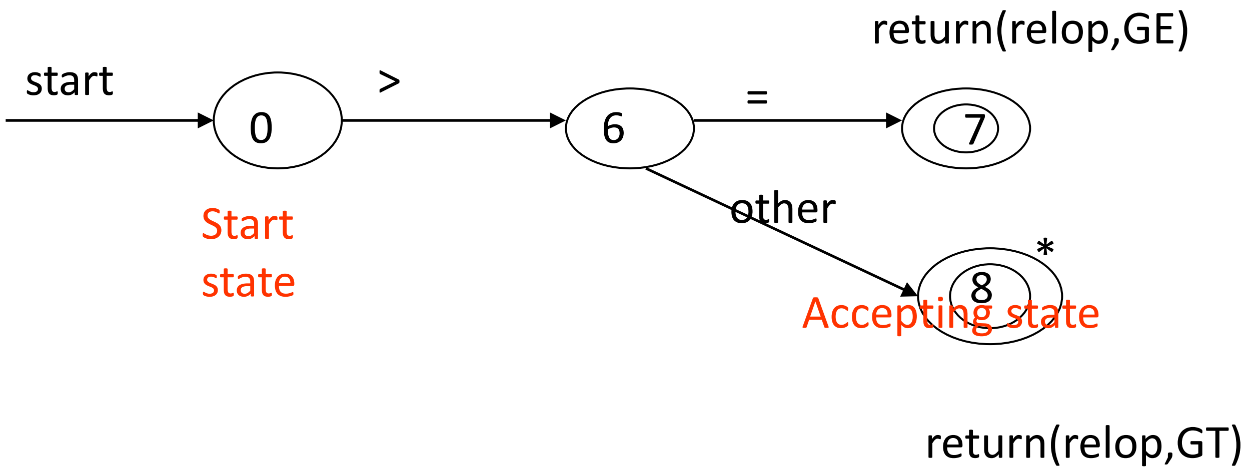
Lexical-Analyser: Automata

Automata – Transition Diagrams

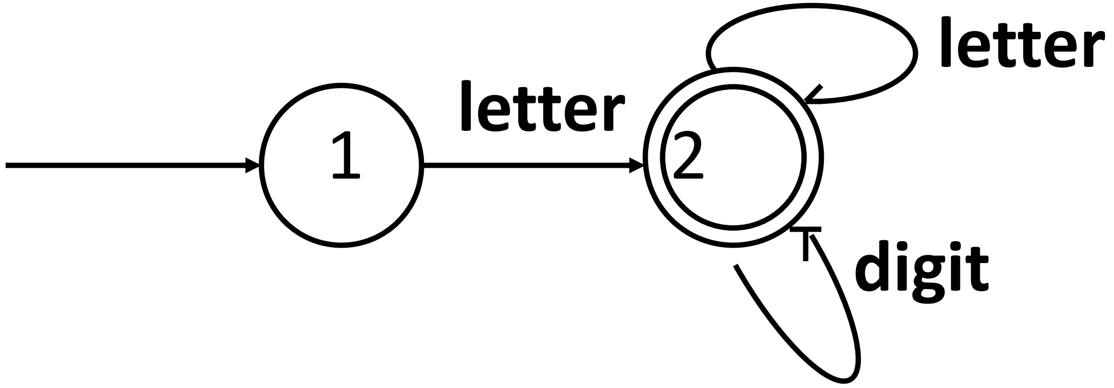
- Transition Diagram(Stylized flowchart)
  - Depict the actions that take place when a lexical analyzer is called by the parser to get the next token

Example NFA



a >= b
a > b

Example for Identifier



- Which represent the rule: **identifier**=**letter**(**letter**|**digit**)*

Finite Automata

- By default a Deterministic one.
- Five tuple representation

  (Q, $\sum$, δ, q_0, F), q_0 belongs to Q and F is a subset of Q

δ is a mapping from $Q \times \sum$ to Q

- Every string has exactly one path and hence faster string matching

DFA

- In a DFA, no state has an ε-transition
- In a DFA, for each state *s* and input symbol *a*, there is at most one edge labeled *a* leaving *s*
- To describe a FA, we use the transition graph or transition table

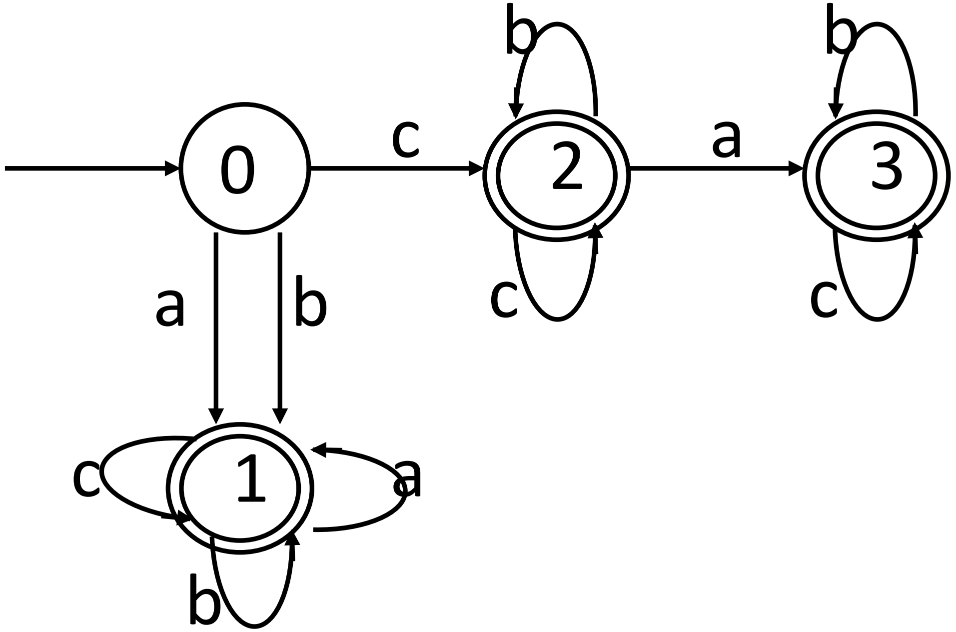
DFA

- A DFA accepts an input string x if and only if there is some path in the transition graph from start state to some accepting state

Example

- Recognition of Tokens
- Construct a DFA M， which can accept the strings which begin with *a* or *b*, or begin with *c* and contain at most one *a*。

Example



c b b c c

c c c b a

c c c a a b x

Non-deterministic Finite automata

- Same as deterministic, gives some flexibility.
- Five tuple representation

  $(Q, \sum, \delta, q0, F)$, q0 belongs to Q and F is a subset of Q

  δ is a mapping from $Q \times \sum$ to 2^Q
- More time for string matching as multiple paths exist.

