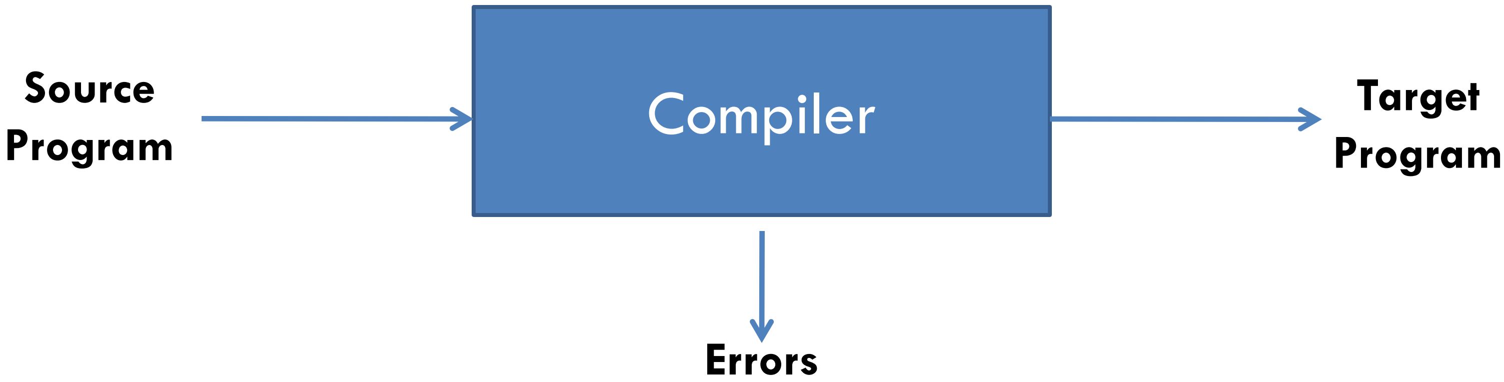
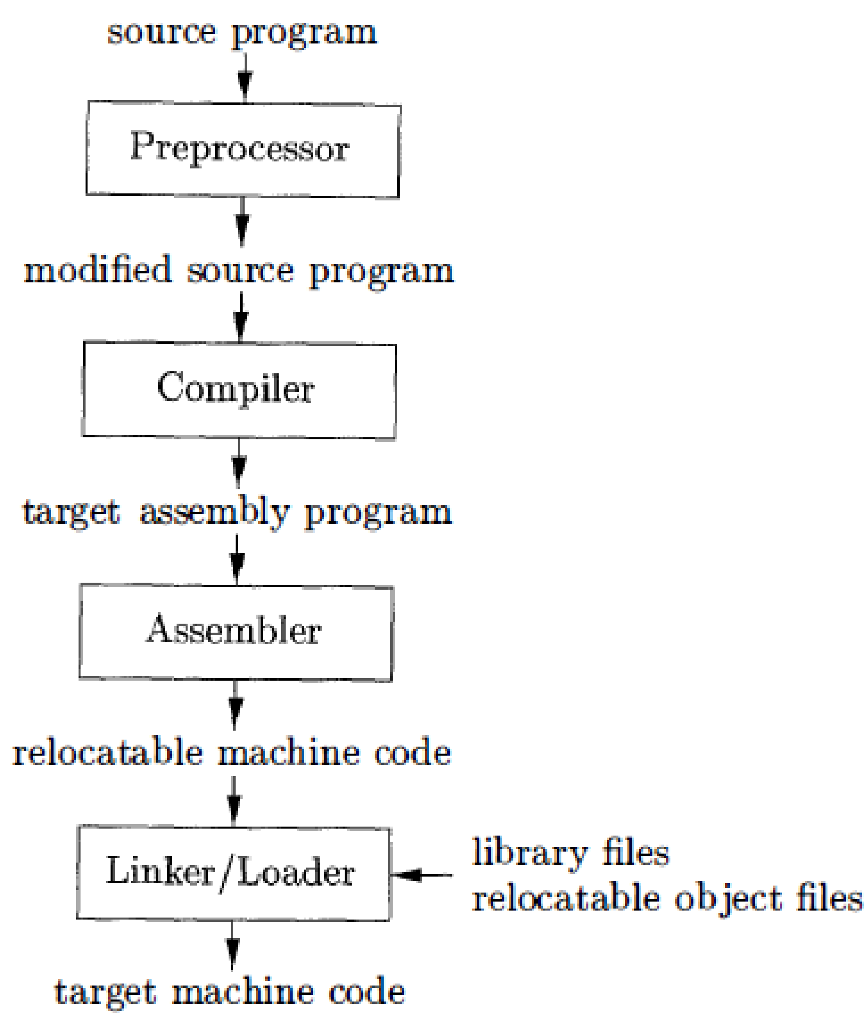
INTRODUCTION TO COMPILERS

