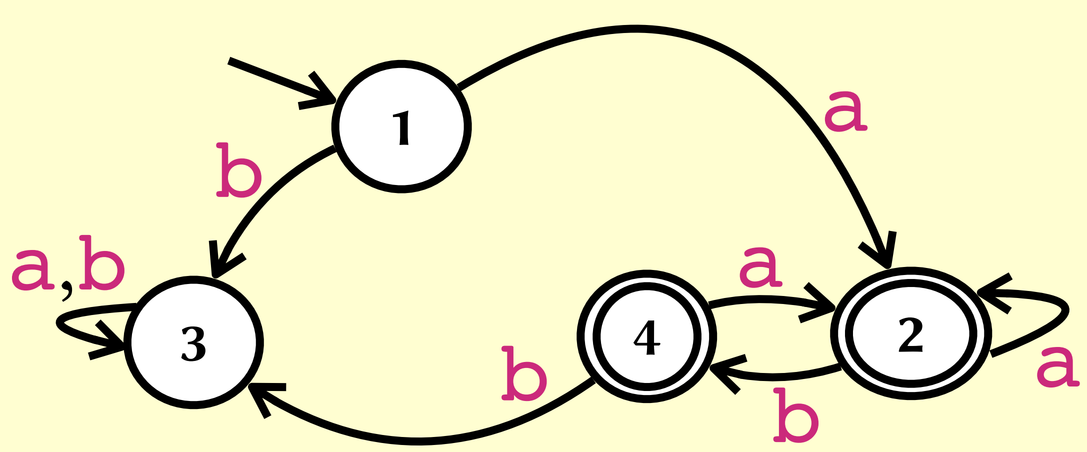


# DFA → Regex

$$R_1 = \mathtt{a}R_2 + \mathtt{b}R_3$$

$$R_2 = \mathtt{a}R_2 + \mathtt{b}(\mathtt{a}R_2 + \mathtt{b}R_3 + \mathbf{1}) + \mathbf{1}$$

$$R_3 = \mathtt{a}R_3 + \mathtt{b}R_3$$

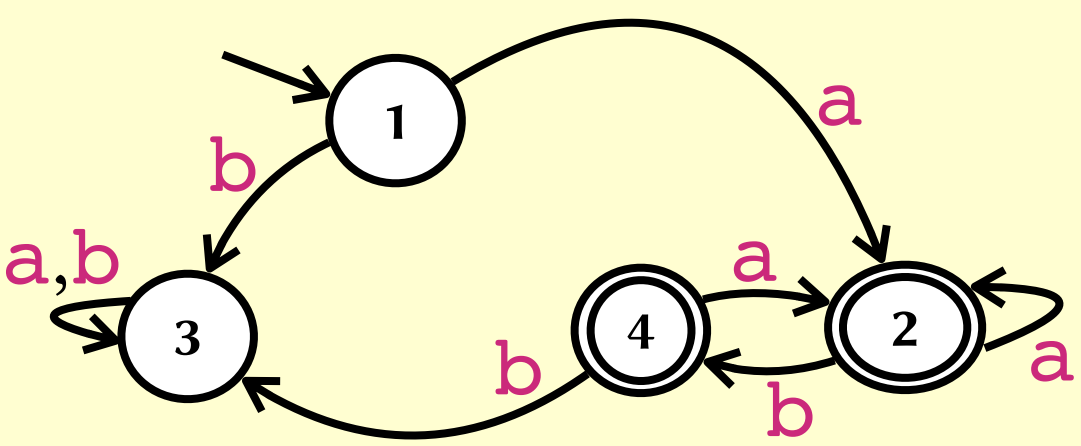


# DFA → Regex

$R_1 = \mathtt{a}R_2 + \mathtt{b}R_3$

$R_2 = \mathtt{a}R_2 + \mathtt{ba}R_2 + \mathtt{bb}R_3 + \mathtt{b}\mathbf{1} + \mathbf{1}$

$R_3 = \mathtt{a}R_3 + \mathtt{b}R_3$

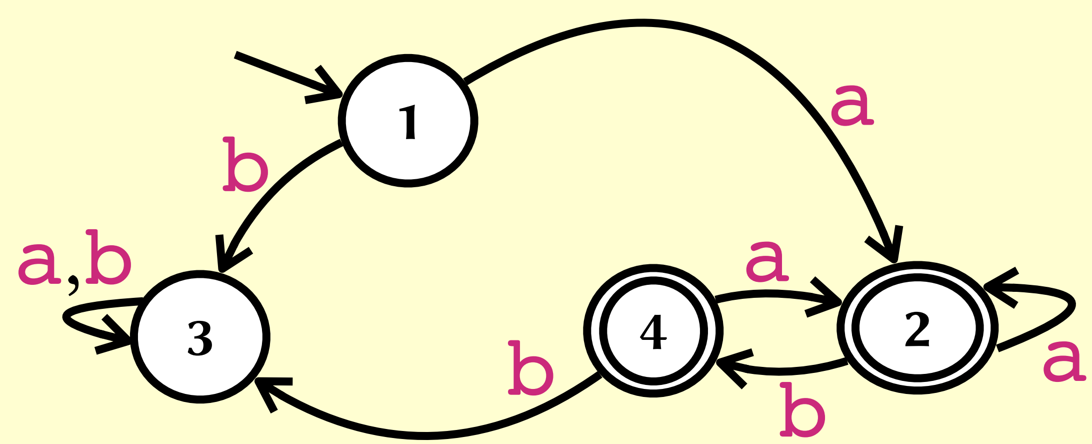


# DFA → Regex

$R_1 = \mathtt{a}R_2 + \mathtt{b}R_3$

$R_2 = \mathtt{a}R_2 + \mathtt{ba}R_2 + \mathtt{bb}R_3 + \mathtt{b} + \mathbf{1}$

$R_3 = \mathtt{a}R_3 + \mathtt{b}R_3$

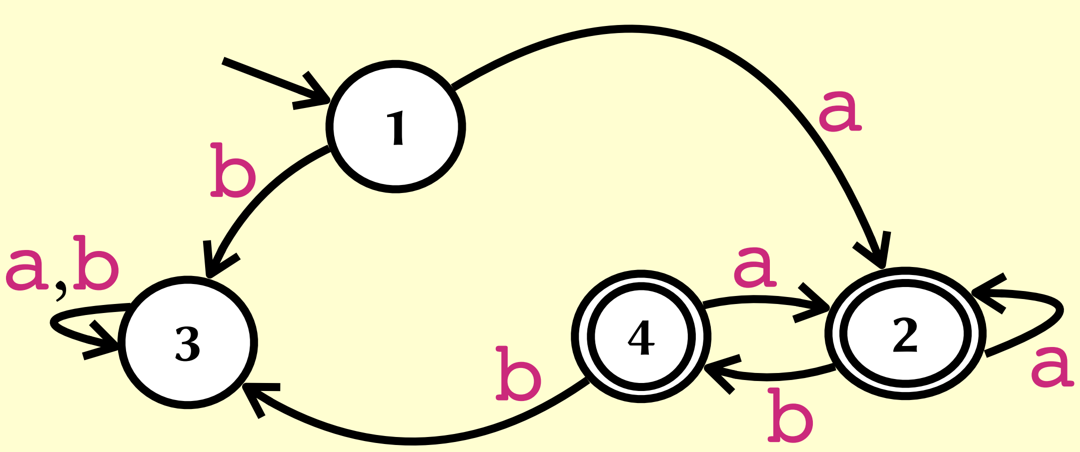


# DFA → Regex

$R_1 = \mathtt{a}R_2 + \mathtt{b}R_3$

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

$R_3 = \mathtt{a}R_3 + \mathtt{b}R_3$

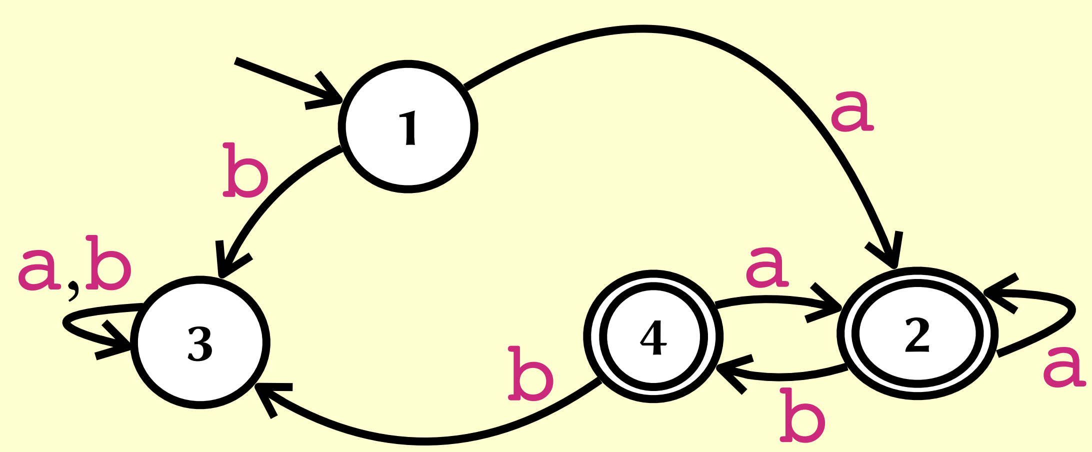


# DFA $\longrightarrow$ Regex

$$R_1 = \mathtt{a}R_2 + \mathtt{b}R_3$$

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

$$R_3 = (\mathtt{a} + \mathtt{b})R_3$$

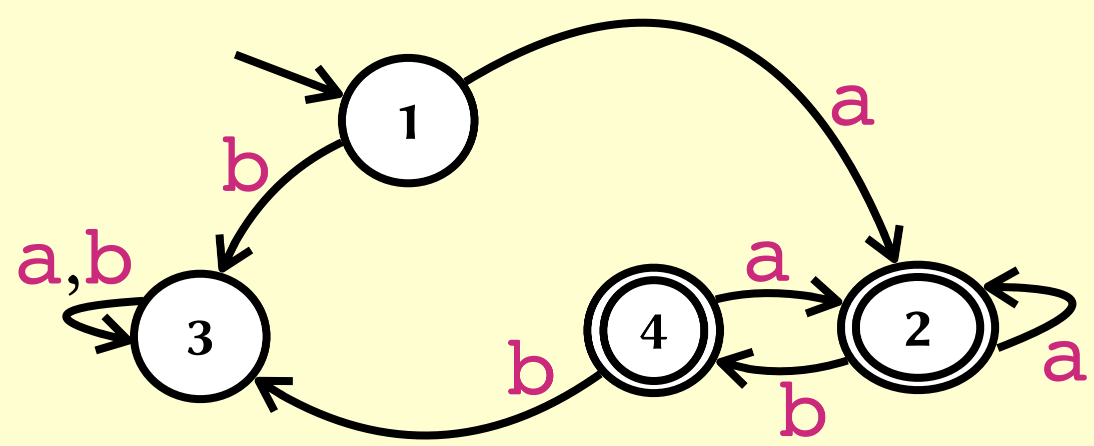


# DFA $\longrightarrow$ Regex

$$R_1 = \mathtt{a}R_2 + \mathtt{b}R_3$$

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

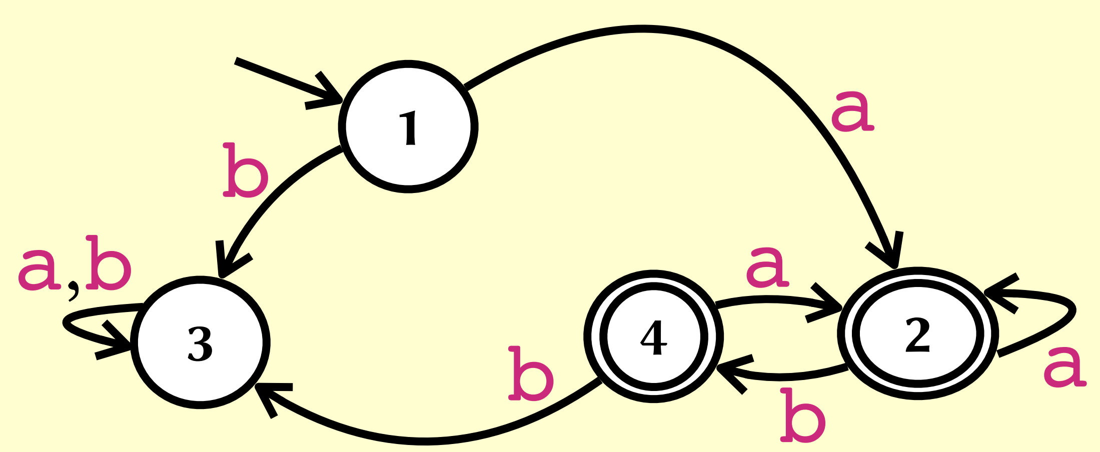
$$R_3 = \mathbf{0}$$



# DFA → Regex

$R_1 = \mathtt{a}R_2 + \mathtt{b}\mathbf{0}$

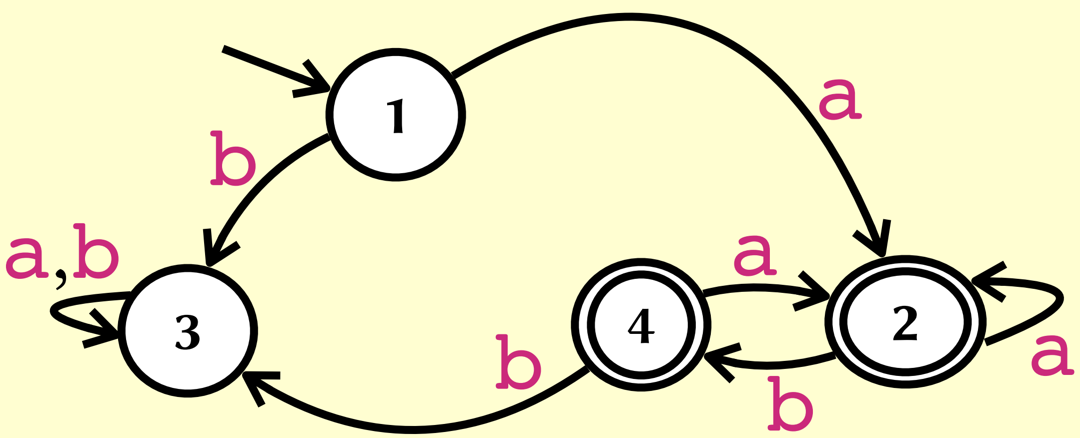
$R_2 = (\mathtt{a} + \mathtt{ba})R_2 + \mathtt{bb}\mathbf{0} + \mathtt{b} + \mathbf{1}$



# DFA → Regex

$R_1 = \mathtt{a}R_2$

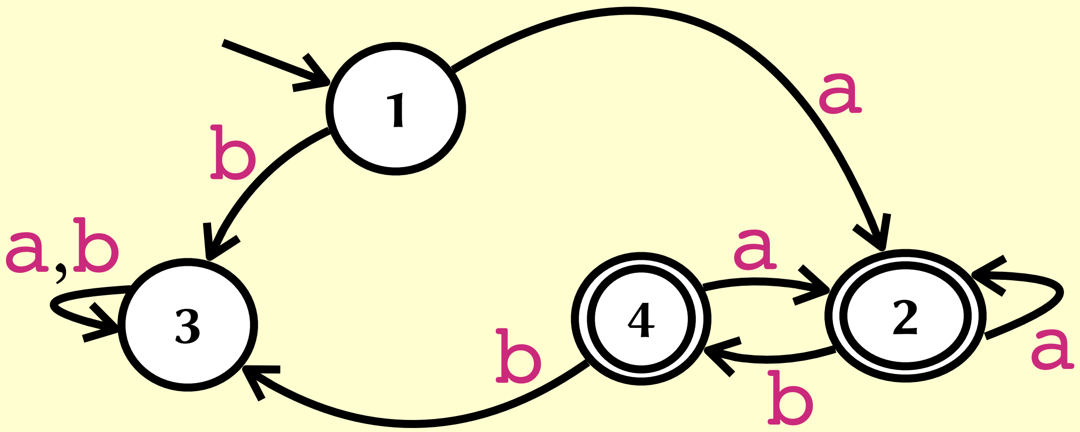
$R_2 = (\mathtt{a} + \mathtt{ba})R_2 + \mathtt{b} + \mathbf{1}$



# DFA → Regex

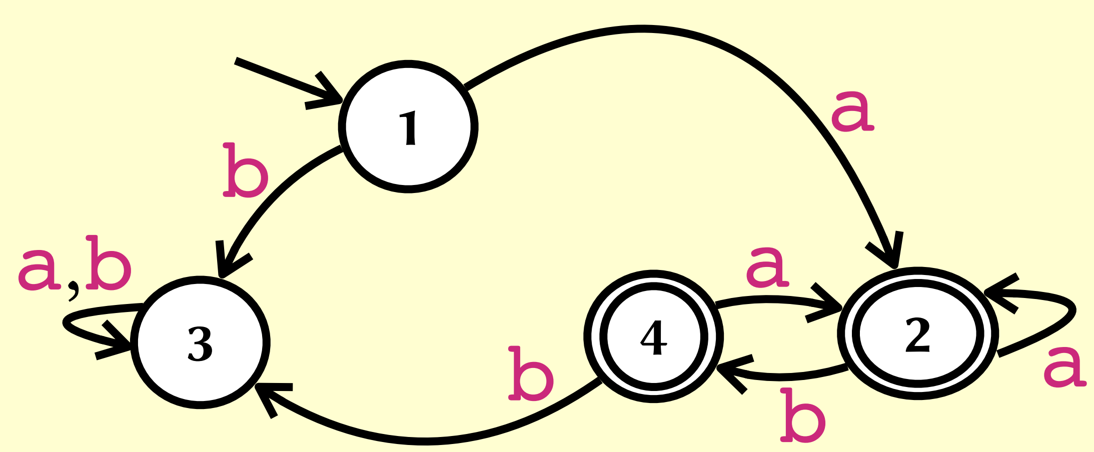
$R_1 = \texttt{a}R_2$

$R_2 = (\texttt{a} + \texttt{ba})^*(\texttt{b} + \mathbf{1})$



# DFA → Regex

$R_1 = \texttt{a}(\texttt{a} + \texttt{ba})^*(\texttt{b} + \mathbf{1})$



# Something to ponder

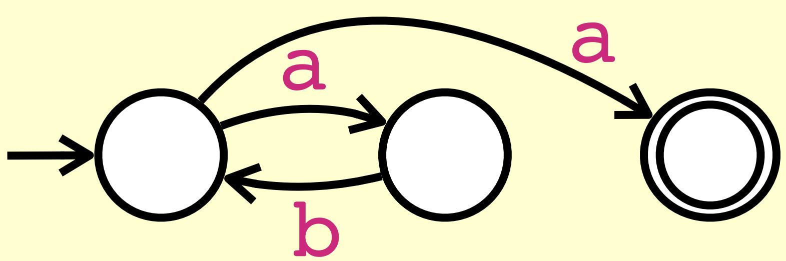
$$\begin{tikzcd}[column sep=huge, row sep=huge]
\textbf{Regular Expressions} \arrow[r, bend left, "\text{Thompson's algorithm}"] & \textbf{NFAs} \arrow[d, bend left, "\text{Subset construction}"] \\
 & \textbf{DFAs} \arrow[u, bend left, "\text{(trivial)}"] \arrow[ul, bend left, "\text{Brzozowski algebraic method}"]
\end{tikzcd}$$

John Brzozowski
1935-2019

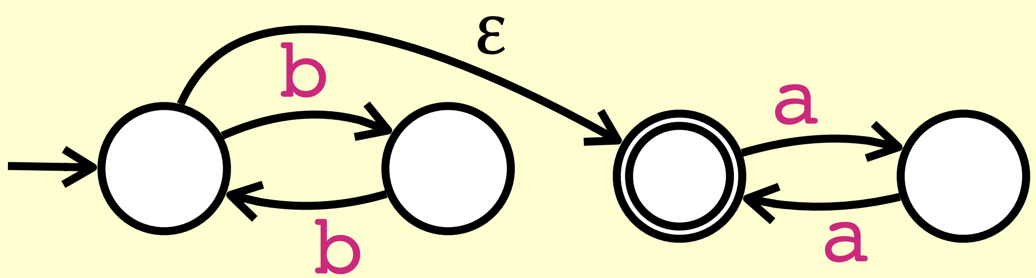
# What about Lex?

- 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.

