Compiler Design
Lexical Analysis
From Regular Expressions to Automata

conf. dr. ing. Ciprian-Bogdan Chirila

chirila@cs.upt.ro

http://www.cs.upt.ro/~chirila

Outline

- Conversion of a NFA to DFA
- Simulation of an NFA
- Construction of an NFA from a Regular Expression

From Regular Expressions to Automata

- regular expression describes
  - lexical analyzers
  - pattern processing software
- implies simulation of DFA or NFA
- NFA simulation is less straightforward
- Techniques
  - to convert NFA to DFA
  - the subset construction technique
    - simulating NFA directly
    - when NFA to DFA is time consuming
  - to convert regular expression to NFA and then to DFA

Conversion of a NFA to a DFA

- subset construction
  - each state of DFA corresponds to a set of NFA states
- DFA states may be exponential in number of NFA states
- for lexical analysis NFA and DFA
  - have approximately the same number of states
  - the exponential behavior is not seen

Subset construction of an DFA from an NFA

- Input
  - an NFA N
- Output
  - DFA D accepting the same language as N
- Method
  - to construct a transition table *Dtran* for D
  - each state of D is a set of NFA states
  - to construct Dtran so D will simulate in parallel all possible moves N can make on a given input string
  - to deal with ε –transitions of N properly

Operations on NFA states

Operation	**Description**
ε-closure(s)	set of NFA states reachable from NFA state *s* on ε-transition alone
ε-closure(T)	set of NFA states reachable from some NFA state *s* in set T on ε-transitions alone
move(T,a)	set of NFA states to which there is a transition on input symbol a from some state s in T

Transitions

- s_0 – start state
- N can be in any states of ε-closure(s_0)
- reading input string *x*
  - N can be in the set of states T after
- reading input *a*
  - N can go in ε-closure(move(T, a))
- accepting states of D are all sets of N states that include at least one accepting state of N

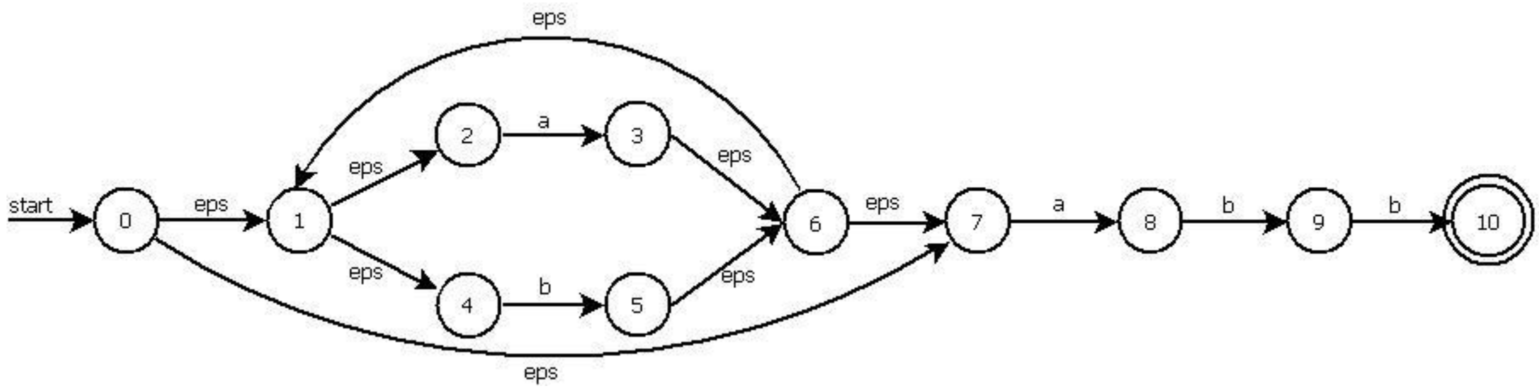
The Subset Construction

```
while(there is an unmarked state T in Dstates)
{
    mark T;
    for(each input symbol a)
    {
        U=ε-closure(move(T,a));
        if (U is not in Dstates)
                add U as unmarked state to Dstates;
        Dtran[T,a]=U;
    }
}
```

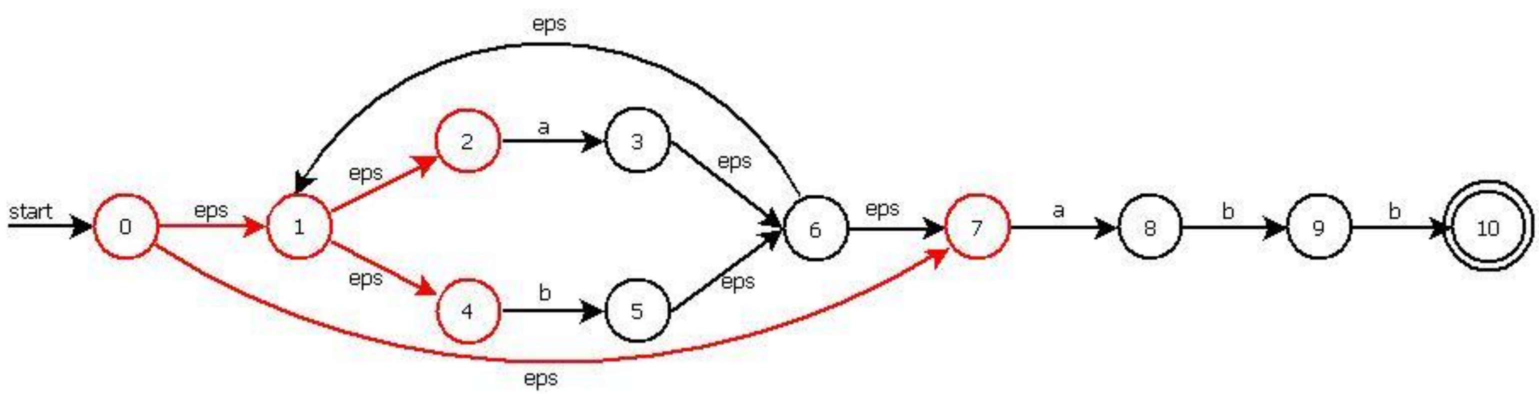
Computing ε-closure(T)

