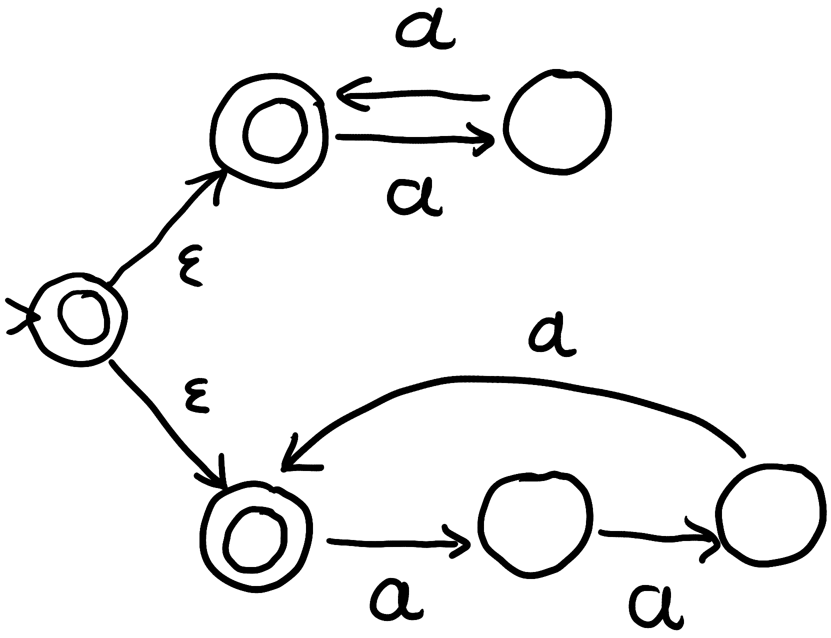
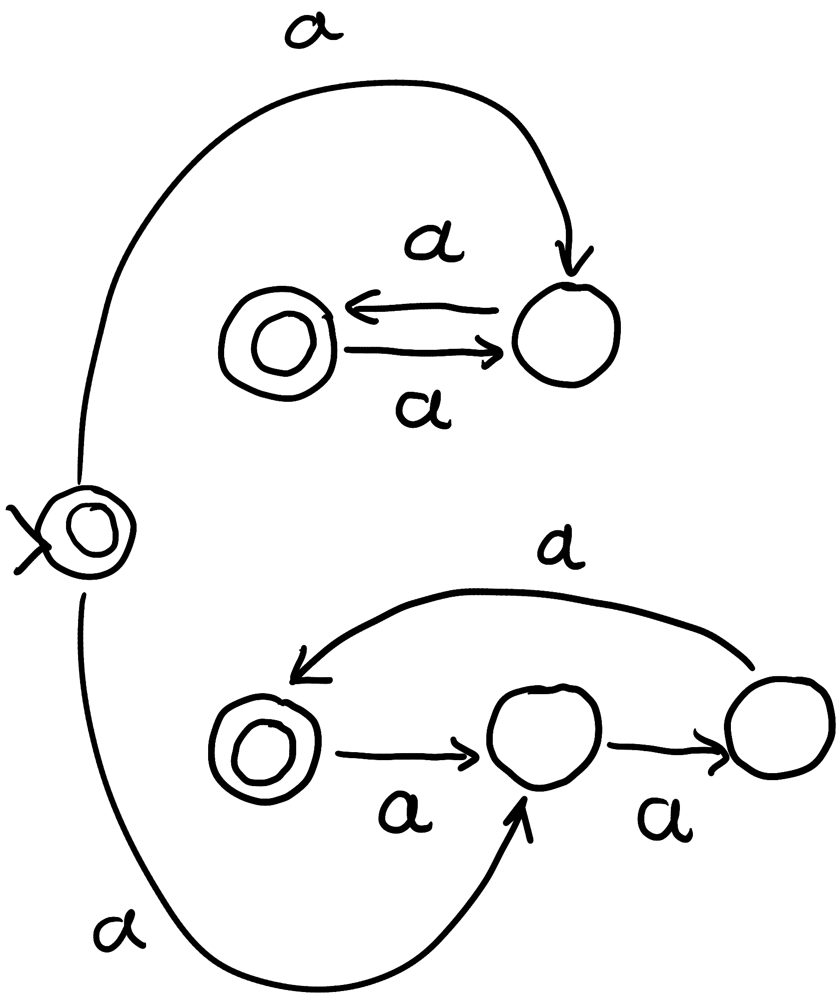
Exercise: (aa)* | (aaa)*