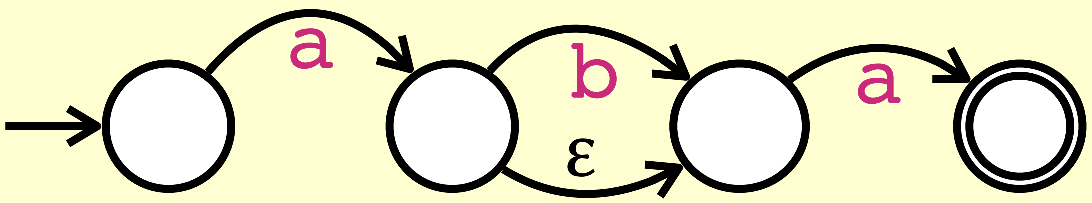
1. (ab)*a



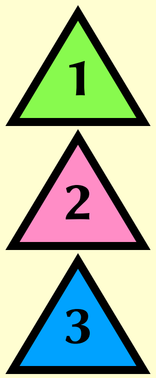

2. (bb)*(aa)*



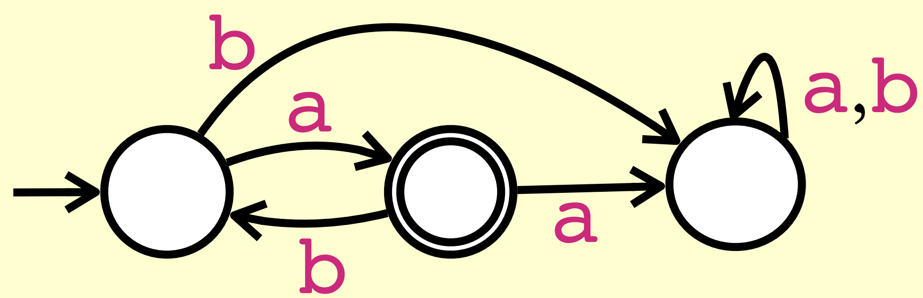

3. a(**1**+b)a



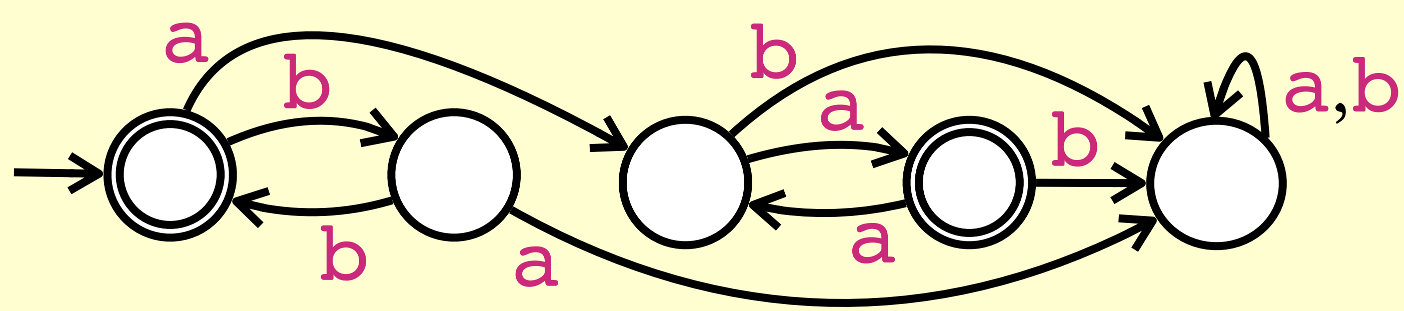
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | b | a | b | a | b | a | $ |



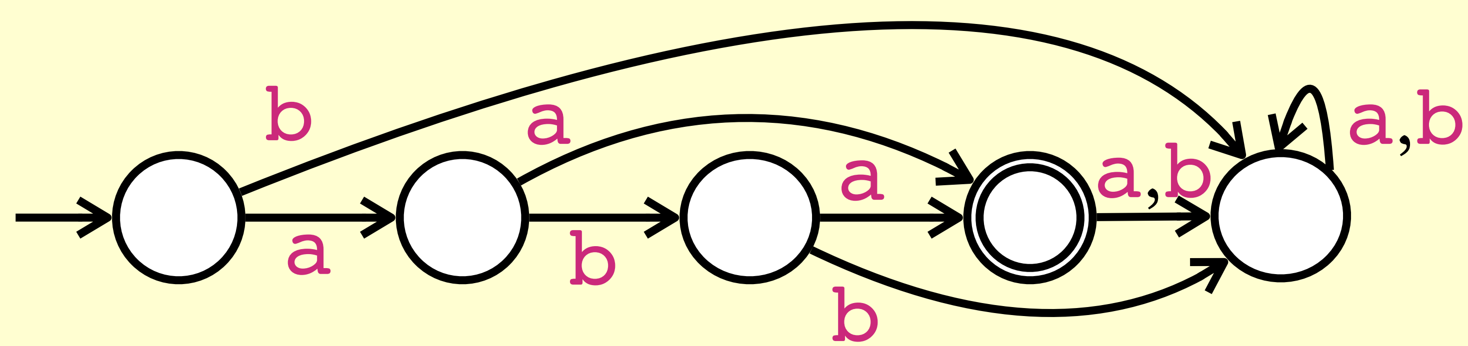

1. (ab)*a



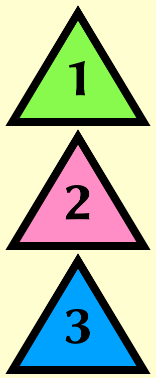

2. (bb)*(aa)*



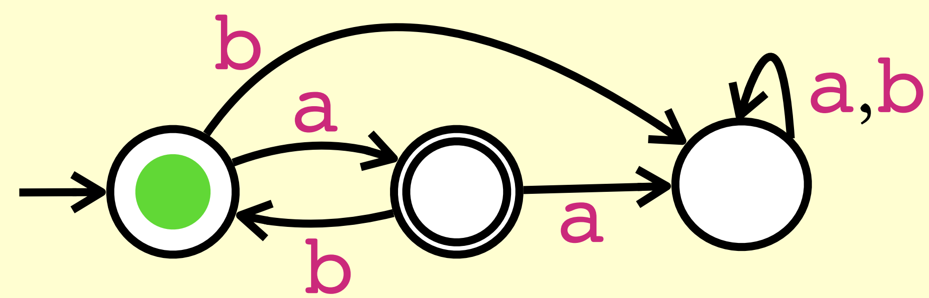

3. a(**1**+b)a



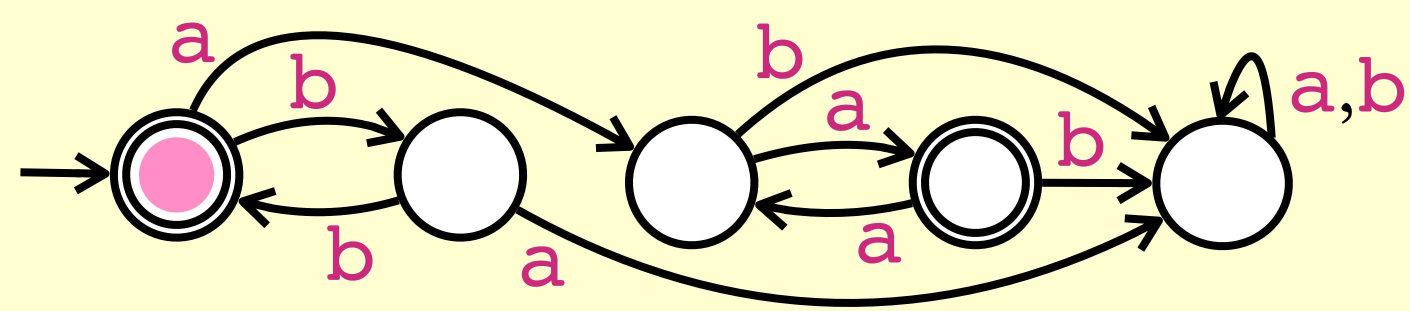
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | b | a | b | a | b | a | $ |