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

Example (a|b)*abb

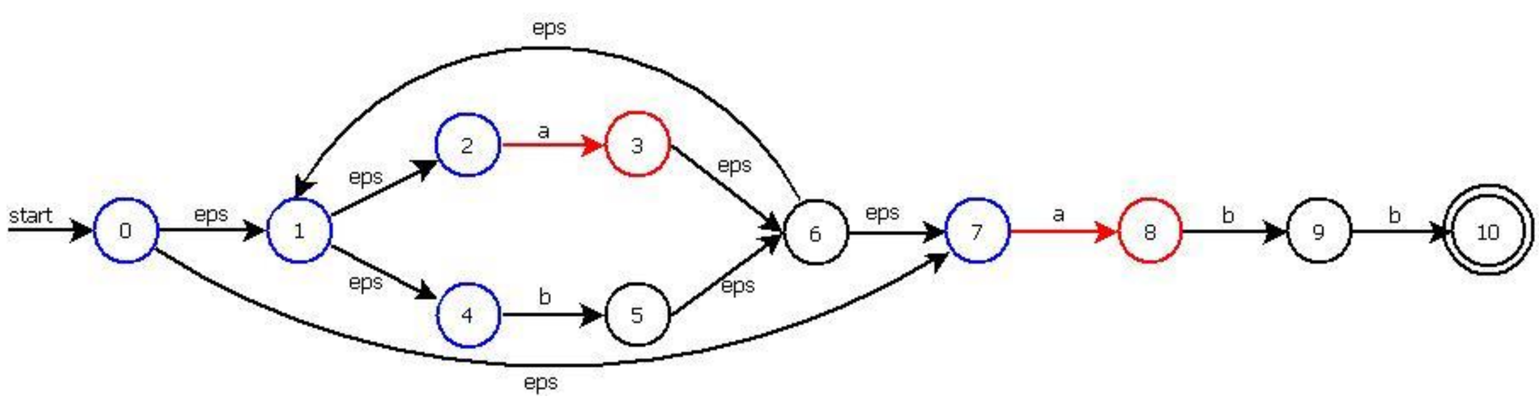


- A= ε-closure(0) or A={0,1,2,4,7}



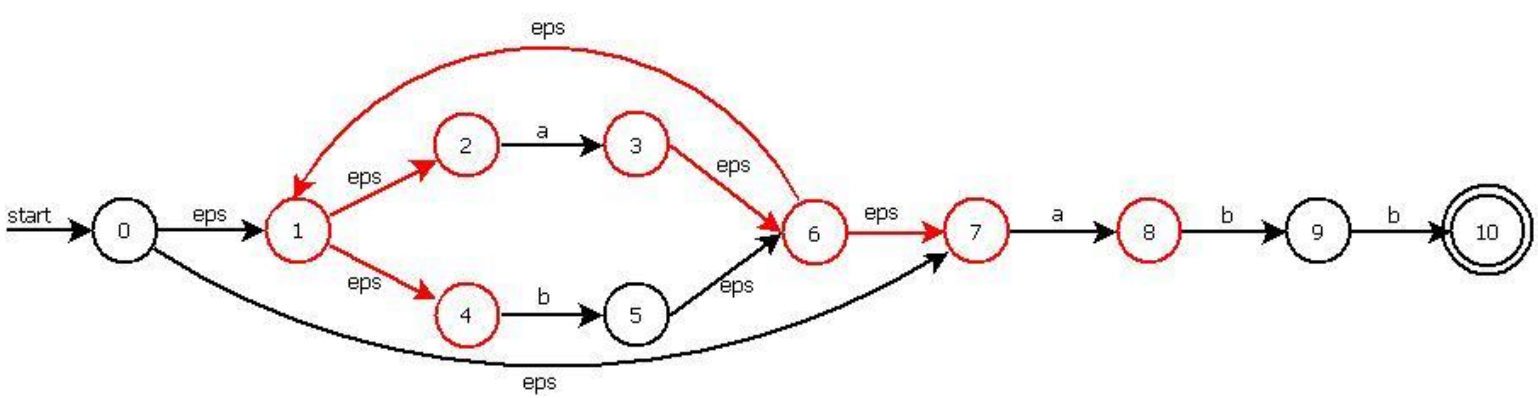
Example (a|b)*abb

- A={0,1,2,4,7}
- Dtran(A,a)= ε-closure(move(A,a))
- from {0,1,2,4,7} only {2,7} have a transition on *a* to {3,8}