Non-Deterministic Finite automata with ϵ

- Same as NFA. Still more flexible in allowing to change state without consuming any input symbol.
- δ is a mapping from $Q \times \sum \cup \{\epsilon\}$ to 2^Q
- Slower than NFA for string matching

NFA Some Observations

- In a NFA, the same character can label two or more transitions out of one state;
- In a NFA, ε is a legal input symbol.
- A DFA is a special case of a NFA

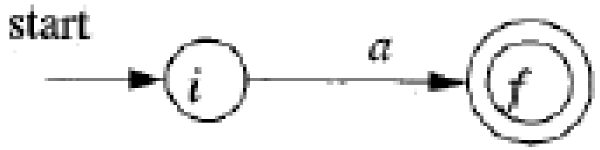
NFA Some Observations

- A NFA accepts an input string ‘x’ if and only if there is some path in the transition graph from start state to some accepting state. A path can be represented by a sequence of state transitions called moves.
- The language defined by a NFA is the set of input strings it accepts

RE to DFA

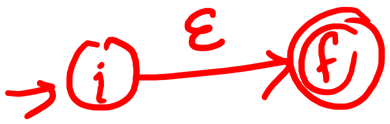
- Regular Expression could be converted to E-NFA using Thompson Construction Algorithm
- E-NFA could be converted to DFA using Subset construction algorithm

Basic Regular Expression and its NFA

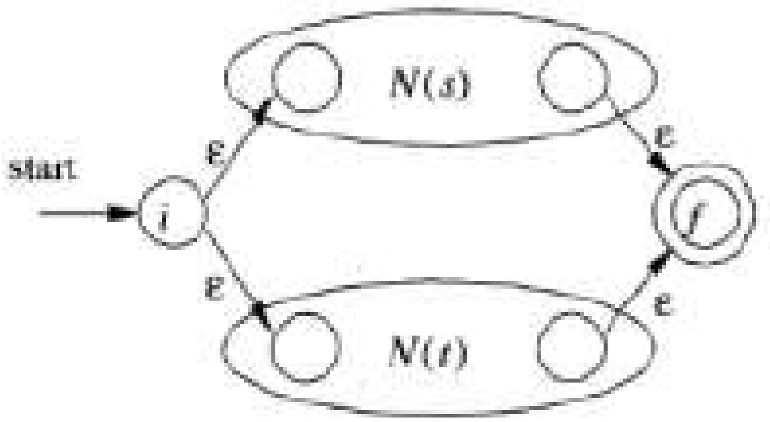


$r = a$

ε



Regular expression – Union operator and its corresponding NFA

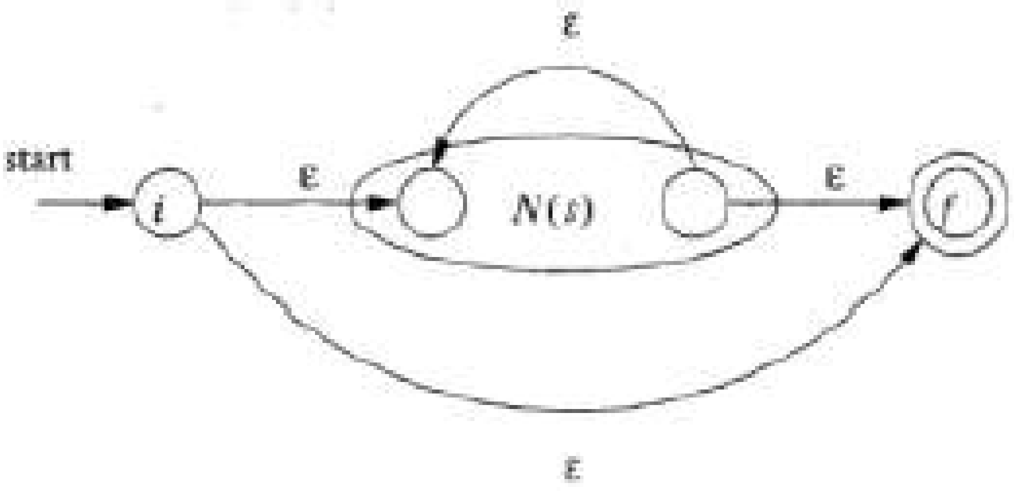


r = s | t

Regular expression with concatenation operator and its corresponding NFA

r = st

Regular expression involving kleene closure operator and its NFA



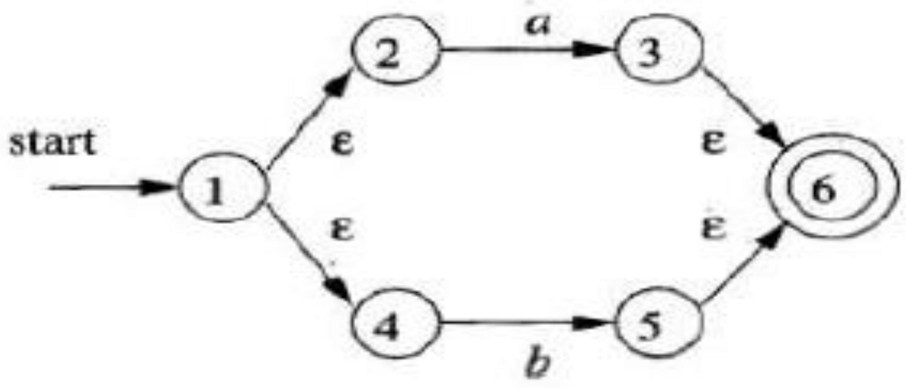
$r = s^*$

Algorithm

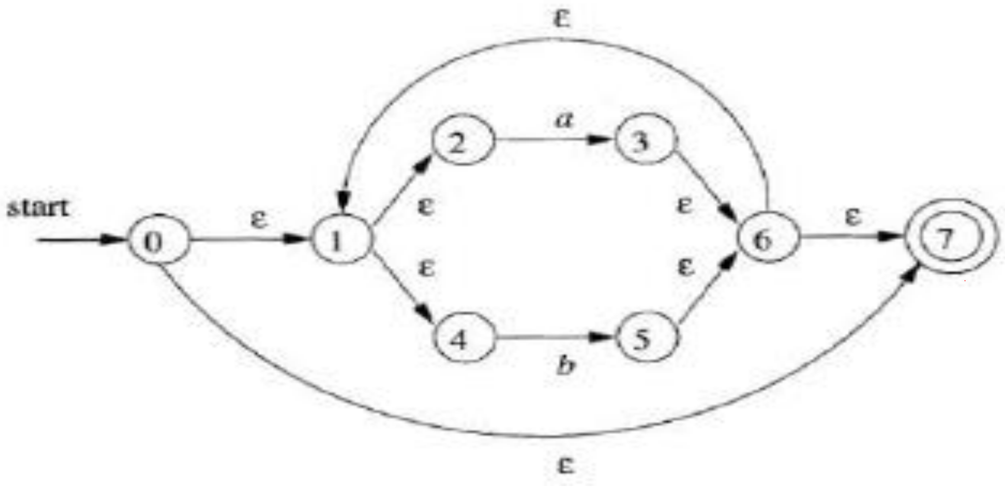
- Construct the basic NFA for each of the input symbols
- Prioritize the operators (), *, **.**, |
- Use the discussed variations and form an NFA.