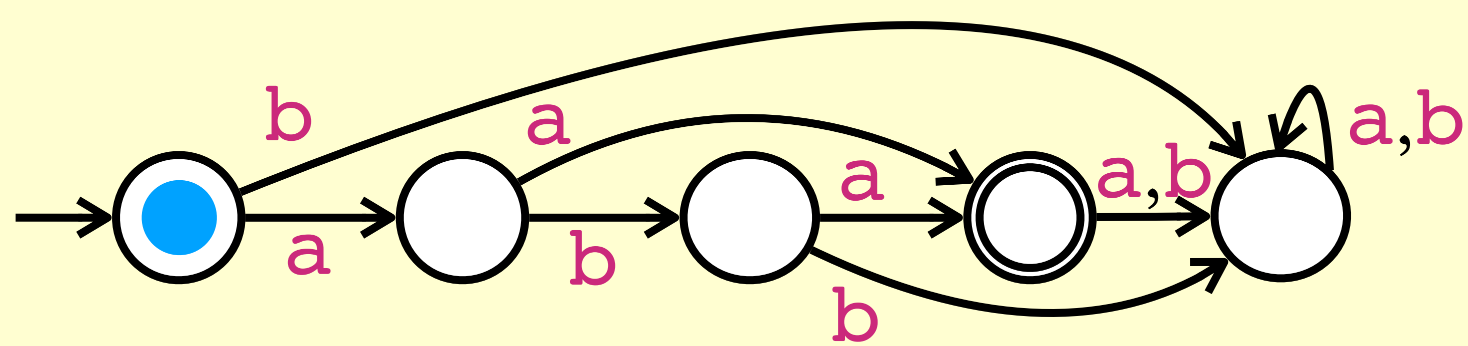

1. (ab)*a

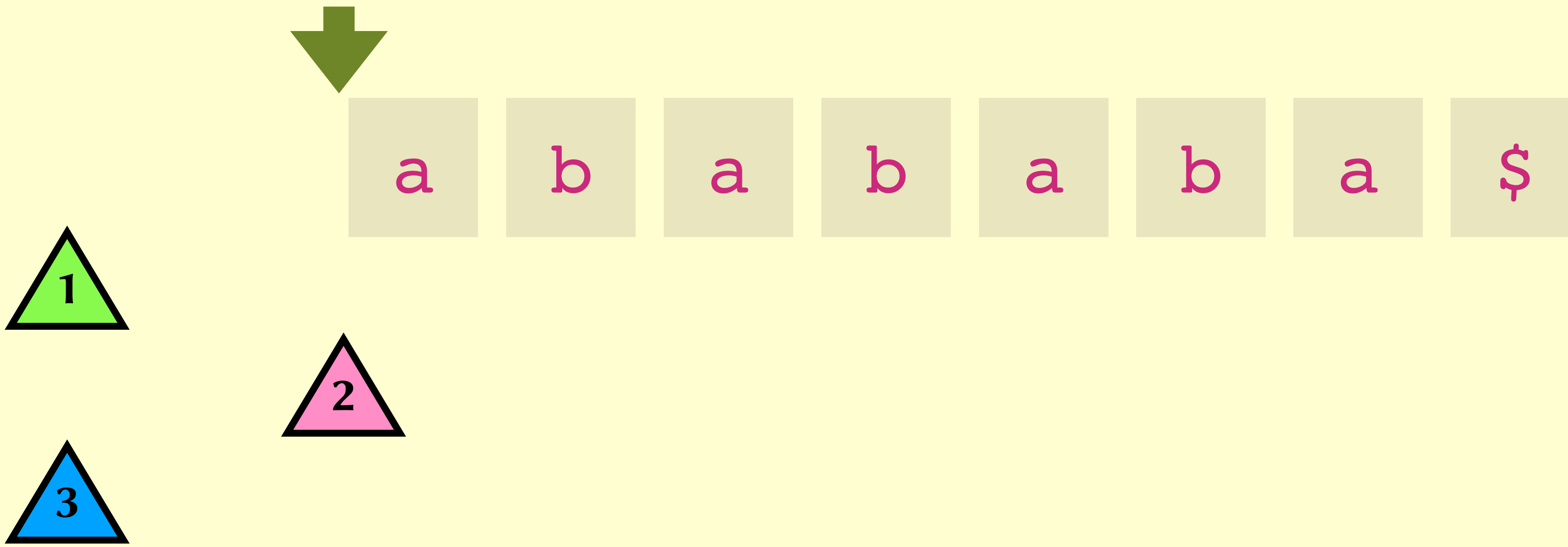



2. (bb)*(aa)*

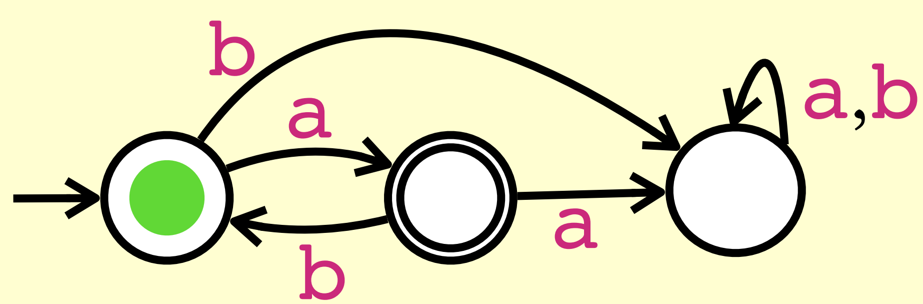



3. a(**1**+b)a

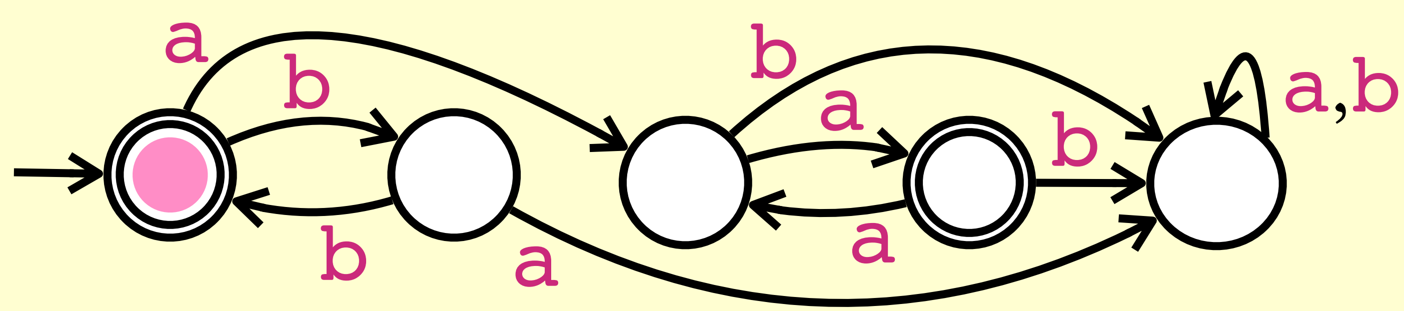




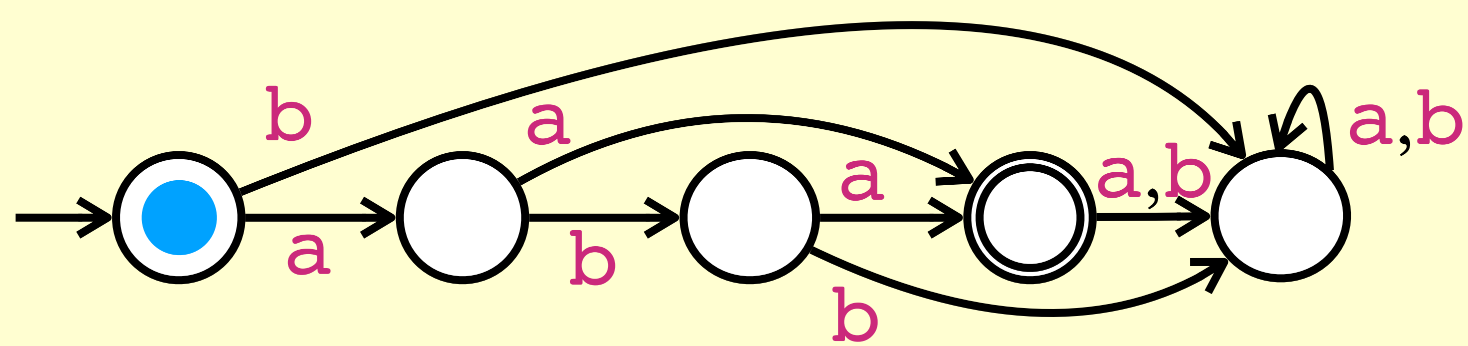

1. (ab)*a

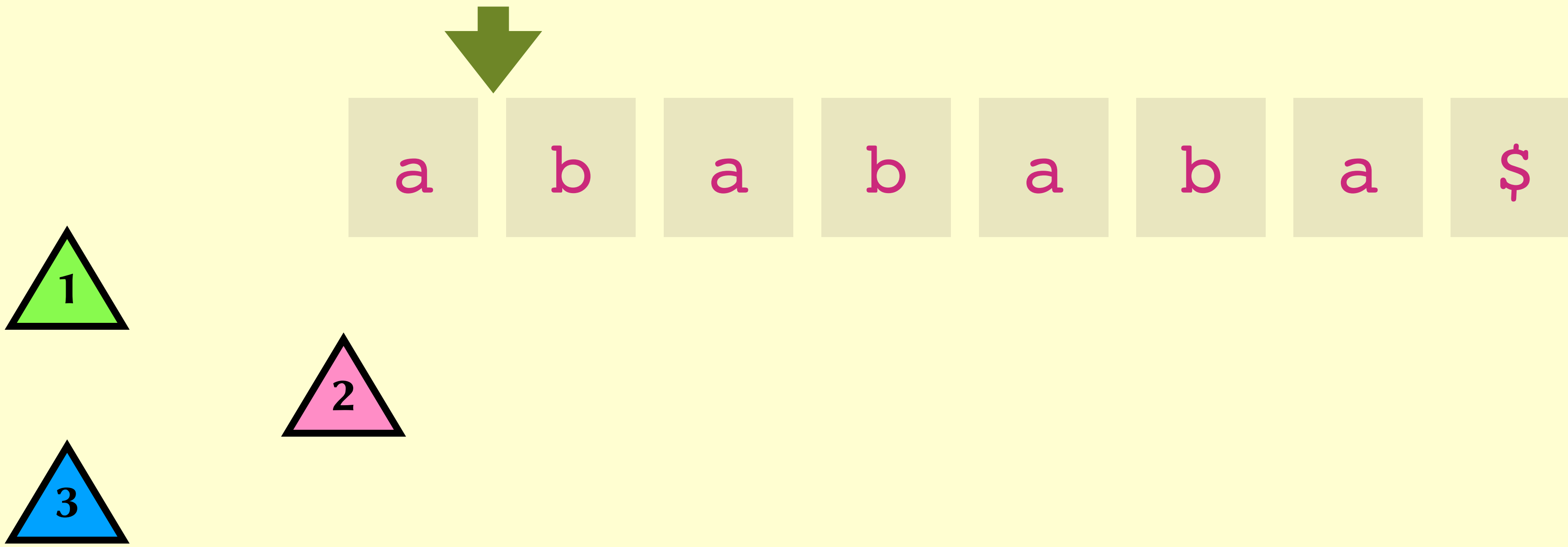



2. (bb)*(aa)*

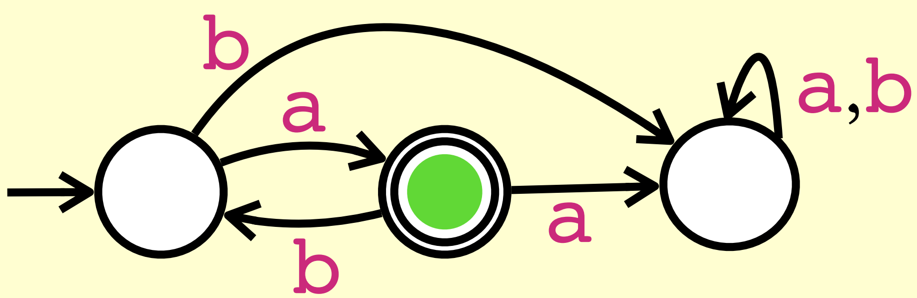



3. a(**1**+b)a

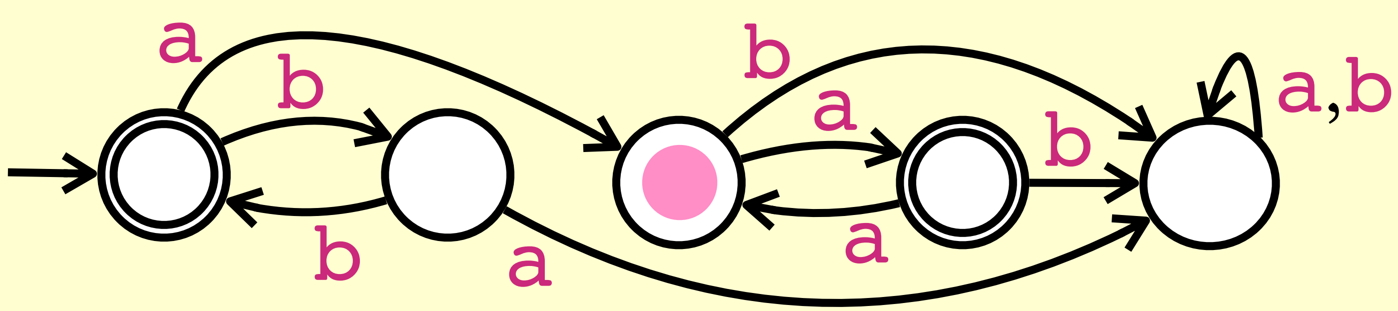




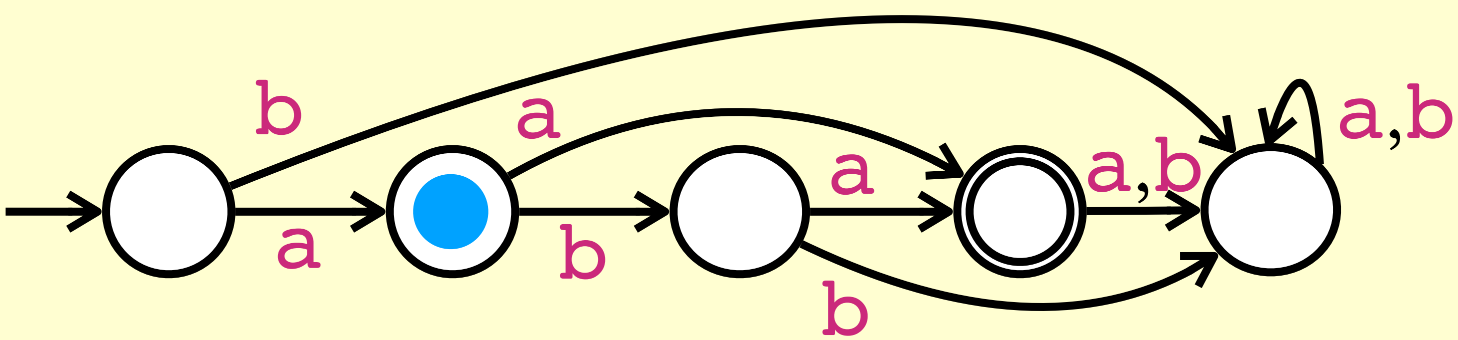

1. (ab)*a

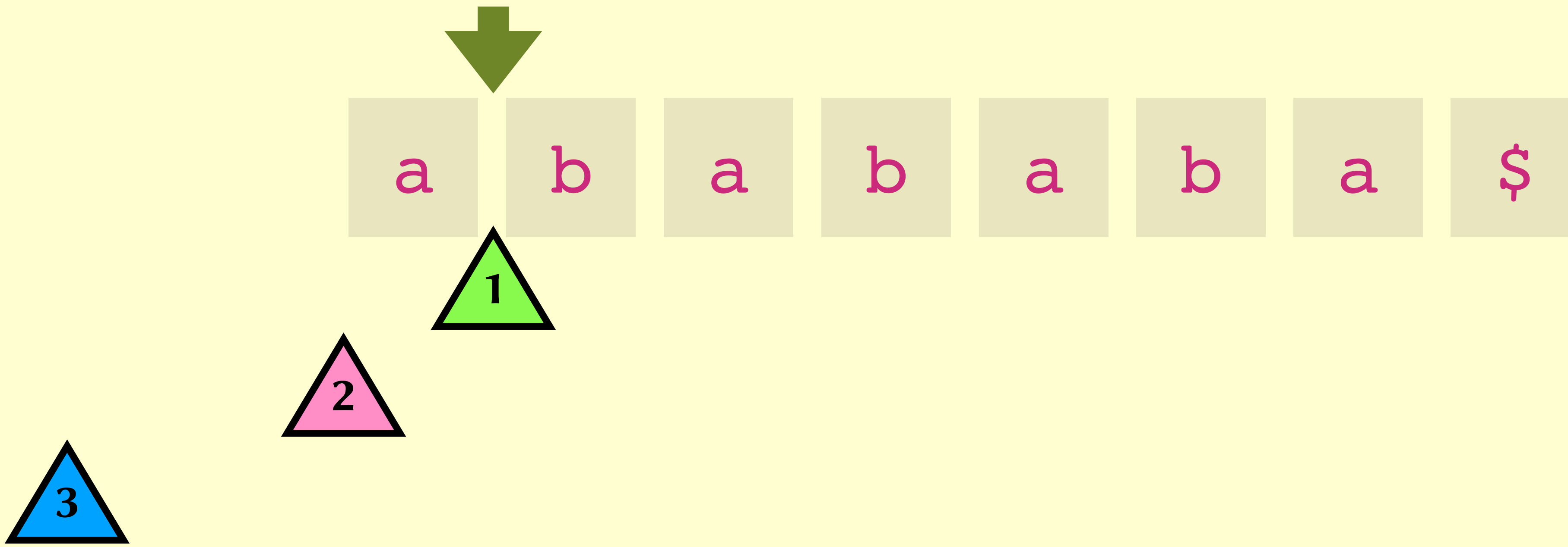



2. (bb)*(aa)*

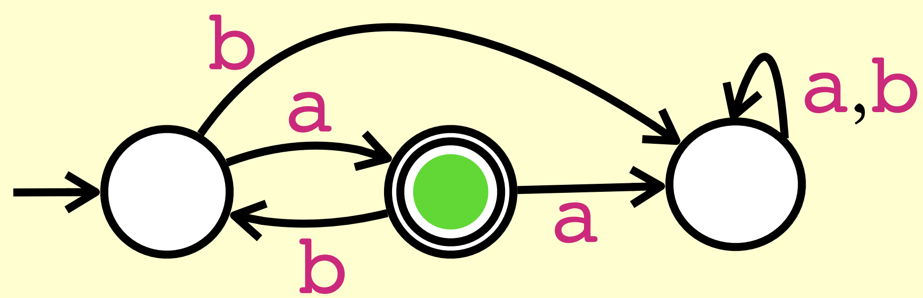



3. a(**1**+b)a

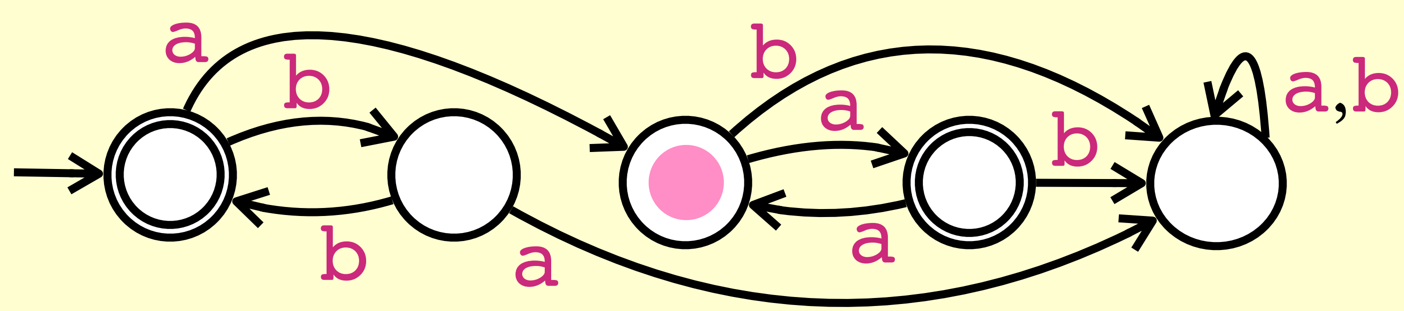




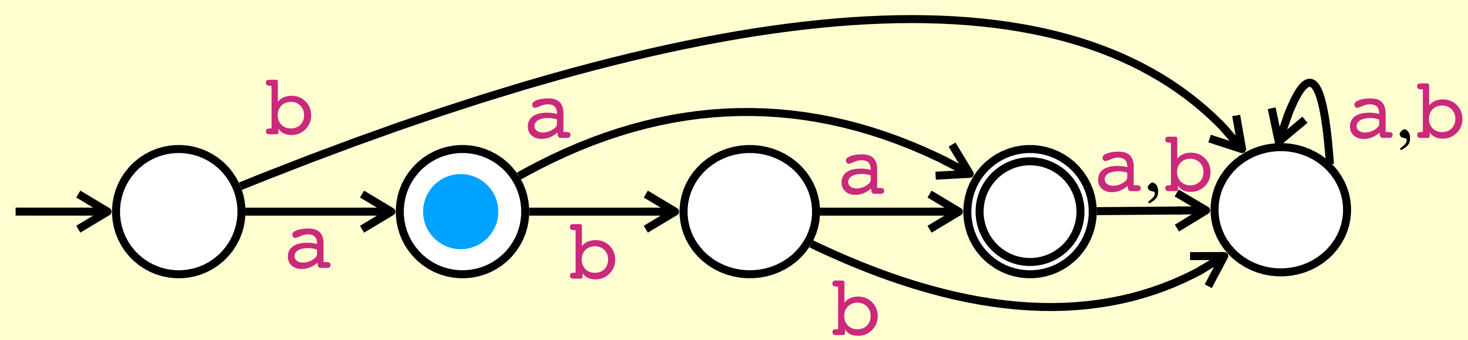

1. (ab)*a

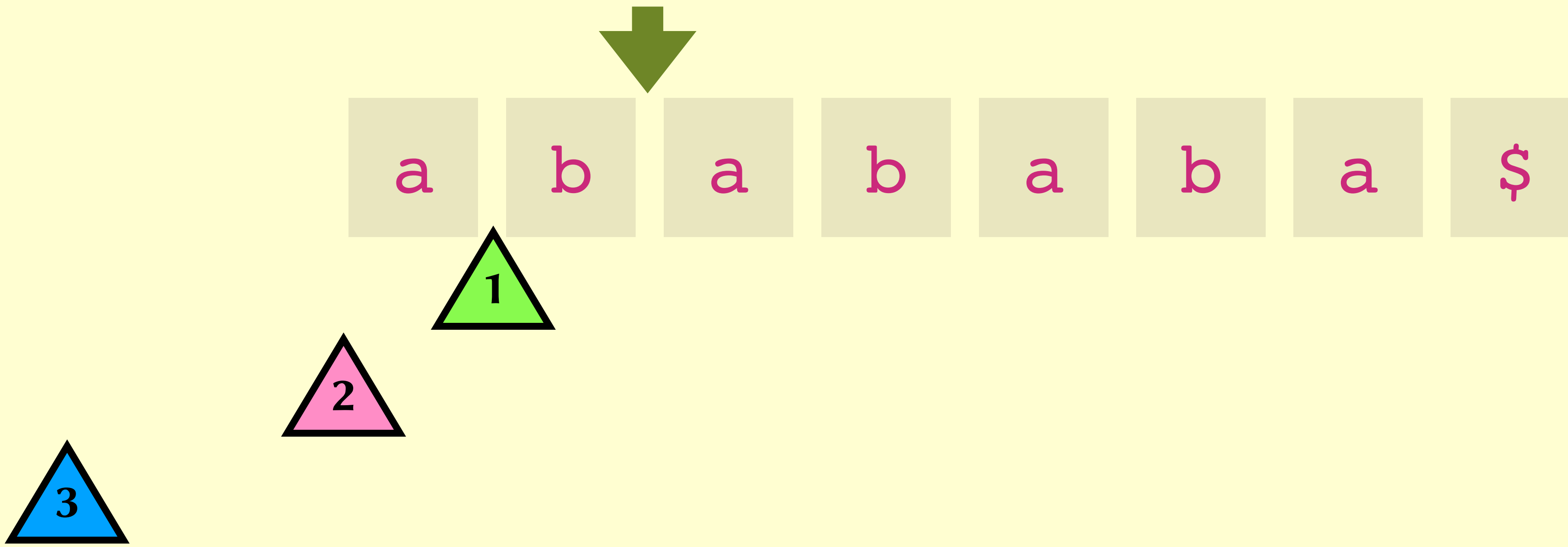



2. (bb)*(aa)*

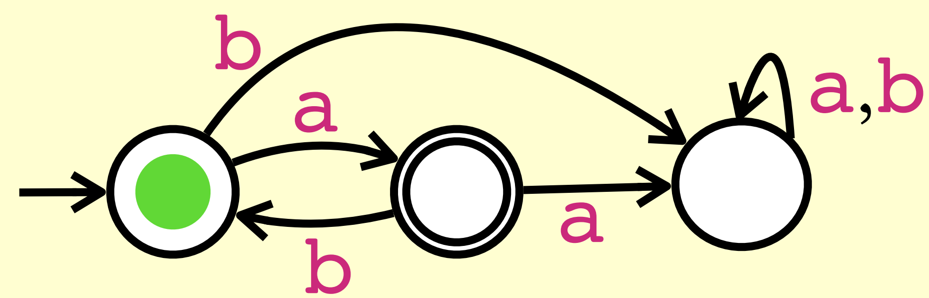



3. a(**1**+b)a