Example (a|b)*abb

- Dtran[A,a]= ε-closure(move(A,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8}
- Dtran[A,a]=B



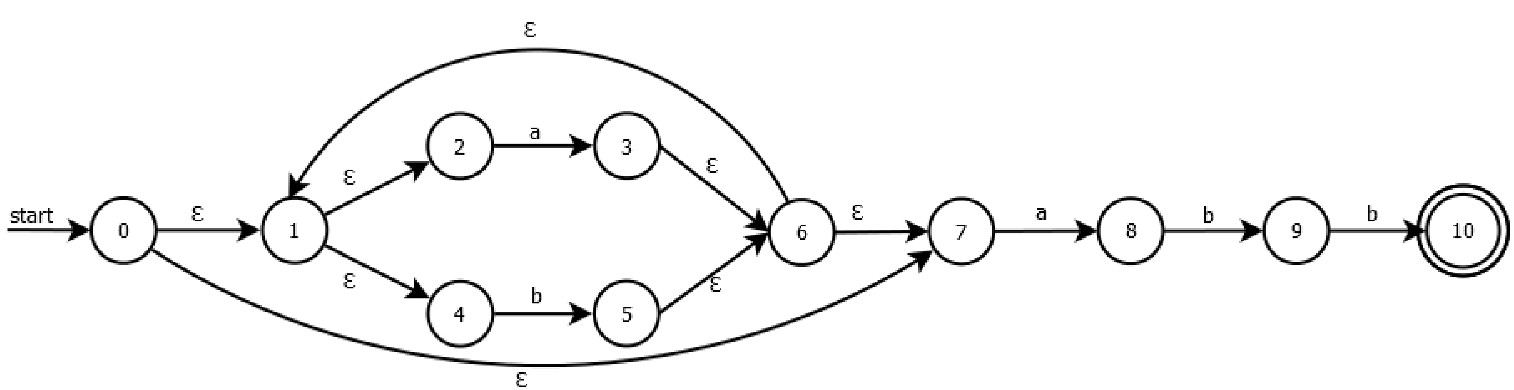
Example (a|b)*abb

- from {0,1,2,4,7} only {4} has a transition on *b* to {5}
- Dtran[A,b]= ε-closure({5})={1,2,4,5,6,7}
- Dtran[A,b]=C
- …

Example (a|b)*abb

- ft(B,a)={3,8}
- eps-closure({3,8})=B

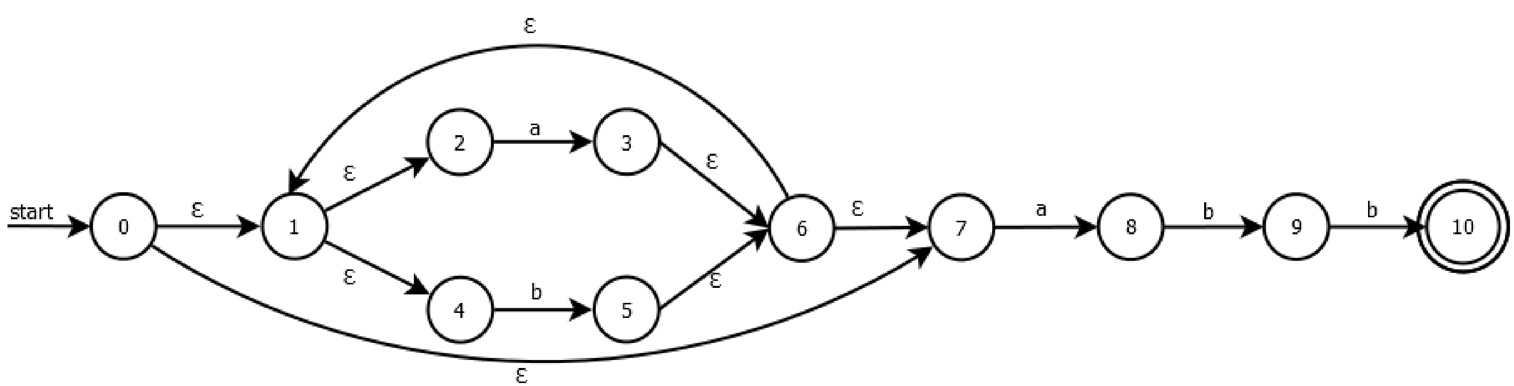
- ft(B,b)={5,9}
- eps-closure({5,9})={5,9,6,7,1,2,4}=D



