Recognition of Tokens

Introduction

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

Example: assume the following **pattern** to generate **if-stmt** in a specific language:

stmt ⟶ **if (*expr*) *stmt* | if (*expr*) *stmt* else *stmt* |** ∈

expr ⟶ ***term* relop *term* | *term***

term ⟶ **id | num**

Where the terminals ***if***, ***then***, ***else***, ***relop***, ***id***, and ***num*** generate sets of strings given by the following regular definitions:

if ⟶ **if**

else ⟶ **else**

relop ⟶ <| <=| ==| !=| >| >=

id ⟶ **letter(letter|digit)***

num ⟶ **digits optional-fraction optional-exponent**

Where ***letter***([A,Z],[a,z]) and ***digits*** ([0,9]) are as defined previously. For this language fragment the lexical analyzer will recognize the keywords ***if, then, else***, as well as the lexemes denoted by ***relop, id, and num***. To simplify matters, we assume ***keywords are reserved***; that is, they cannot be used as identifiers. The **num** represents the unsigned integer and real numbers of C program.

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws → (blank/tab/newline)$^{+}$

Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space.

Transition Diagrams (TD)

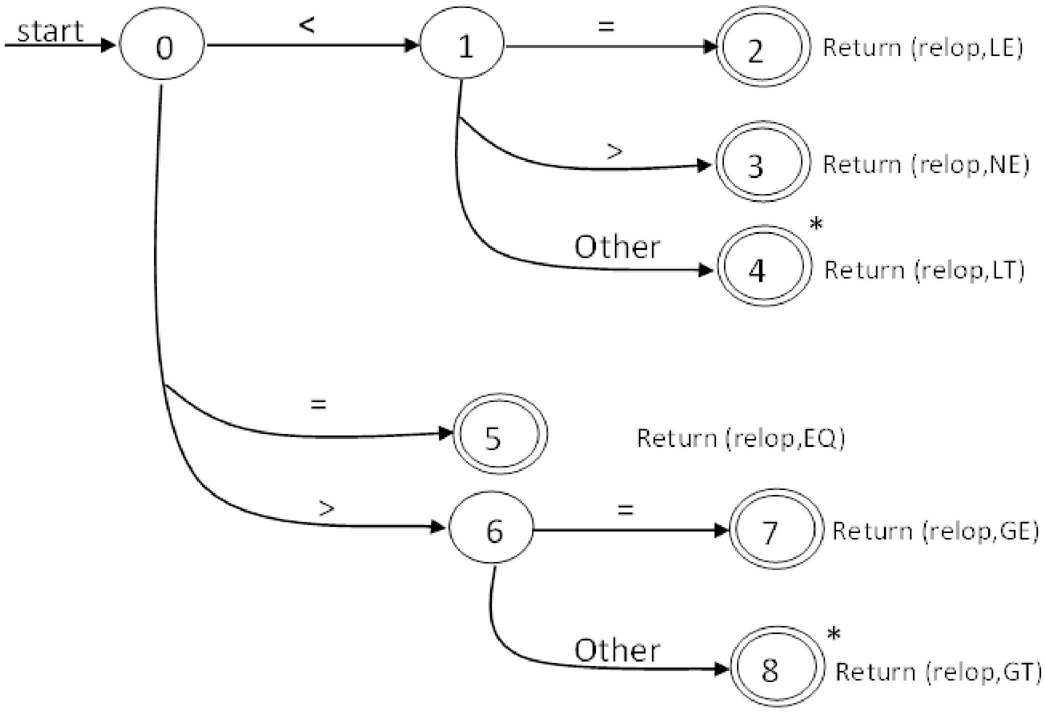
As an intermediate step in the construction of a lexical analyzer, we first produce flowchart, called a ***Transition diagram***. Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token.

The **TD** uses to keep track of information about characters that are seen as the forward pointer scans the input. It dose that by moving from position to position in the diagram as characters are read.