Prof. S.J. Soni, SPCE - Visnagar

Compiler



Language Processing System



Phases of a compiler

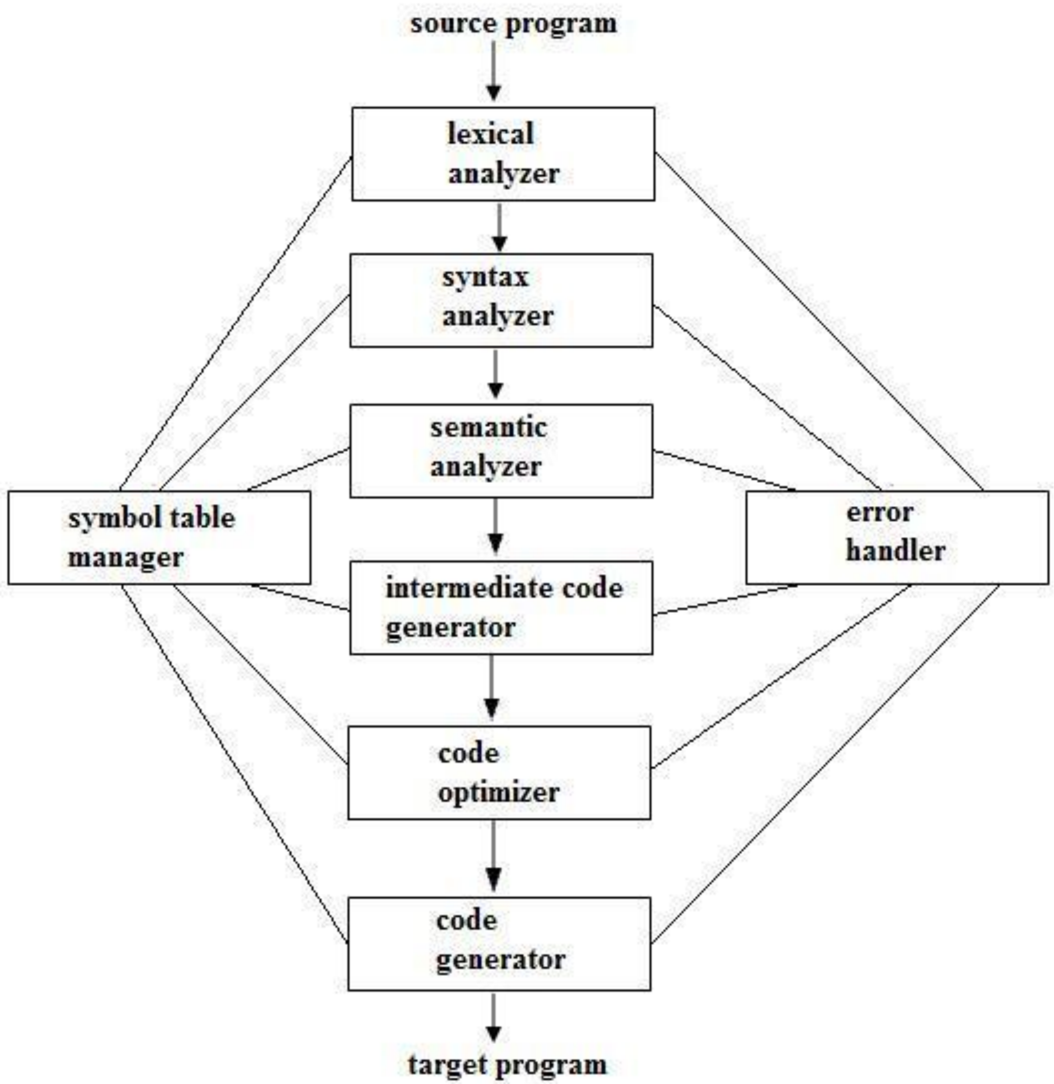
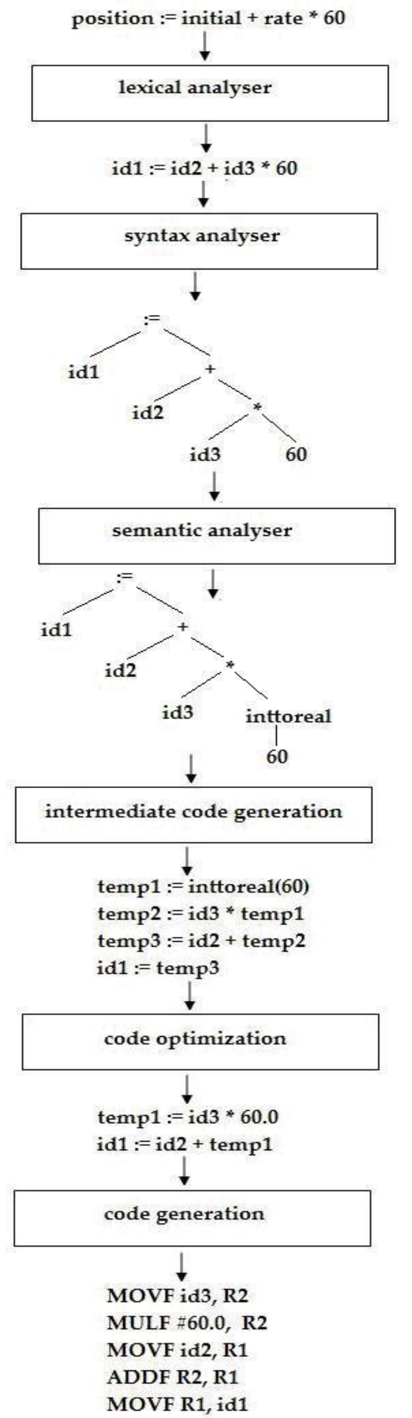


Fig 1.5 Phases of a compiler

Translation of Statement



Lexical Analysis

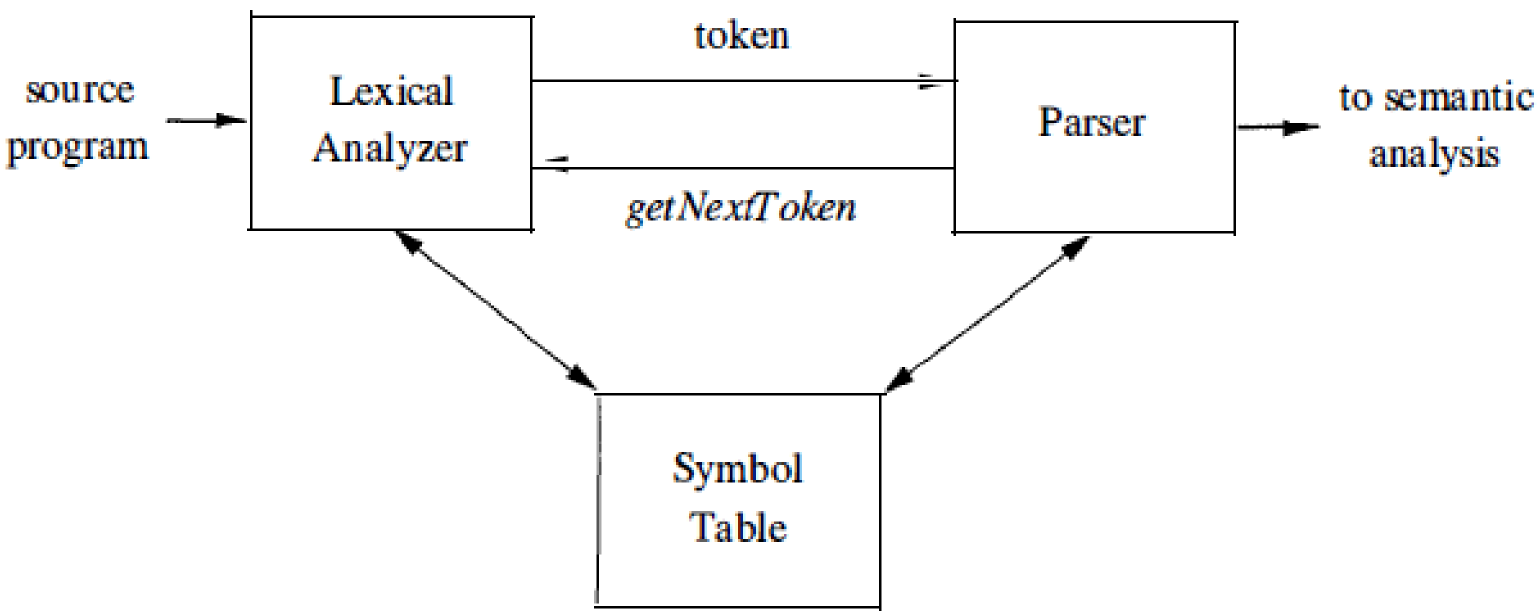


Figure 3.1: Interactions between the lexical analyzer and the parser

Tokens, Patterns, Lexemes

```
const pi = 3.1416;
```

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Figure 3.2: Examples of tokens

Strings and Languages

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s. For example, **ban**, **banana**, and ϵ are prefixes of **banana**.