Example : (a|b)*abb

a|*b*

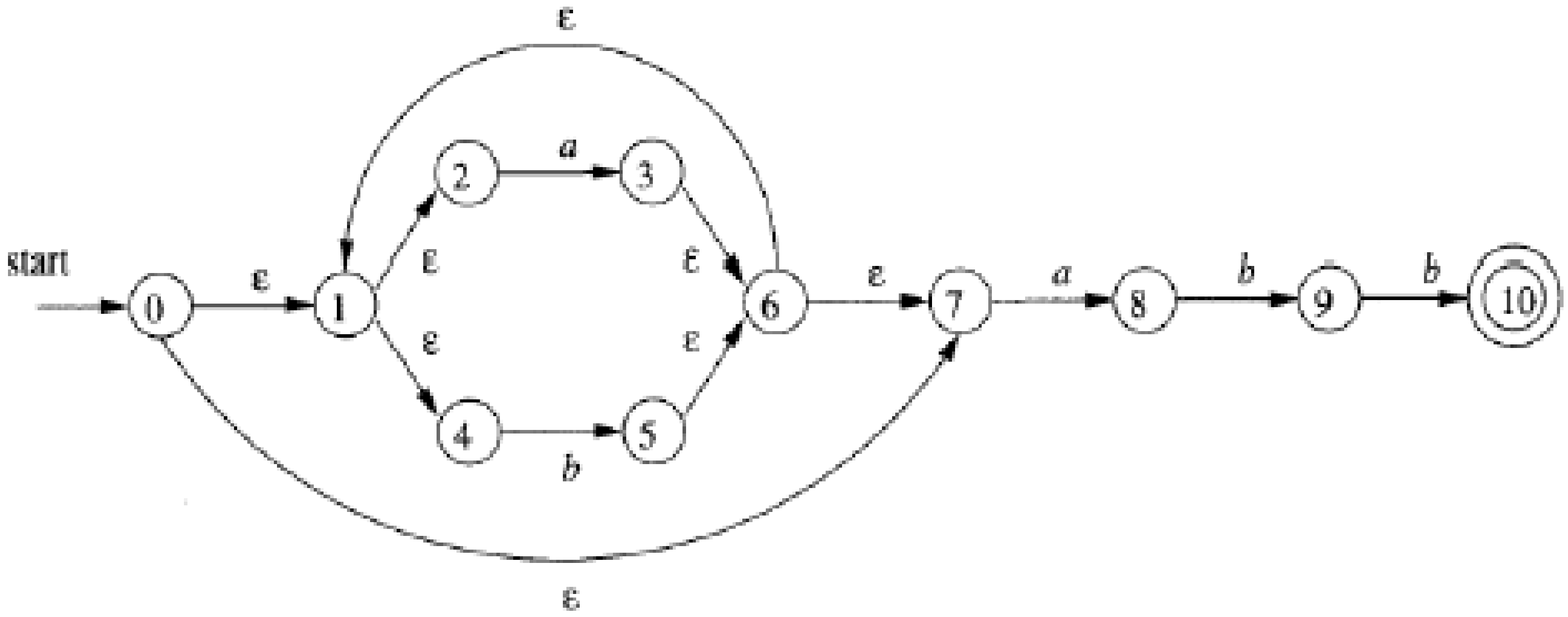


(*a*|*b*)*



Example

$(a|b)^*abb$



Conversion from NFA to DFA

- Reasons to conversion

  Avoiding ambiguity

- The algorithm idea

  Subset construction: The state set of a state in a NFA is thought of as a following STATE of the state in the converted DFA

Subset Construction algorithm

- Input. An NFA $N=(S,\Sigma,move,S_0,Z)$
- Output. A DFA $D=(Q,\Sigma,\delta,I_0,F)$, accepting the same language
- Requires Pre-processing - Determination of E-Closure

Pre-process-- ε-closure(T)

- Obtain ε-closure(T) $T \subseteq S$
- ε-closure(T) definition
  - A set of NFA states reachable from NFA state *s* in *T* on *ε-transitions* alone

Conversion from NFA to DFA – The pre-process--- ε-closure(T)

- ε-closure(T) algorithm

```
push all states in T onto stack;
initialize ε-closure(T)  to T;
 while stack is not empty do {
    pop the top element of the stack into t;
    for each state u with an edge from t to u labeled ε do {
      if u is not in ε-closure(T) {
         add u to ε-closure(T)
         push u into stack}}}
```

Subset Construction Algorithm

- $I_0 = \varepsilon$-closure(S_0), $I_0 \in Q$
- For each I_i , $I_i \in Q$,

  let $I_t = \varepsilon$-closure(move(I_i,a))

  if $I_t \notin Q$, then put I_t into Q
- Repeat above step until there are no new states to put into Q
- Let F={I | I $\in$ Q, such that I $\cap$ Z <>Φ}

Example

A or $q_0 \Rightarrow \varepsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\}$

