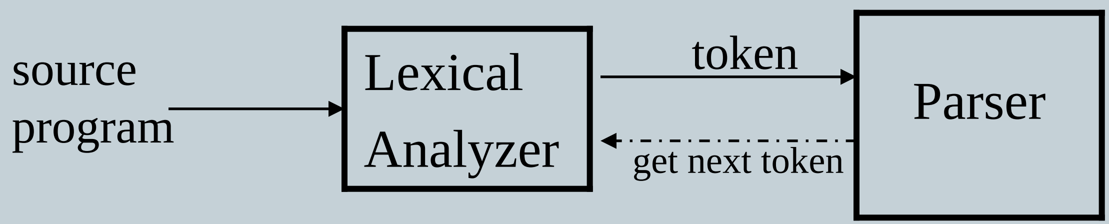
Lexical Analyzer

- Lexical Analyzer reads the source program character by character to produce tokens.
- Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.



Token

- Token represents a set of strings described by a pattern.
  - Identifier represents a set of strings which start with a letter continues with letters and digits
  - The actual string (newval) is called as lexeme.
  - Tokens: identifier, number, addop, delimeter, …
- Since a token can represent more than one lexeme, additional information should be held for that specific lexeme. This additional information is called as the attribute of the token.
- For simplicity, a token may have a single attribute which holds the required information for that token.
  - For identifiers, this attribute a pointer to the symbol table, and the symbol table holds the actual attributes for that token.
- Some attributes:
  - <id,attr> where attr is pointer to the symbol table
  - <assgop,_> no attribute is needed (if there is only one assignment operator)
  - <num,val> where val is the actual value of the number.
- Token type and its attribute uniquely identifies a lexeme.
- Regular expressions are widely used to specify patterns.

Terminology of Languages

- Alphabet : a finite set of symbols (ASCII characters)
- String :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - |s| is the length of string s.
- Language: sets of strings over some fixed alphabet
  - ∅ the empty set is a language.
  - {ε} the set containing empty string is a language
  - The set of well-formed C programs is a language
  - The set of all possible identifiers is a language.
- Operators on Strings:
  - Concatenation: xy represents the concatenation of strings x and y. $s\,\varepsilon = s$ $\varepsilon\, s = s$
  - $s^n = s\, s\, s \ldots s$ (n times) $s^0 = \varepsilon$

Operations on Languages

- Concatenation:
  - $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
- Union
  - $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$
- Exponentiation:
  - $L^0 = \{\varepsilon\}$ $L^1 = L$ $L^2 = LL$
- Kleene Closure
  - $L^* = \bigcup_{i=0}^{\infty} L^i$
- Positive Closure
  - $L^+ = \bigcup_{i=1}^{\infty} L^i$

Example

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- L_1^3 = all strings with length three (using a,b,c,d}
- L_1^* = all strings using letters a,b,c,d and empty string
- L_1^+ = doesn't include the empty string

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\ (r_2)$	$L(r_1)\ L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules.
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$

- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon, 0, 00, 000, 0000, \ldots.\}$
  - $(0|1)^* \Rightarrow$ all strings with 0 and 1, including the empty string

Regular Definitions

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

basic symbols previously defined names

Regular Definitions (cont.)

- Ex: Identifiers in Pascal

  letter → A | B | ... | Z | a | b | ... | z
  digit → 0 | 1 | ... | 9
  id → letter (letter | digit) *

  - If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

    (A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9)) *

- Ex: Unsigned numbers in Pascal

  digit → 0 | 1 | ... | 9
  digits → digit +
  opt-fraction → (. digits) ?
  opt-exponent → (E (+|-)? digits) ?
  unsigned-num → digits opt-fraction opt-exponent

C-language identifiers

letter_ → [A-Za-z_]
digit → [0-9]
CId → letter_ (letter_ | digit)*

Unsigned integer or floating point numbers

digit → [0-9]
digits → digit+
number → digits (. digits)?(E[+-]? digits)?