2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, **nana**, **banana**, and ϵ are suffixes of **banana**.

3. A *substring* of s is obtained by deleting any prefix and any suffix from s. For instance, **banana**, **nan**, and ϵ are substrings of **banana**.

4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.

5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, **baan** is a subsequence of **banana**.

Notational Shorthands

1. *One or more instances.* The unary, postfix operator $^+$ represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then $(r)^+$ denotes the language $\left(L(r)\right)^+$. The operator $^+$ has the same precedence and associativity as the operator $*$. Two useful algebraic laws, $r^* = r^+|\epsilon$ and $r^+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one occurrence." That is, $r?$ is equivalent to $r|\epsilon$, or put another way, $L(r?) = L(r) \cup \{\epsilon\}$. The ? operator has the same precedence and associativity as $*$ and $+$.

3. *Character classes.* A regular expression $a_1|a_2|\cdots|a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2\cdots a_n]$. More importantly, when $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n, that is, just the first and last separated by a hyphen. Thus, $[\mathbf{abc}]$ is shorthand for $\mathbf{a}|\mathbf{b}|\mathbf{c}$, and $[\mathbf{a}\text{-}\mathbf{z}]$ is shorthand for $\mathbf{a}|\mathbf{b}|\cdots|\mathbf{z}$.

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.
2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.
3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.
5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

Regular Expressions - Examples

RE Pattern	RE
The set of strings over (0,1) that have at least one 1.	0*1(0|1)*
The set of strings over (0,1) that have at most one 1.	0*|(0*10*)
The set of strings over (0,1) with at least two consecutive 0's.	(0|1)*00(0|1)*
The set of strings over (0,1) without two consecutive 0's.	(1|01)*(0|epsilon)
The set of strings over (0,1) that not end with 0.	(0*1)*

Finite Automata

- A ***recognizer*** for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language and "no" otherwise.
- We compile a regular expression into a recognizer by constructing a *generalized transition diagram* called a ***finite automata.***
- A finite automata can be ***deterministic*** or ***nondeterministic*** , <u>where "nondeterministic" means that more than one transition out of a state may be possible on the same input symbol.</u>

Nondeterministic Finite Automata

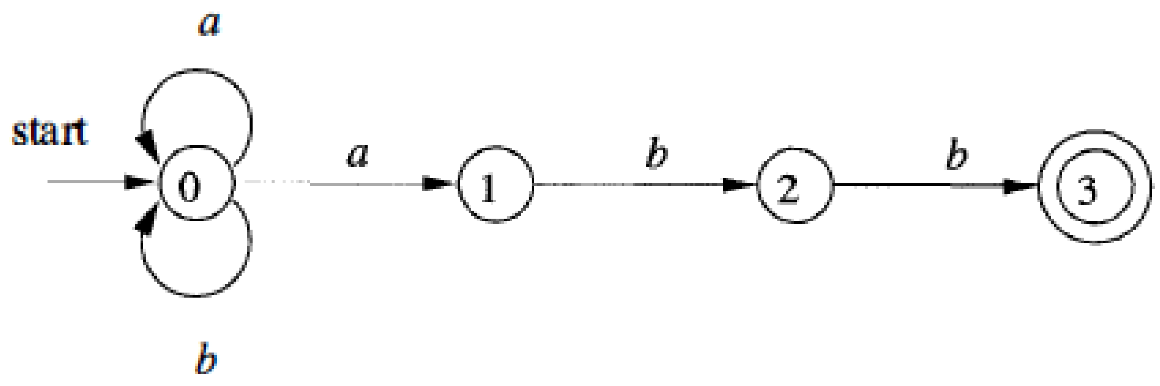
A *nondeterministic finite automaton* (NFA) consists of:

1. A finite set of states S.
2. A set of input symbols Σ, the *input alphabet*. We assume that ϵ, which stands for the empty string, is never a member of Σ.
3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.
4. A state s_0 from S that is distinguished as the *start state* (or *initial state*).
5. A set of states F, a subset of S, that is distinguished as the *accepting states* (or *final states*).

Nondeterministic Finite Automata

RE $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

NFA

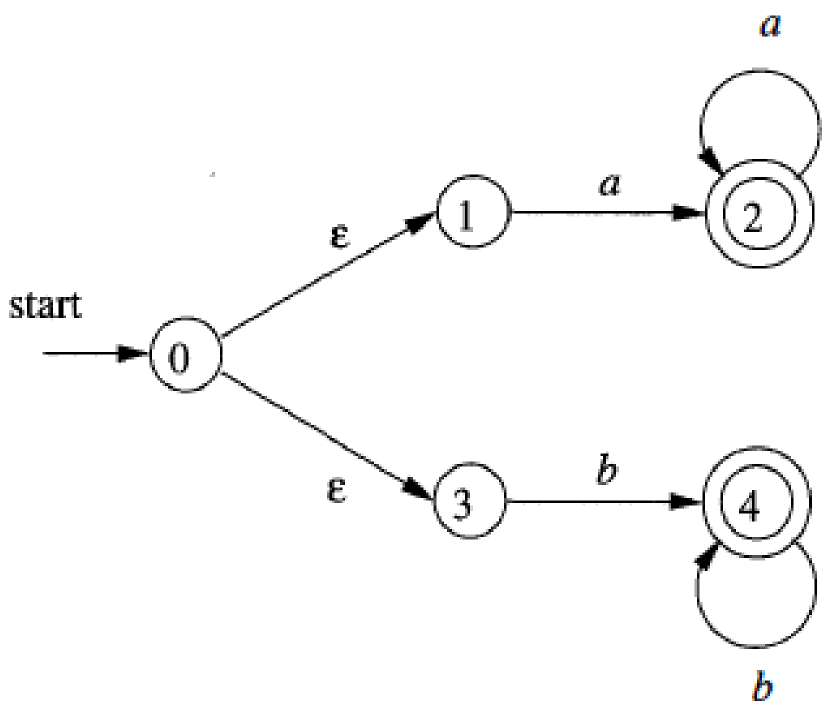


Transition Table

STATE	a	b	ϵ
0	$\{0, 1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Nondeterministic Finite Automata

