Automating Construction of Lexers

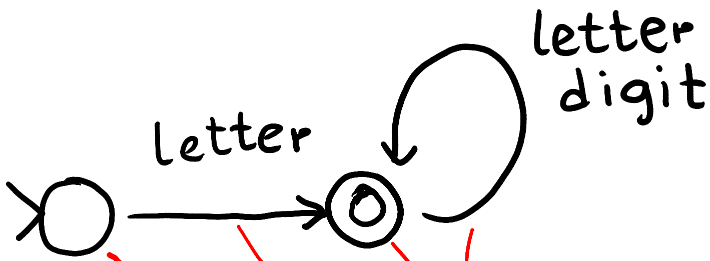
Example in javacc

```
TOKEN: {
   <IDENTIFIER: <LETTER> (<LETTER> | <DIGIT> | "_")* >
  | <INTLITERAL: <DIGIT> (<DIGIT>)* >
  | <LETTER: ["a"-"z"] | ["A"-"Z"]>
  | <DIGIT: ["0"-"9"]>
}
SKIP: {
  " "  |  "\n"  |   "\t"
}
```

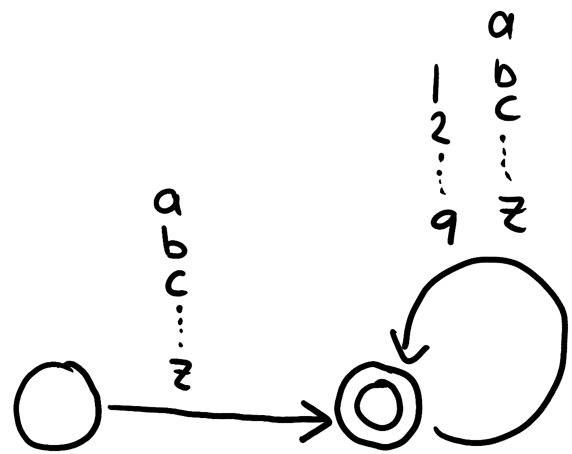
--> get automatically generated code for lexer!

But how does javacc do it?

Finite Automaton (Finite State Machine)



i.e.



$$A = (\Sigma, Q, q_0, \delta, F)$$

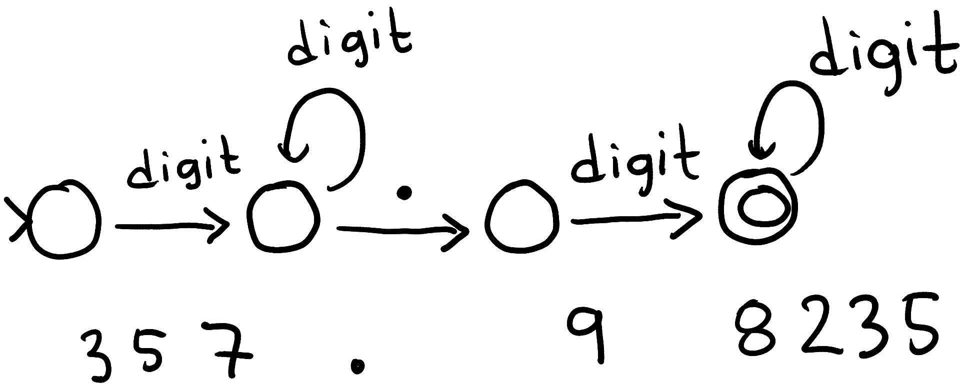
$$\delta \subseteq Q \times \Sigma \times Q$$

$$(q_1, a, q_2) \in \delta$$

$$q_1 \xrightarrow{a} q_2$$

- Σ - alphabet
- Q - states (nodes in the graph)
- q_0 - initial state (with '>' sign in drawing)
- δ - transitions (labeled edges in the graph)
- F - final states (double circles)

Numbers with Decimal Point



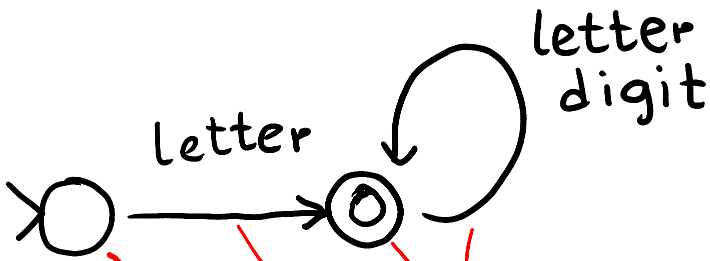
digit digit* . digit digit*

What if the decimal part is optional?