Components of Transition Diagram

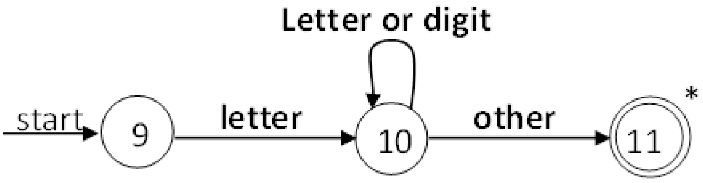
1. One state is labeled the **Start State**; start→○ it is the initial state of the transition diagram where control resides when we begin to recognize a token.
2. **Positions** in a transition diagram are drawn as circles ○ and are called states.
3. The states are connected by **Arrows**, called edges. Labels on edges are indicating the input characters. ⟶
4. The **Accepting** states in which the tokens has been found. ◎
5. **Retract** one character use ***** to indicate states on which this input retraction.

Example: A Transition Diagram for the token relation operators in Pascal language "**relop**" {<=, <, >, >=, =, <>} is shown in Figure:



Transition Diagram for relation operators in **Pascal**.

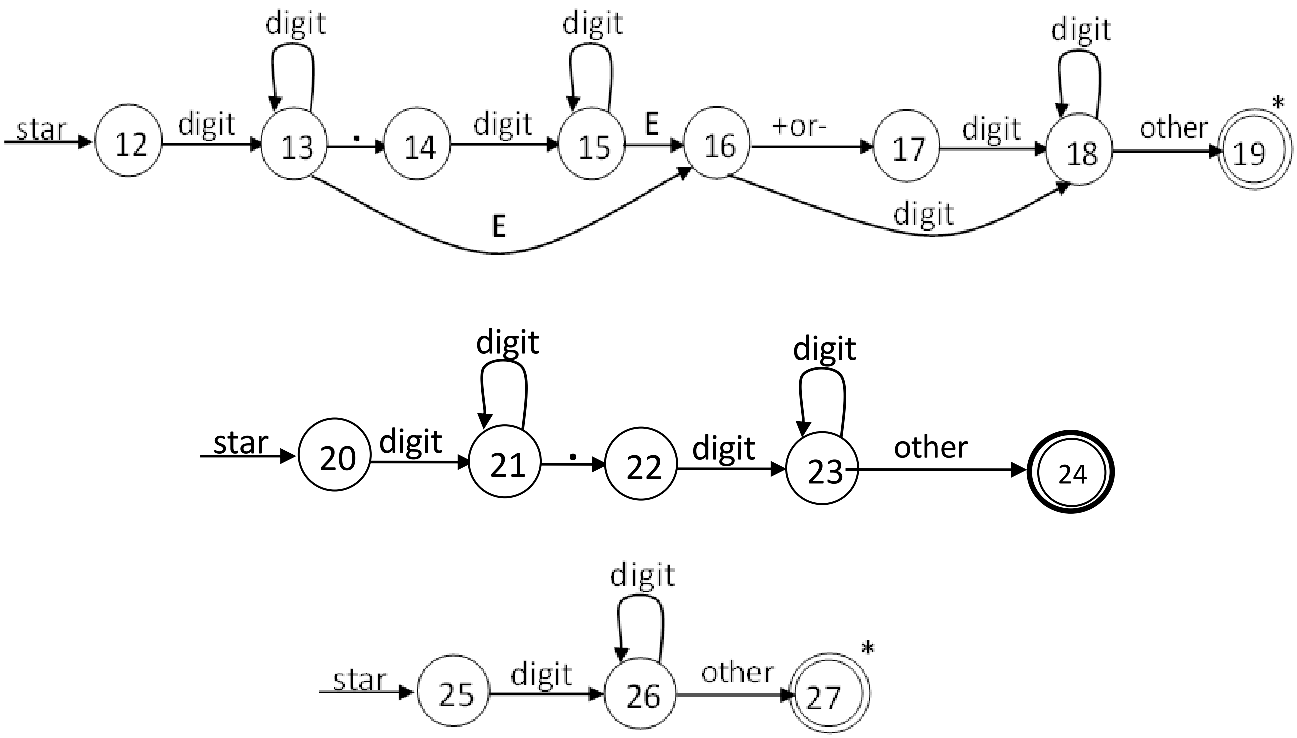
Example: A Transition Diagram for the **identifiers** and **keywords**:



Transition Diagram for identifiers and keywords

The above TD for an identifier, defined to be a letter followed by any no of letters or digits.