$$aa^*|bb^*$$



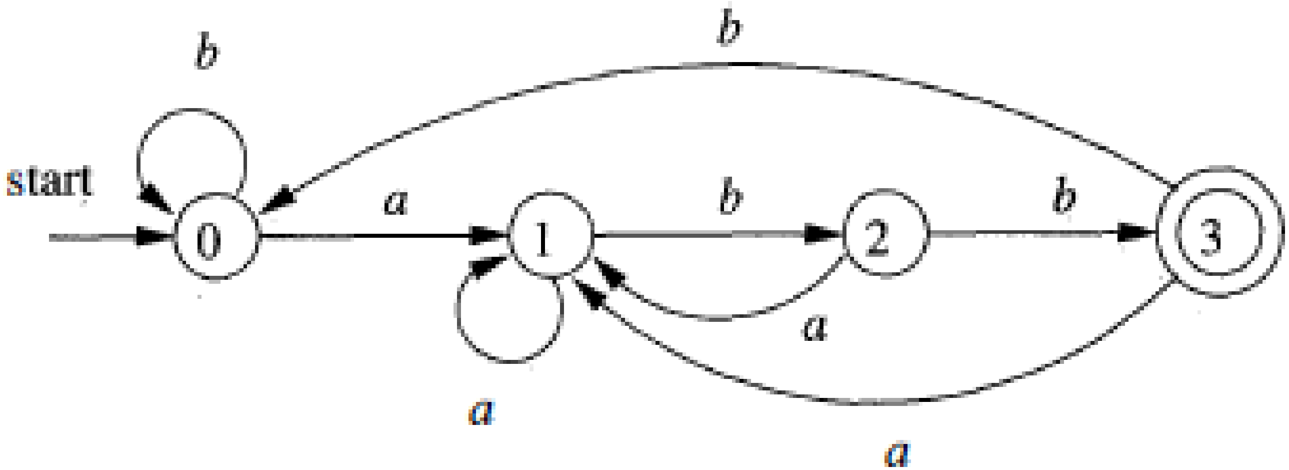
Deterministic Finite Automata (DFA)

A *deterministic finite automaton* (DFA) is a special case of an NFA where:

1. There are no moves on input ϵ, and
2. For each state s and input symbol a, there is exactly one edge out of s labeled a.

DFA - Example

$(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

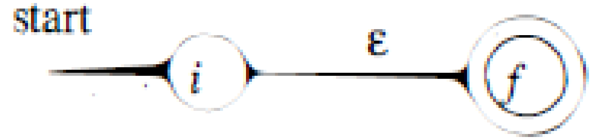


Construction of an NFA from a RE

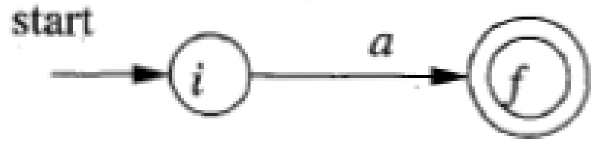
Thompson's construction rules

The McNaughton-Yamada-Thompson algorithm

BASIS: For expression ϵ construct the NFA

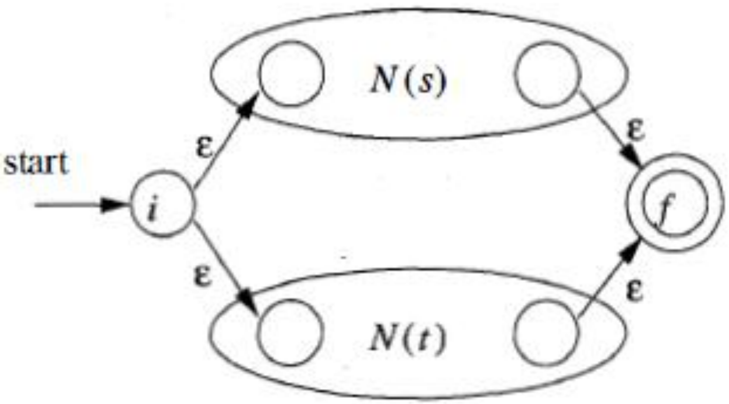


For any subexpression a in Σ, construct the NFA



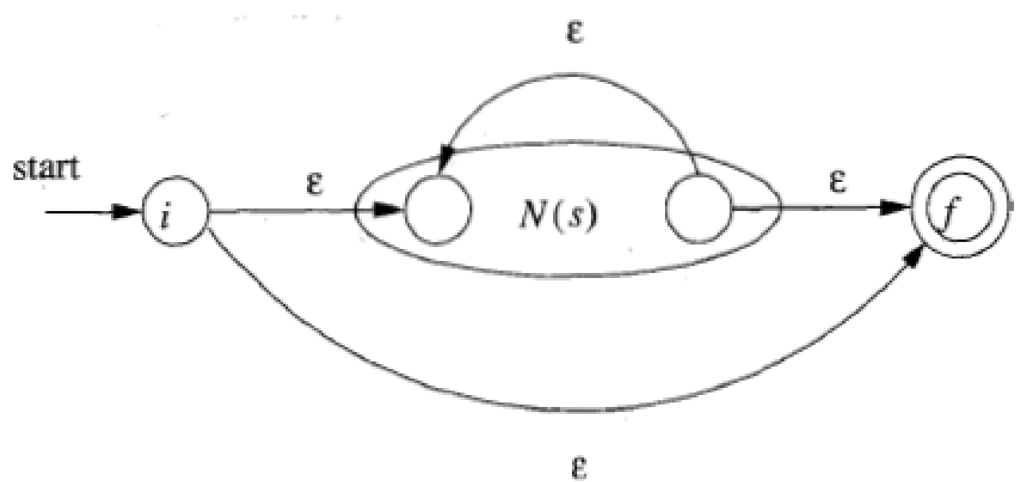
Construction of an NFA from a RE

$r = s|t.$



$r = st.$

$r = s^*.$



Conversion from NFA to DFA

- The algorithm, called **<u>subset construction</u>** is used for conversion from NFA to DFA.
- This algorithm is also useful for simulating an NFA by a computer program.
- The general idea behind the NFA-to-DFA construction is that each DFA state corresponds to a set of NFA states.