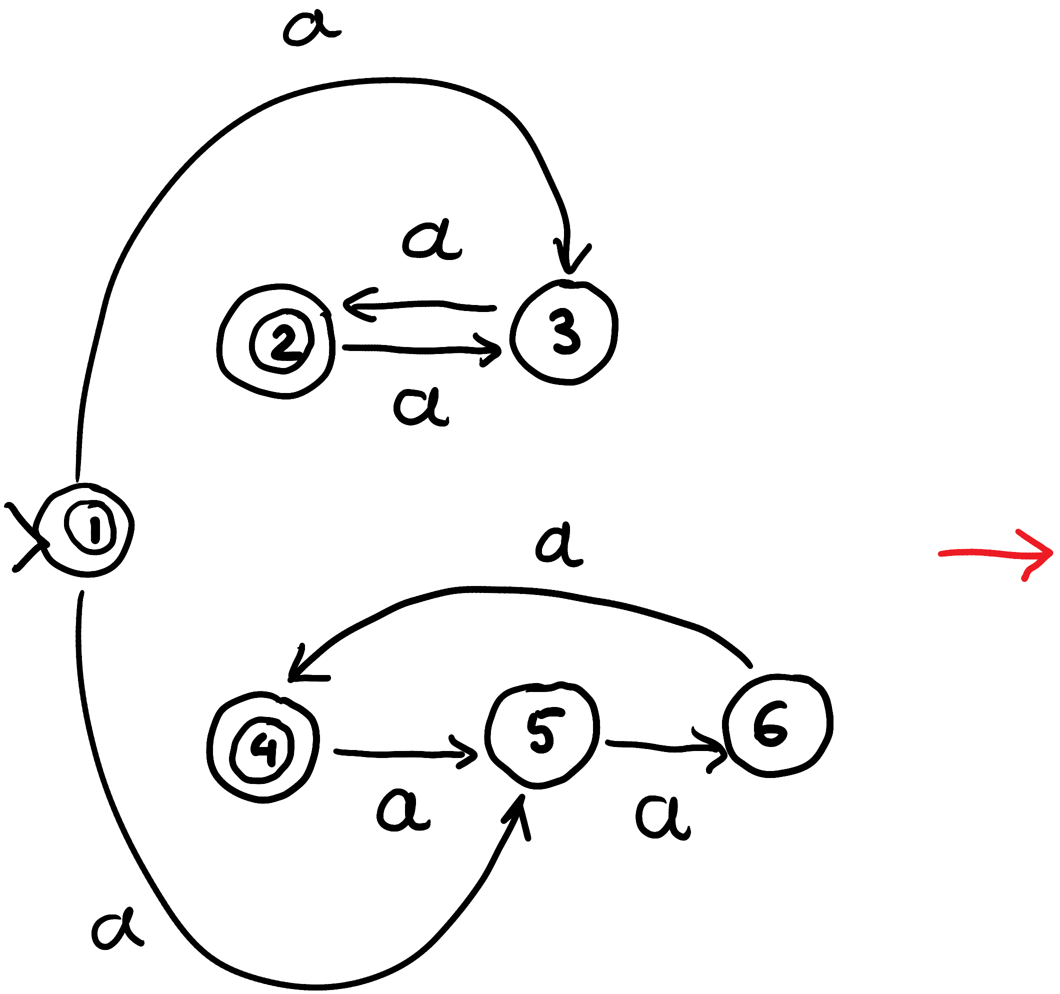
Construct automaton and eliminate epsilons