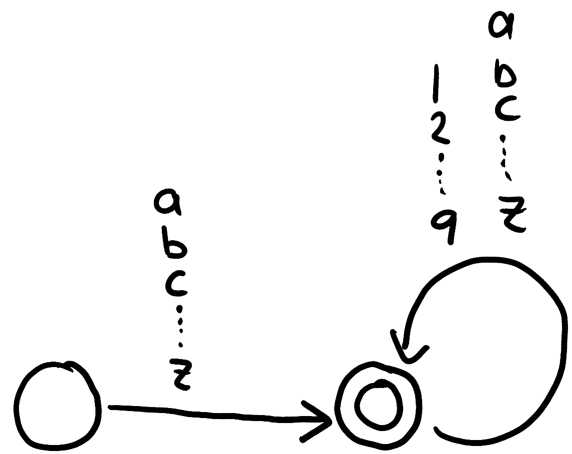
Exercise

- Design a DFA which accepts all the numbers written in binary and divisible by 6. For example your automaton should accept the words 0, 110 (6 decimal) and 10010 (18 decimal).

Kinds of Finite State Automata



i.e.

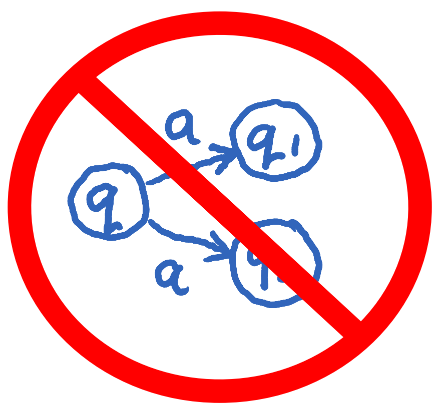


$$A = (\Sigma, Q, q_0, \delta, F)$$

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

$$(q_1, a, q_2) \in \delta$$

- Deterministic: δ is a function

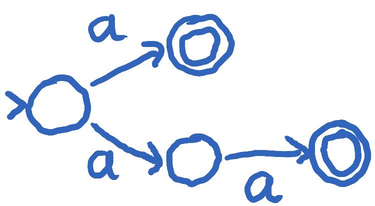


$$\frac{(q, a, q_1) \in \delta \qquad (q, a, q_2) \in \delta}{q_1 = q_2}$$

- Otherwise: non-deterministic

Interpretation of Non-Determinism

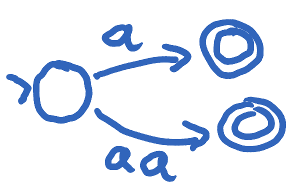
A:



$\Sigma = \{a\}$

$L(A) = \{a, aa\}$

- For a given word (string), a path in automaton lead to accepting, another to a rejecting state
- Does the automaton accept in such case?
  - yes, if there **exists** an accepting path in the automaton graph whose symbols give that word
- Epsilon transitions: traversing them does not consume anything (empty word)
- More generally, transitions labeled by a word: traversing such transition consumes that entire word at a time



Regular Expressions and Automata

Theorem:

If L is a set of words, then it is a value of a regular expression if and only if it is the set of words accepted by some finite automaton.

Algorithms:

- regular expression → automaton (important!)
- automaton → regular expression (cool)