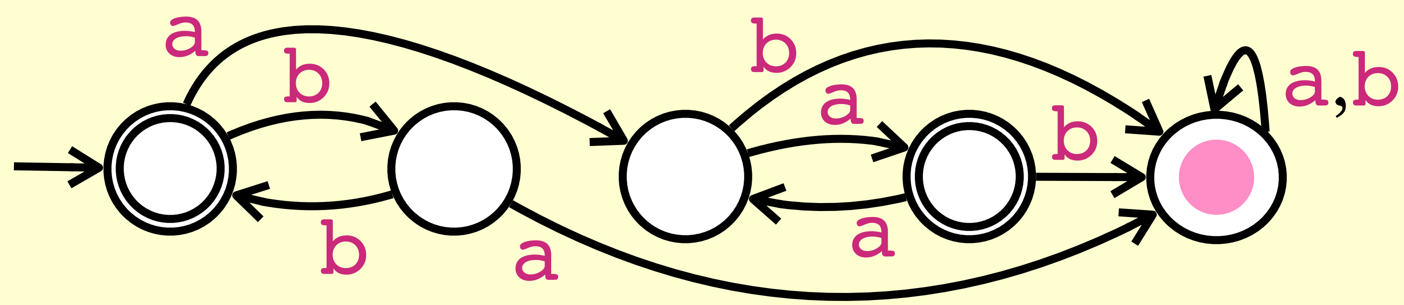




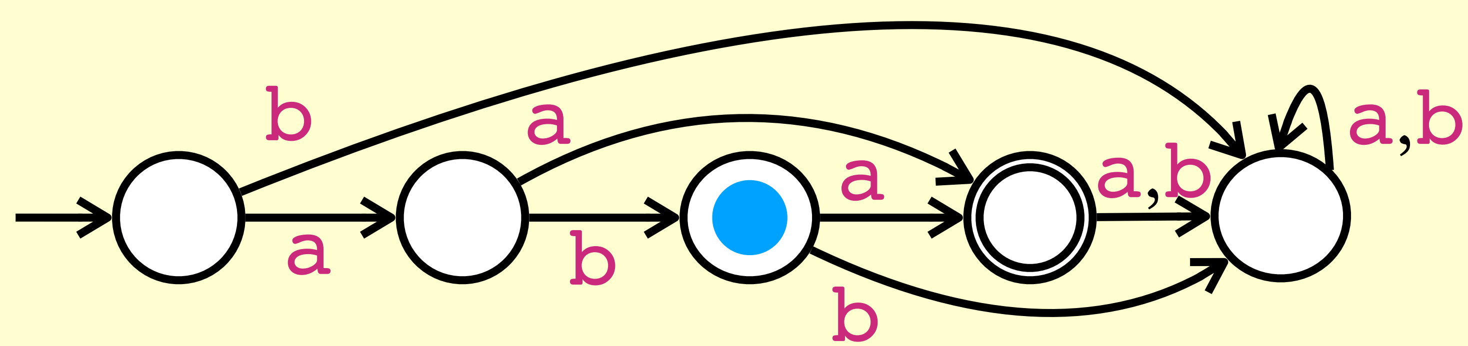

1. (ab)*a

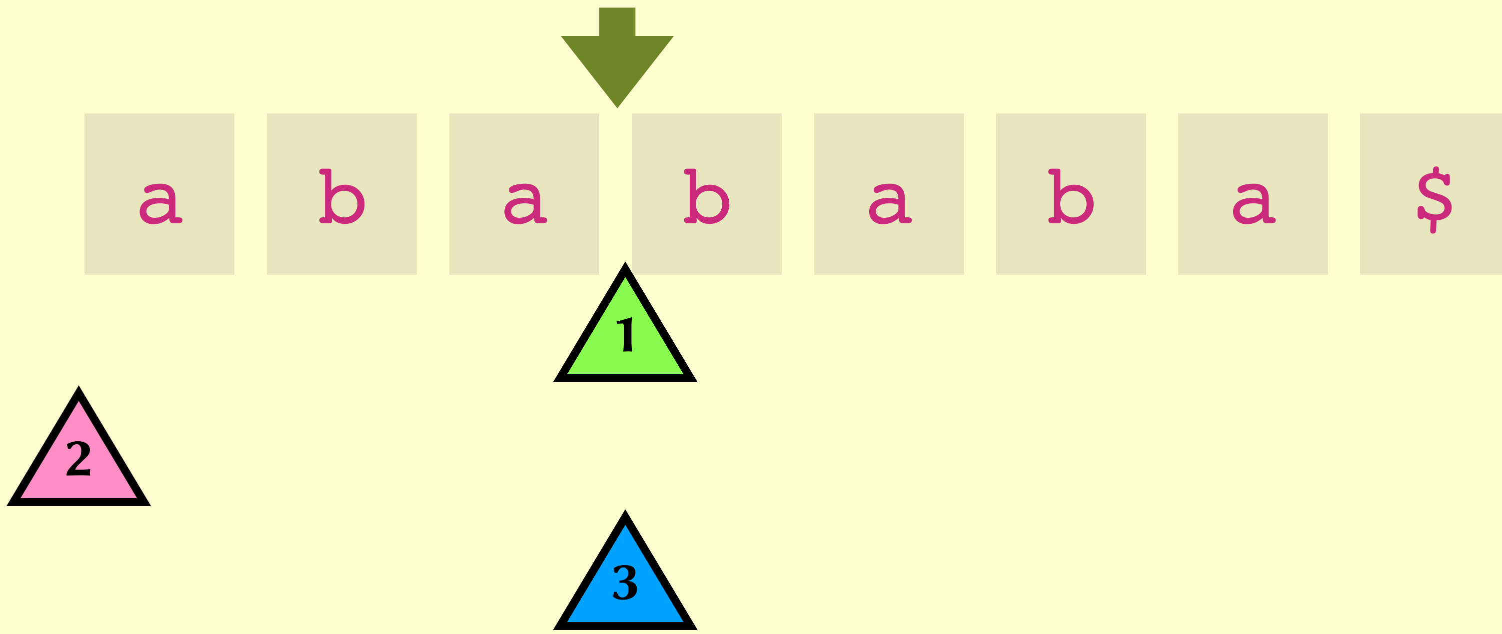



2. (bb)*(aa)*

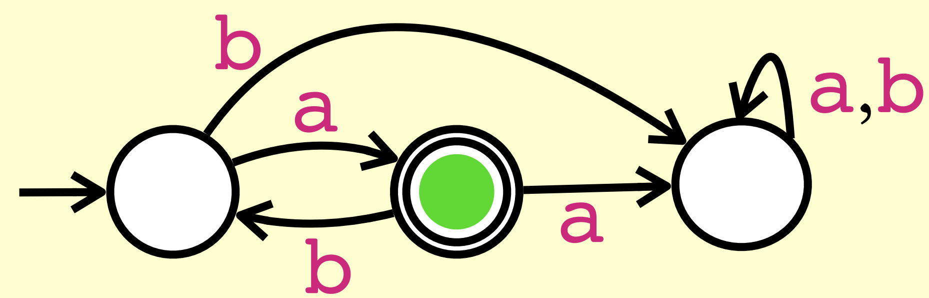



3. a(**1**+b)a

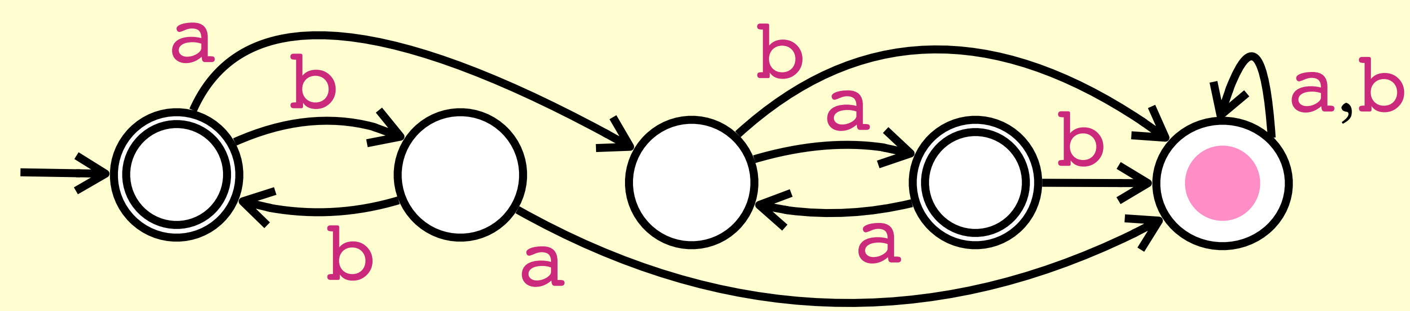




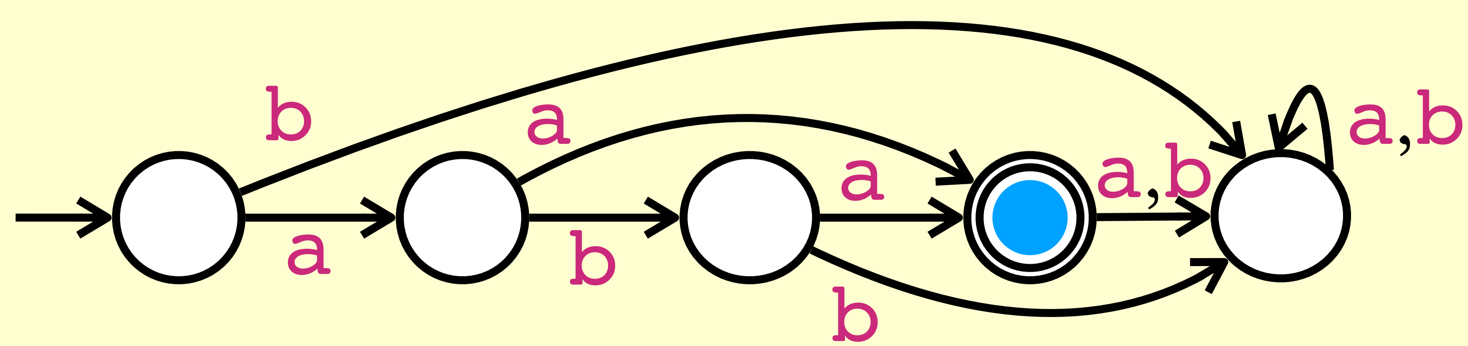

1. $(ab)^*a$

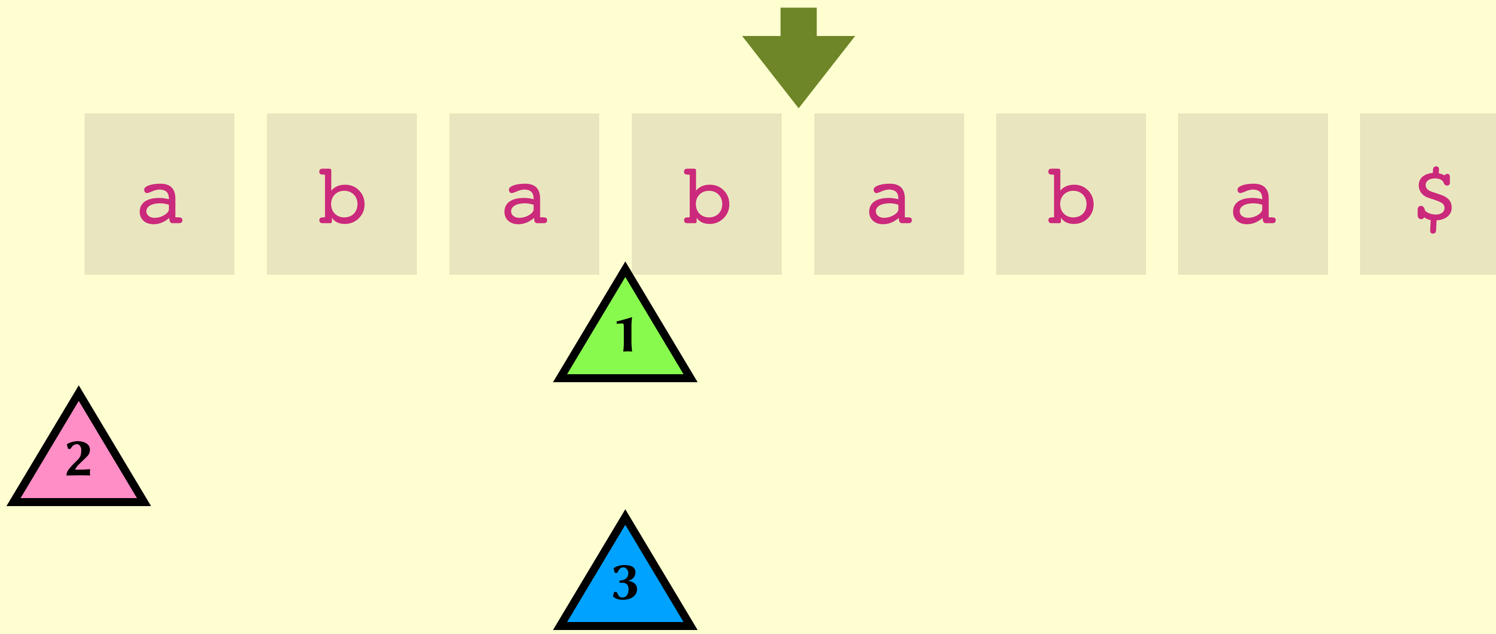



2. $(bb)^*(aa)^*$

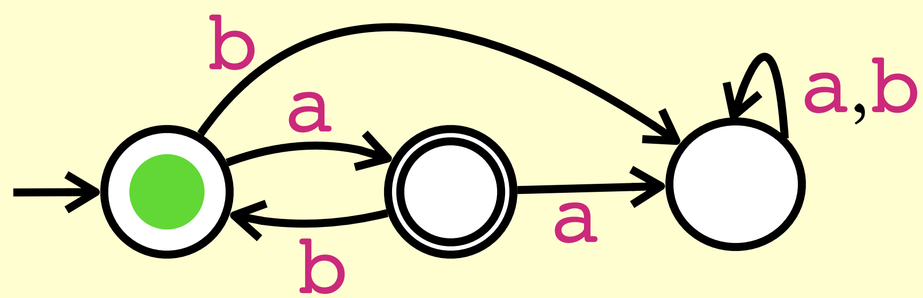



3. $a(\mathbf{1}+b)a$

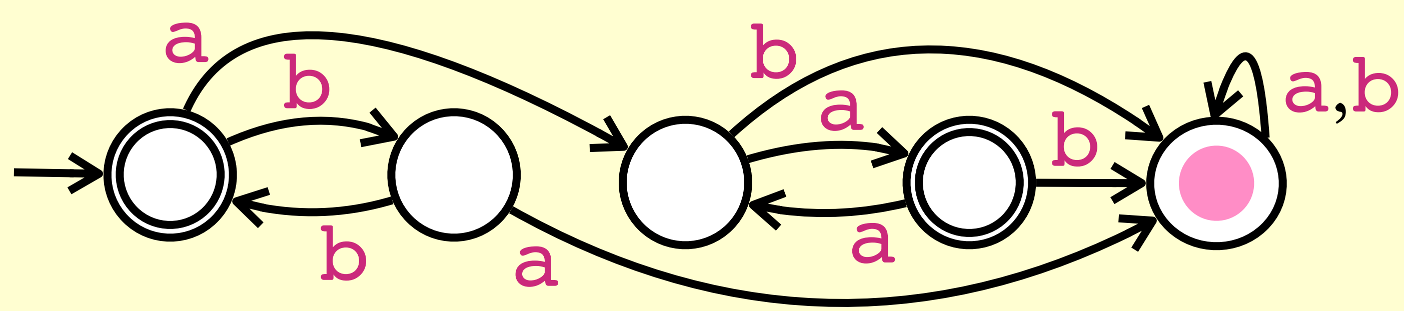




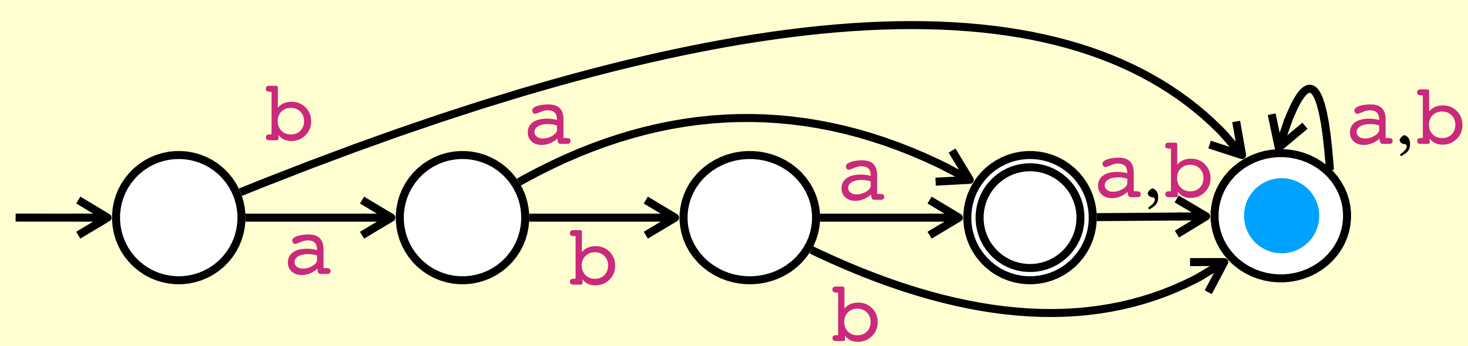

1. $(\mathtt{ab})^*\mathtt{a}$

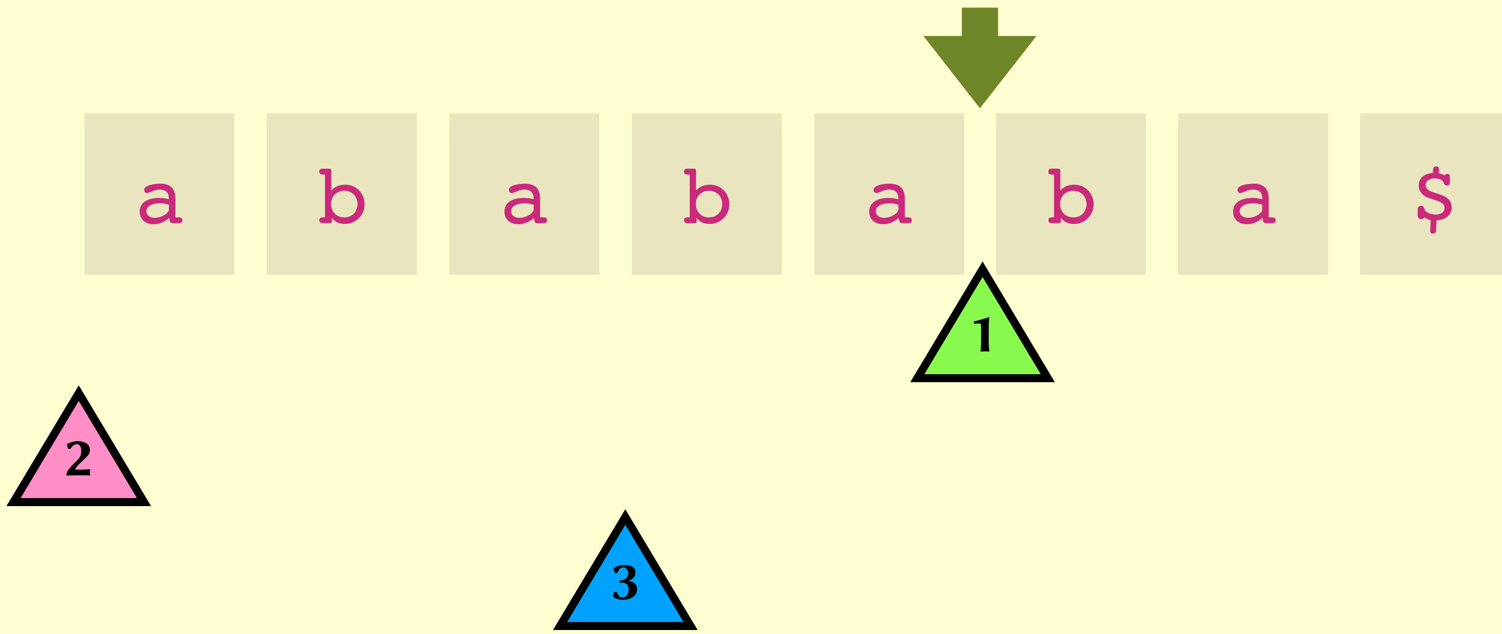



2. $(\mathtt{bb})^*(\mathtt{aa})^*$

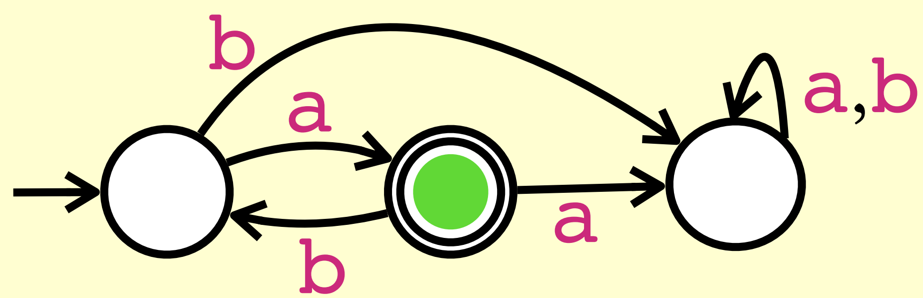



3. $\mathtt{a}(\mathbf{1}+\mathtt{b})\mathtt{a}$

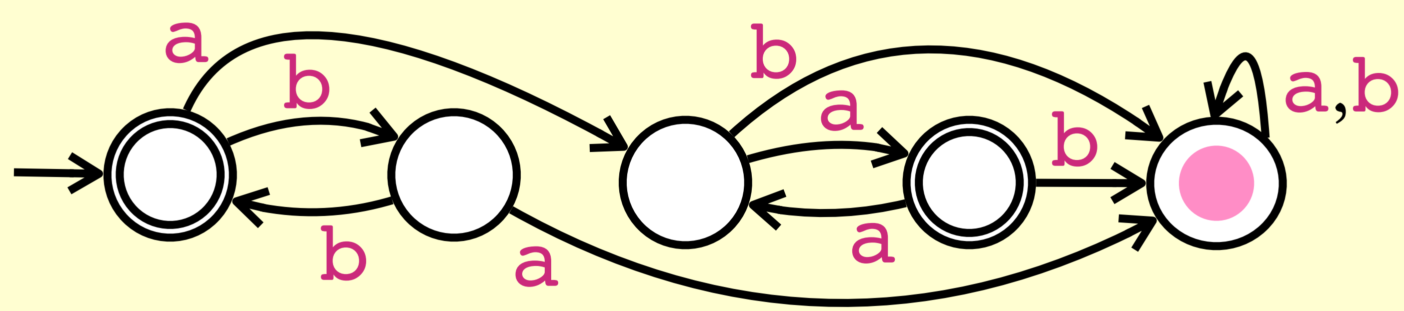




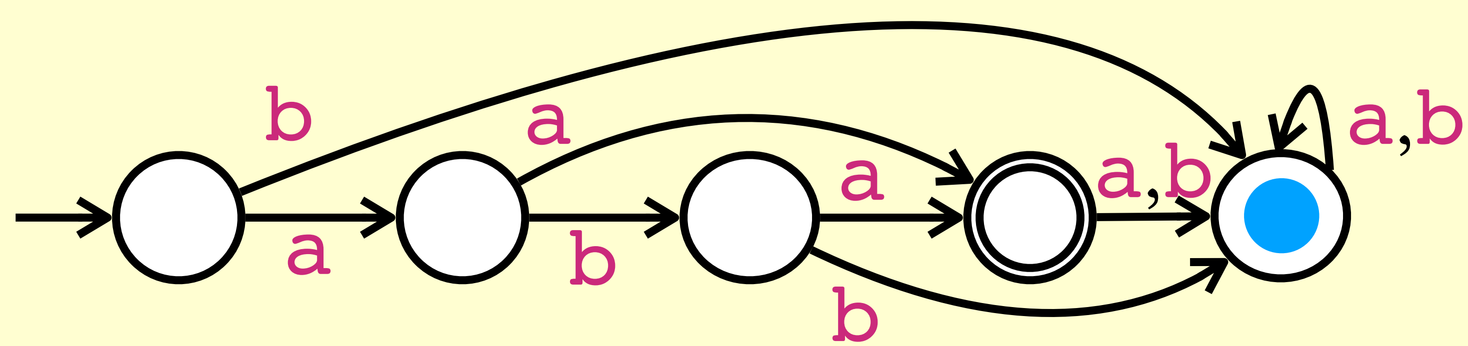