Subset Construction - Algorithm

- This algorithm constructs a transition table Dtran for DFA. Each state of DFA is a set of NFA states, and we construct Dtran so DFA will simulate "in parallel" all possible moves N can make on a given input string.
- Note that s is a single state of N, while T is a set of states of N.

OPERATION	DESCRIPTION
$\epsilon\text{-}closure(s)$	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
$\epsilon\text{-}closure(T)$	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}\ \epsilon\text{-}closure(s)$.
$move(T, a)$	Set of NFA states to which there is a transition on input symbol a from some state s in T.

Operations on NFA states

Subset Construction - Algorithm

```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
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 an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

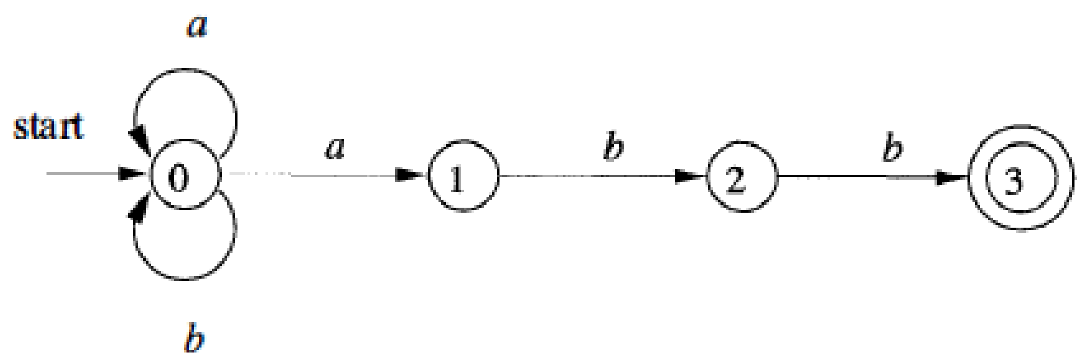
The subset construction

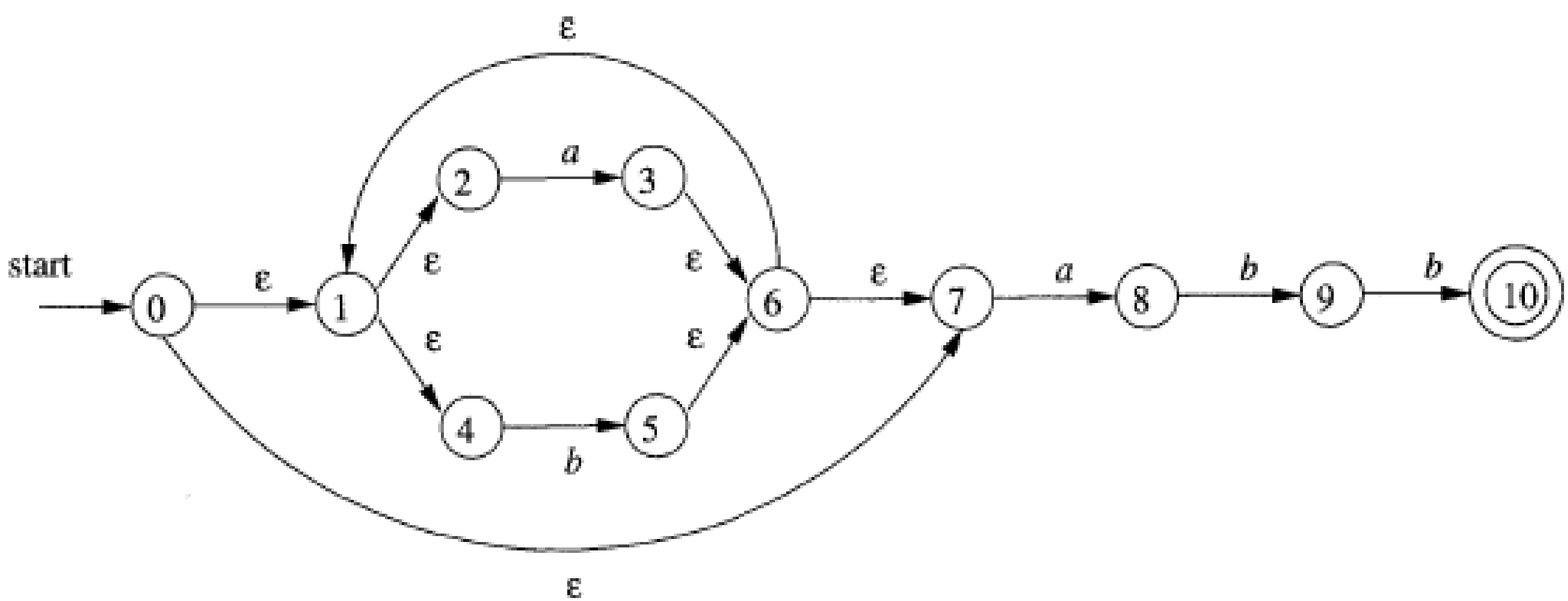
```
push all states of T onto stack;
initialize ε-closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off stack;
        for ( each state u with an edge from t to u labeled ε )
                if ( u is not in ε-closure(T) ) {
                        add u to ε-closure(T);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

Computing ϵ-*closure*(T)

Subset Construction - Example

$(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$





Subset Construction - Example

The start state A of the equivalent DFA is ϵ-$closure(0)$, or $A = \{0, 1, 2, 4, 7\}$,

The input alphabet is $\{a, b\}$.

first step is to mark A and compute

$$Dtran[A, a] = \epsilon\text{-}closure(move(A, a))$$

Among the states 0, 1, 2, 4, and 7, only 2 and 7 have transitions on a, to 3 and 8, respectively.

$$\text{Thus, } move(A, a) = \{3, 8\}.$$

$$\epsilon\text{-}closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

$$Dtran[A, a] = \epsilon\text{-}closure(move(A, a)) = \epsilon\text{-}closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

Subset Construction - Example

Let us call this set B, so $Dtran[A, a] = B$.

Now, we must compute $Dtran[A, b]$.