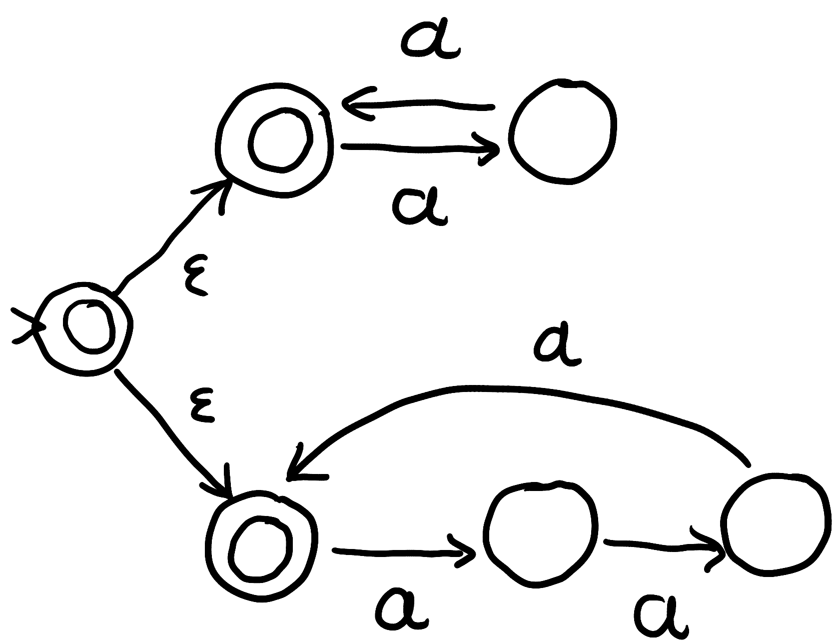
Recursive Constructions

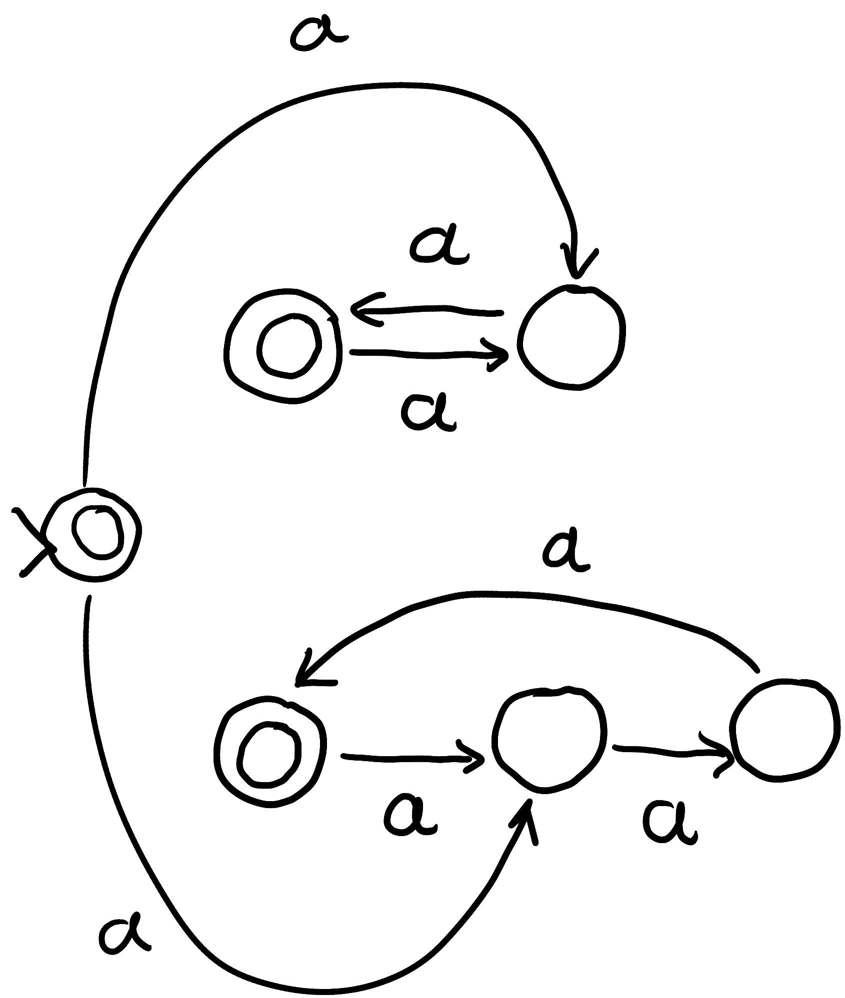
- Union $r_1 \mid r_2$
- Concatenation $r_1 r_2$
- Star r^*