Example (a|b)*abb

- ft(C,a)={3,8}
- eps-closure({3,8})=B

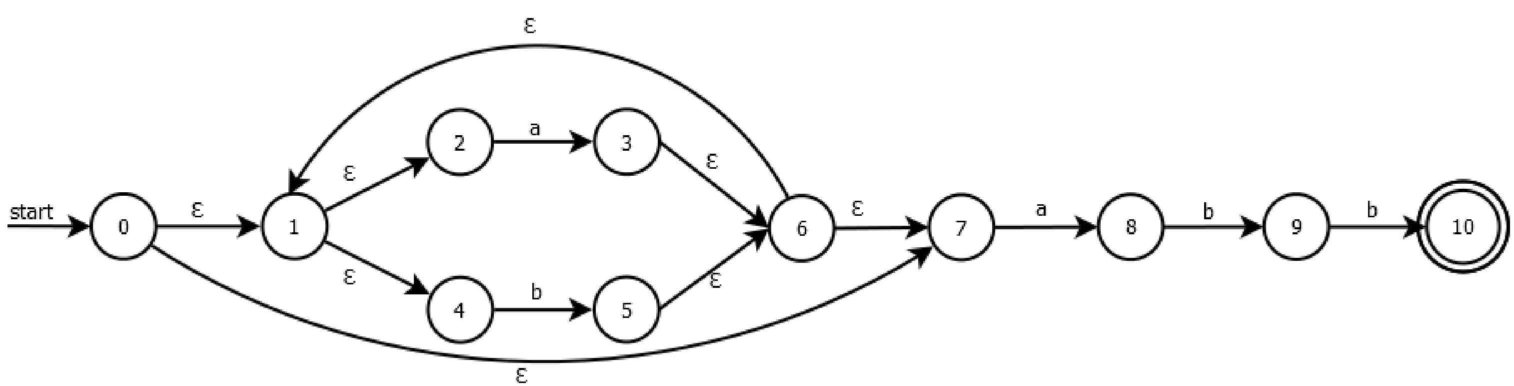
- ft(C,b)={5}
- eps-closure({5})=C



Example (a|b)*abb

- ft(D,a)={3,8}
- eps-closure({3,8})=B

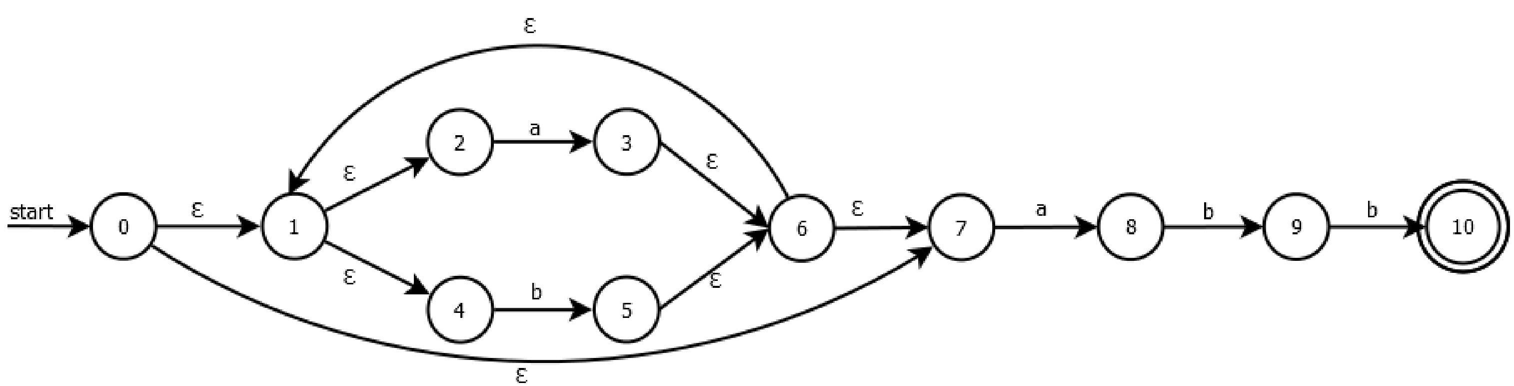
- ft(D,b)={5,10}
- eps-closure({5,10})={5,10,6,7,1,2,4}=E



Example (a|b)*abb

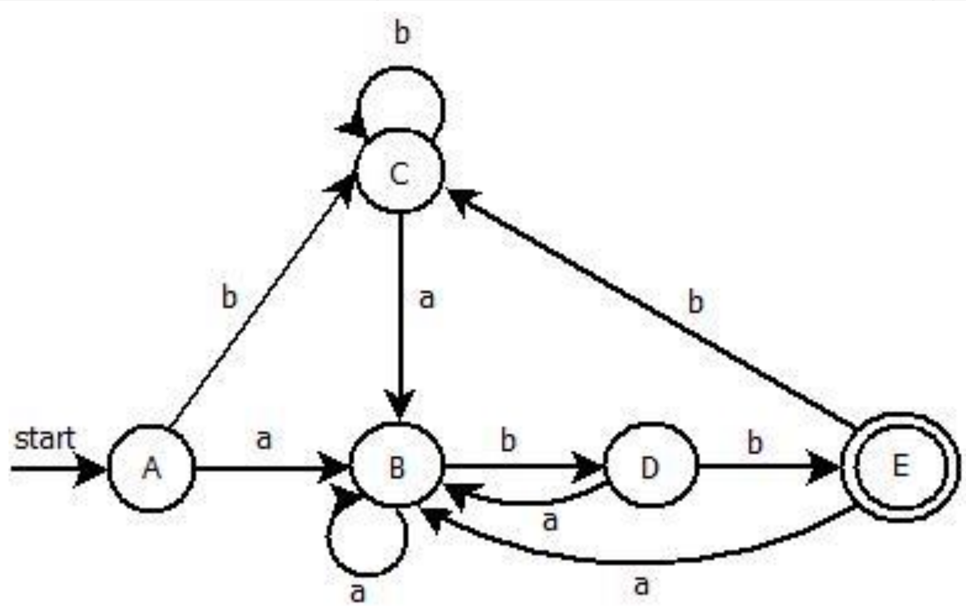
- ft(E,a)={3,8}
- eps-closure({3,8})=B

- ft(E,b)={5}
- eps-closure({5})=C



Example (a|b)*abb

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



Simulation of an NFA

- the strategy in text editing programs is
  - to construct a NFA from a regular expression
  - to simulate NFA using on-the-fly subset construction
- Input
  - input string x terminated by **eof**
  - NFA N
    - start state s_0
    - accepting states F
    - transition function move
- Output
  - yes / no
- Method
  - to keep the current states S reached from s_0
  - if *c* is the next input read by *nextChar()*
  - we compute *move(S,c)* and then we use ε-closure()