1. (ab)*a

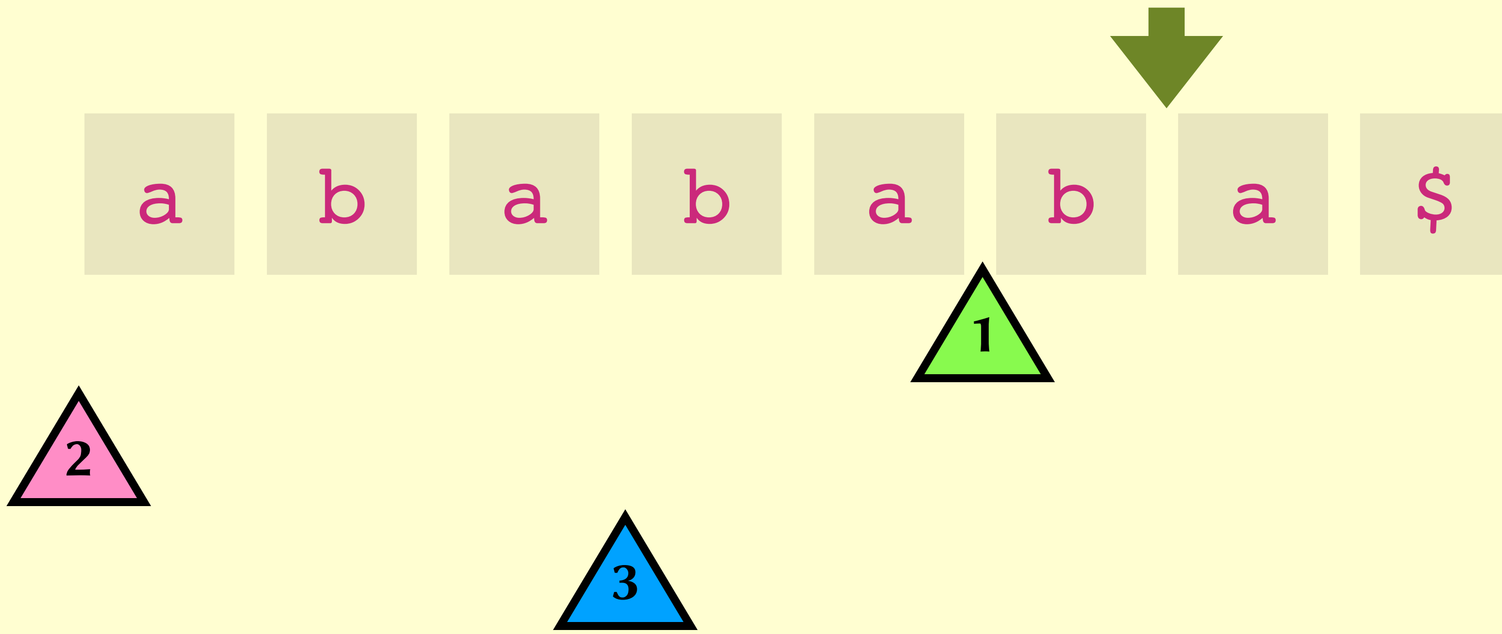



2. (bb)*(aa)*

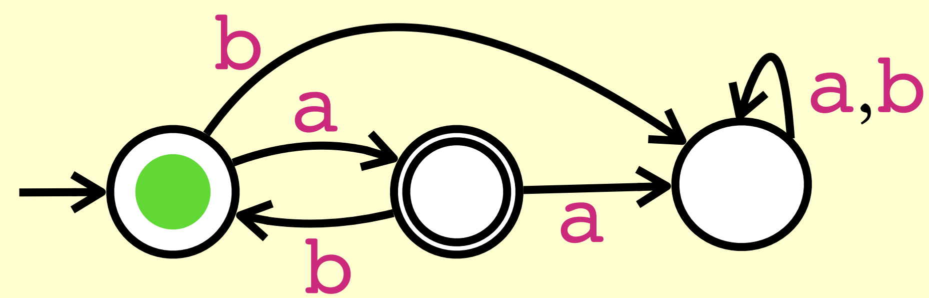



3. a(**1**+b)a

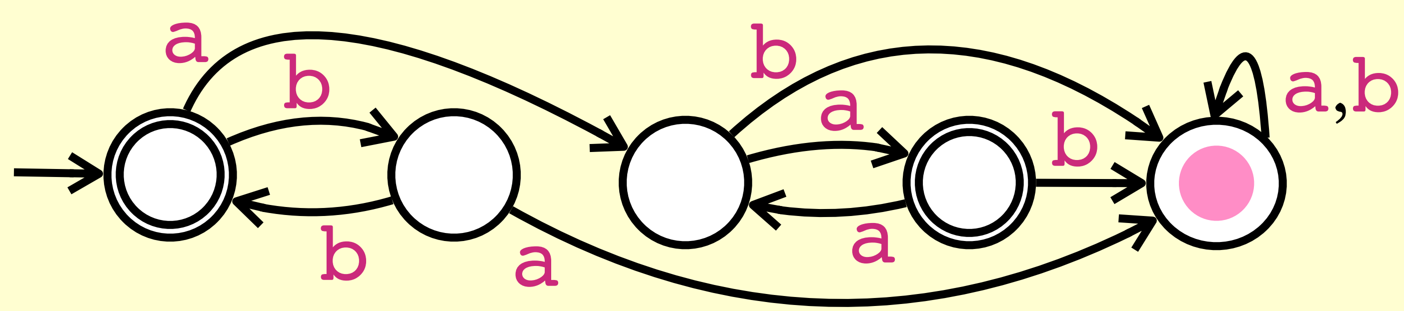




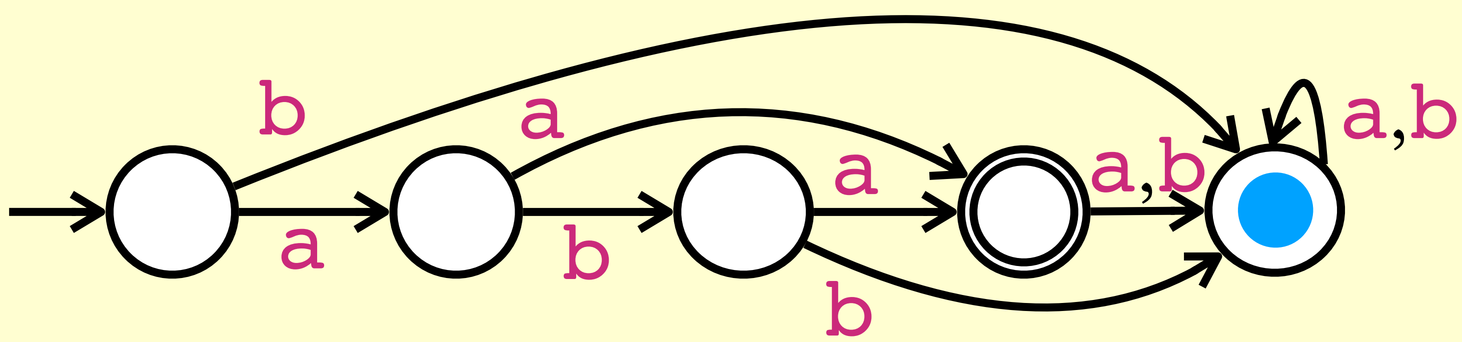

1. (ab)*a

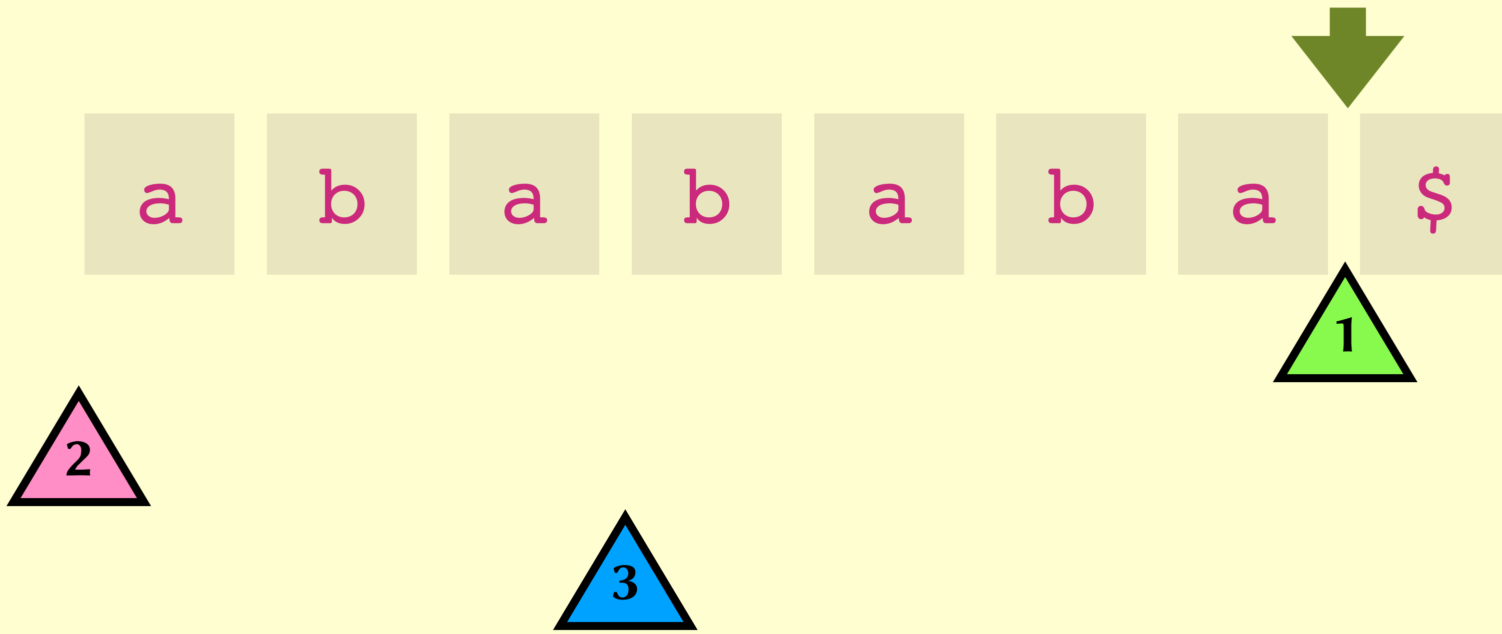



2. (bb)*(aa)*

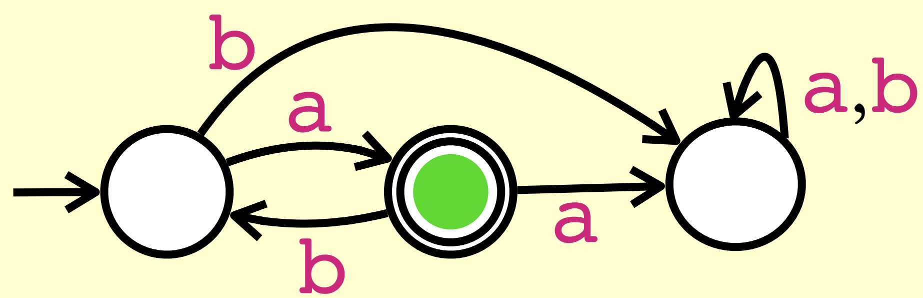



3. a(**1**+b)a

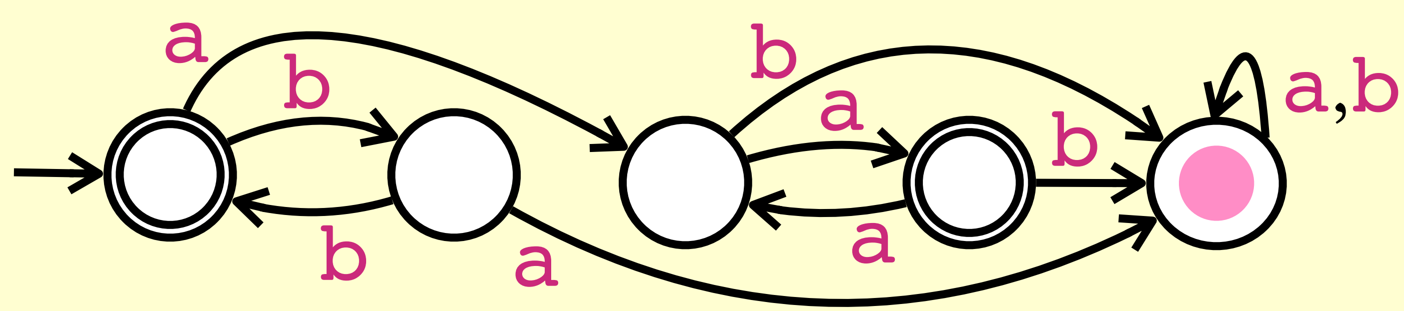




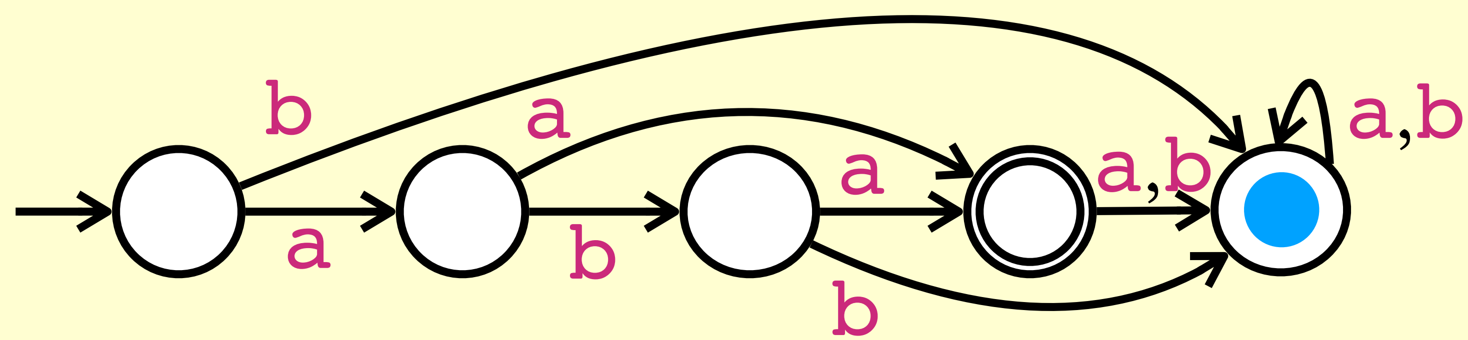

1. (ab)*a

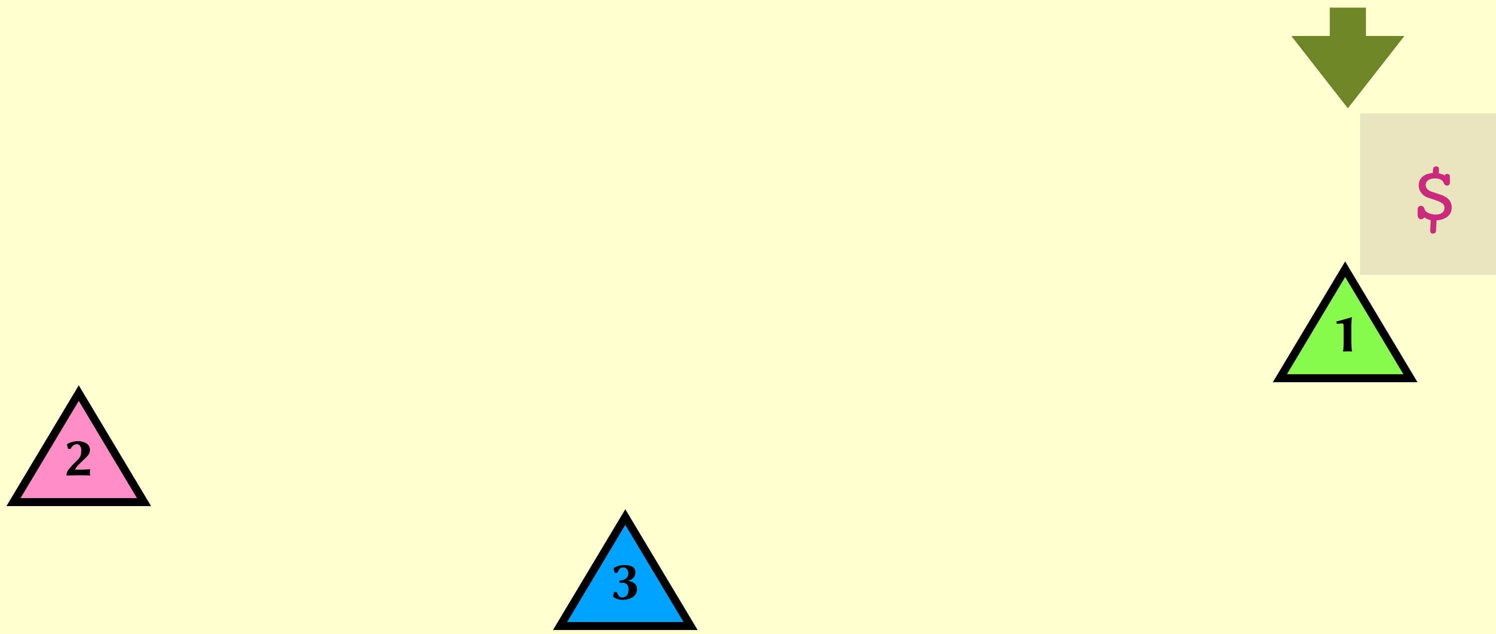



2. (bb)*(aa)*

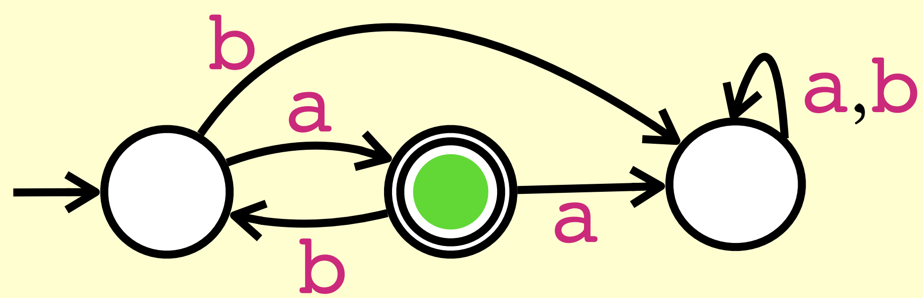



3. a(**1**+b)a

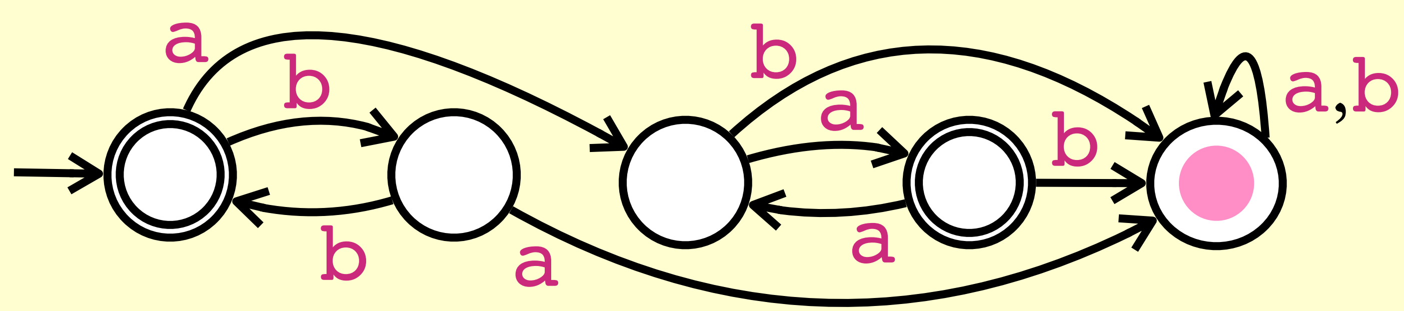




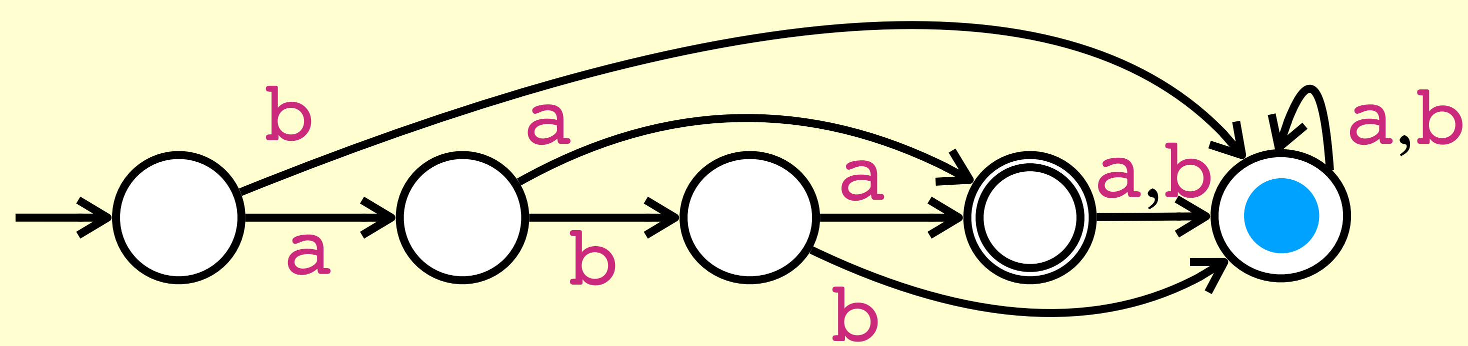

1. $(ab)^*a$



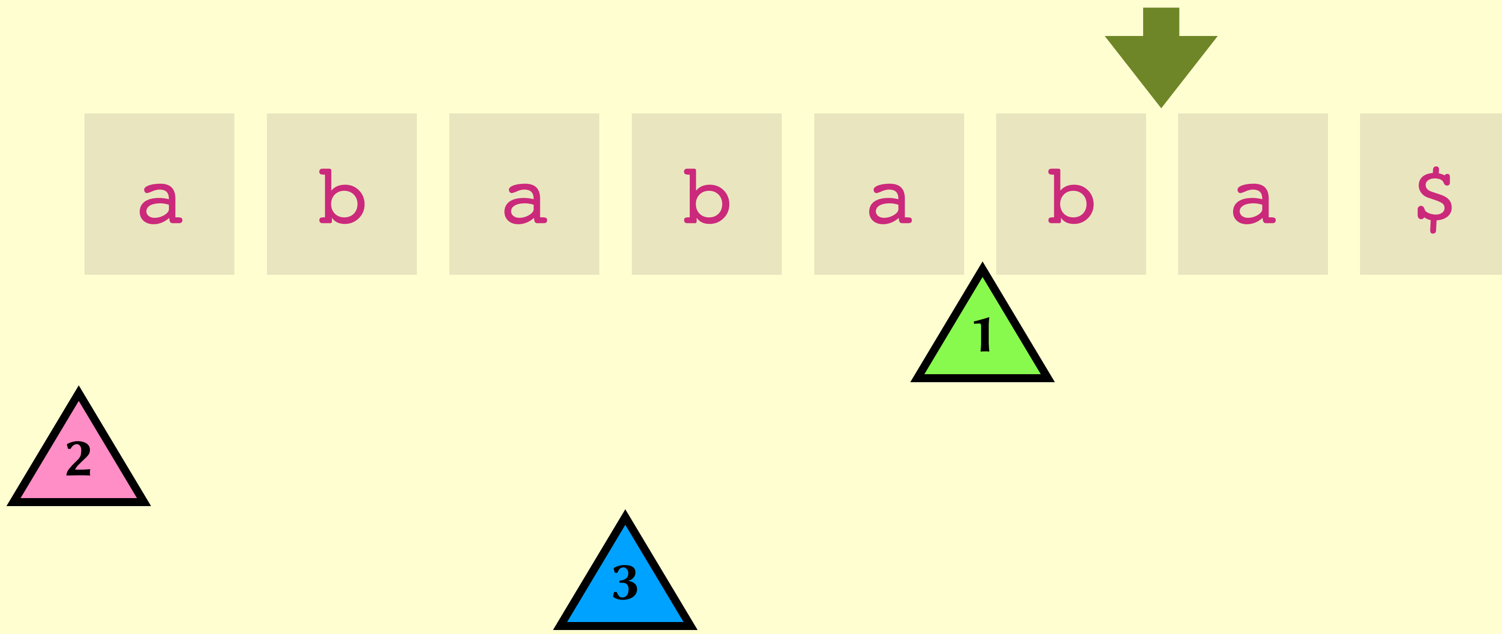

2. $(bb)^*(aa)^*$



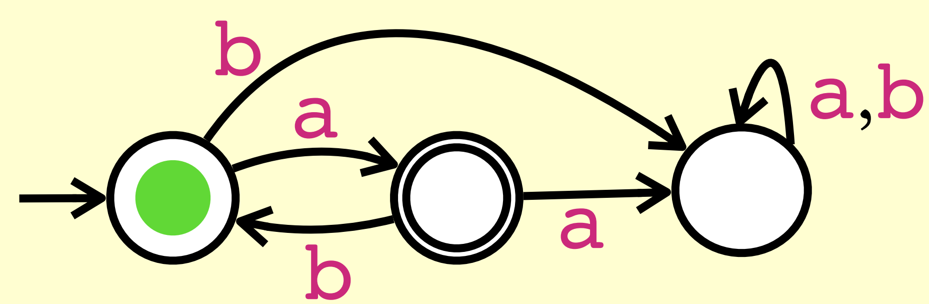

3. $a(\mathbf{1}+b)a$