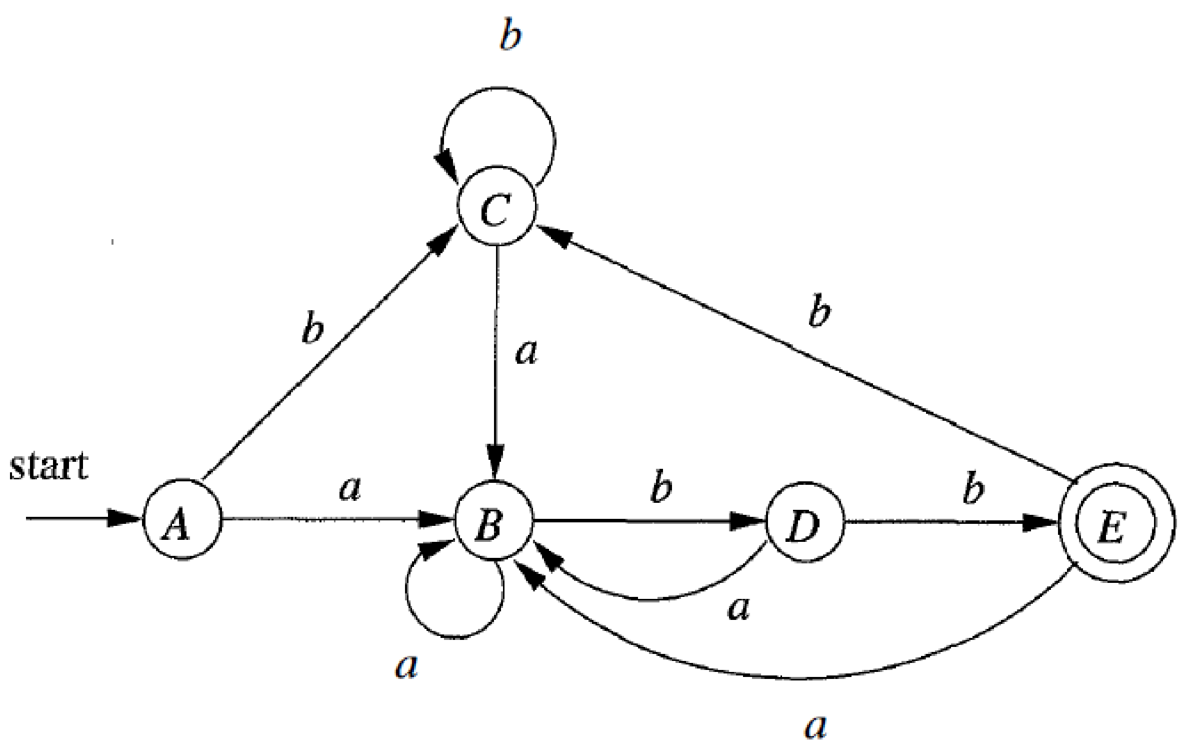
$$Dtran[A, b] = \epsilon\text{-}closure(\{5\}) = \{1, 2, 4, 5, 6, 7\}$$

Let us call the above set C, so $Dtran[A, b] = C$.

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

Transition table $Dtran$ for DFA D

Subset Construction - Example



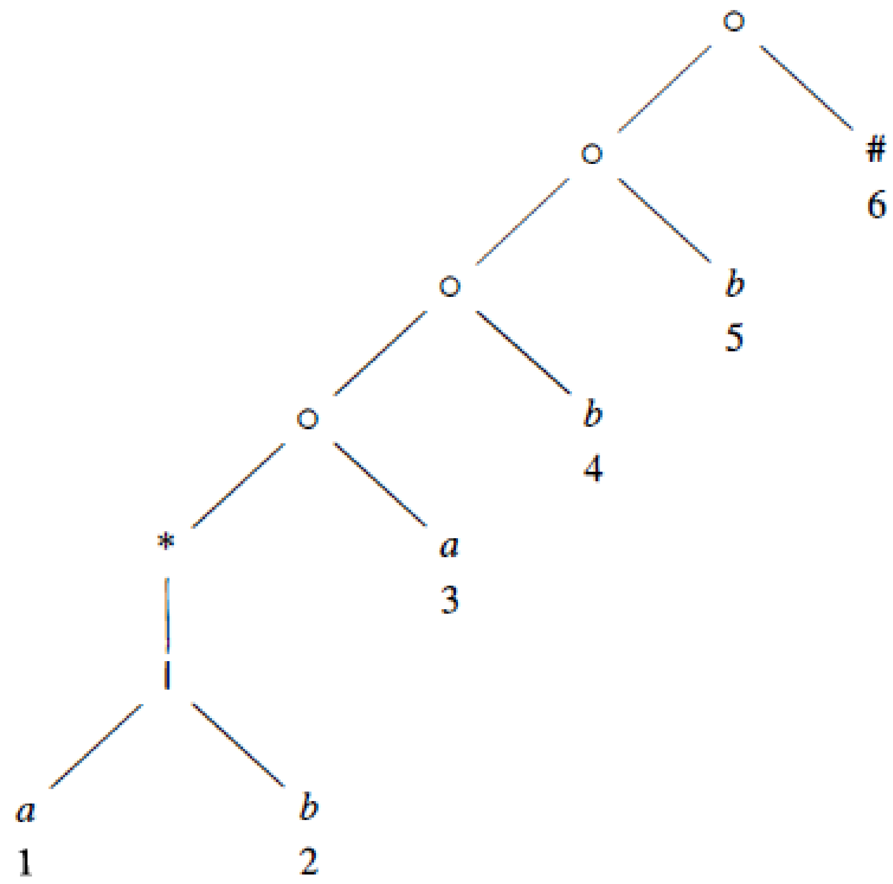
Result of applying the subset construction

From a RE to a DFA

The constructed NFA has only one accepting state, but this state, having no out-transitions, is not an important state. By concatenating a unique right endmarker # to a regular expression r, we give the accepting state for r a transition on #, making it an important state of the NFA for $(r)\#$. In other words, by using the *augmented* regular expression $(r)\#$, we can forget about accepting states as the subset construction proceeds; when the construction is complete, any state with a transition on # must be an accepting state.