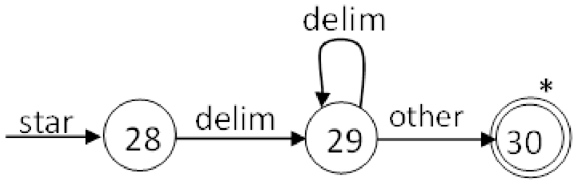
<u>Example:</u> A Transition Diagram for **Unsigned Numbers** in Pascal:



$$\text{num} \rightarrow \text{digit}^{+}(.\ \text{digit}^{+} \mid \in)(E(+\mid-\mid\in)\text{digit}^{+} \mid \in)$$

Transition Diagram for unsigned numbers in Pascal

Treatment of White Space (WS):



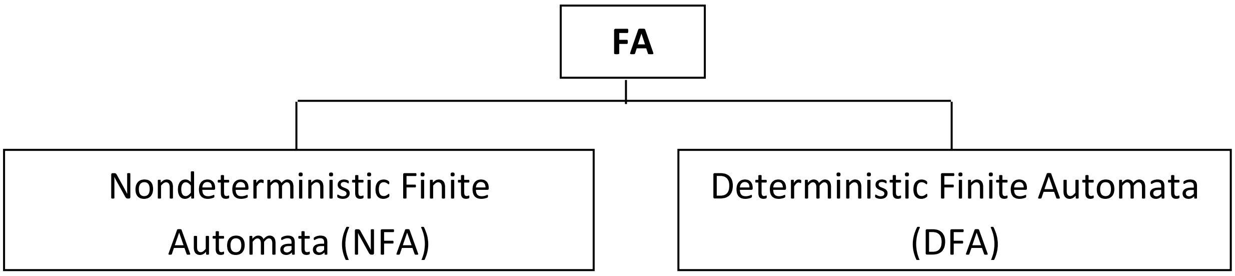
Transition Diagram for White Space

Nothing is returned when the accepting state is reached; we merely go back to the start state of the first transition diagram to look for another pattern.

Finite Automata (FA)

It a generalized transition diagram TD, constructed to compile a regular expression RE into recognizer.

Recognizer for a Language: is a program that takes a string **X** as an input and answers "Yes" if **X** is a sentence of the language and "No" otherwise.



FA

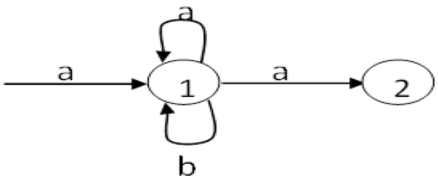
Nondeterministic Finite Automata (NFA)

Deterministic Finite Automata (DFA)

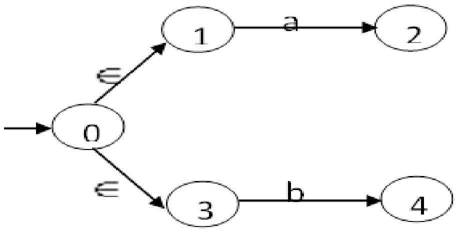
Note: Both **NFA** and **DFA** are capable of recognizing what regular expression can denote.

Nondeterministic Finite Automata (NFA)

NFA: means that more than one transition out of a state may be possible on a same input symbol.



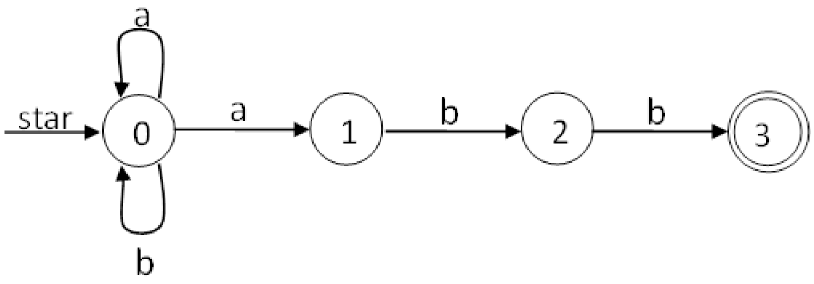
Also a transition on input ∈ (∈-Transition) is possible.