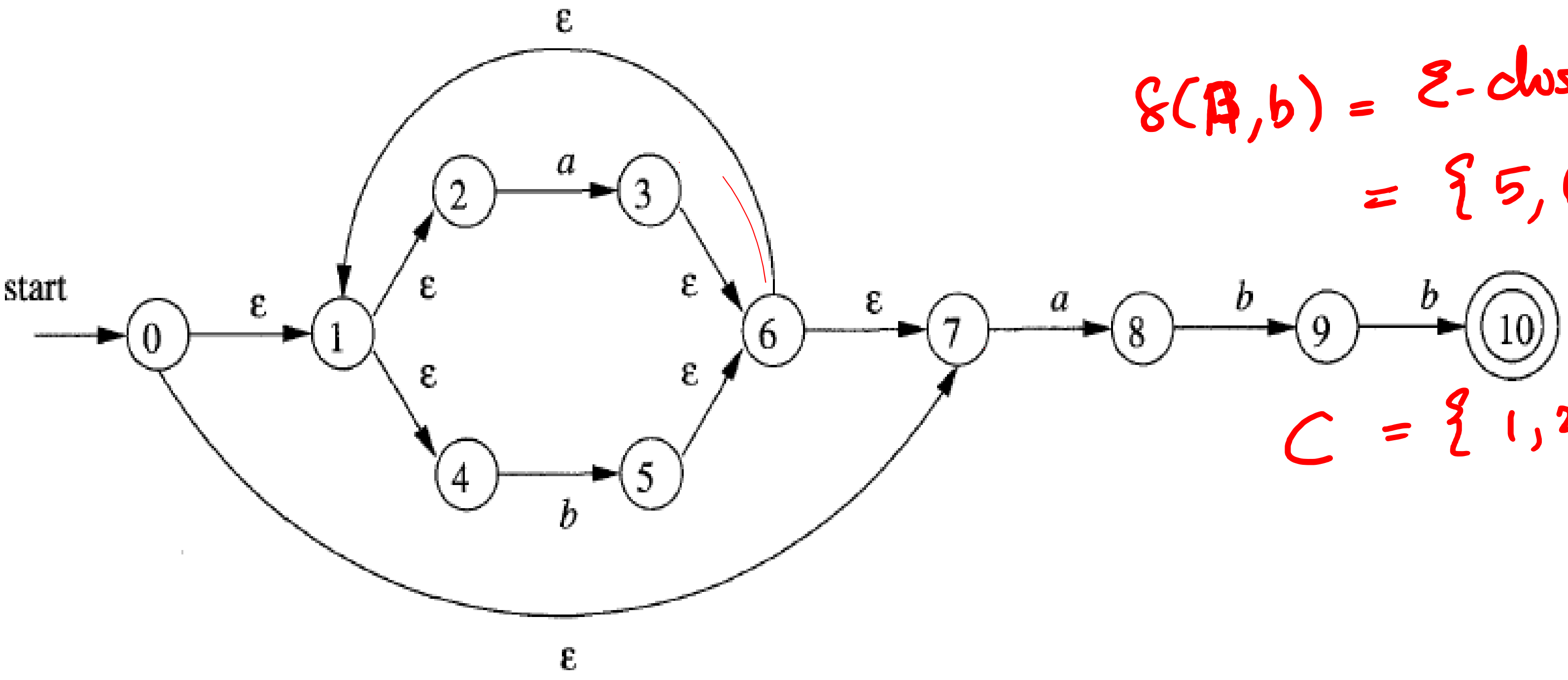
$\varepsilon\text{-closure}\{3, 8)$

$= \{3, 6, 7, 1, 2, 4, 8\}$

$B = \{1, 2, 3, 4, 6, 7, 8\}$

$\delta(B, b) = \varepsilon\text{-closure}(5)$

$= \{5, 6, 7, 1, 2, 4\}$



$C = \{1, 2, 4, 5, 6, 7\}$

Result

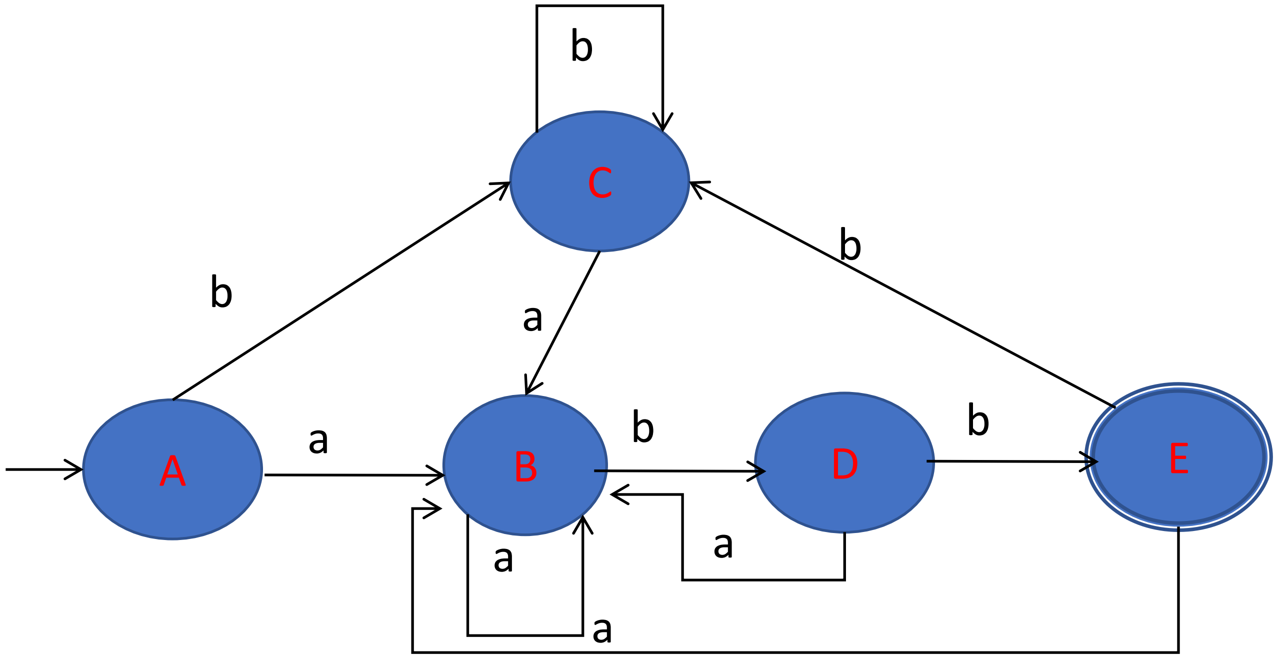
I	a	b
A={0,1,2,4,7}	B={1,2, 3, 4, 6, 7, 8}	C = {1,2,4,5,6,7}
B={1,2, 3, 4, 6, 7, 8}	B={1,2, 3, 4, 6, 7, 8}	D = {1,2,4,5,6,7,9}
C = {1,2,4,5,6,7}	B={1,2, 3, 4, 6, 7, 8}	C = {1,2,4,5,6,7}
D = {1,2,4,5,6,7,9}	B={1,2, 3, 4, 6, 7, 8}	E = {1,2,3,5,6,7,10}
E = {1,2,3,5,6,7,10}	B={1,2, 3, 4, 6, 7, 8}	C = {1,2,4,5,6,7}

$\delta(A, a)$

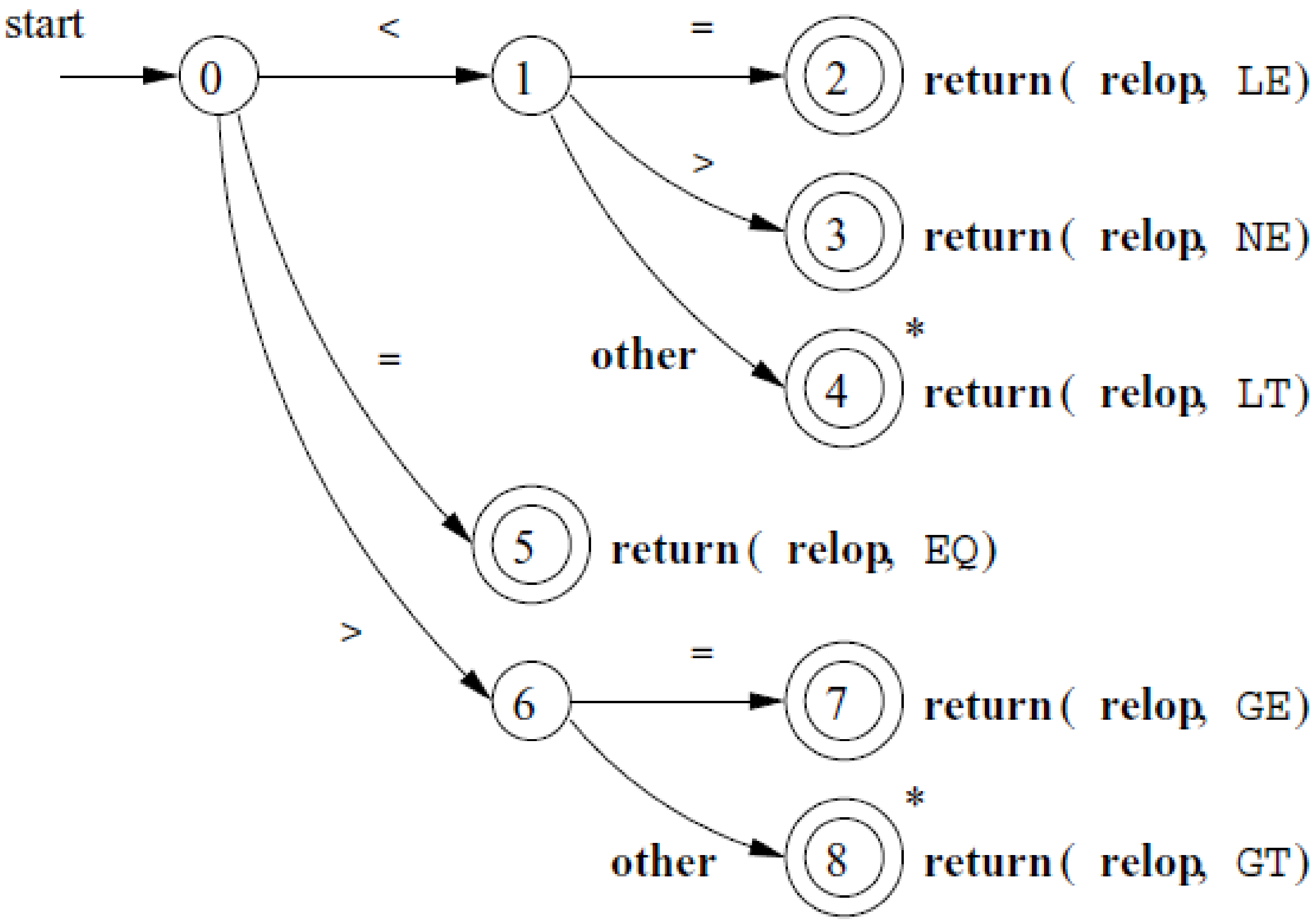
$= \varepsilon\text{-closure}(\text{move}(A, a))$