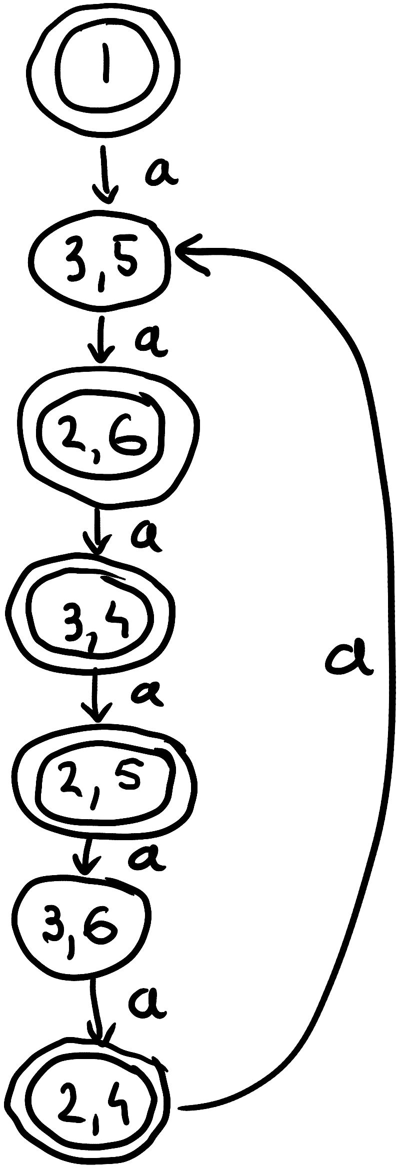



a
a
a
a
a
a
a
a

Determinization: Subset Construction

  - keep track of a set of all possible states in which automaton could be
  - view this finite set as one state of new automaton
- Apply to (aa)* | (aaa)*
  - can also eliminate epsilons during determinization





$\delta \subseteq Q \times \Sigma \times Q$

$\delta = \{ \ldots (1,a,3), (1,a,5), \ldots \}$

$\delta'(S,a) = \{ q_2 \mid \exists q_1 \in S . (q_1,a,q_2) \in \delta \}$ $\delta' : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$

Remark: Relations and Functions

- Relation $r \subseteq B \times C$

  $r = \{ \ldots, (b,c1) , (b,c2) ,\ldots \}$
- Corresponding function: $f : \underline{B} \to \underline{\mathscr{P}(C)}$ $\underline{2^C}$

  $f = \{ \ldots (b,\{c1,c2\}) \ldots \}$

$f(b) = \{ c \mid (b,c) \in r \}$

- Given a state, next-state function returns the set of new states
  - for deterministic automaton, the set has exactly 1 element