Algorithm: Simulating an NFA

```
S=ε-closure(s0);
c=nextChar();
while(c!=eof) {
  S=ε-enclosure(move(S,c));
  c=nextChar();
}
if(S∩F!=ø) return "yes";
else return "no";
```

Implementation of NFA Simulation

- two stacks each holding a set of NFA states
- a boolean array *alreadyOn*
- a two dimensional array *move[s,a]*

NFA Simulation Data Structures

- two stacks each holding a set of NFA states
  - used for the values of S in both sides of assign = operator in line 4

    `S=ε-enclosure(move(S,c));`

    - right side – *oldStates*
    - left side – *newStates*
  - *newStates->oldStates*

NFA Simulation Data Structures

- boolean array *alreadyOn*
  - indexed by NFA states
  - indicates which states are in *newStates*
  - array and stack hold the same information
  - it is much faster to interrogate the array than to search the stack
- two dimensional array *move[s,a]*
  - the entries are set of states
  - implemented by linked lists

Implementation of step 1

```
S=ε-closure(s0);

addState(s)
{
  push s onto newStates;
  alreadyOn[s]=TRUE;
  for(t on move[s,ε])
     if(!alreadyOn(t))
          addState(t);
}
```

Implementation of step 4

```
04    S=ε-enclosure(move(S,c));

for (s on oldStates)
{
  for (t on move[s,c])
      if(!alreadyOn[t])
            addState(t);
  pop s from oldStates;
}

for (s on newStates)
{
  pop s from newStates;
  push s onto oldStates;
  alreadyOn[s]=FALSE;
}
```

Construction of an NFA from a Regular Expression

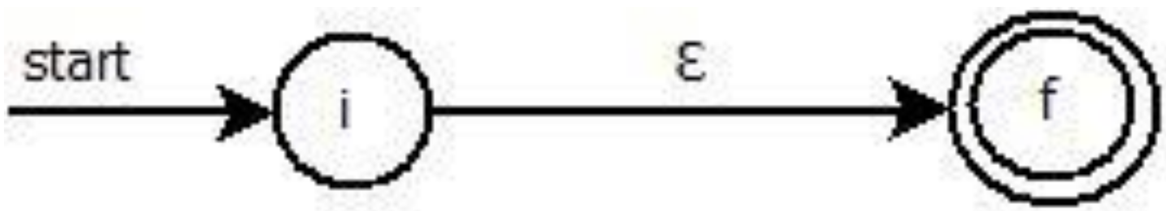
- to convert a regular expression to a NFA
- McNaughton-Yamada-Thompson algorithm
- syntax-directed
  - it works recursively up the parse tree of the regular expression
- for each subexpression a NFA with a single accepting state is built

Construction of an NFA from a Regular Expression

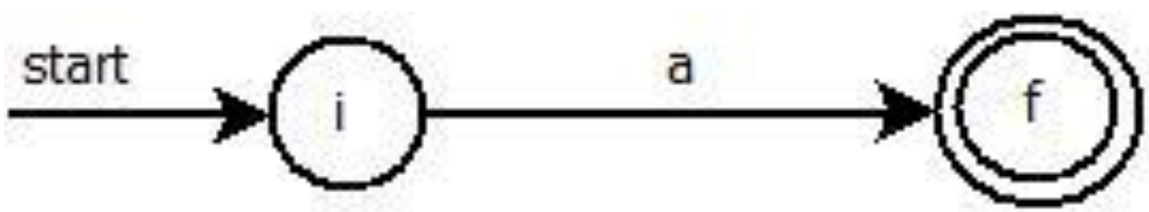
- Input
  - regular expression *r* over an alphabet *Σ*
- Output
  - An NFA accepting L(r)
- Method
  - to parse r into constituent subexpressions
  - basis rules for handling subexpressions with no operators
  - inductive rules for creating larger NFAs from subexpressions NFAs
    - union, concatenation, closure