Finite Automata

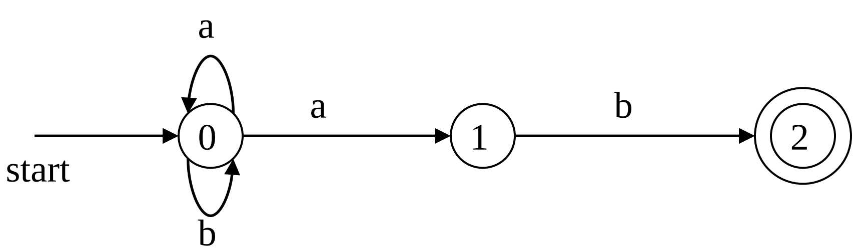
- A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
  - Algorithm1: Regular Expression ➔ NFA ➔ DFA (two steps: first to NFA, then to DFA)
  - Algorithm2: Regular Expression ➔ DFA (directly convert a regular expression into a DFA)

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - S - a set of states
  - Σ - a set of input symbols (alphabet)
  - move – a transition function move to map state-symbol pairs to sets of states.
  - s_o - a start (initial) state
  - F – a set of accepting states (final states)

- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

NFA (Example)



Transition graph of the NFA

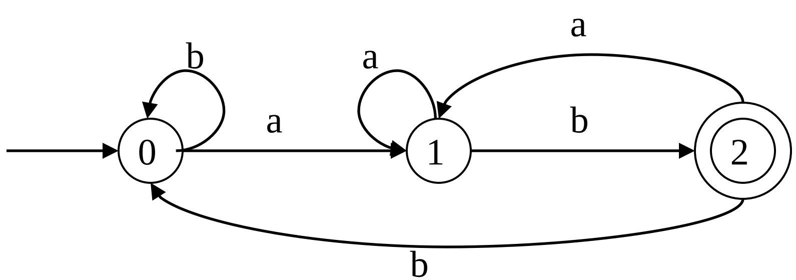
0 is the start state s_0
{2} is the set of final states F
$\Sigma = \{a,b\}$
$S = \{0,1,2\}$
Transition Function:

	a	b
0	{0,1}	{0}
1	_	{2}
2	_	_

The language recognized by this NFA is $(a|b)^*\ a\ b$

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
  - no state has ε- transition
  - for each symbol a and state s, there is at most one labeled edge a leaving s.
    i.e. transition function is from pair of state-symbol to state (not set of states)



The language recognized by

this DFA is also $(a|b)^*$ a b

Implementing a DFA

- Le us assume that the end of a string is marked with a special symbol (say eos). The algorithm for recognition will be as follows: (an efficient implementation)

```
s ← s0                  { start from the initial state }
c ← nextchar            { get the next character from the input string }
while (c != eos) do     { do until the en dof the string }
    begin
        s ← move(s,c)   { transition function }
        c ← nextchar
    end
if (s in F) then                { if s is an accepting state }
    return "yes"
else
    return "no"
```

Implementing a NFA


```
S ← ε-closure({s0})                 { set all of states can be accessible from s0 by ε-
    transitions }
c ← nextchar
while (c != eos) {
    begin
        s ← ε-closure(move(S,c))    { set of all states can be accessible from a state in S
        c ← nextchar                  by a transition on c }
    end
if (S∩F != Φ) then                  { if S contains an accepting state }
    return "yes"
else
    return "no"
```

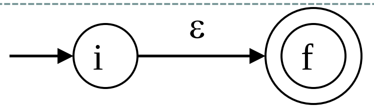
- This algorithm is not efficient.

Converting A Regular Expression into A NFA (Thomson's Construction)

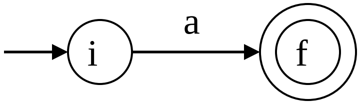
- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
  It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA,

Thomson's Construction (cont.)

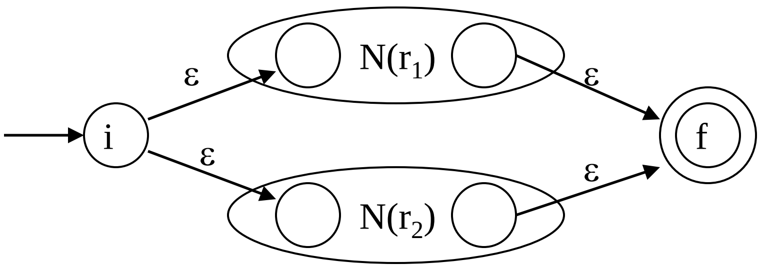
- To recognize an empty string ε



- To recognize a symbol a in the alphabet Σ



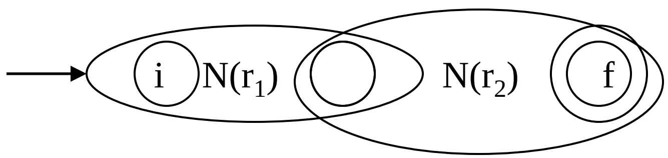
- If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2
  - For regular expression $r_1 \mid r_2$



NFA for $r_1 \mid r_2$

Thomson's Construction (cont.)