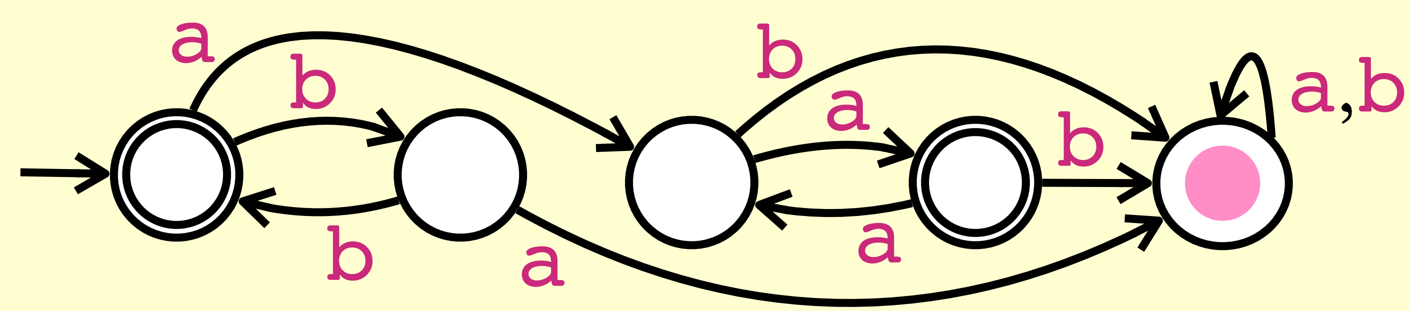

a b a b a b a



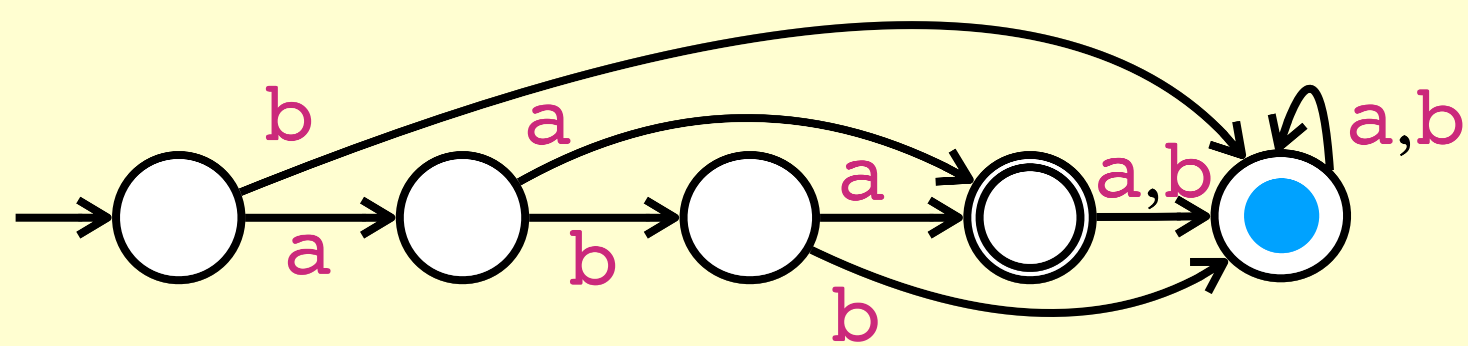

1. $(\mathtt{ab})^*\mathtt{a}$

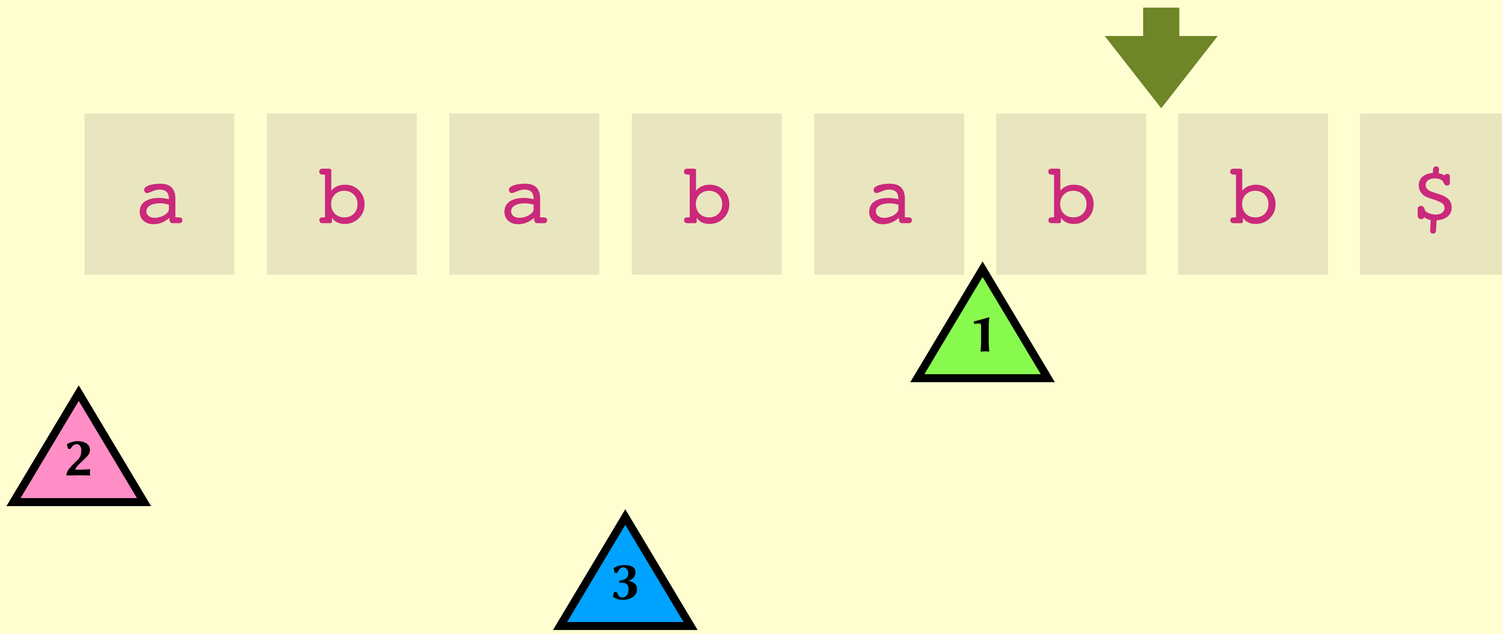



2. $(\mathtt{bb})^*(\mathtt{aa})^*$

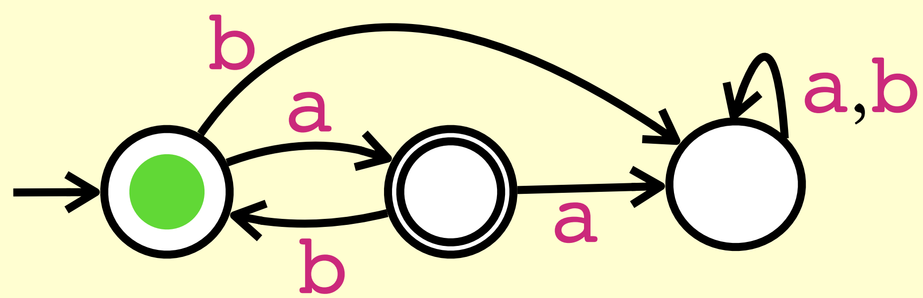



3. $\mathtt{a}(\mathbf{1}+\mathtt{b})\mathtt{a}$

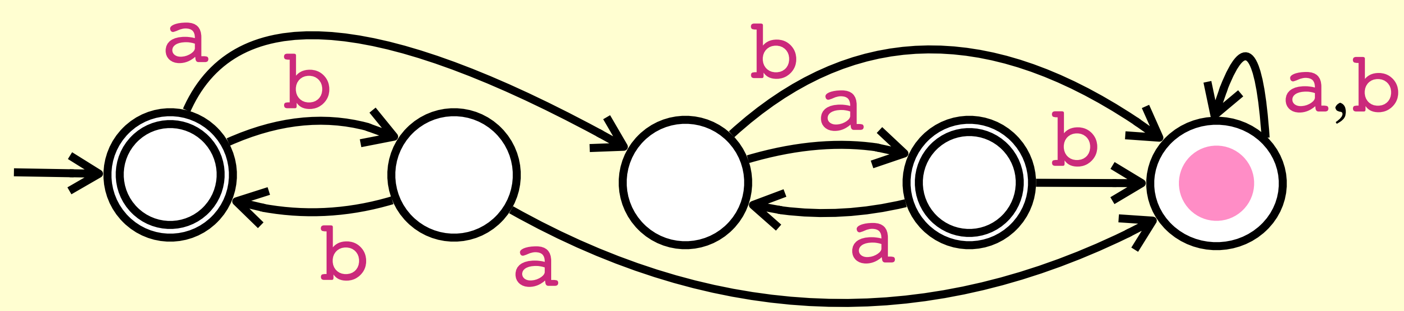




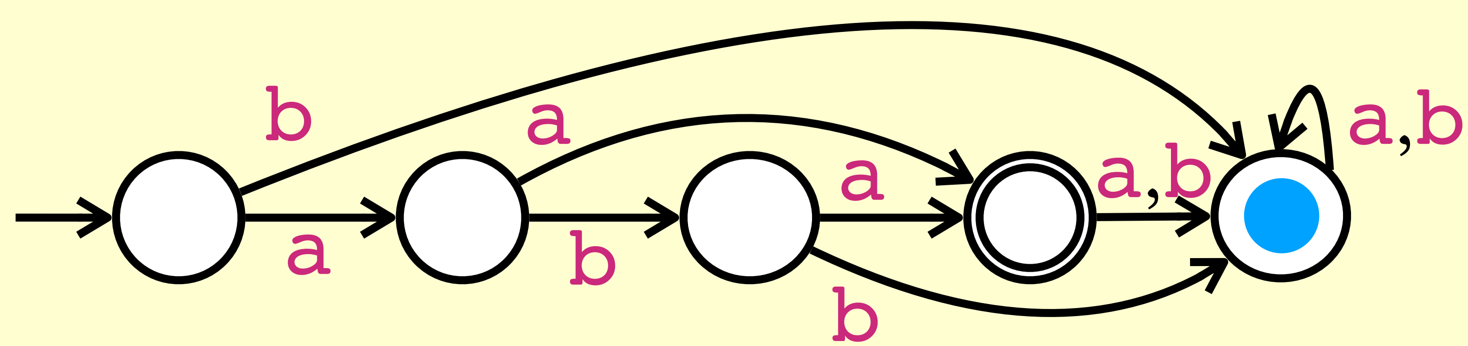

1. (ab)$^*$a

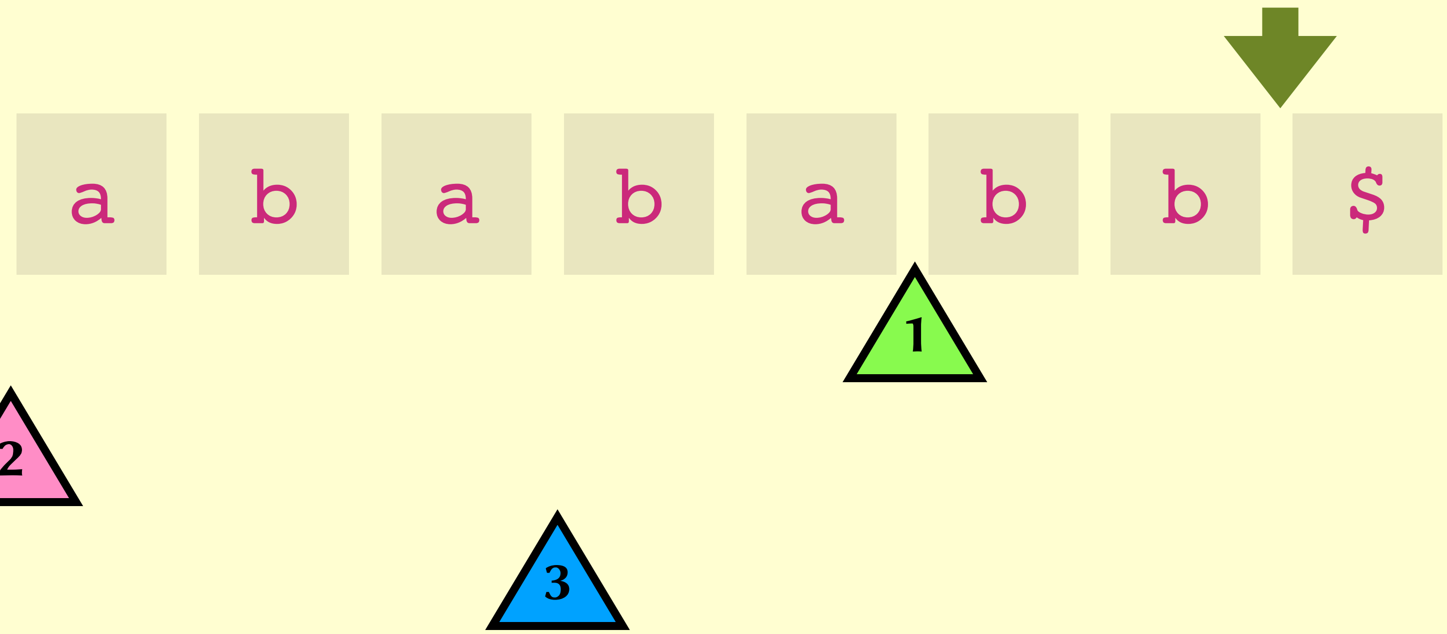



2. (bb)$^*$(aa)$^*$

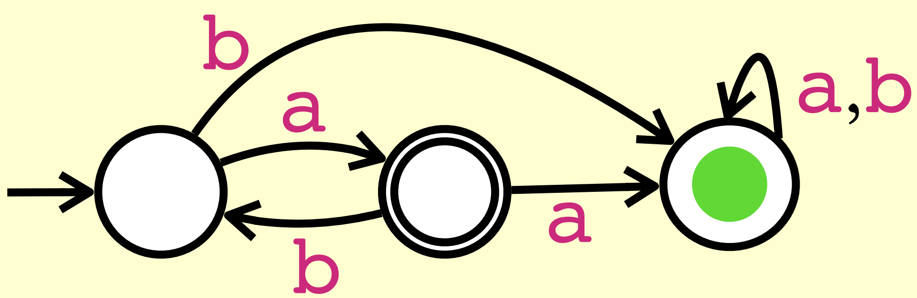



3. a(**1**+b)a

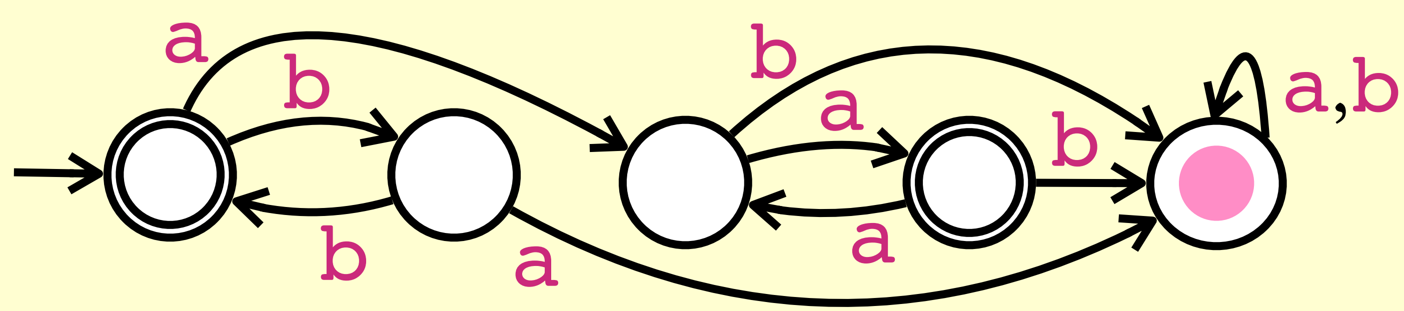




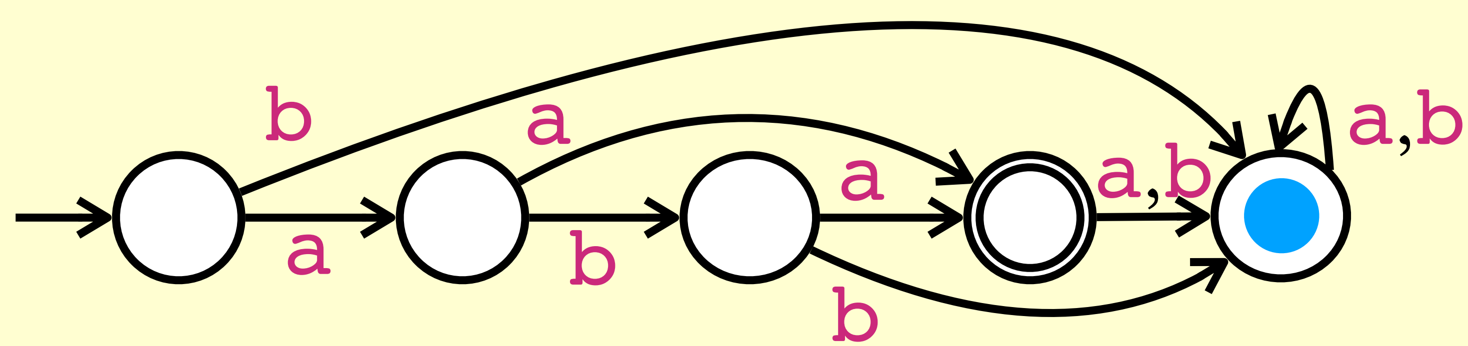

1. (ab)*a

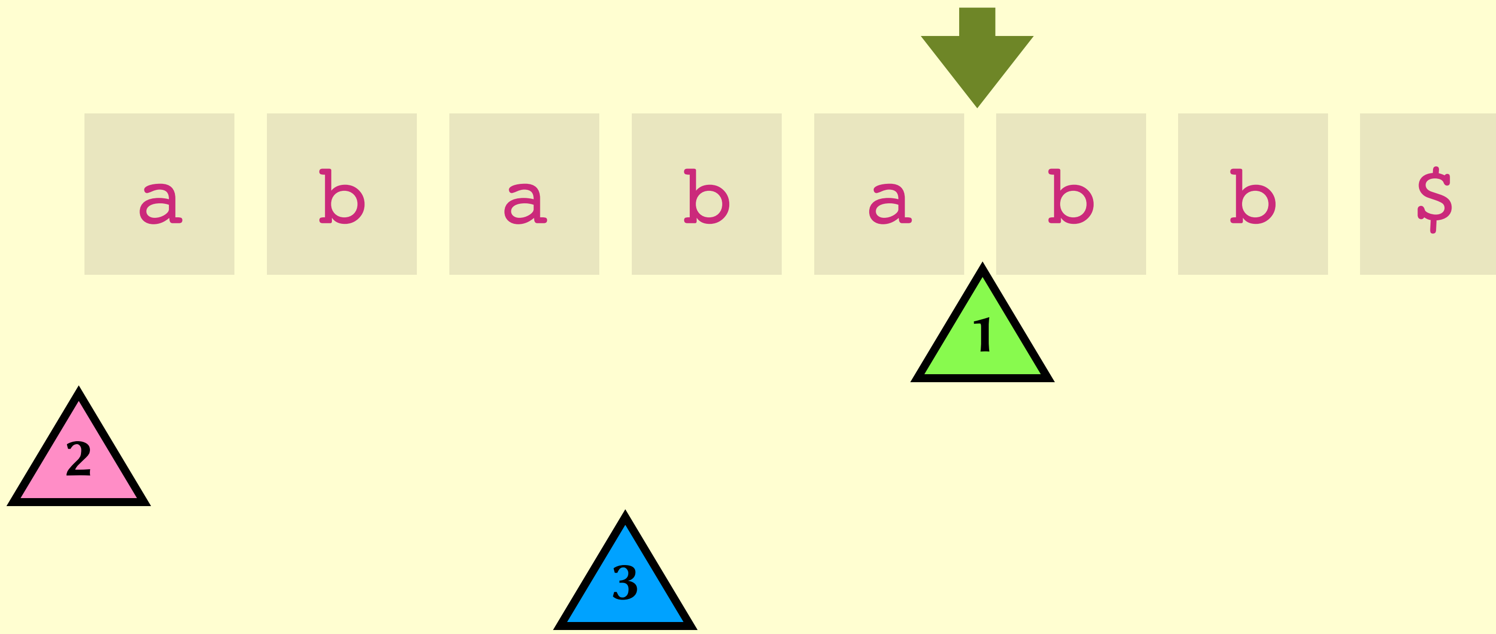



2. (bb)*(aa)*

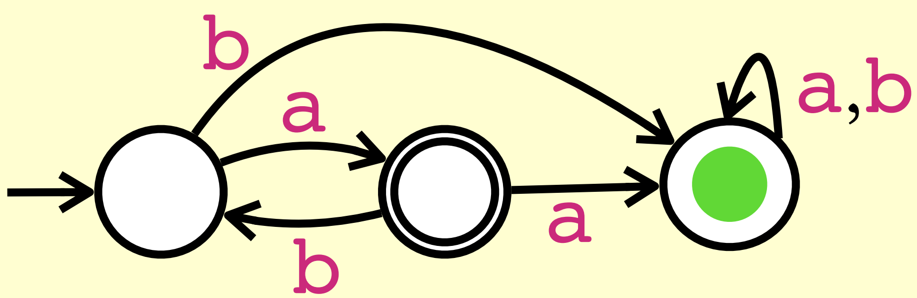



3. a(**1**+b)a

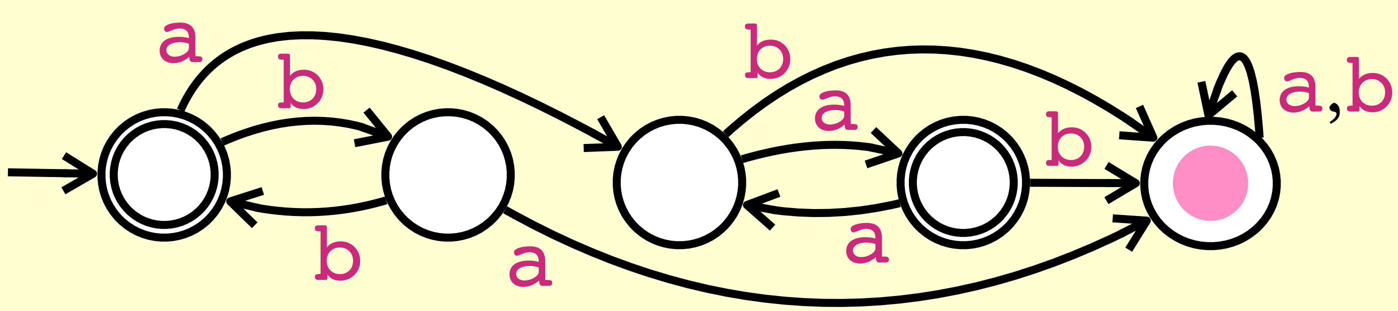




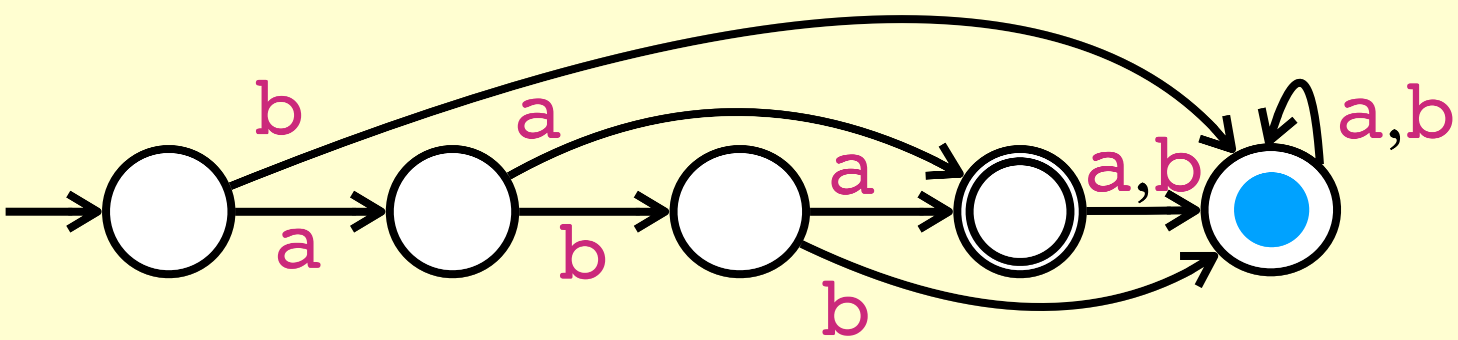