Basis Rules for Constructing NFA

- for expression ε

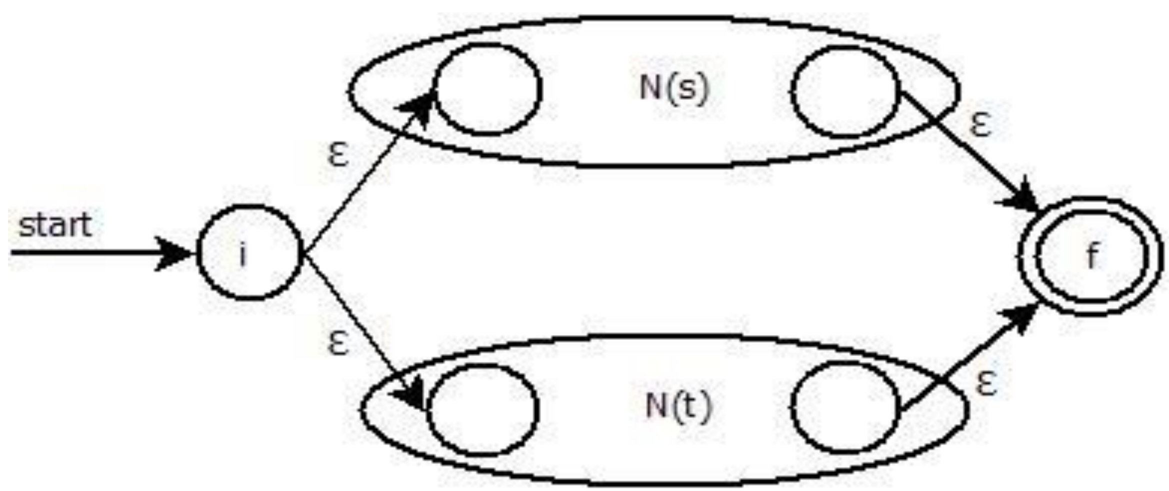


- for expression a



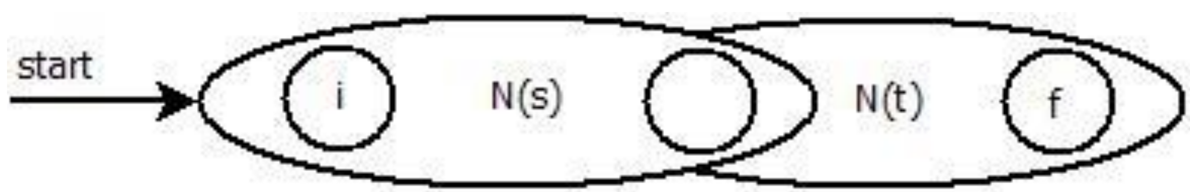
NFA for the Union of Two Regular Expressions

- r=s|t
- N(s) and N(t) are NFA's for regular expressions s and t



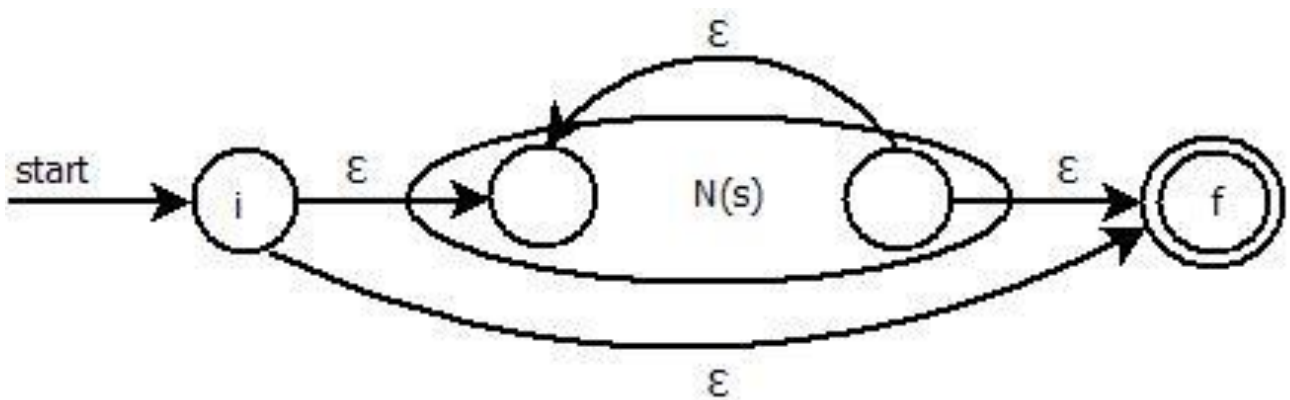
NFA for the Concatenation of Two Regular Expressions