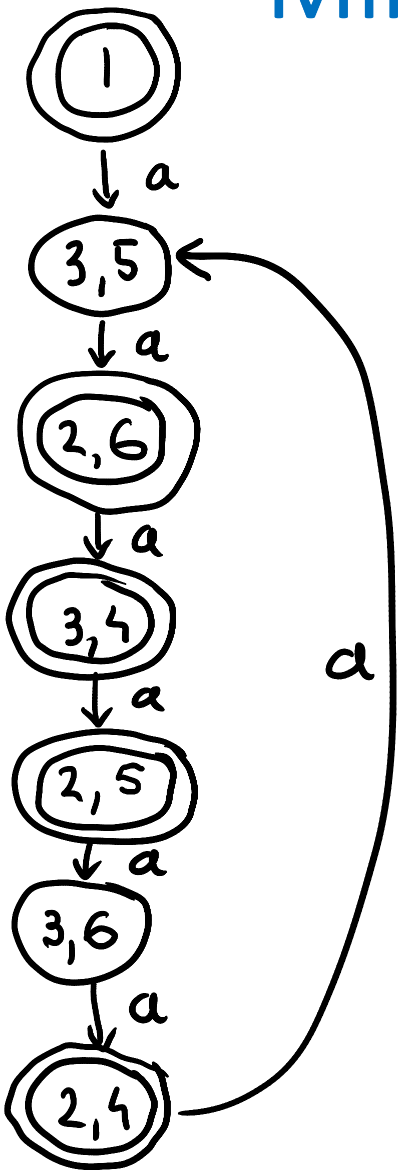
Running NFA in Scala

```
def δ(q : State, a : Char) : Set[States] = { ... }
def δ'(S : Set[States], a : Char) : Set[States] = {
  for (q1 <- S, q2 <- δ(q1,a)) yield q2
}
def accepts(input : MyStream[Char]) : Boolean = {
 var S : Set[State] = Set(q0) // current set of states
 while (!input.EOF) {
  val a = input.current
  S = δ'(S,a)          // next set of states
 }
 !(S.intersect(finalStates).isEmpty)
}
```

Minimization: Merge States

- Only limit the freedom to merge (prove !=) if we have evidence that they behave differently (final/non-final, or leading to states shown !=)
- When we run out of evidence, merge the rest
  - merge the states in the previous automaton for (aa)* | (aaa)*
- Very special case: if successors lead to same states on all symbols, we know immediately we can merge
  - but there are cases when we can merge even if successors lead to merged states

Minimization for example



Start from all accepting disequal all non-accepting.

Result:
only {1} and {2,4} are merged.

Here, the special case is sufficient, but in general, we need the above construction (take two copies of same automaton and union them).

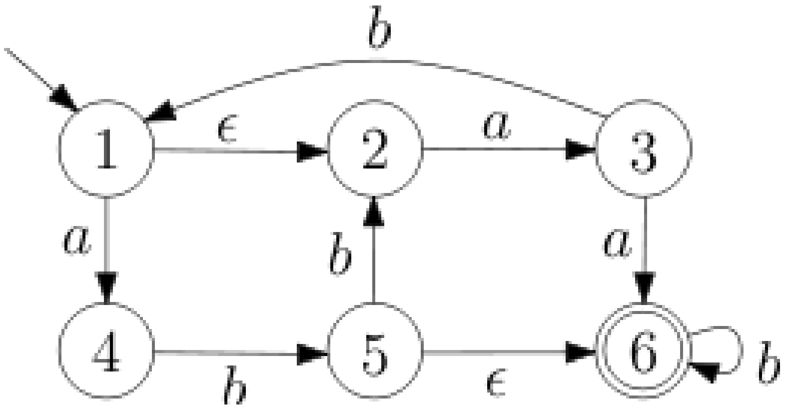
Clarifications

- Non-deterministic state machines where a transition on some input is not defined
- We can apply determinization, and we will end up with
  - singleton sets
  - empty set (this becomes trap state)
- Trap state: a state that has self-loops for all symbols, and is non-accepting.

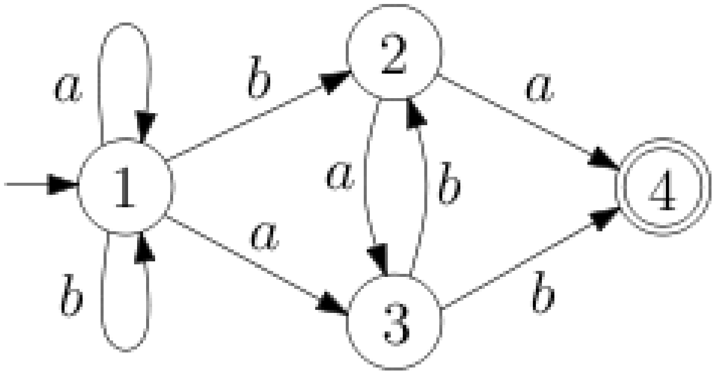
Exercise

Convert the following NFAs to deterministic finite automata.