$= B$

$\delta(A, b)$

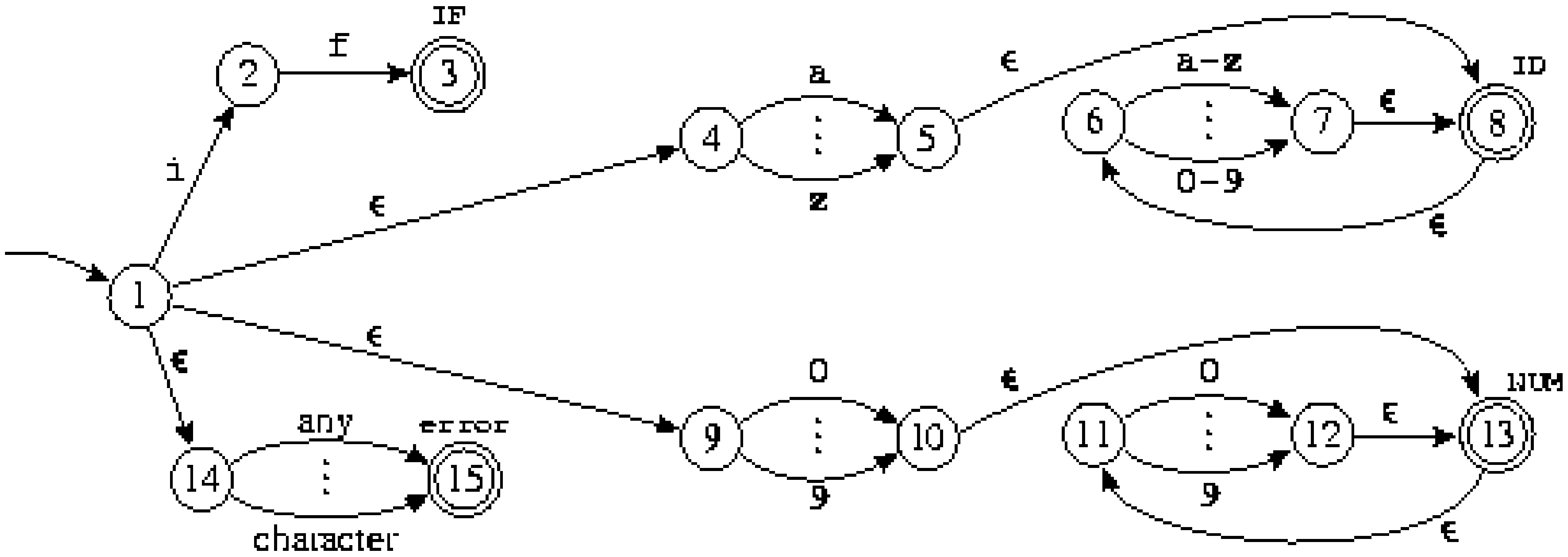

b
C
b
a
b
a
b
b
A
B
D
E
a
a
a

Example



start
<
=
return(relop, LE)
>
return(relop, NE)
=
other
*
return(relop, LT)
return(relop, EQ)
>
=
return(relop, GE)
*
other
return(relop, GT)
0
1
2
3
4
5
6
7
8

Example



IF
f
i
ε
a
z
ε
a-z
0-9
ε
ID
ε
ε
ε
0
9
ε
0
9
ε
NUM
ε
any
character
error
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Subset Construction Algorithm

- RE to E-NFA and then to DFA is time consuming and results in redundant states in the DFA
- Need to minimize the DFA for faster string matching

Summary till now

- DFA,NFA and NFA with ϵ as ways of defining patterns.
- DFA is faster, but construction is difficult
- NFA construction is easier but slower during string matching
- Conversion of RE to E-NFA
- Convert NFA to DFA

NFA and DFA

- Constructing NFA is easier. But string matching with DFA is faster.
- RE to DFA – done by converting to E-NFA and then to DFA
- This results in an increased number of states in the DFA – need for minimization

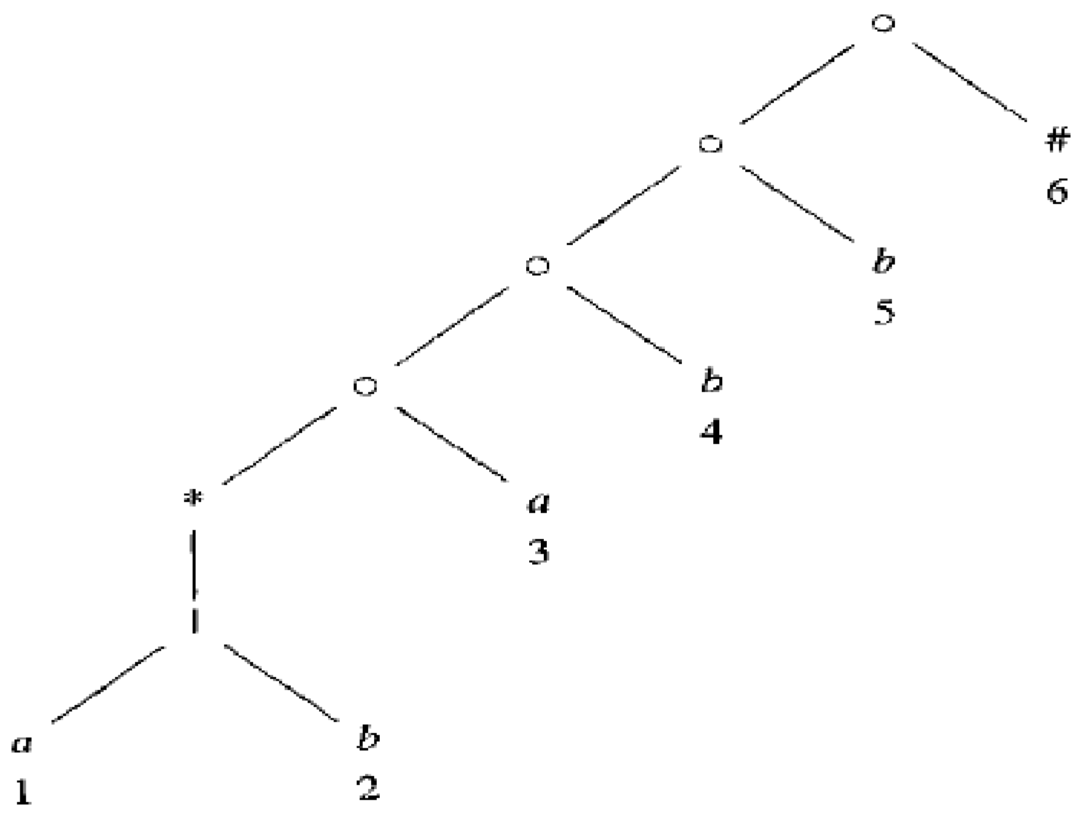
Minimized DFA

- Construct the DFA directly from RE by using a new algorithm
- Table filling minimization algorithm
  - Construct DFA and then use a procedure to eliminate redundant state