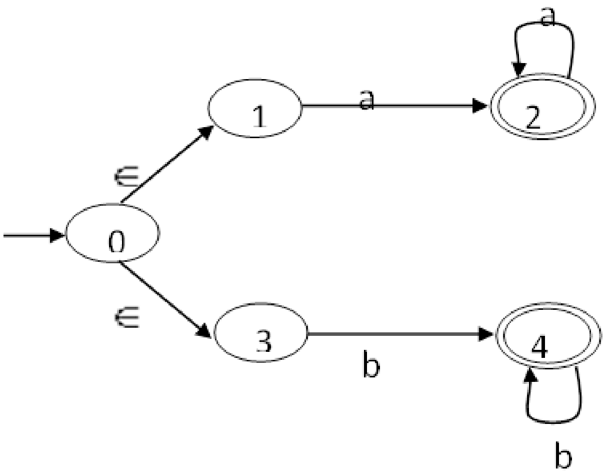
A nondeterministic finite automation **NFA** is a mathematical model consists of

1) A set of states' **S**;
2) A set of input symbol, ∑, called the input symbols alphabet.
3) A set of transition to move the symbol to the sets of states.
4) A state $\mathbf{S_0}$ called the initial or the **start** state.
5) A set of states **F** called the accepting or **final** state.

Example: The NFA that recognizes the language **(a | b)*abb** is shown below:



Example: The NFA that recognizes the language **aa*|bb*** is shown below



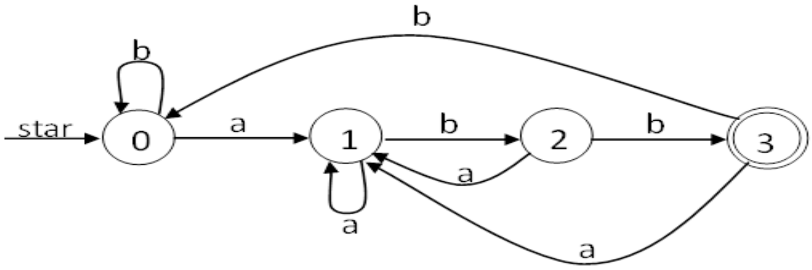
Deterministic Finite Automata (DFA)

A deterministic finite automation (DFA, for short) is a special case of a non-deterministic finite automation (NFA) in which

1. No state has an ∈-transition, i.e., a transition on input ∈, and
2. For each state ***S*** and input symbol ***a***, there is at most one edge labeled ***a*** leaving ***S***.

A deterministic finite automation DFA has at most one transition from each state on any input.

Example: The following figure shows a DFA that recognizes the language **(a|b)*abb**.



A DFA is represented by a ***transition table T***, The Transition Table is:

State	a	b
0	1	0
1	1	2
2	1	3
3	1	0

DFAs have the same expressive power as NFAs: A DFA is a special case of NFA and any NFA can be converted to an equivalent DFA. However, this comes at a cost: The resulting DFA can be exponentially larger than the NFA.

Conversion of an NFA into a DFA

It is hard for a computer program to simulate an NFA because the transition function is multivalued. The algorithm that called the **subset construction** will convert an NFA for any language into a DFA that recognizes the same languages.

Algorithm: (Subset construction): constructing DFA from NFA.

Input: ***NFA*** *N.*

Output: ***DFA*** *D* accepting the same language.

Method: this algorithm constructs a transition table ***Dtran*** for ***D***. Each **DFA** state is a set of **NFA** states and we construct ***Dtran*** so that ***D*** will simulate "**in parallel**" all possible moves ***N*** can make on a given input string.

It use the operations in below to keep track of sets of **NFA** states (***s*** represents an NFA state and ***T*** a set of NFA states).

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

1) ε -closure (s_0) is the start state of ***D***.

2) A state of ***D*** is accepting if it contains at least one accepting state in ***N***.