1. (ab)*a

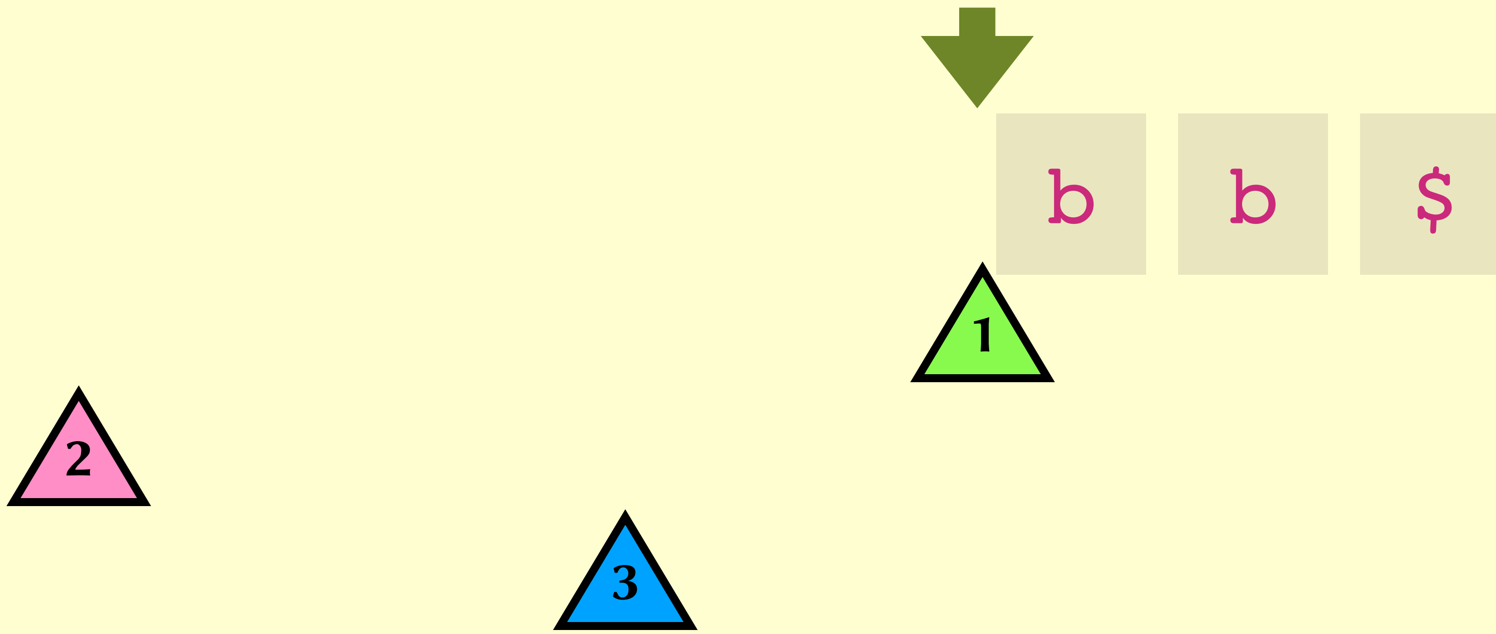



2. (bb)*(aa)*

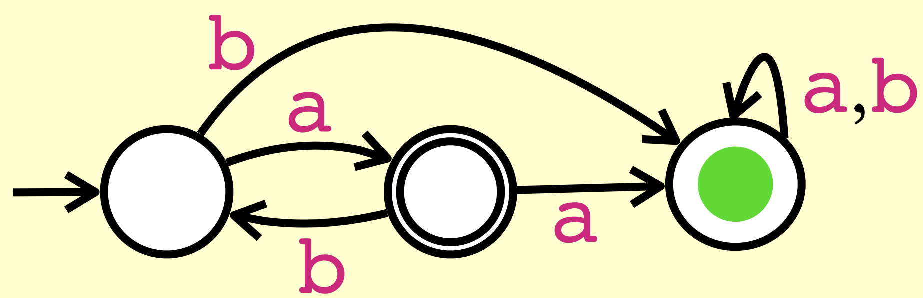



3. a(**1**+b)a

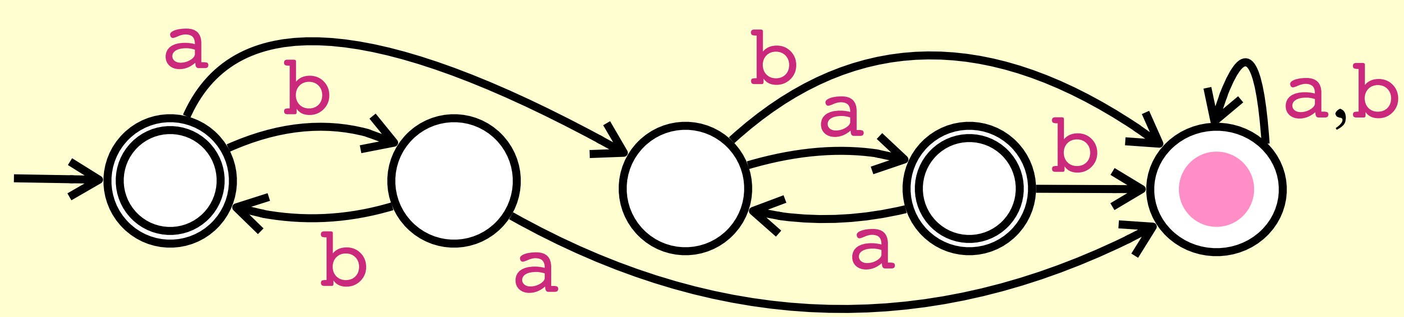




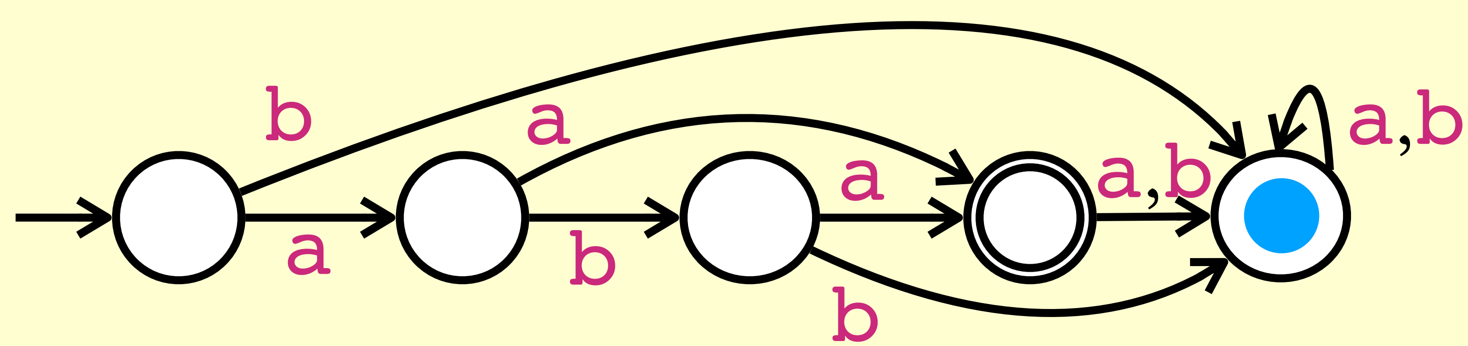

1. (ab)*a



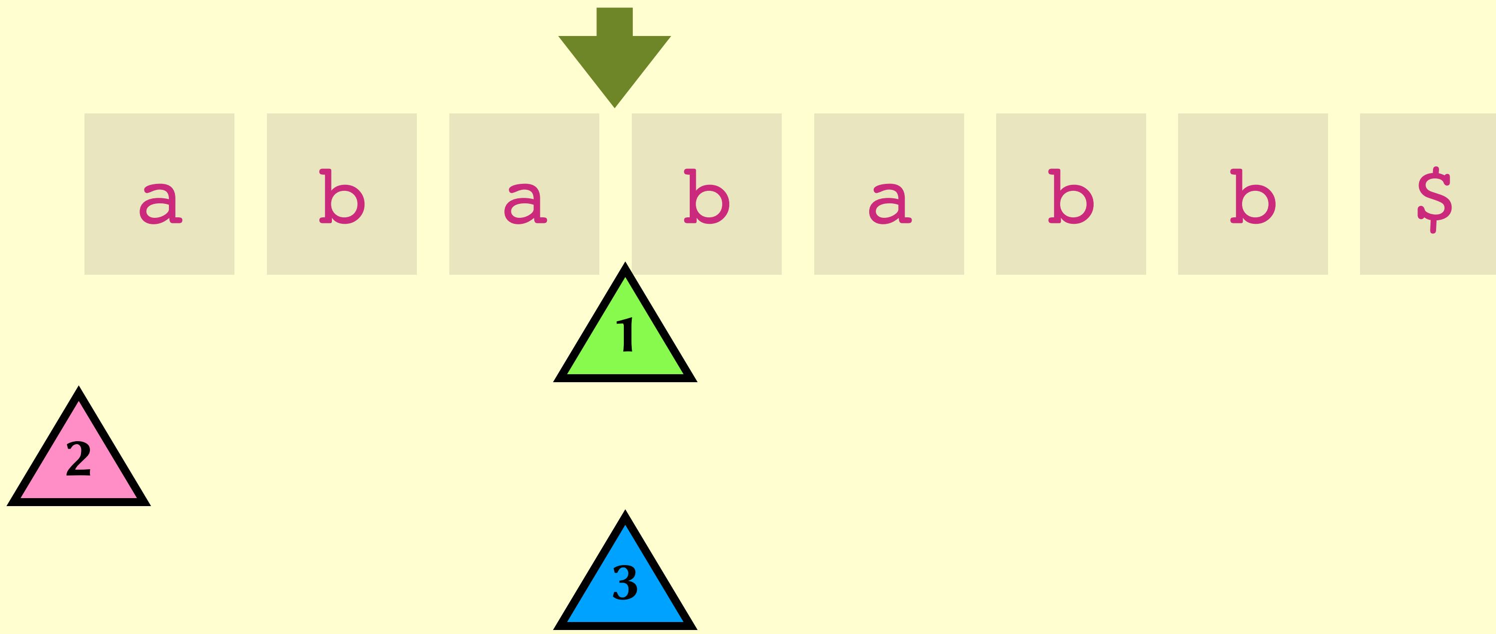

2. (bb)*(aa)*



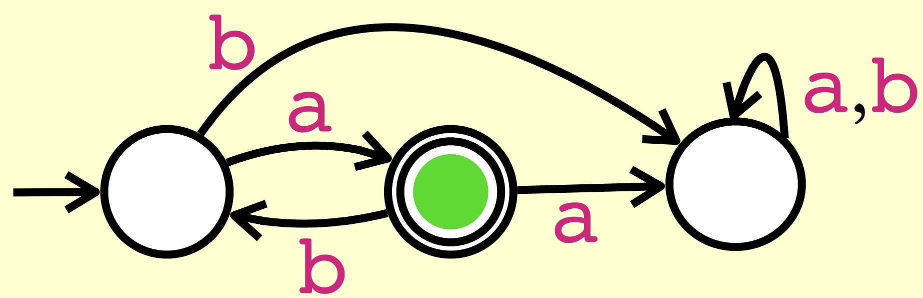

3. a(**1**+b)a



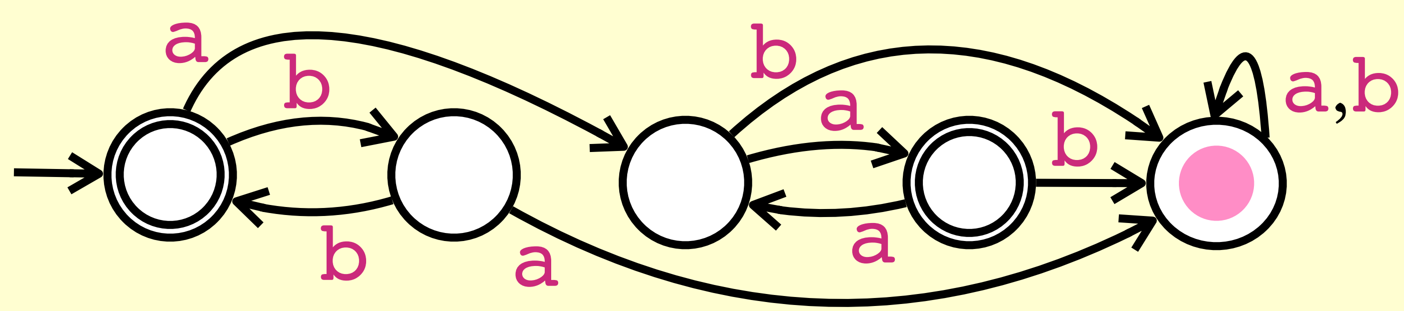

a b a b a



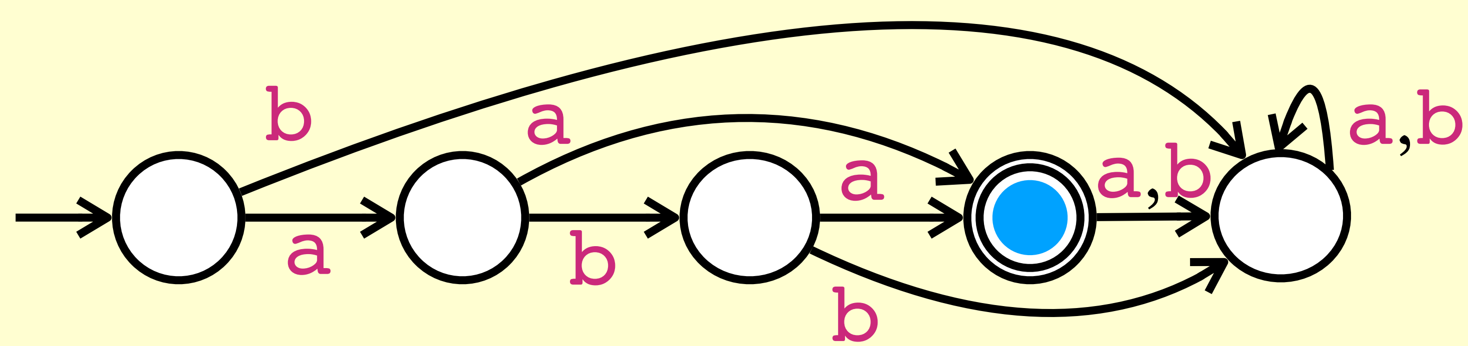

1. $(ab)^{*}a$

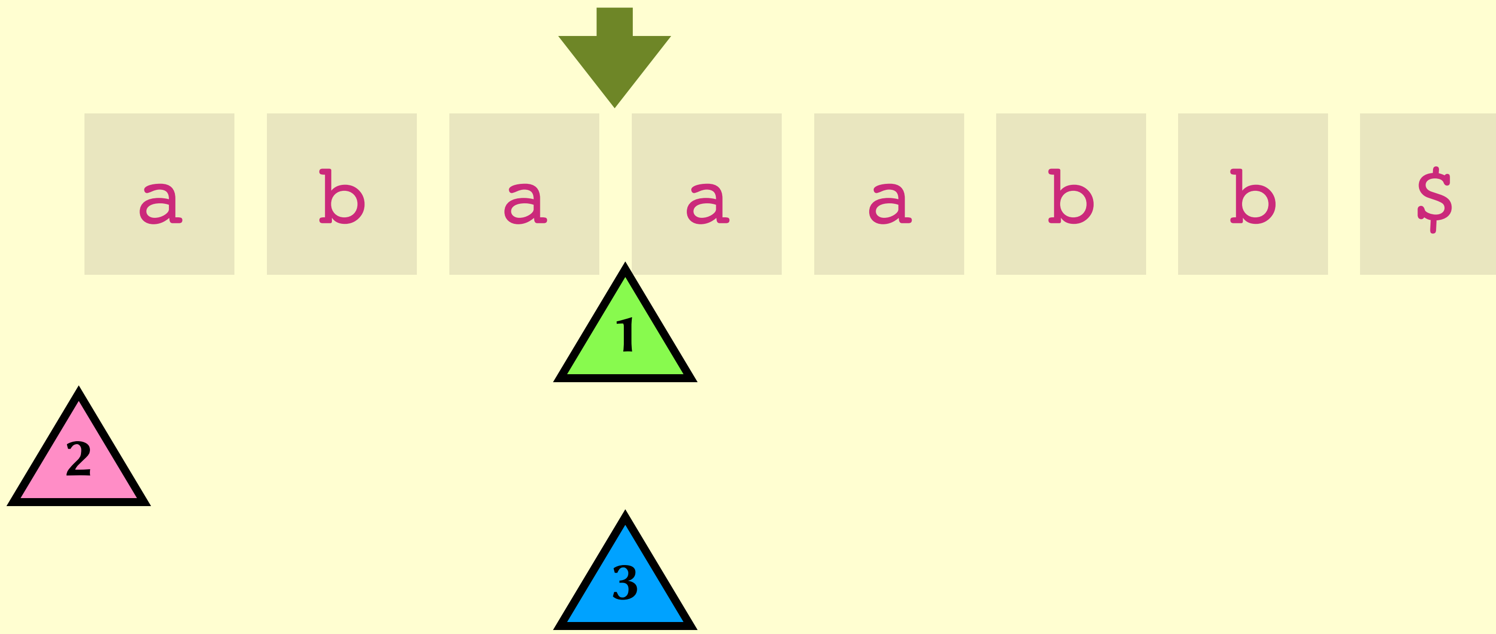



2. $(bb)^{*}(aa)^{*}$

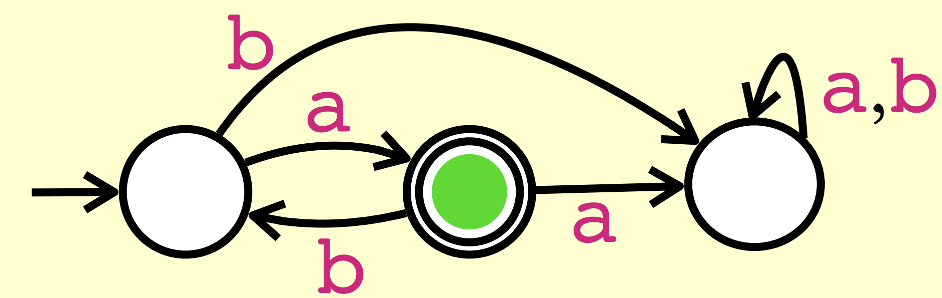



3. $a(\mathbf{1}+b)a$

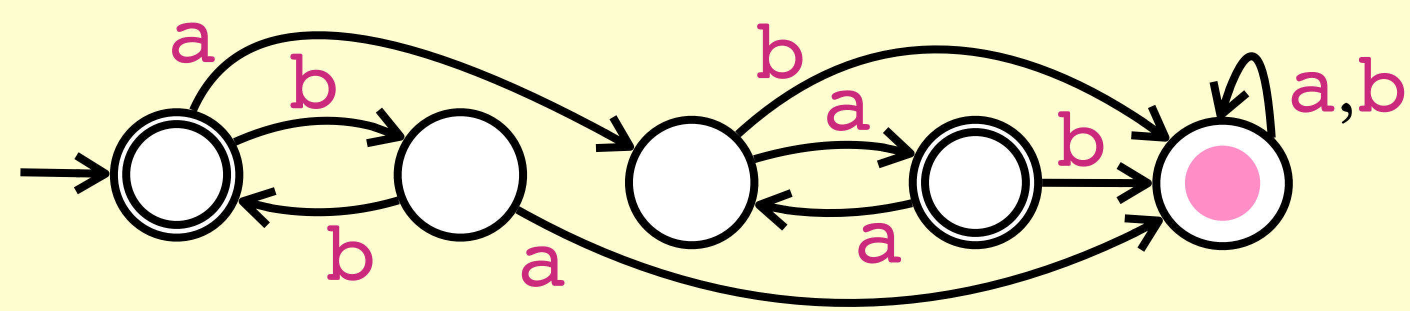




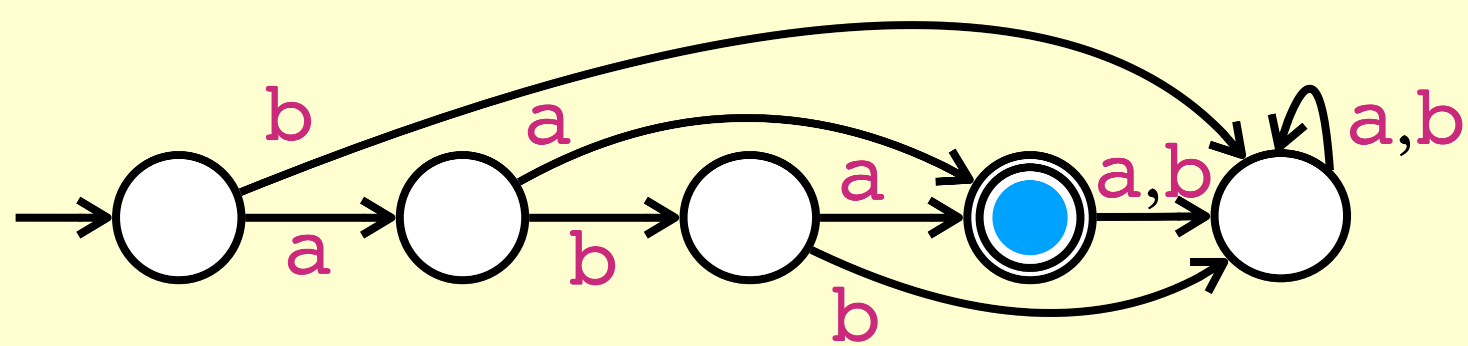

1. $(ab)^*a$

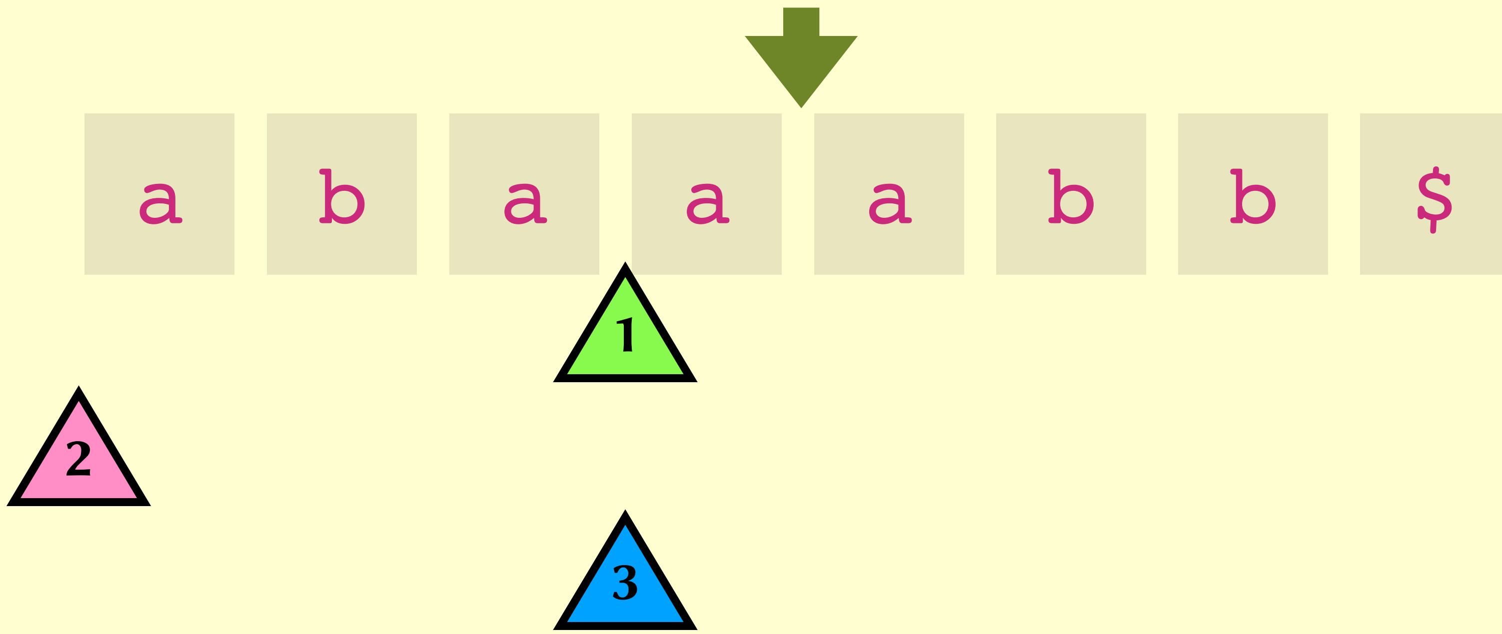



2. $(bb)^*(aa)^*$

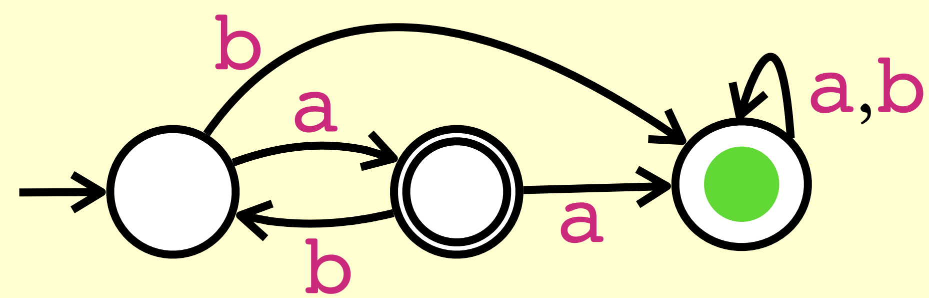