- r=st
- N(s) and N(t) are NFA's for regular expressions s and t



Induction Rules for Constructing NFA

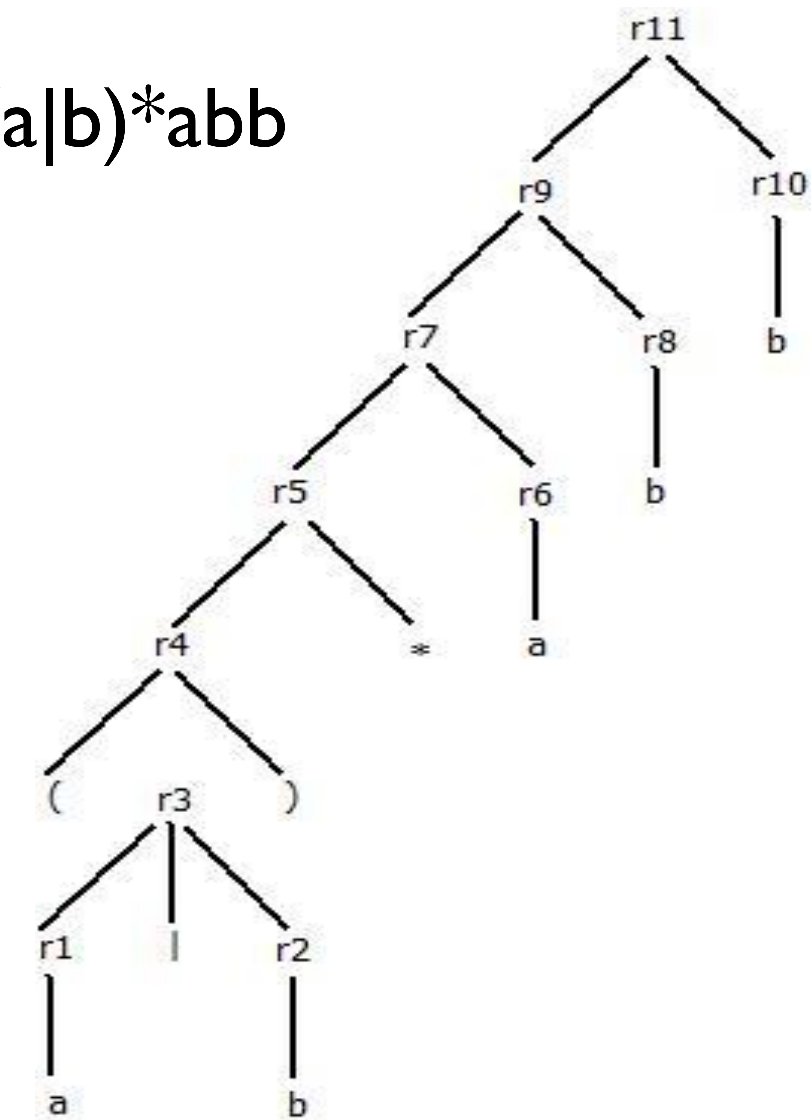
- r=s*
- N(s) is the NFA for the regular expressions s



- r=(s)
  - L(r)=L(s)
  - N(s) is equivalent to N(r)