From Regular Expression to DFA Directly (Algorithm)

- Augment the regular expression *r* with a special end symbol # to make accepting states important: the new expression is *r* #
- Construct a syntax tree for *r*#
- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

Example Syntax tree for (a|b)* abb



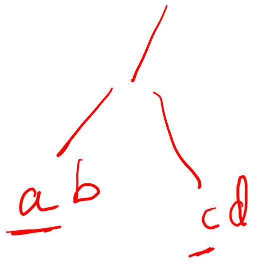
From Regular Expression to DFA Directly: Annotating the Tree

- *nullable*(*n*): the subtree at node *n* generates languages including the empty string
- *firstpos*(*n*): set of positions that can match the first symbol of a string generated by the subtree at node *n*

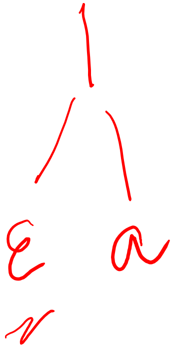
Algorithm

- *lastpos*(*n*): the set of positions that can match the last symbol of a string generated be the subtree at node *n*
- *followpos*(*i*): the set of positions that can follow position *i* in the tree

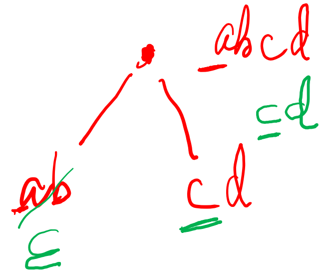
Annotating tree



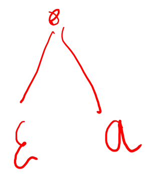
Node n	*nullable*(n)	*firstpos*(n)	*lastpos*(n)
Leaf ε	true	$\varnothing$	$\varnothing$
Leaf i	false	$\{i\}$	$\{i\}$
$\vert$ / \ c_1 c_2	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1)$ $\cup$ $firstpos(c_2)$	$lastpos(c_1)$ $\cup$ $lastpos(c_2)$