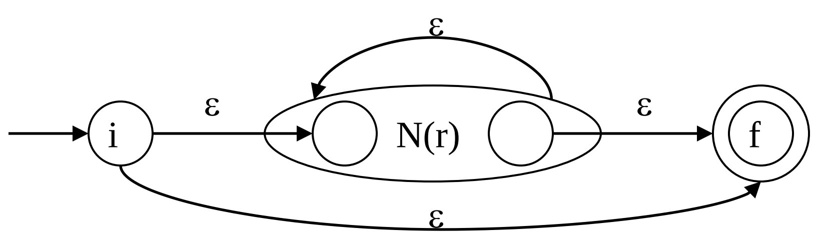
- For regular expression $r_1 \, r_2$



NFA for $r_1 \, r_2$

Final state of $N(r_2)$ become final state of $N(r_1 r_2)$

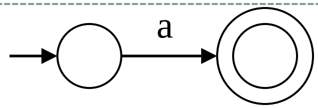
- For regular expression r^*



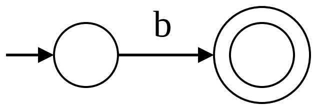
NFA for r^*

Thomson's Construction (Example - $(a|b)^*a$)

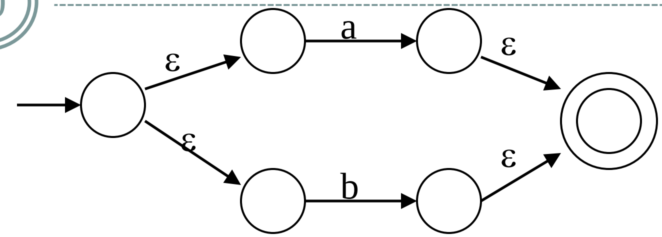
a:



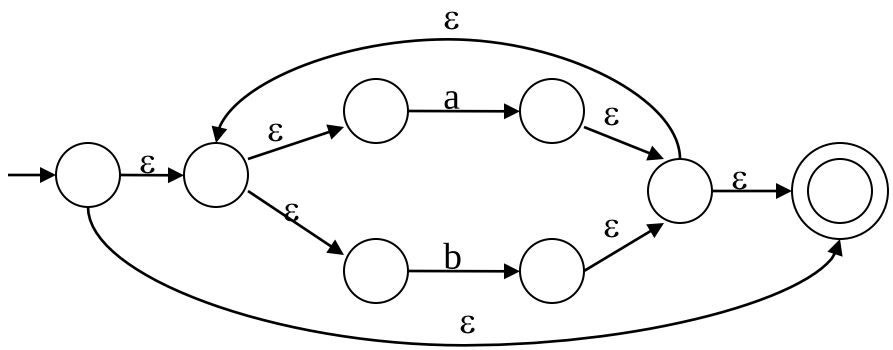
b:



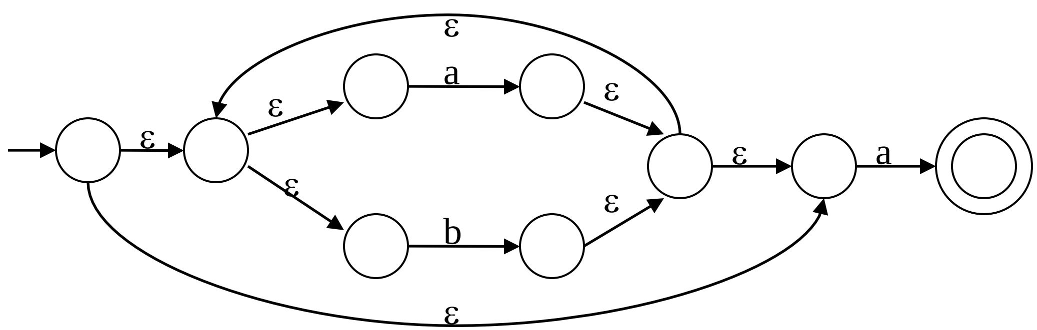
(a | b)