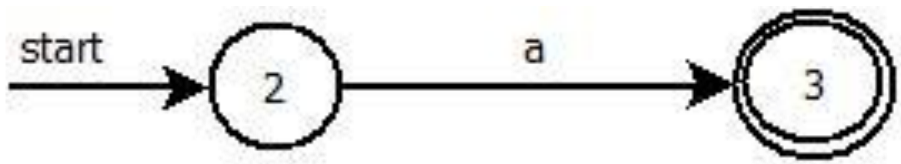
Example

- parse tree for (a|b)*abb

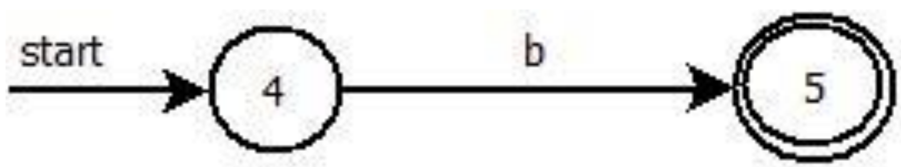


Example

- NFA for r1

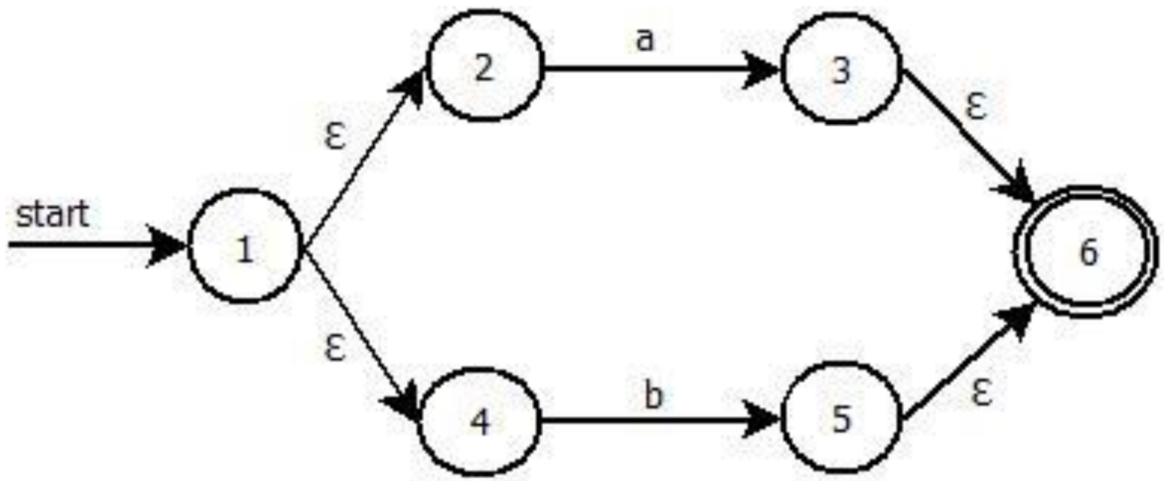


- NFA for r2



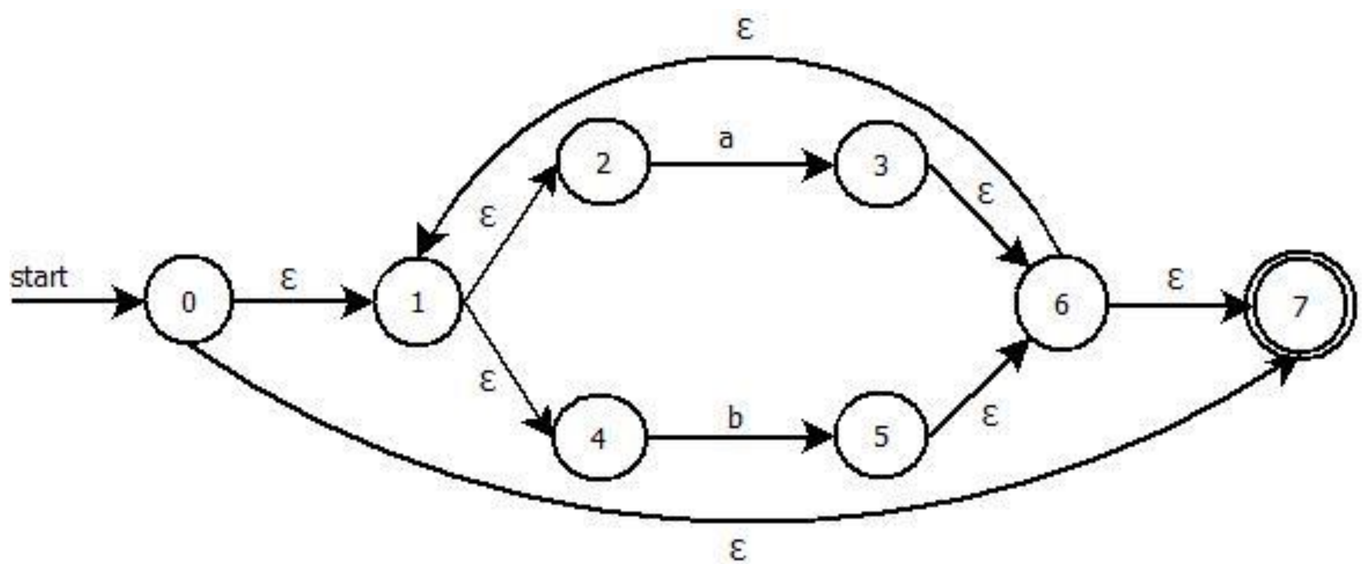
Example

- NFA for r3=r1 | r2



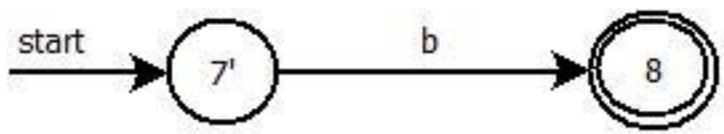
Example

- NFA for r5=(r3)*

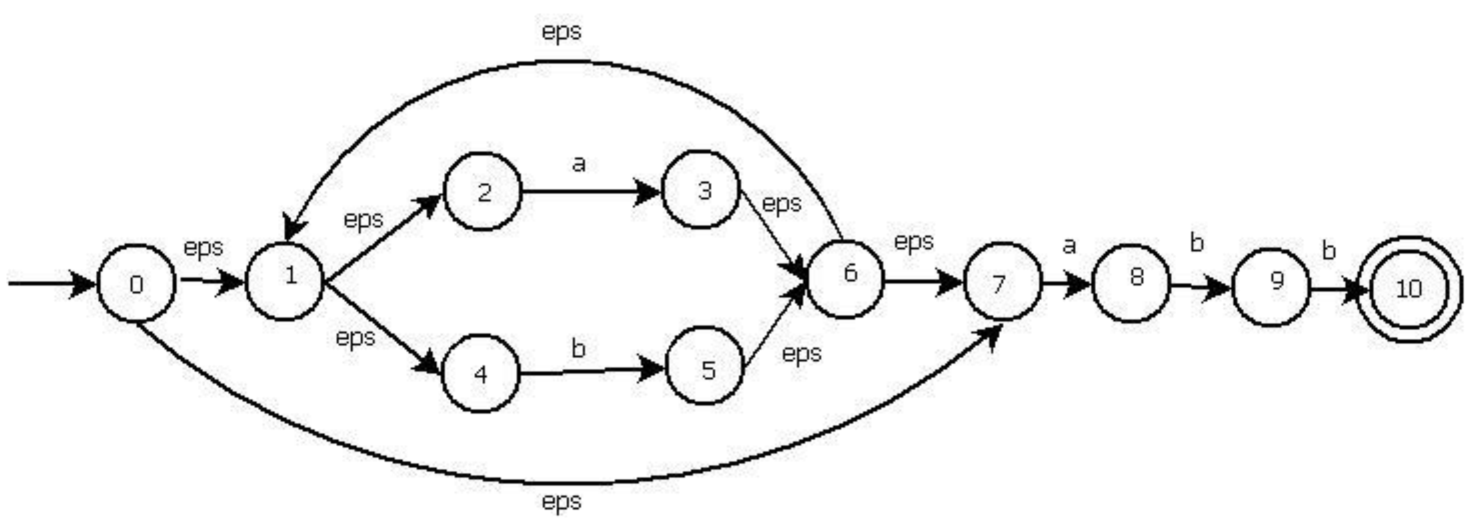


Example

- NFA for r7=r5r6



- …



Bibliography

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers, Principles, Techniques and Tools, Second Edition, 2007