Algorithm: (Subset construction):

```
Initially, $\varepsilon$-closure($s_0$) is the only state in Dstates and it is unmarked;
 while there is an unmarked state T in Dstates do begin
         mark T;
         For each input symbol a do begin
              U: = ($\varepsilon$-closure (move (T, a)) ;
              if U is not in Dstates then
                   add U as an unmarked state to Dstates;
              Dtran [T, a]: = U
        End
End
```

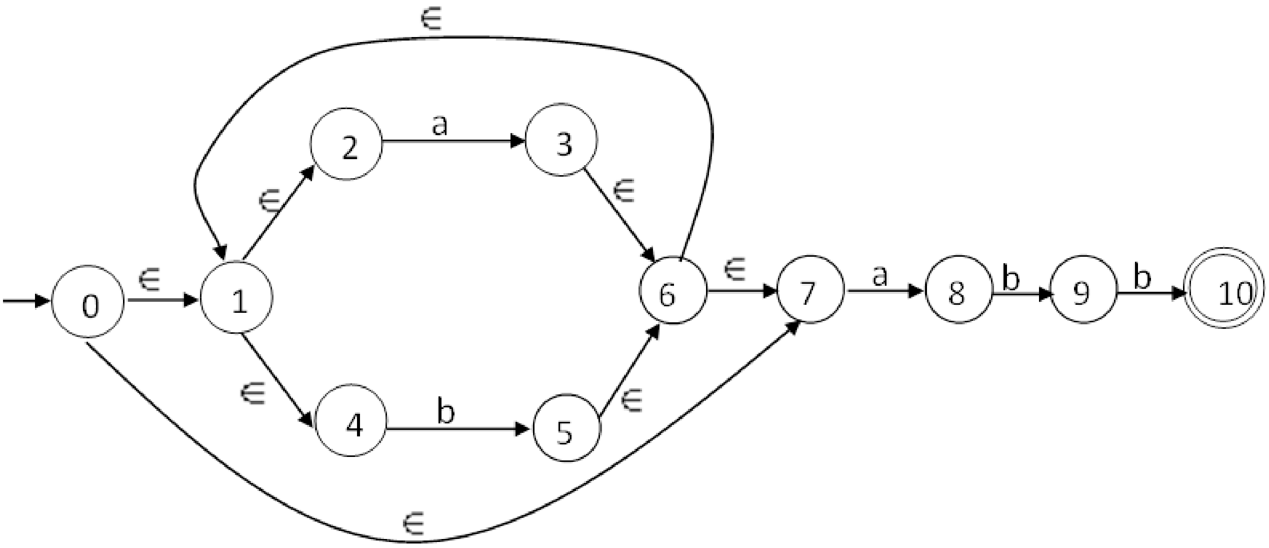
We construct ***Dstates***, the set of states of ***D***, and ***Dtran***, the transition table for ***D***, in the following manner. Each state of ***D*** corresponds to a set of **NFA** states that ***N*** could be in after reading some sequence of input symbols including all possible ε-transitions before or after symbols are read.

Algorithm: Computation of ε-closure

```
Push all states in T onto slack;
Initialize $\varepsilon$-closure (T) to T;
While slack is not empty do begin
         Pop t, the top clement, of the stack;
         For each state u with an edge from t to u labeled $\varepsilon$ do
             If u is not in $\varepsilon$-closure (T) do begin
                 Add u to $\varepsilon$-closure (T);
                 Push u onto stack
             End
End
```

A simple algorithm to compute ε-closure (***T***) uses a stack to hold states whose edges have not been checked for ε-labeled transitions.

<u>Example:</u> The figure below shows **NFA *N*** accepting the language **(a | b)*abb**.



Sol: apply the Algorithm of Subset construction as follow:

1) Find the **start state** of the equivalent **DFA** is ε-closure (**0**), which is consist of **start state** of **NFA** and the **<u>all states</u>** reachable from **state 0** via a path in which every edge is labeled ε .

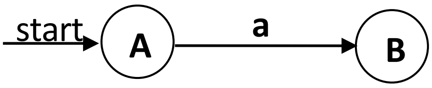
   A= {0, 1, 2, 4, 7**}**

2) Compute move (A, a), the set of states of NFA having transitions on **a** from members of A. Among the states 0, 1, 2, 4 and 7, only 2 and 7 have such transitions, to 3 and 8, so

   move (A, *a*)={3, 8}

Compute the ε **-closure (move (A, *a*))** = ε **-closure ({3, 8})**,

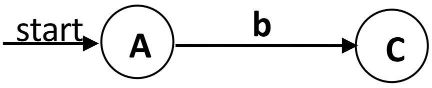
ε **-closure ({3, 8})** = **{1, 2, 3, 4, 6, 7, 8}** Let us call this set **B**.