3. $a(\mathbf{1}+b)a$

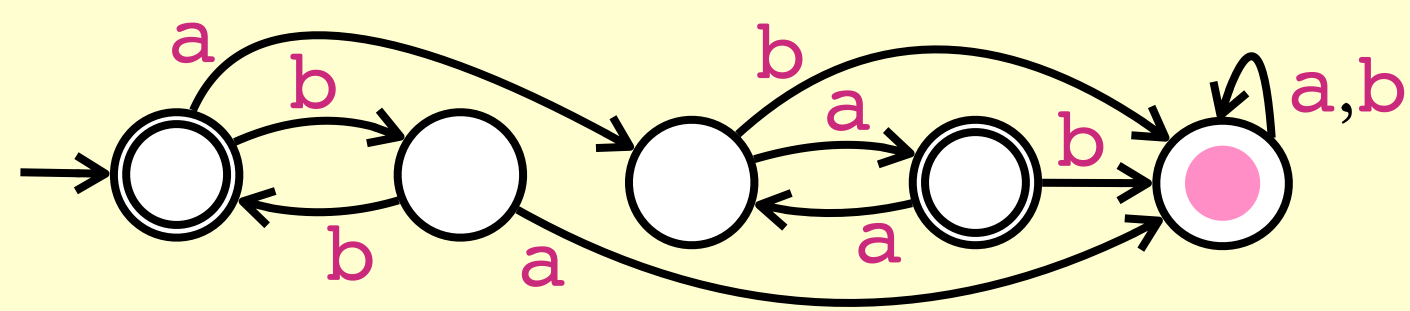




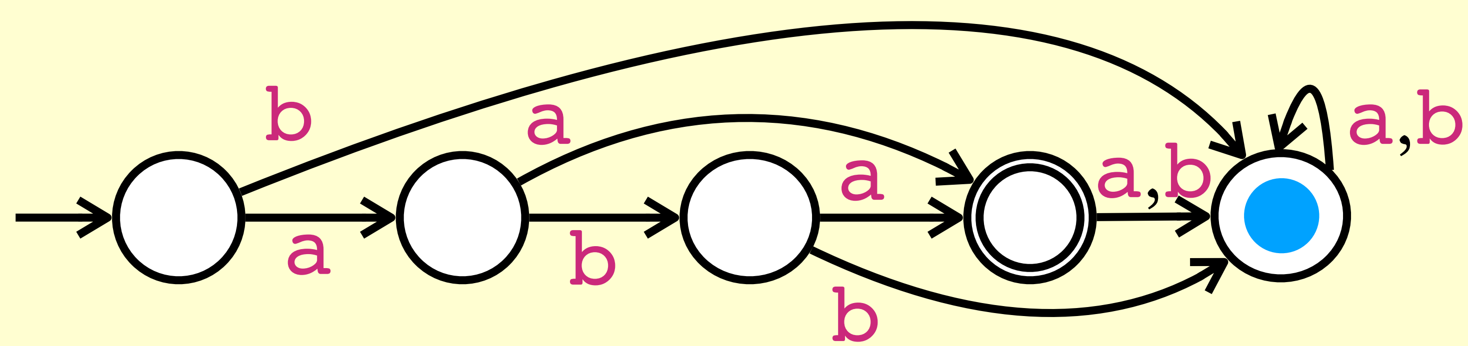

1. $(\mathtt{ab})^*\mathtt{a}$

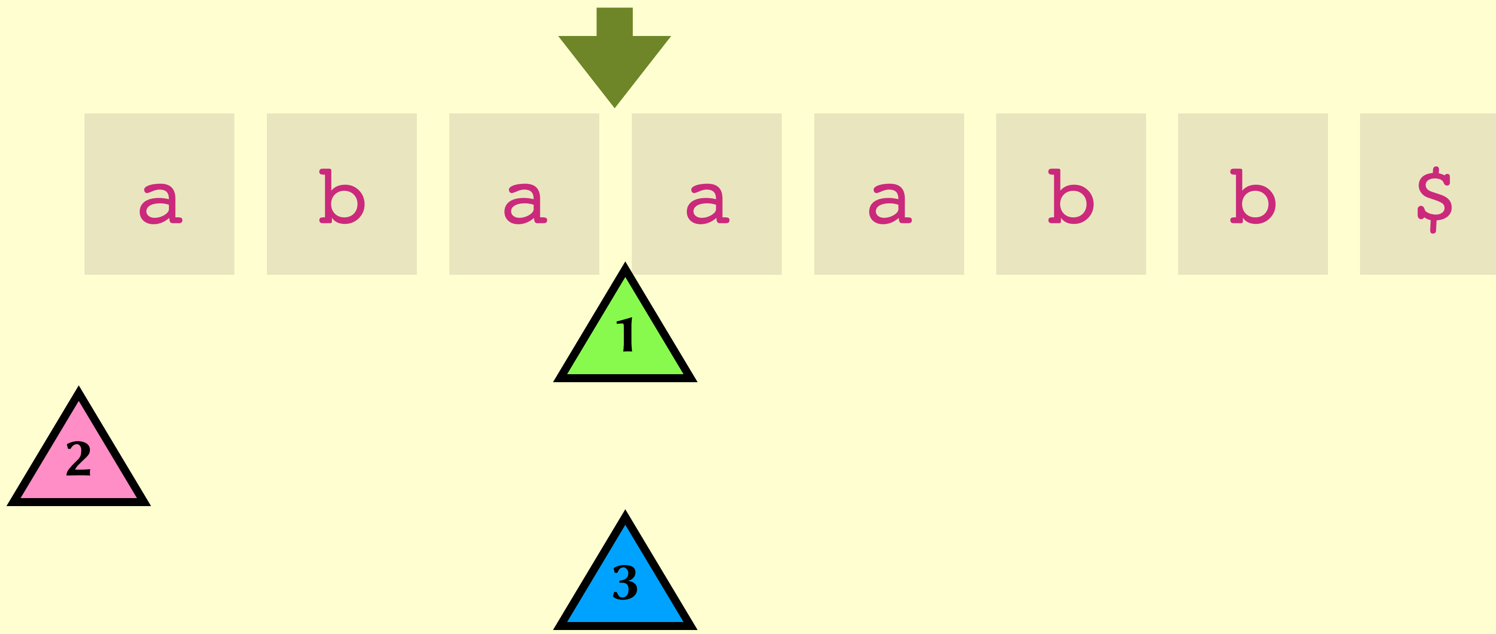



2. $(\mathtt{bb})^*(\mathtt{aa})^*$

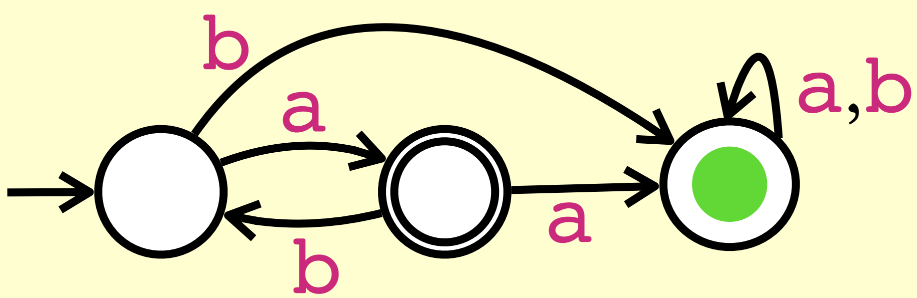



3. $\mathtt{a}(\mathbf{1}+\mathtt{b})\mathtt{a}$

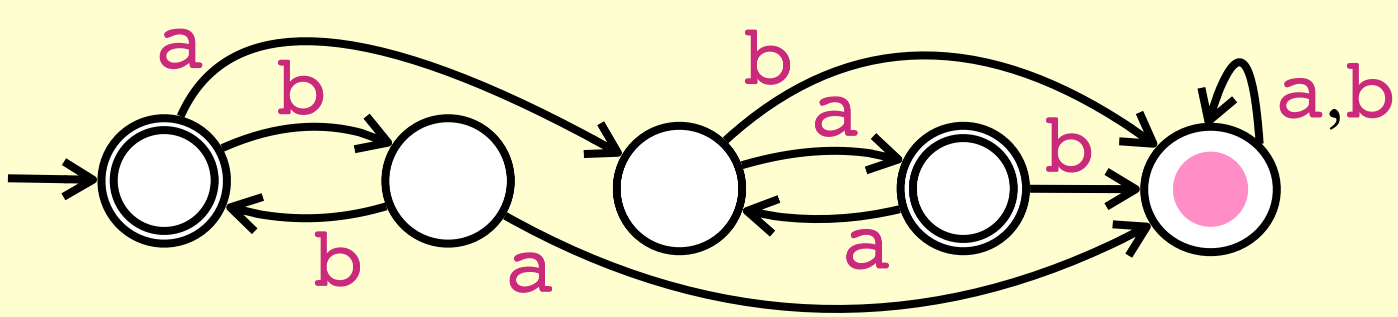




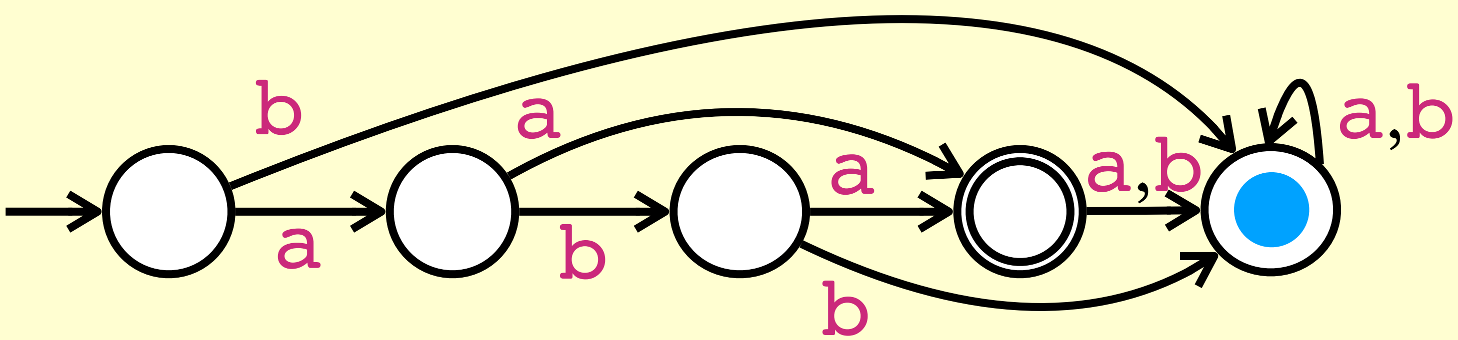

1. $(\mathtt{ab})^*\mathtt{a}$

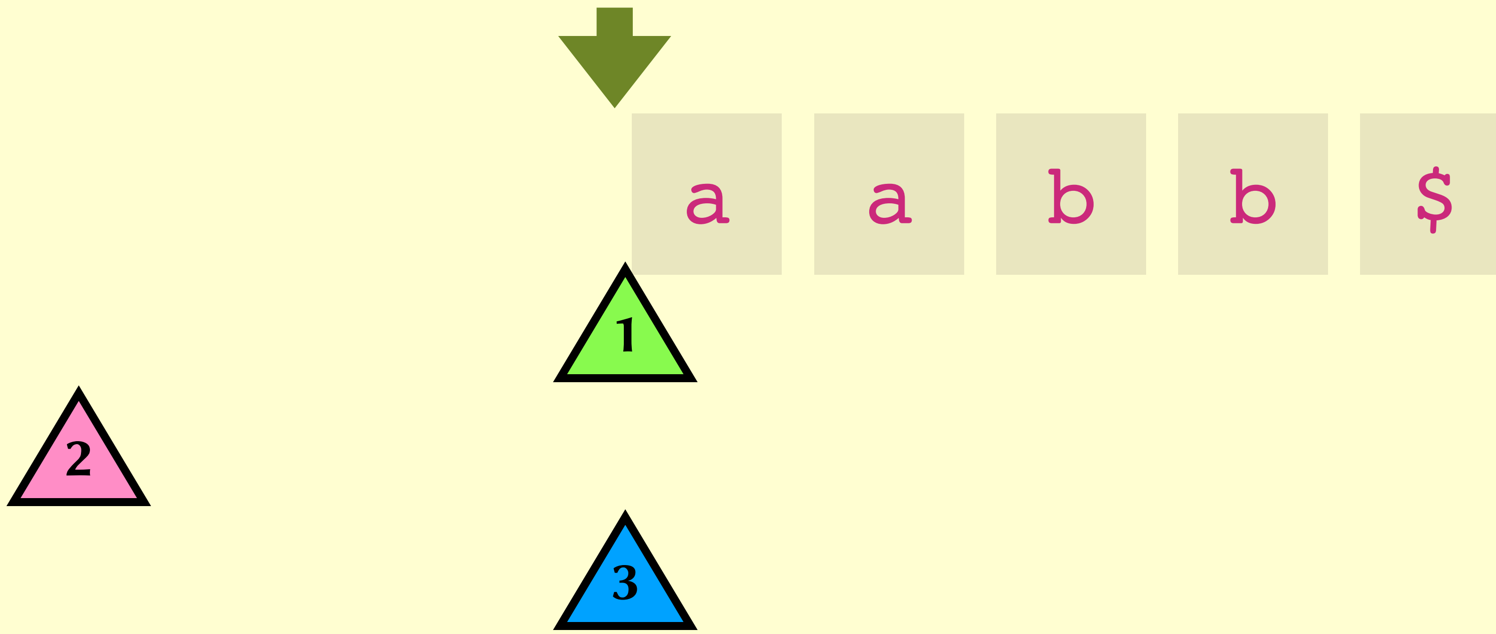



2. $(\mathtt{bb})^*(\mathtt{aa})^*$

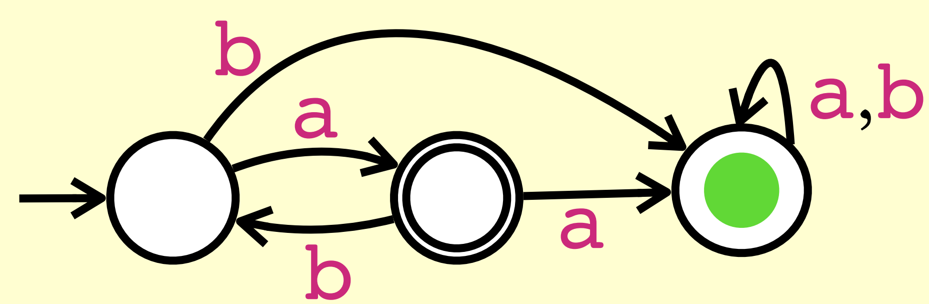



3. $\mathtt{a}(\mathbf{1}+\mathtt{b})\mathtt{a}$

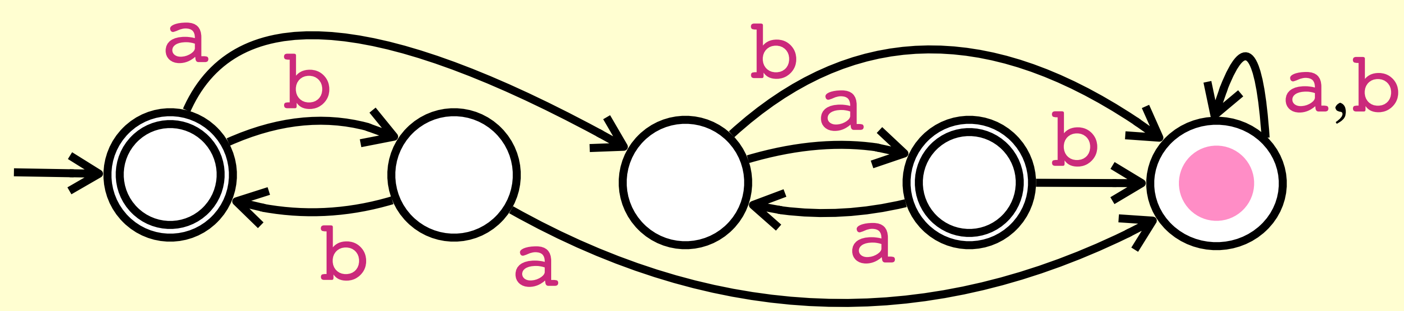




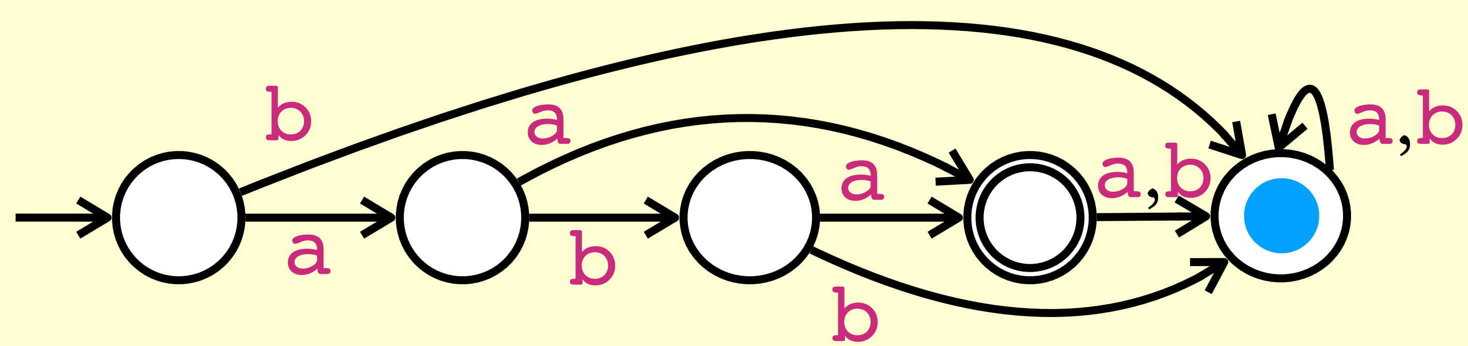

1. (ab)*a




2. (bb)*(aa)*




3. a(**1**+b)a




a b a

# What's next?

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