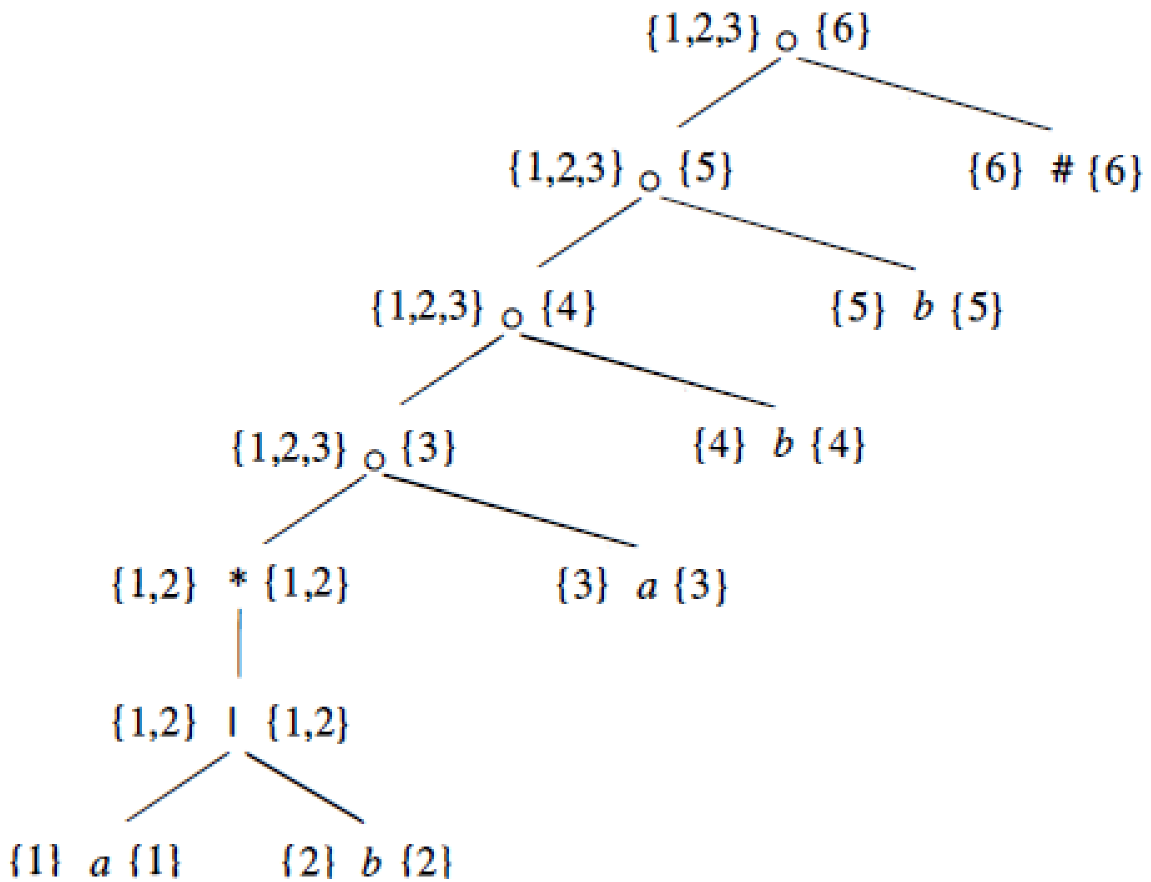
The important states of the NFA correspond directly to the positions in the regular expression that hold symbols of the alphabet. It is useful, as we shall see, to present the regular expression by its *syntax tree*, where the leaves correspond to operands and the interior nodes correspond to operators. An interior node is called a *cat-node*, *or-node*, or *star-node* if it is labeled by the concatenation operator (dot), union operator $|$, or star operator $*$, respectively.

From a RE to a DFA



Syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

From a RE to a DFA



firstpos and *lastpos* for nodes in the syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

From a RE to a DFA

Finally, we need to see how to compute *followpos*. There are only two ways that a position of a regular expression can be made to follow another.

1. If n is a cat-node with left child c_1 and right child c_2, then for every position i in $lastpos(c_1)$, all positions in $firstpos(c_2)$ are in $followpos(i)$.

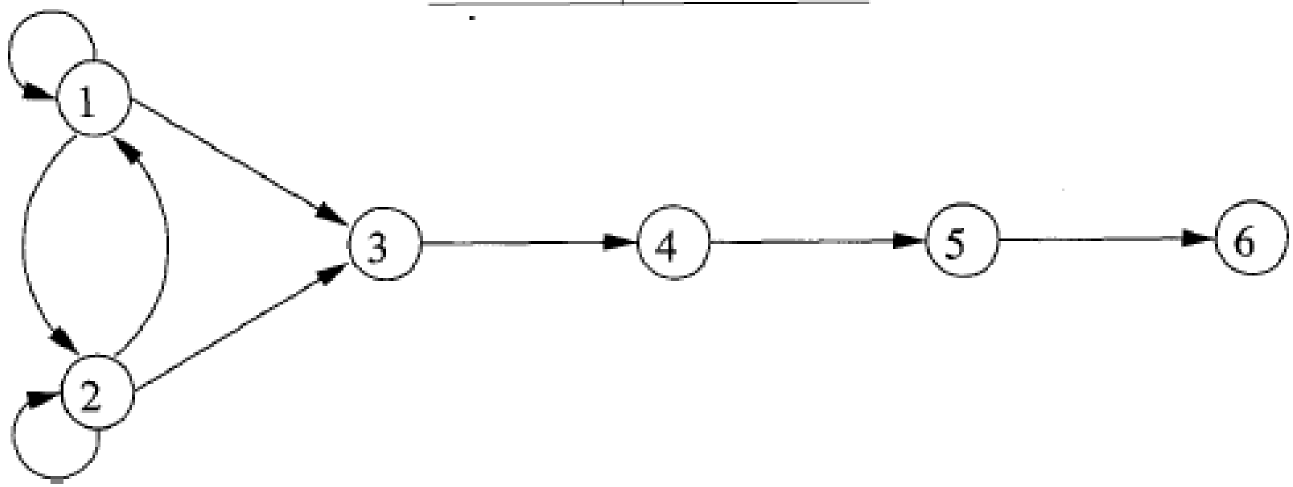
2. If n is a star-node, and i is a position in $lastpos(n)$, then all positions in $firstpos(n)$ are in $followpos(i)$.

NODE n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

The function *followpos*

From a RE to a DFA

NODE n	$followpos(n)$
1	$\{1,2,3\}$
2	$\{1,2,3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$



Directed graph for the function *followpos*

From a RE to a DFA

The value of *firstpos* for the root of the tree is $\{1, 2, 3\}$, so this set is the start state of D.