$(a|b)^*$



$(a|b)^*a$



Converting a NFA into a DFA (subset construction)

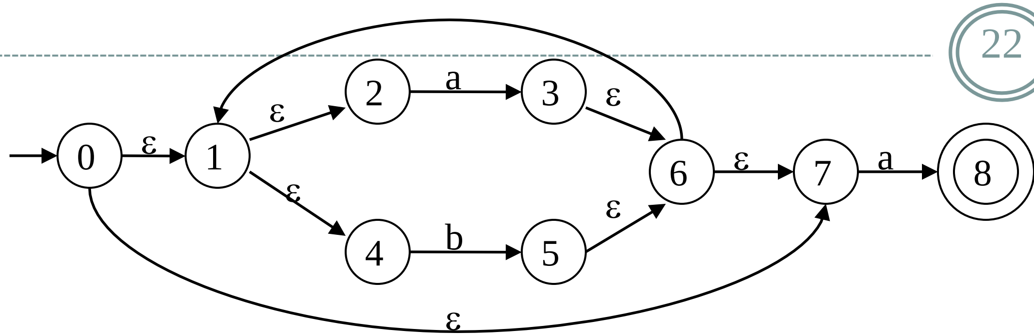
put ε-closure($\{s_0\}$) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S_1 in DS) do
    begin
        mark S_1
        for each input symbol a do
            begin
                $S_2 \leftarrow \varepsilon$-closure(move($S_1$,a))
                if (S_2 is not in DS) then
                    add S_2 into DS as an unmarked state
                transfunc[S_1,a] $\leftarrow S_2$
            end
    end

ε-closure($\{s_0\}$) is the set of all states can be accessible from s_0 by ε-transition.

set of states to which there is a transition on a from a state s in S_1

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure($\{s_0\}$)

Converting a NFA into a DFA (Example)



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into DS as an unmarked state

⇓ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into DS

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into DS

transfunc[S_0,a] 🡐 S_1 transfunc[S_0,b] 🡐 S_2

⇓ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_1,a] 🡐 S_1 transfunc[S_1,b] 🡐 S_2

⇓ mark S_2

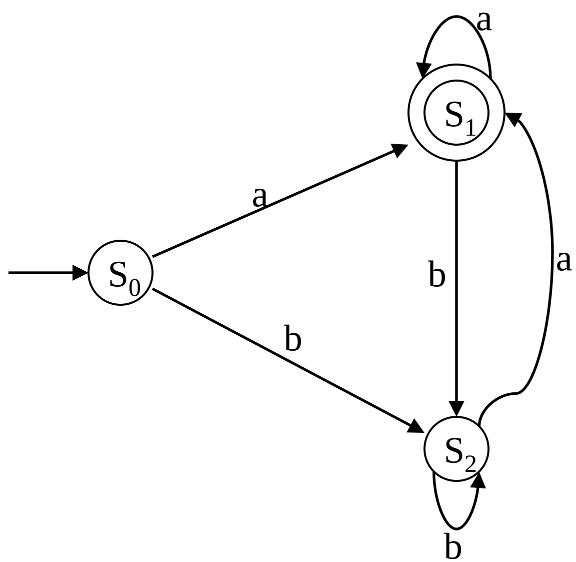
ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_2,a] 🡐 S_1 transfunc[S_2,b] 🡐 S_2

Converting a NFA into a DFA (Example – cont.)

S_0 is the start state of DFA since 0 is a member of $S_0=\{0,1,2,4,7\}$
S_1 is an accepting state of DFA since 8 is a member of $S_1 = \{1,2,3,4,6,7,8\}$



Stop here

Converting Regular Expressions Directly to DFAs

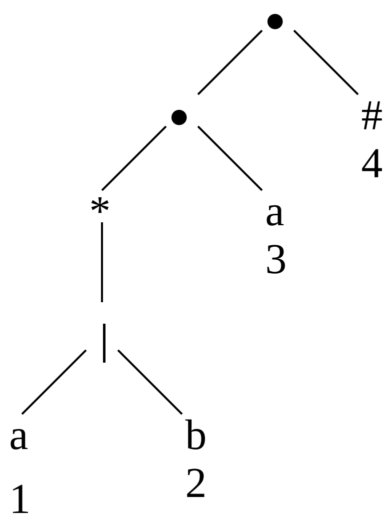
- We may convert a regular expression into a DFA (without creating a NFA first).
- First we augment the given regular expression by concatenating it with a special symbol #.

  r ➔ (r)# augmented regular expression

- Then, we create a syntax tree for this augmented regular expression.
- In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.
- Then each alphabet symbol (plus #) will be numbered (position numbers).