Eliminating Epsilon Transitions

Exercise: (aa)* | (aaa)*

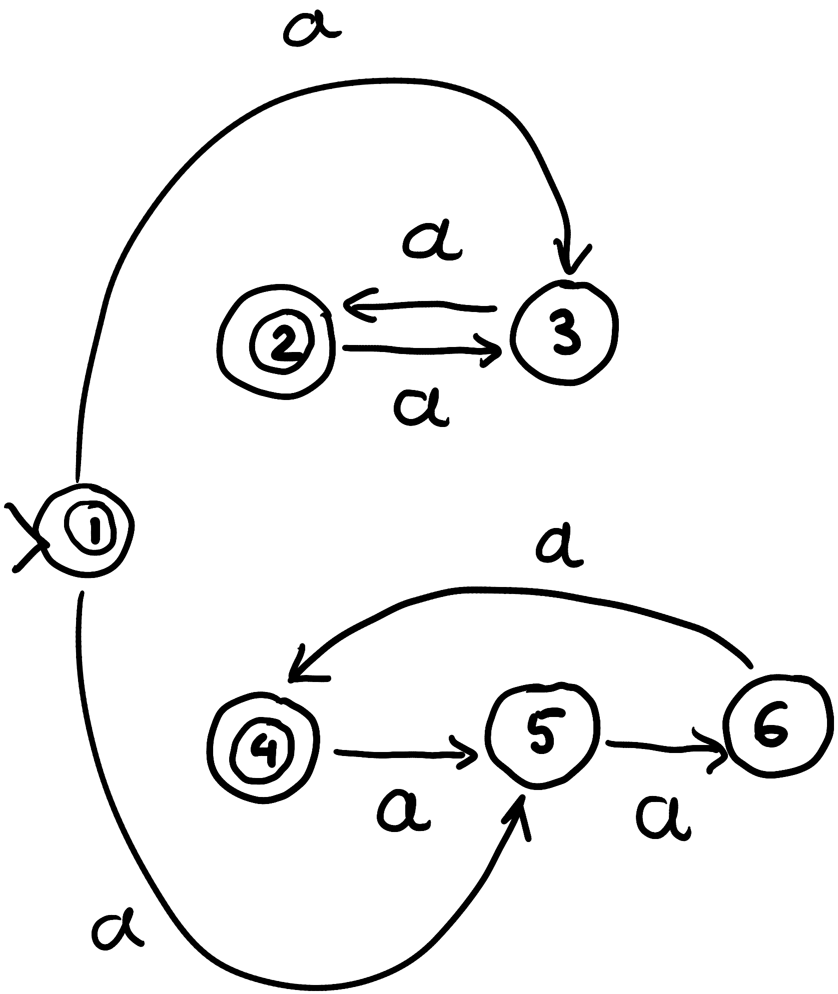
Construct automaton and eliminate epsilons

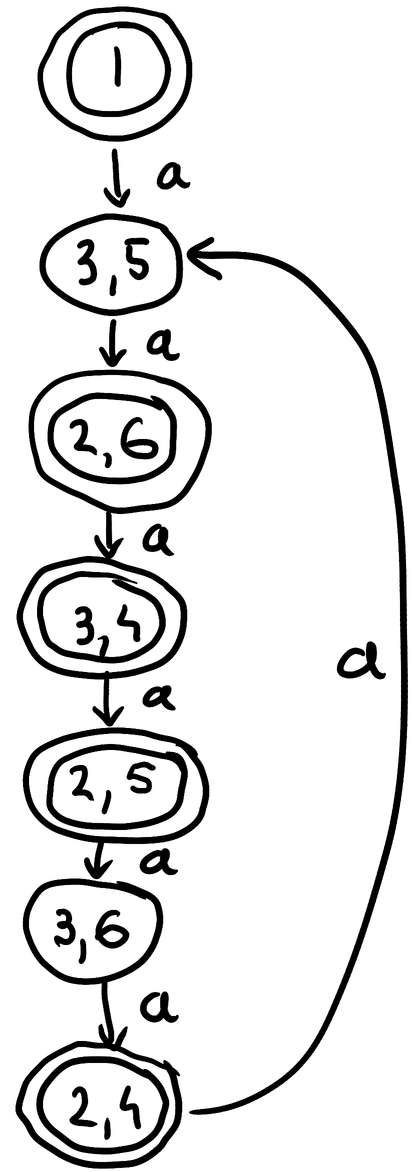



a
a
a
a
a
a
a

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





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

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

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

Remark: Relations and Functions

- Relation $r \subseteq B \times C$

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

- Corresponding function: $f : B \to \mathscr{P}(C)$ 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