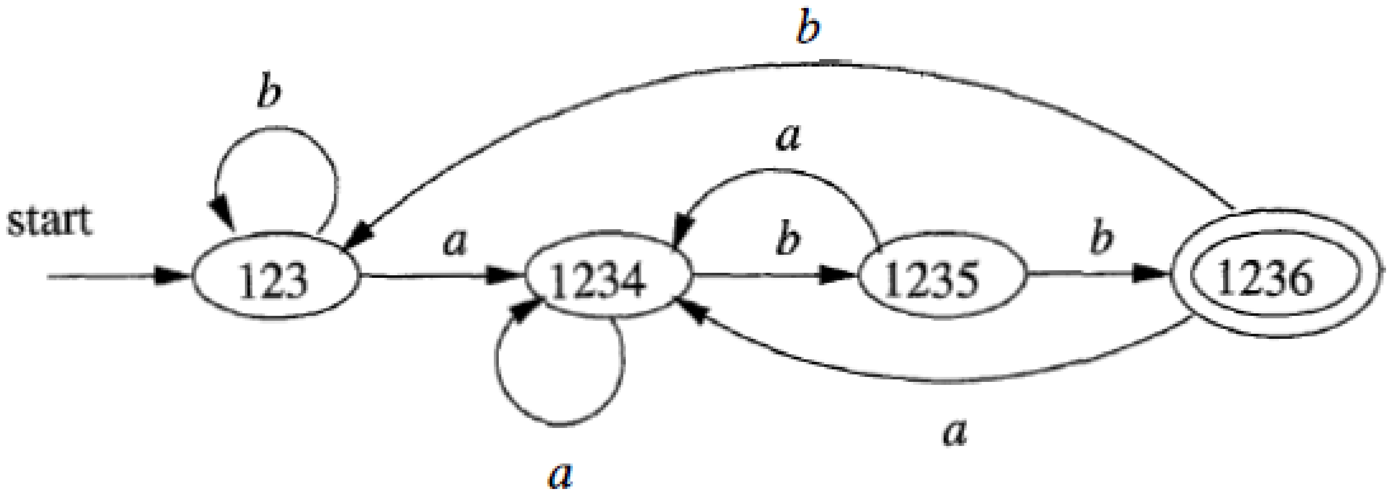
Call this set of states A. We must compute $Dtran[A, a]$ and $Dtran[A, b]$.

Among the positions of A, 1 and 3 correspond to a, while 2 corresponds to b.

Thus, $Dtran[A, a] = followpos(1) \cup followpos(3) = \{1, 2, 3, 4\}$,
and $Dtran[A, b] = followpos(2) = \{1, 2, 3\}$.

$B = \{1, 2, 3, 4\}$, is new, add it to *Dstates* and proceed to compute its transitions.

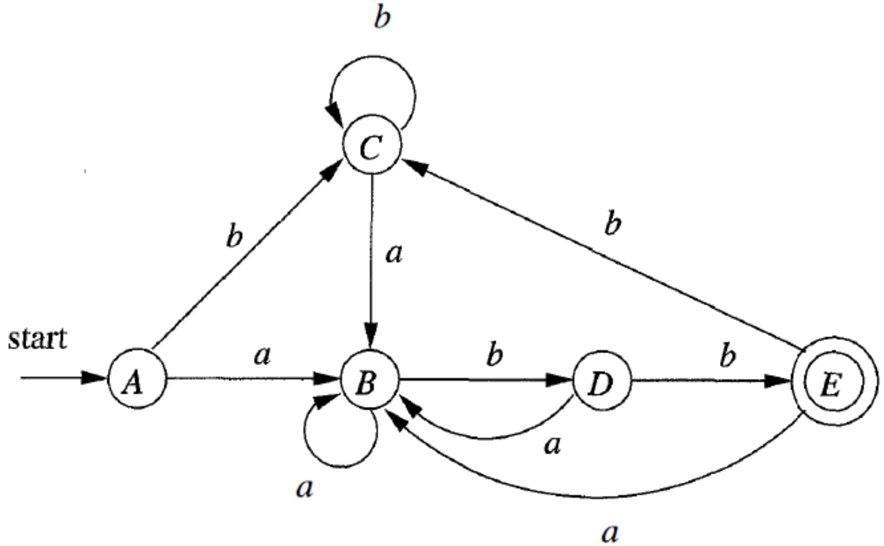
From a RE to a DFA



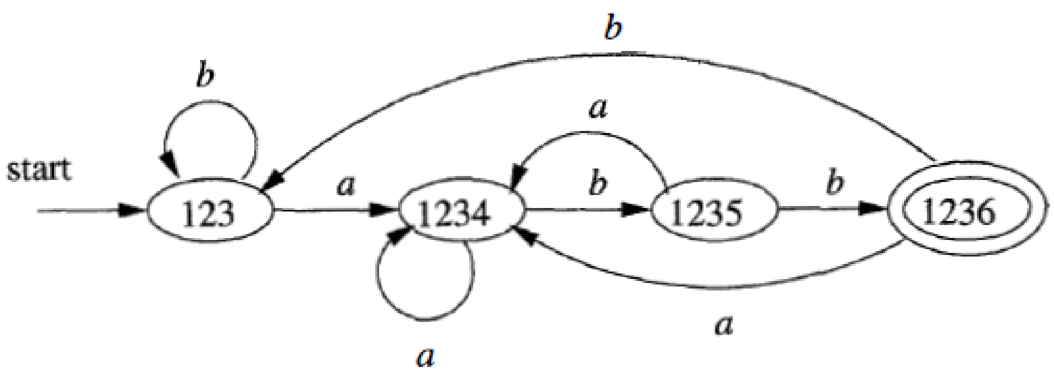
DFA constructed

DFA to Optimized DFA (Minimizing the number of states of a DFA)

$(a|b)^*abb$



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



Partition Algorithm

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

- The initial partition consists of the two groups {A, B, C, D} {E } , which are respectively the nonaccepting states and the accepting states.
- The group {E} cannot be split, because it has only one state. The other group {A, B, C, D} can be split, so we must consider the effect of each input symbol.
- On input a, each of these states goes to state B, so there is no way to distinguish these states using strings that begin with a. On input b, states A, B, and C go to members of group {A, B, C ,D}, while state D goes to E, a member of another group . Thus, group {A, B, C, D} is split into {A, B, C} {D}, and we get {A, B, C} {D} {E}.

Partition Algorithm

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

- In the next round, we can split {A, B, C} into {A,C} {B} , since A and C each go to a member of {A,B,C} on input b, while B goes to a member of another group, {D} . Thus, after the second round, {A, C} {B} {D} {E}.
- For the third round, we cannot split the one remaining group with more than one state, since A and C each go to the same state (and therefore to the same group) on each input. We conclude that final states= {A, C} { B} {D} {E} .

Partition Algorithm

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

NFA States	DFA State	a	b
{0,1,2,4,7}	A	B	~~C~~ **A**
{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~~ **A**