Regular Expression ➔ DFA (cont.)

$(a|b)^* a \rightarrow (a|b)^* a \#$ augmented regular expression



Syntax tree of $(a|b)^* a \#$

- each symbol is numbered (positions)
- each symbol is at a leave

- inner nodes are operators

followpos

Then we define the function **followpos** for the positions (positions assigned to leaves).

followpos(i) -- is the set of positions which can follow the position i in the strings generated by the augmented regular expression.

For example, $(a \mid b)^* \; a \;\; \#$

$\quad\quad 1 \;\; 2 \quad\;\; 3 \;\; 4$

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

followpos is just defined for leaves, it is not defined for inner nodes.

firstpos, lastpos, nullable

- To evaluate followpos, we need three more functions to be defined for the nodes (not just for leaves) of the syntax tree.

- **firstpos(n)** -- the set of the positions of the **first** symbols of strings generated by the sub-expression rooted by n.

- **lastpos(n)** -- the set of the positions of the **last** symbols of strings generated by the sub-expression rooted by n.

- **nullable(n)** -- *true* if the empty string is a member of strings generated by the sub-expression rooted by n *false* otherwise

How to evaluate firstpos, lastpos, nullable

n	nullable(n)	firstpos(n)	lastpos(n)
leaf labeled ε	true	Φ	Φ
leaf labeled with position i	false	{i}	{i}
$\mid$ c_1 c_2	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
● c_1 c_2	nullable(c_1) and nullable(c_2)	if (nullable(c_1)) firstpos(c_1) ∪ firstpos(c_2) else firstpos(c_1)	if (nullable(c_2)) lastpos(c_1) ∪ lastpos(c_2) else lastpos(c_2)
$*$ c_1	true	firstpos(c_1)	lastpos(c_1)

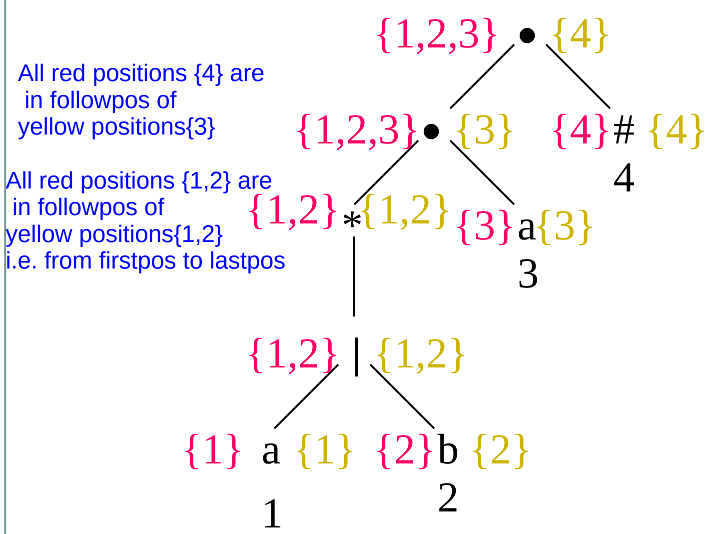
How to evaluate followpos

- Two-rules define the function followpos:

1. If **n** is concatenation-node with left child c_1 and right child c_2, and **i** is a position in **lastpos(c_1)**, then all positions in **firstpos(c_2)** are in **followpos(i)**.
2. If **n** is a star-node, and **i** is a position in **lastpos(n),** then all positions in **firstpos(n)** are in **followpos(i)**.

- If firstpos and lastpos have been computed for each node, followpos of each position can be computed by making one depth-first traversal of the syntax tree.

Example -- $(a \mid b)^* a \ \#$



red– firstpos
yellow– lastpos

Then we can calculate followpos

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

- After we calculate follow positions, we are ready to create DFA for the regular expression.