a)



b)



done on board

left for self-study

Complementation, Inclusion, Equivalence

- Can compute complement: switch accepting and non-accepting states in **deterministic** machine (wrong for non-deterministic)
- We can compute intersection, inclusion, equivalence
- Intersection: complement union of complements
- Set difference: intersection with complement
- Inclusion: emptiness of set difference
- Equivalence: two inclusions

Short Demo

- Automated interactive tool for practicing finite-state automata constructions

- See home page of **Damien Zufferey**

  http://pub.ist.ac.at/~zufferey/

Exercise: first, nullable

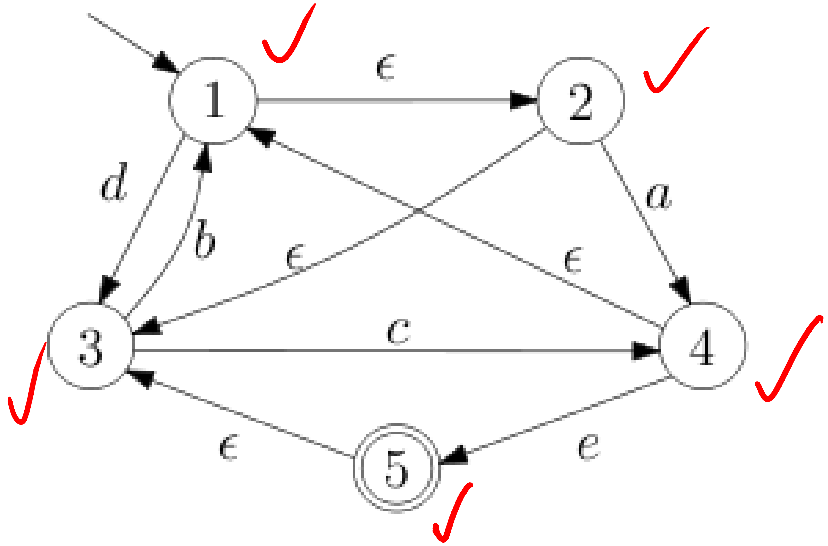
- For each of the following languages find the *first* set. Determine if the language is *nullable*.

$\text{first}\left(\underbrace{(a|b)^*\ \underbrace{(b|d)}}\ ((c|a|d)^* \mid a^*)\right) = \{a, b, d\}$

  - language given by automaton: closure(1) = {1, 2, 3}

first(A) = {d, a, b, c}

A:



Automated Construction of Lexers

- let $r_1, r_2, \ldots, r_n$ be regular expressions for token classes
- consider combined regular expression: $(\underline{r_1} \mid \underline{r_2} \mid \ldots \mid \underline{r_n})^* \leftarrow$
- recursively map a regular expression to a non-deterministic automaton
- eliminate epsilon transitions and determinize
- optionally minimize A_3 to reduce its size → A_4
- **the result only checks that input can be split into tokens, does not say how to split it**

From $(r_1|r_2|\ldots|r_n)^*$ to a Lexer

- Construct machine for each r_i labelling different accepting states differently
- for each accepting state of r_i specify the token class *i* being recognized
- longest match rule: remember last token and input position for a last accepted state
- when no accepting state can be reached (effectively: when we are in a trap state)
  - revert position to last accepted state
  - return last accepted token