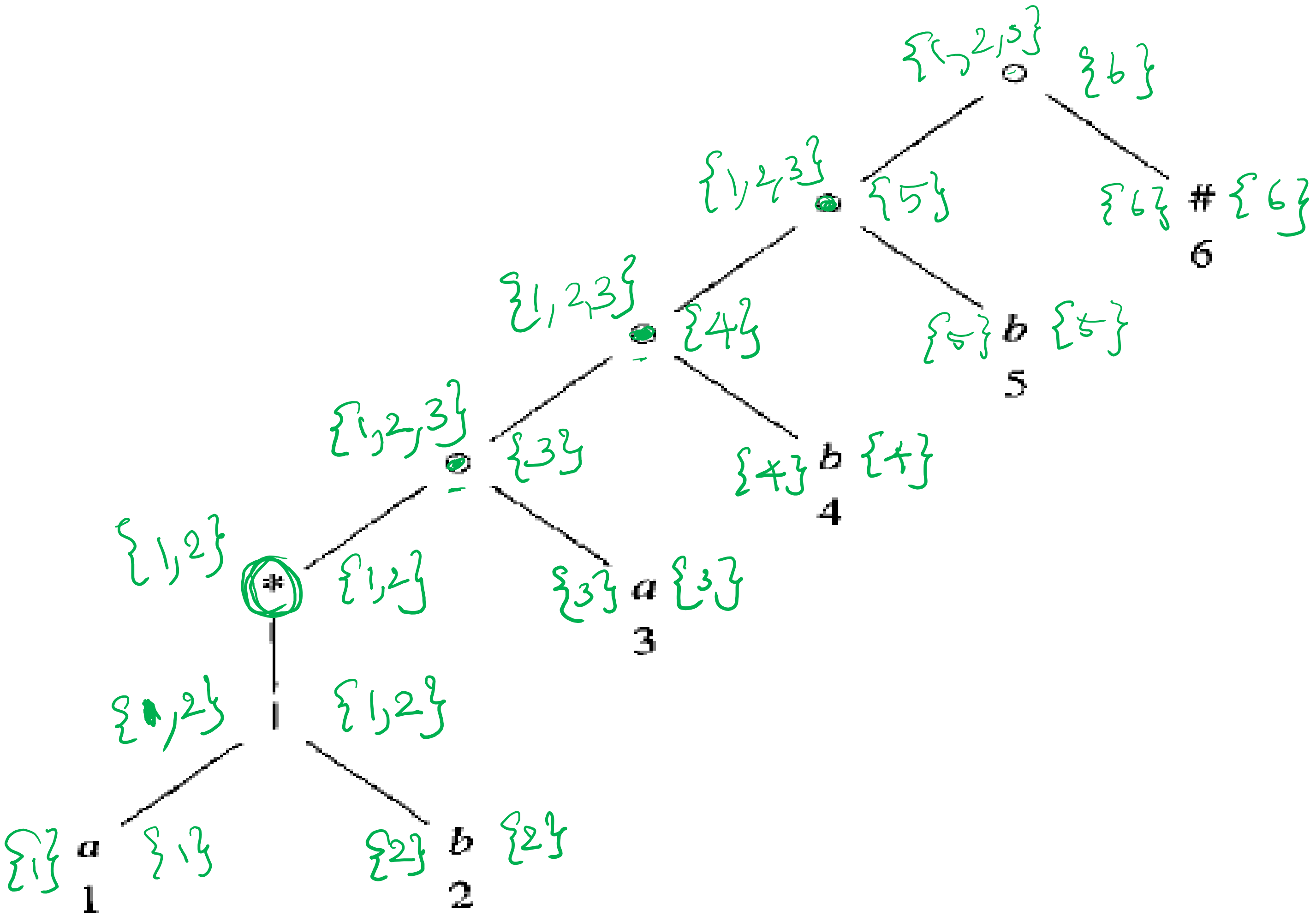
Annotating tree



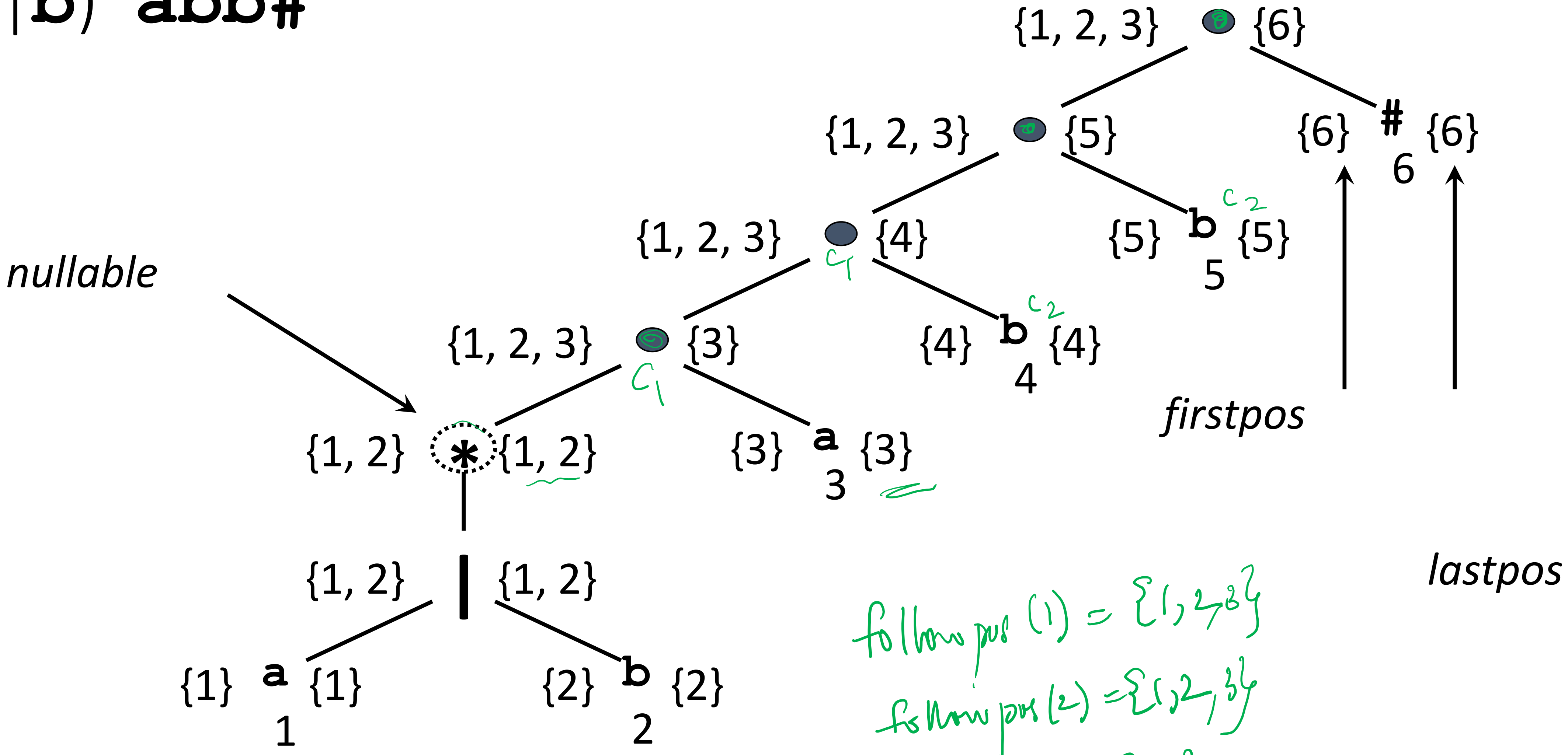
Node n	*nullable*(n)	*firstpos*(n)	*lastpos*(n)
• / \ c_1 c_2	*nullable*(c_1) and *nullable*(c_2)	**if** *nullable*(c_1) **then** *firstpos*(c_1) ∪ *firstpos*(c_2) **else** *firstpos*(c_1)	**if** *nullable*(c_2) **then** *lastpos*(c_1) ∪ *lastpos*(c_2) **else** *lastpos*(c_2)
* | c_1	true	*firstpos*(c_1)	*lastpos*(c_1)



Syntax tree for (a|b)* abb



(a|b)*abb#



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

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

followpos(3) = {4}

followpos

for each node n in the tree **do**

if n is a cat-node with left child c_1 and right child c_2 **then**

for each i in $lastpos(c_1)$ **do**

$followpos(i) := followpos(i) \cup firstpos(c_2)$

end do

else if n is a star-node

for each i in $lastpos(n)$ **do**

$followpos(i) := followpos(i) \cup firstpos(n)$

end do **end if** **end do**

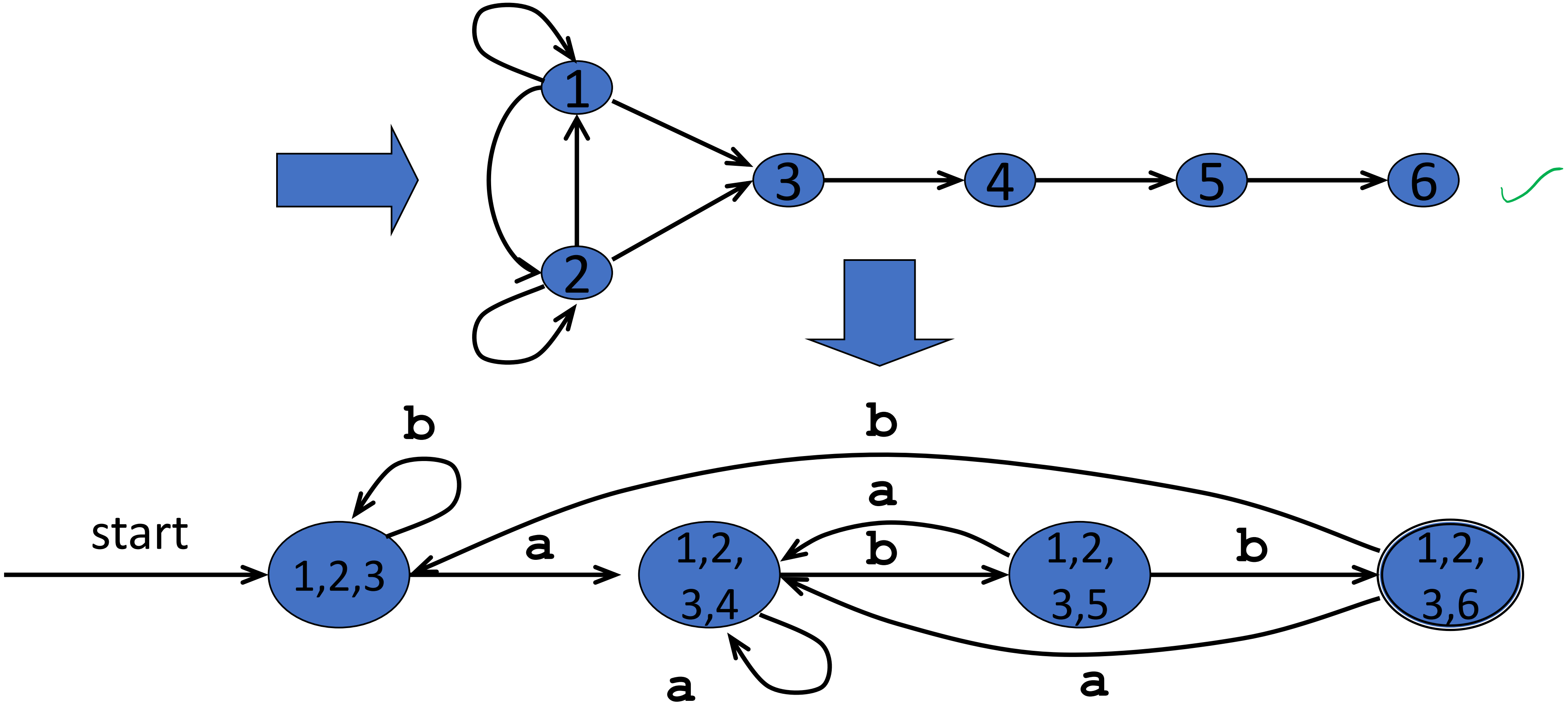
Follow pos

Node	**Followpos(n)**
1	{1, 2, 3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	Φ