Algorithm (RE → DFA)

- Create the syntax tree of (r) #
- Calculate the functions: followpos, firstpos, lastpos, nullable
- Put firstpos(root) into the states of DFA as an unmarked state.
- *while* (there is an unmarked state S in the states of DFA) *do*
  - mark **S**
  - *for each* input symbol **a** *do*
    - let $s_1,\ldots,s_n$ are positions in **S** and symbols in those positions are **a**
    - **S'** ← followpos(s_1) ∪ ... ∪ followpos(s_n)
    - **move(S,a) ← S'**
    - if (**S'** is not empty and not in the states of DFA)
      - put **S'** into the states of DFA as an unmarked state.

- *the start state of DFA is firstpos(root)*
- *the accepting states of DFA are all states containing the position of #*

Example -- $(a_1 | b_2)^* a_3 \#_4$

followpos(1)={1,2,3}   followpos(2)={1,2,3}   followpos(3)={4}   followpos(4)={}

S_1=firstpos(root)={1,2,3}

$\Downarrow$ mark S_1

a: followpos(1) $\cup$ followpos(3)={1,2,3,4}=S_2 move(S_1,a)=S_2

b: followpos(2)={1,2,3}=S_1 move(S_1,b)=S_1

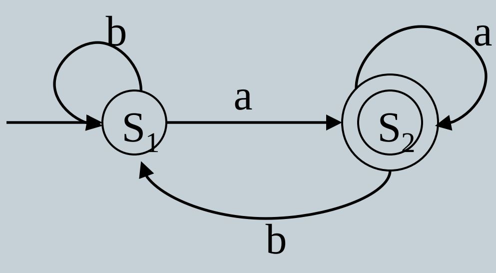
$\Downarrow$ mark S_2

a: followpos(1) $\cup$ followpos(3)={1,2,3,4}=S_2 move(S_2,a)=S_2

b: followpos(2)={1,2,3}=S_1 move(S_2,b)=S_1

start state: S_1

accepting states: {S_2}



Example -- $(a_1 | \varepsilon)\, b_2\, c_3^*\, \#_4$

followpos(1)={2}   followpos(2)={3,4}   followpos(3)={3,4}   followpos(4)={}

S_1=firstpos(root)={1,2}

⇓ mark S_1

a: followpos(1)={2}=S_2 move(S_1,a)=S_2

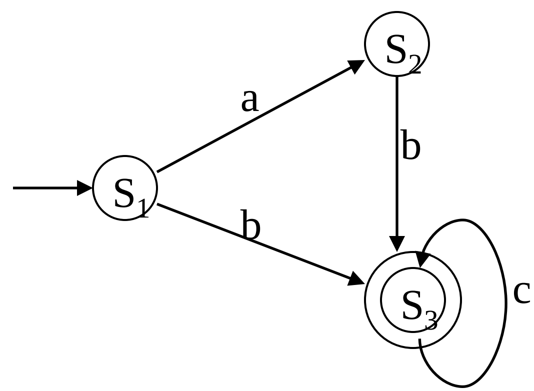
b: followpos(2)={3,4}=S_3 move(S_1,b)=S_3

⇓ mark S_2

b: followpos(2)={3,4}=S_3 move(S_2,b)=S_3

⇓ mark S_3

c: followpos(3)={3,4}=S_3 move(S_3,c)=S_3



start state: S_1

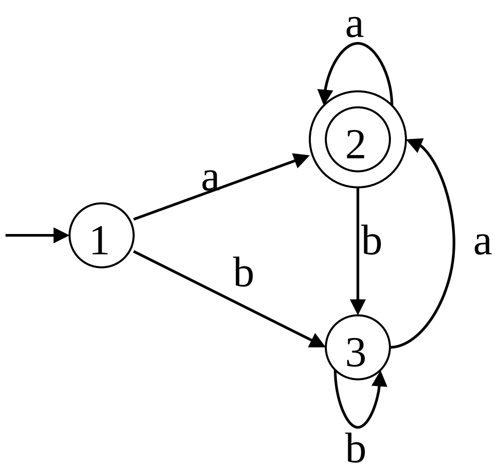
accepting states: {S_3}

Minimizing Number of States of a DFA

- partition the set of states into two groups:
  - G_1 : set of accepting states
  - G_2 : set of non-accepting states
- For each new group G
  - partition G into subgroups such that states s_1 and s_2 are in the same group iff

    for all input symbols a, states s_1 and s_2 have transitions to states in the same group.
- Start state of the minimized DFA is the group containing the start state of the original DFA.
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