3) Compute **move (A, *b*)**, the set of states of **NFA** having transitions on ***b*** from members of **A**. Among the states **0**, **1**, **2**, **4** and **7**, only **4** have such transitions, to **5** so: **move (A, *b*)={5}**

Compute the ε **-closure (move (A, *b*))** = ε **-closure ({5})**,

ε **-closure ({5})** = **{1, 2, 4, 5, 6, 7}** Let us call this set **C**. So the **DFA** has a transition on ***b*** from **A** to **C.**



4) We apply the **steps** 2 and 3 on the B and C, this process continues for every new state of the **DFA** until all sets that are states of the **DFA** are marked.

The five different sets of states we actually construct are:

A = {0, 1, 2, 4, 7}

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

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

D = {1, 2, 4, 5, 6, 7, 9}

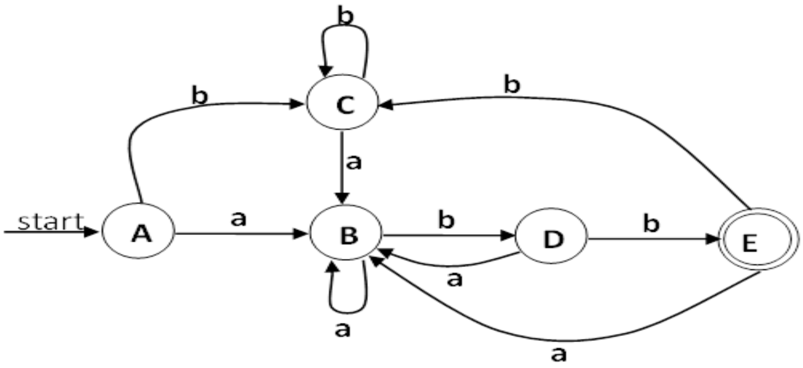
E = {1, 2, 4, 5, 6, 7, 10}

State **A** is the **start state**, and state **E** is the only **accepting state**. The complete **transition table *Dtran*** is shown in below:

STATE	**INPUT SYMBOL**	
	a	***b***
A	***B***	***C***
B	***B***	***D***
C	***B***	***C***
D	***B***	***E***
E	***B***	***C***

Transition table Dtran for DFA

Also, a transition graph for the resulting **DFA** is shown in below. It should be noted that the **DFA** also accepts **(a | b)*abb**.



From a Regular Expression to an NFA

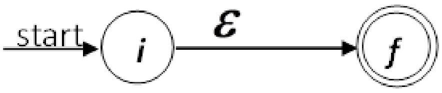
Now give an algorithm to construct an NFA from a regular expression. The algorithm is syntax-directed in that it uses the syntactic structure of the regular expression to guide the construction process.

Algorithm: (Thompson's construction):

Input: a regular expression ***R*** over alphabet ∑.

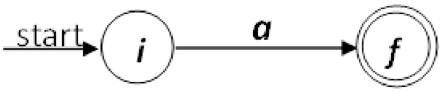
Output: ***NFA*** *N* accepting ***L(R)***.

1- For ε **,** construct the ***NFA***



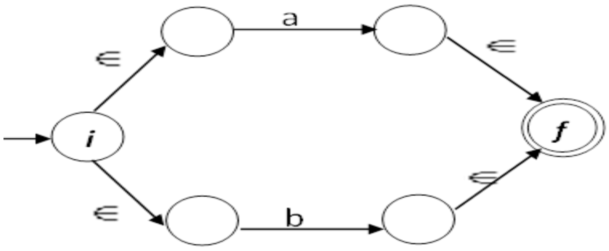
Here ***i*** is a start state and ***f*** a accepting state. Clearly this ***NFA*** recognizes {ε }.

2- For ***a*** in ∑, construct the ***NFA***

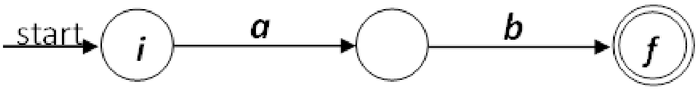


Again ***i*** is a start state and ***f*** a accepting state. This machine recognizes {***a***}.