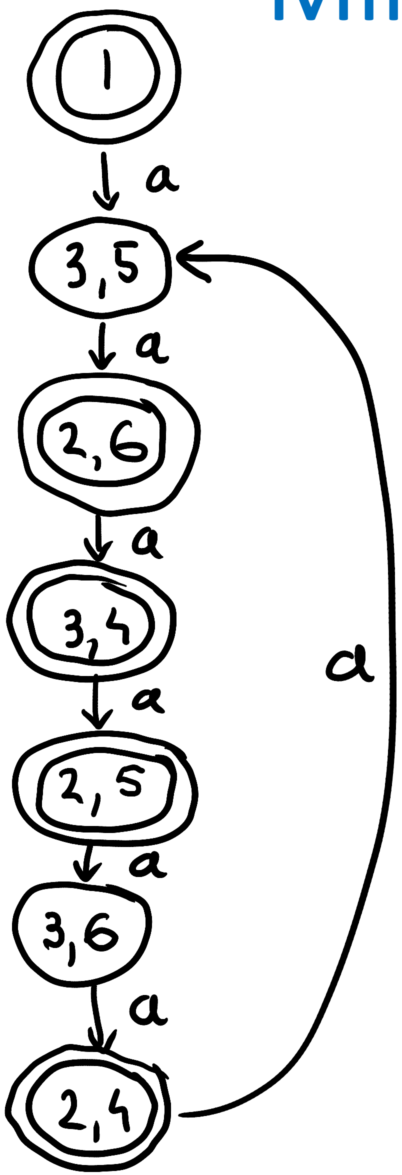
Running NFA in Scala

```
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.

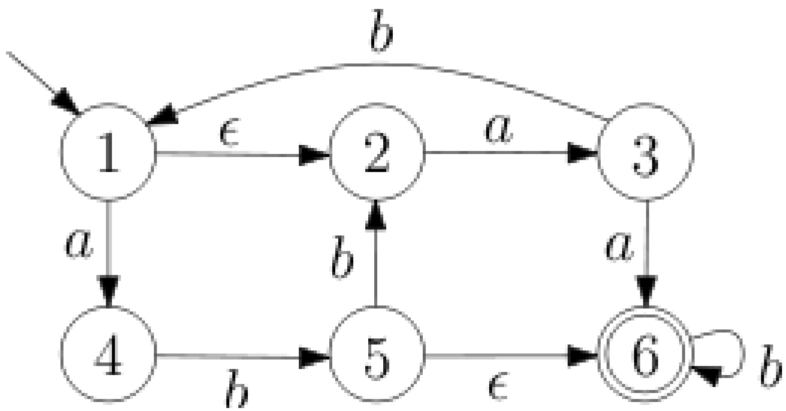
Here, the special case is sufficient, but in general, we need the above construction (take two copies of same automaton and union them).

Clarifications

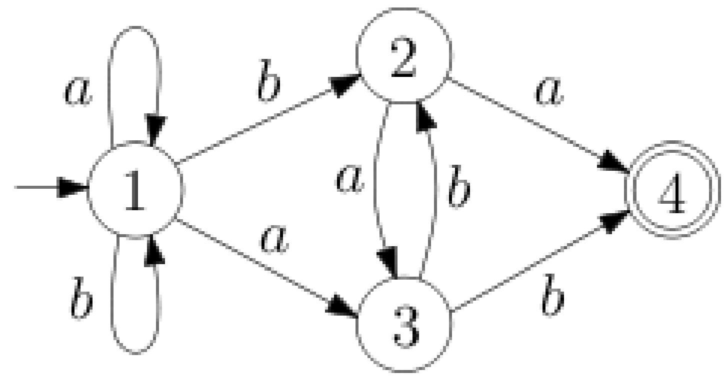
- Non-deterministic state machines where a transition on some input is not defined
- We can apply determinization, and we will end up with
  - singleton sets
  - empty set (this becomes trap state)
- Trap state: a state that has self-loops for all symbols, and is non-accepting.

Exercise

Convert the following NFAs to deterministic finite automata.



a)



b)

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

Exercise: first, nullable

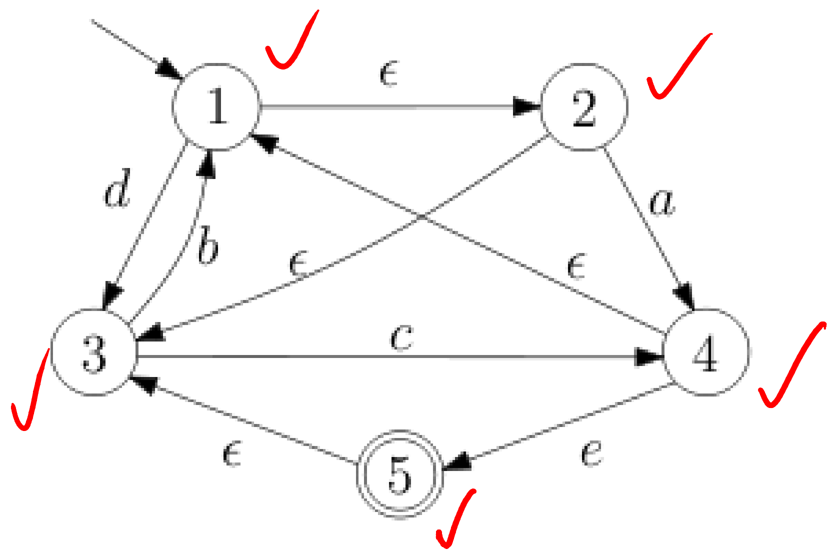
- For each of the following languages find the *first* set. Determine if the language is *nullable*.

first ((a|b)* (b|d) ((c|a|d)* | a*)) = {a, b, d}

– language given by automaton:

closure(1) = {1, 2, 3}

A:



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

Automated Construction of Lexers

- let $r_1, r_2, \ldots, r_n$ be regular expressions for token classes