Algorithm

s_0 := *firstpos*(*root*) where *root* is the root of the syntax tree
Dstates := {s_0} and is unmarked
while there is an unmarked state *T* in *Dstates* **do**
    mark *T*
    for each input symbol $a \in \Sigma$ **do**
    let *U* be the set of positions that are in *followpos*(*p*)
      for some position *p* in *T*,
      such that the symbol at position *p* is *a*
      if *U* is not empty and not in *Dstates* **then**
      add *U* as an unmarked state to *Dstates*
      end if
      Dtran[*T,a*] := *U*
    end do
end do

From Regular Expression to DFA Directly: Example



Minimized DFA - Table filling minimization algorithm

- Table filling minimization algorithm
  - Construct DFA and then use a procedure to eliminate redundant state
- Construct the DFA directly from RE by using a new algorithm

Basic Idea

- Find all groups of states that can be distinguished by some input string.
- At beginning of the process, we assume two distinguished groups of states:
  - the group of non-accepting states
  - the group of accepting states..
- Then we use the method of partition of equivalent class on input string to partition the existed groups into smaller groups

Minimization Algorithm

- Input: A DFA M={S, Σ, move, s_0, F}
- Output: A DFA M' accepting the same language as M and having as few states as possible.

Minimization Algorithm

1. Construct an initial partition $\prod$ of the set of states with two groups: the accepting states *F* and the non-accepting states *S-F.* $\prod_0 = \{I_0^1, I_0^2\}$
2. For each group *I* of $\prod_i$,partition *I* into subgroups such that two states *s* and *t* of *I* are in the same subgroup if and only if for all input symbols *a*, states *s* and *t* have transitions on *a* to states in the same group of $\prod_i$; replace *I* in $\prod_{i+1}$ by the set of subgroups formed.
3. If $\prod_{i+1} = \prod_i$, let $\prod_{final} = \prod_{i+1}$ and continue with step (4). Otherwise, repeat step (2) with $\prod_{i+1}$

Minimization Algorithm

- Choose one state in each group of the partition Π_{final} as the representative for that group which will be the states of the reduced DFA M'.
- Let *s* and *t* be representative states for *s*'s and *t*'s group respectively, and suppose on input *a* there is a transition of *M* from *s* to *t*. Then *M'* has a transition from *s* to *t* on *a*.

Minimization Algorithm

- If M’ has a dead state(a state that is not accepting and that has transitions to itself on all input symbols),then remove it. Also remove any states not reachable from the start state.

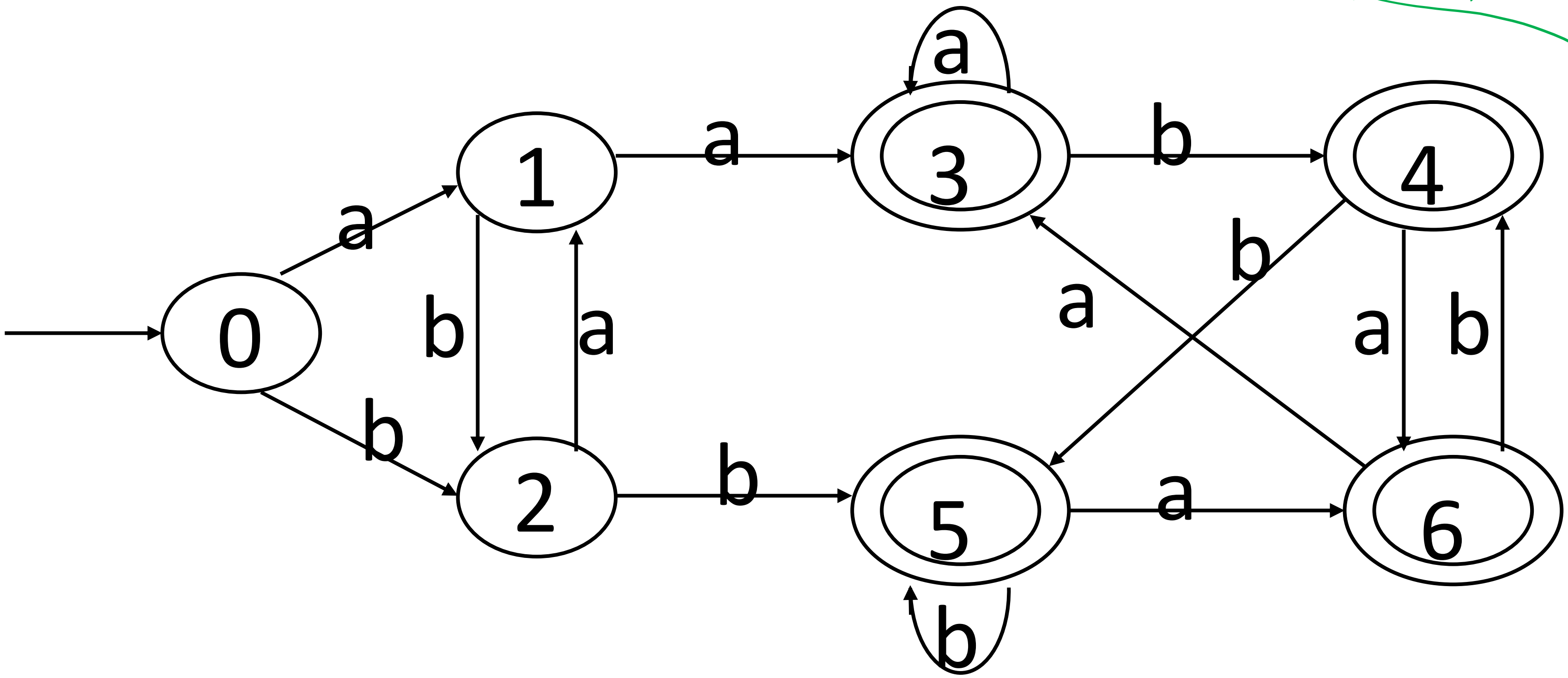
Example

{0, 1, 2} {3, 4, 5, 6}

{0, 2}, {1}, {3, 4, 5, 6}

{0}, {2}, {1}, {3, 4, 5, 6}

{3}



Example

- Initialization: $\prod_0 = \{\{0,1,2\}, \{3,4,5,6\}\}$
- For Non-accepting states in $\prod_0$:
  - a: move({0,2},a)={1} ; move({1},a)={3} . 1,3 do not in the same subgroup of $\prod_0$.
  - So ,$\prod_1$` = {{1}, {0,2}, {3,4,5,6}}
  - b: move({0},b)={2}; move({2},b)={5}. 2,5 do not in the same subgroup of $\prod_1$'.
  - So, $\prod_1$`` = {{1}, {0}, {2}, {3,4,5,6}}

Example

- For accepting states in $\prod_0$:
  - a: move({3,4,5,6},a)={3,6}, which is the subset of {3,4,5,6} in $\prod_1$“
  - b: move({3,4,5,6},b)={4,5}, which is the subset of {3,4,5,6} in $\prod_1$“
  - So, $\prod_1$ = {{1}, {0}, {2}, {3,4,5,6}}.
- Apply the same step again to $\prod_1$,and get $\prod_2$.
  - $\prod_2$ = {{1}, {0}, {2}, {3,4,5,6}}= $\prod_1$,
  - So, $\prod_{final}$ = $\prod_1$
- Let state 3 represent the state group {3,4,5,6}

Minimized DFA