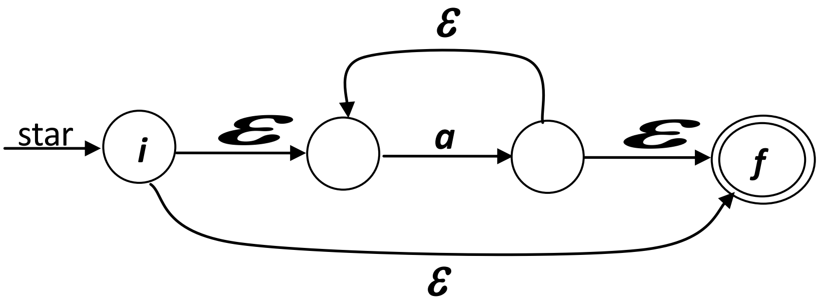
3- For the regular expression a | b construct the following composite NFA N(a | b).



4- For the regular expression ***ab*** construct the following composite NFA ***N(ab)***.

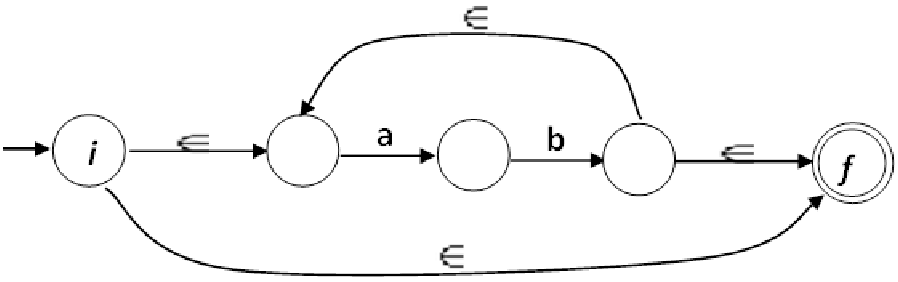


5- For the regular expression ***a**** construct the following composite NFA ***N(a*)***.

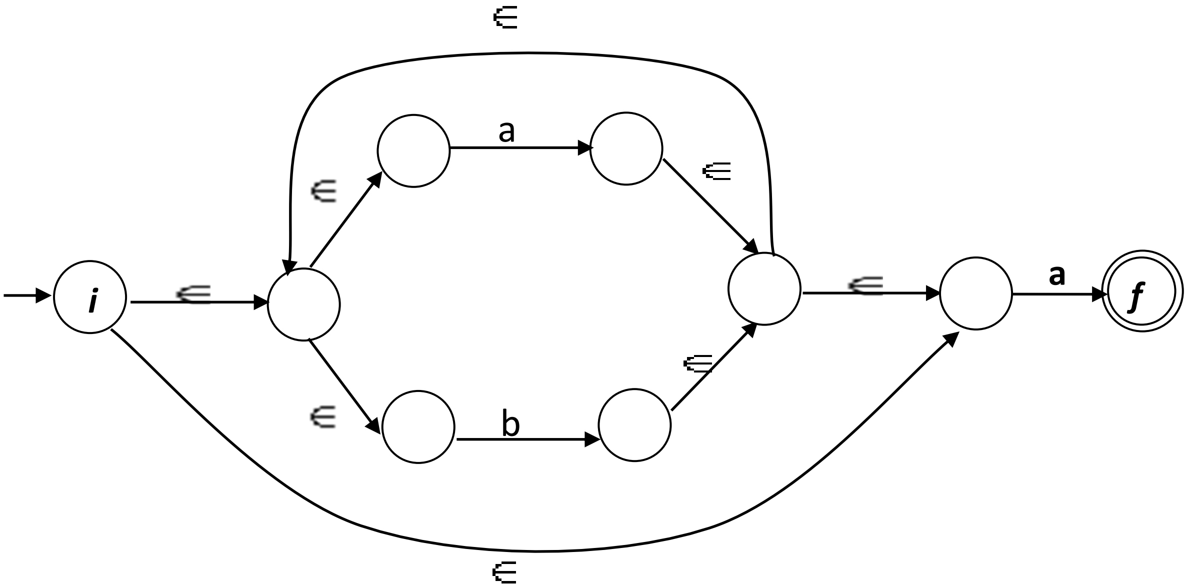


<u>Example:</u> let us use algorithm **Thompson's** construction to construct the following regular expressions:

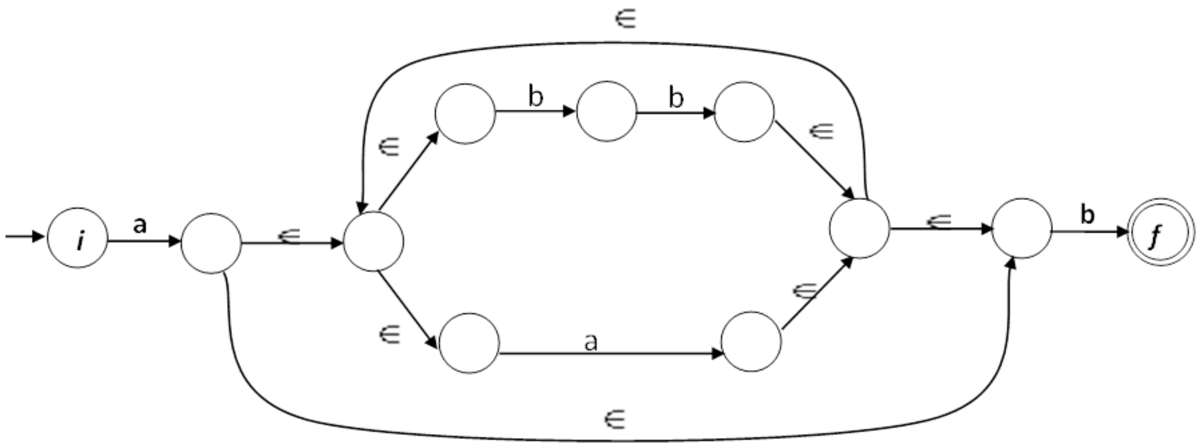
1) RE = (***ab***)*



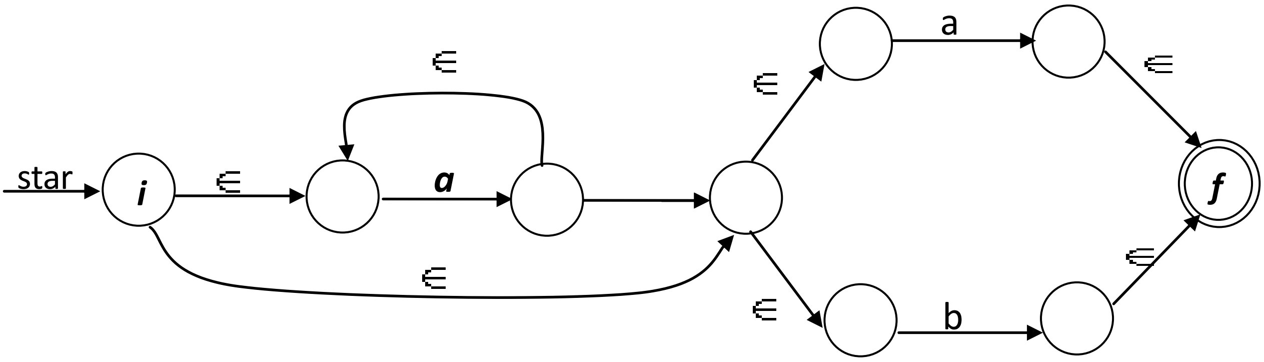
2) RE= (***a*** | ***b***)****a***



3) RE= ***a*** (***bb***| ***a***)****b***



4) RE= ***a**** (***a***



Lexical Errors

What if user omits the space in "Fori"?

No lexical error, single token IDENT ("Fori") is produced instead of sequence For, IDENT ("i").

Typically few lexical error types

1. Delete one character from the input (Keyword Token).

2. Insert an illegal character into the input.

3. Replace a character by another character (Keyword Token).

4. Transpose two adjacent characters (Keyword Token).

5. Misspelling of the keyword.

How is a Scanner Programmed?

1) Describe tokens with regular expressions.

2) Draw transition diagrams.

3) Code the diagram as table/program.