Exercise: Build Lexical Analyzer Part

For these two tokens, using longest match,

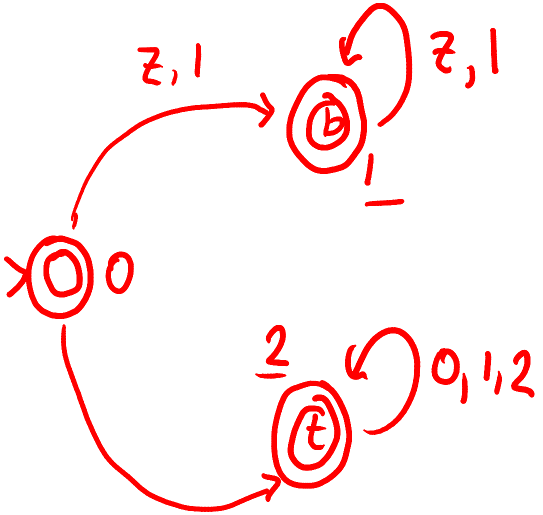
where first has the priority:

binaryToken ::= (**z**|1)*

ternaryToken ::= (0|1|2)*

$(z|1)^* \mid (0|1|2)^*$

1111z1021z1 →



$\{0\} \xrightarrow{1} \{1,2\} \xrightarrow{1} \{1,2\} \ldots \xrightarrow{1} \{1,2\} \xrightarrow{z} \overset{0,1,2}{\{1\}} \xrightarrow{1} \{1\} \xrightarrow{0} \emptyset$

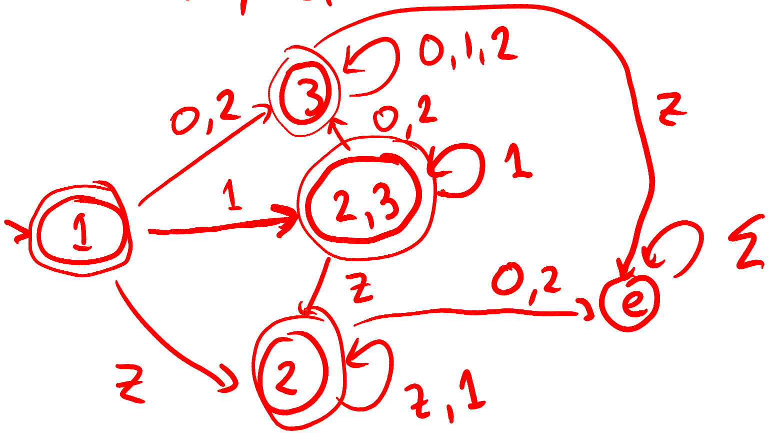
Lexical Analyzer

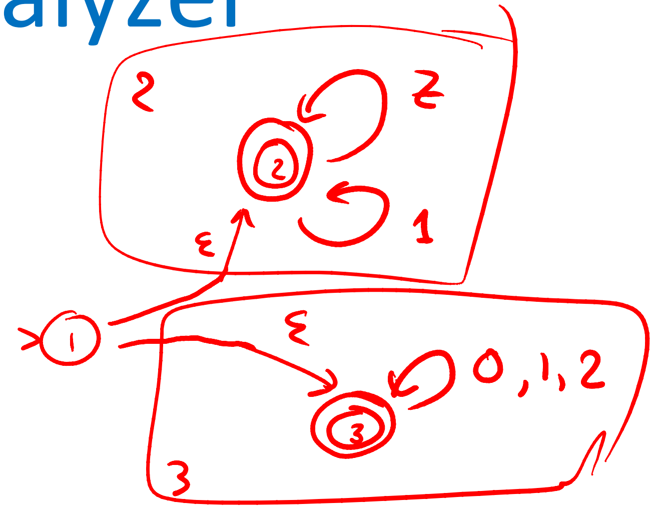
1) binaryToken ::= $(\mathbf{z}|1)^*$

2) ternaryToken ::= $(0|1|2)^*$

$\uparrow\underbrace{1111z1}_{\text{binary}}\underbrace{021}_{\text{tern}}\underbrace{z1}_{\text{binary}} \rightarrow$

$\underbrace{1\ 1\ 1\ 1\ 1}_{\text{binary (priority)}}\ 5$





(binaryToken | ternaryToken)*
↑

Σ = {0,1,2,z}