- consider combined regular expression: $(r_1 \mid r_2 \mid \ldots \mid r_n)^*$
- 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,

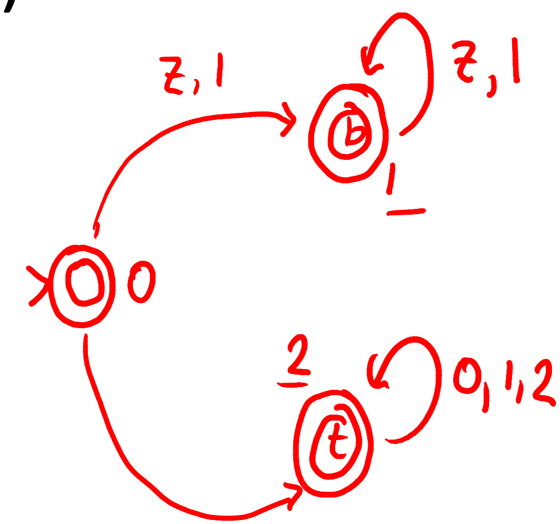
where first has the priority:

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

ternaryToken ::= $(0 \mid 1 \mid 2)^*$

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

1111z1021z1 $\rightarrow$



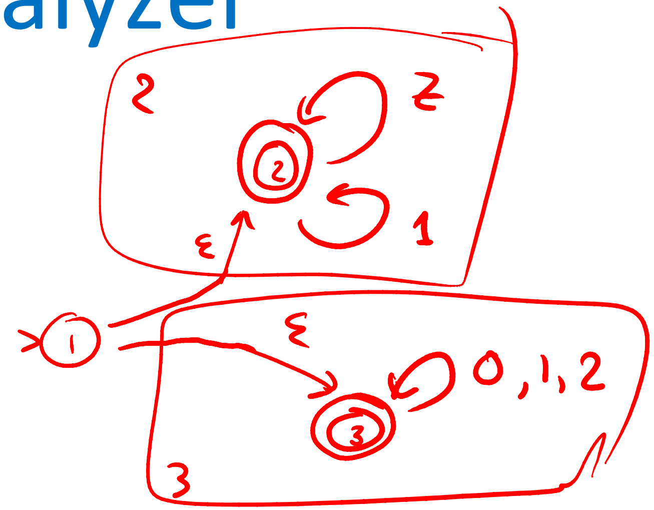
0, 1, 2

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

Lexical Analyzer

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

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



1111z1021z1 →

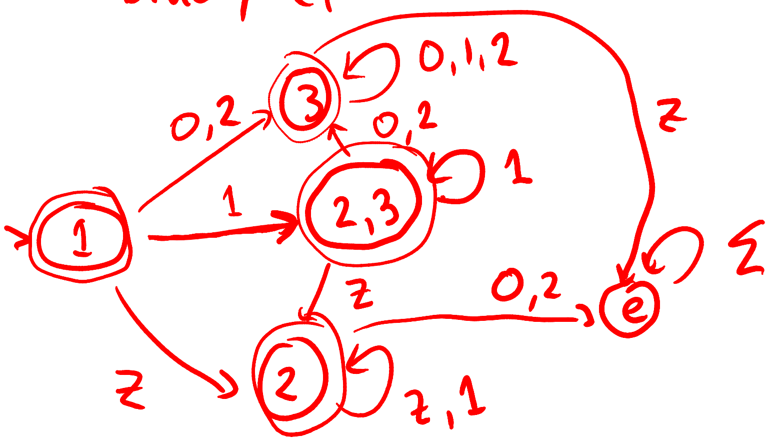
binary terr binary

(binaryToken | ternaryToken)*

1 1 1 1 1 5

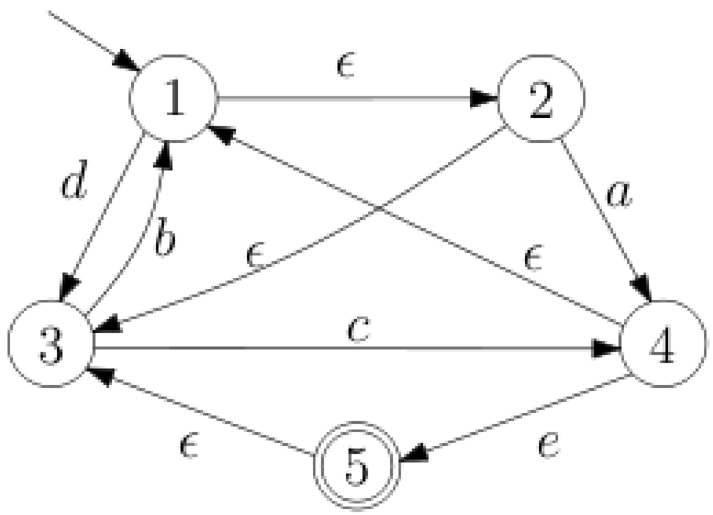
binary (priority)

Σ = {0,1,2,z}



Exercise: first, nullable

- For each of the following languages find the *first* set. Determine if the language is *nullable*.
  - (a|b)*(b|d)((c|a|d)* | a*)
  - language given by automaton:



Exercise: Realistic Integer Literals

- Integer literals are in three forms in Scala: decimal, hexadecimal and octal. The compiler discriminates different classes from their beginning.
  - Decimal integers are started with a non-zero digit.
  - Hexadecimal numbers begin with 0x or 0X and may contain the digits from 0 through 9 as well as upper or lowercase digits A to F afterwards.
  - If the integer number starts with zero, it is in octal representation so it can contain only digits 0 through 7.
  - l or L at the end of the literal shows the number is Long.
- Draw a single DFA that accepts all the allowable integer literals.
- Write the corresponding regular expression.

Exercise

- Let L be the language of strings A = {<, =} defined by regexp (<|=| <====*), that is, L contains <,=, and words $<=^n$ for n>2.
- Construct a DFA that accepts L
- Describe how the lexical analyzer will tokenize the following inputs.

1) <=====

2) ==<==<==<==<==

3) <=====<

More Questions

- Find automaton or regular expression for:
  - Sequence of open and closed parentheses of even length?
  - as many digits before as after decimal point?
  - Sequence of balanced parentheses
    ((()) ()) - balanced
    ( )) (( ) - not balanced
  - Comment as a sequence of space,LF,TAB, and comments from // until LF
  - Nested comments like /* ... /* */ ... */