Minimizing DFA - Example

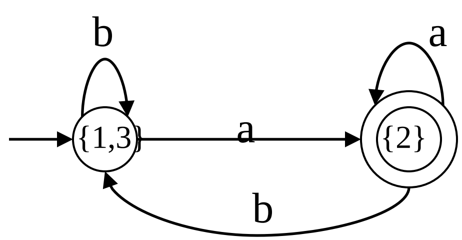


$G_1 = \{2\}$
$G_2 = \{1,3\}$

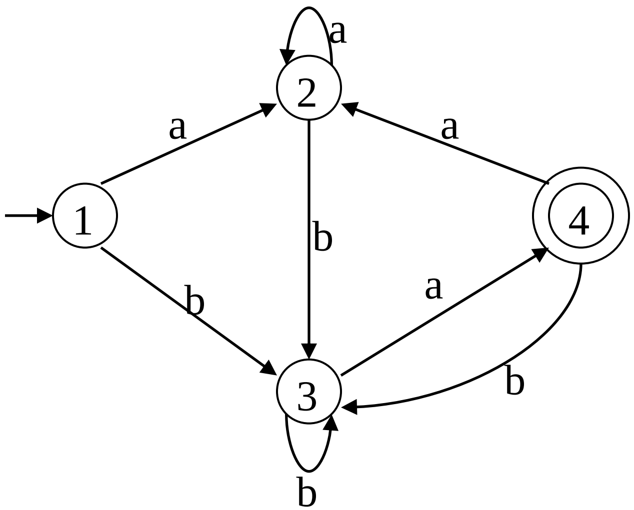
G_2 cannot be partitioned because

move(1,a)=2 move(1,b)=3
move(3,a)=2 move(3,b)=3

So, the minimized DFA (with minimum states)



Minimizing DFA – Another Example



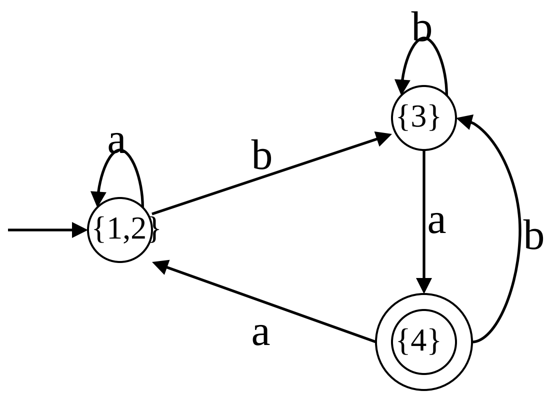
Groups: {1,2,3} {4}

{1,2} {3}

no more partitioning

a	b
1->2	1->3
2->2	2->3
3->4	3->3

So, the minimized DFA



Some Other Issues in Lexical Analyzer

- The lexical analyzer has to recognize the longest possible string.
  - Ex: identifier `newval` -- `n`   `ne`   `new`   `newv`   `newva`   `newval`

- What is the end of a token? Is there any character which marks the end of a token?
  - It is normally not defined.
  - If the number of characters in a token is fixed, in that case no problem: + -
  - But < → < or <> (in Pascal)
  - The end of an identifier : the characters cannot be in an identifier can mark the end of token.
  - We may need a lookhead
    - In Prolog: p :- X is 1. p :- X is 1.5.
      The dot followed by a white space character can mark the end of a number.
      But if that is not the case, the dot must be treated as a part of the number.

Some Other Issues in Lexical Analyzer (cont.)

- Skipping comments
  - Normally we don't return a comment as a token.
  - We skip a comment, and return the next token (which is not a comment) to the parser.
  - So, the comments are only processed by the lexical analyzer, and the don't complicate the syntax of the language.

- Symbol table interface
  - symbol table holds information about tokens (at least lexeme of identifiers)
  - how to implement the symbol table, and what kind of operations.
    - hash table – open addressing, chaining
    - putting into the hash table, finding the position of a token from its lexeme.

- Positions of the tokens in the file (for the error handling).