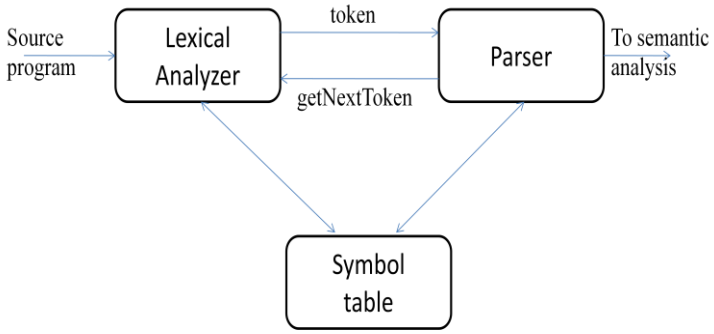
Unit-II lexical analysis

Need and Role of Lexical Analyzer-Lexical Errors-Expressing Tokens by Regular Expressions-Converting Regular Expression to DFA-Minimization of DFA-Language for Specifying Lexical Analyzers-LEX-Design of Lexical Analyzer for a sample Language.

Lexical Analyser

The first phase of compiler is Lexical Analysis. This is also known as linear analysis in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.



Up on receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Its secondary tasks are,

- One task is stripping out from the source program comments and white space is in the form of blank, tab, new line characters.
- Another task is correlating error messages from the compiler with the source program.

Sometimes lexical analyzer is divided in to cascade of two phases.

1) Scanning
2) lexical analysis.

The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

Issues in a lexical analyzer

There are several reason for separating the analysis phase of compiling into lexical analysis and parsing.

1.Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.

2. Compiler efficiency is improved.

3. Compiler portability is enhanced.

Role of lexical analysis phase

The lexical analyzer breaks a sentence into a sequence of words or tokens and ignores white spaces and comments. It generates a stream of tokens from the input. This is modeled through regular expressions and the structure is recognized through finite state automata.

Need for separating the analysis phase into lexical analysis and parsing?

i)Separation of lexical analysis from syntax allows the simplified design.

ii)Efficiency of complier gets increased. Reading of input source file is a time consuming process and if it is been done once in lexical analyzer then efficiency get increased . Lexical analyzer uses buffering techniques to improve the performances.
iii)Input alphabet peculiarities and other device specific anomalies can be restricted to the lexical analyzer.

Possible error recovery actions in a lexical analyzer:

Panic mode

Delete successive characters from the remaining input until the lexical analyzer can find a Well-formed token. This technique may confuse the parser. When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Other possible error recovery actions:

- Deleting an extraneous characters
- Inserting missing characters.
- Replacing an incorrect character by a correct character.
- Transposing two adjacent characters

Lexeme : Lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Regular set : A language denoted by a regular expression is said to be a **regular set**.

Tokens: A token is a syntactic category. Sentences consist of a string of tokens. For example number, identifier, keyword, string etc are tokens

Lexemes: Lexeme: Sequence of characters in a token is a lexeme. For example 100.01, counter, const, "How are you?" etc are lexemes.

Patterns: Rule of description is a pattern. For example letter (letter | digit)* is a pattern to symbolize a set of strings which consist of a letter followed by a letter or digit.

Input buffering: Lexical analyzer scan the sources program line by line. For storing the input string, which is to be read, the lexical analyzer makes use of input buffer. The lexical analyzer

maintains two pointer forward and backward pointer for scanning the input string. There are two types of input buffering schemes- one buffer scheme and two buffer scheme.

- Some efficiency issues concerned with the buffering of input.
- A two-buffer input scheme that is useful when lookahead on the input is necessary to identify tokens.
- Techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.
- There are three general approaches to the implementation of a lexical analyser:
  1. Use a lexical-analyser generator, such as Lex compiler to produce the lexical analyser from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.
  2. Write the lexical analyser in a conventional systems-programming language, using I/O facilities of that language to read the input.
  3. Write the lexical analyser in assembly language and explicitly manage the reading of input.

***Buffer pairs*:**

- Because of a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character. The scheme to be discussed:
- Consists a buffer divided into two N-character halves.

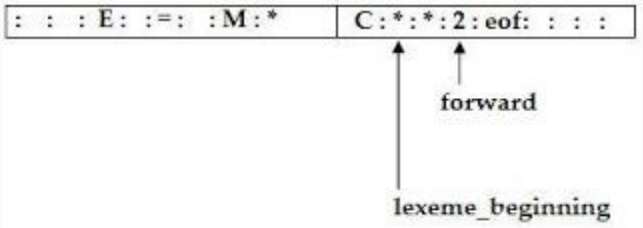


Fig 1.8 An input buffer in two halves

- N – Number of characters on one disk block, e.g., 1024 or 4096.
  - Read N characters into each half of the buffer with one system read command.
  - If fewer than N characters remain in the input, then eof is read into the buffer after the input characters.
  - Two pointers to the input buffer are maintained.
  - The string of characters between two pointers is the current lexeme.
  - Initially both pointers point to the first character of the next lexeme to be found.
  - Forward pointer, scans ahead until a match for a pattern is found.
  - Once the next lexeme is determined, the forward pointer is set to the character at its right end.
  - If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters.
  - If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

– ***Disadvantage of this scheme*:**

  - This scheme works well most of the time, but the amount of lookahead is limited.

- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.
- For example: DECLARE (ARG1, ARG2, … , ARGn) in PL/1 program;
- Cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

***Sentinels*:**

- In the previous scheme, must check each time the move forward pointer that have not moved off one half of the buffer. If it is done, then must reload the other half.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.
- This can reduce the two tests to one if it is extend each buffer half to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program. (eof character is used as sentinel).

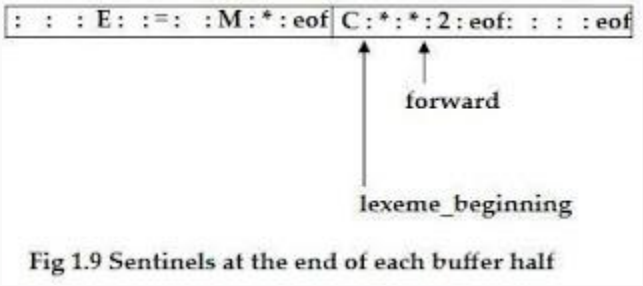


Fig 1.9 Sentinels at the end of each buffer half

- In this, most of the time it performs only one test to see whether forward points to an eof.
- Only when it reach the end of the buffer half or eof, it performs more tests.
- Since N input characters are encountered between eof's, the average number of tests per input character is very close to 1.

Drawbacks of using buffer pairs:

i. This buffering scheme works quite well most of the time but with it amount of look ahead is limited.
ii. Limited look ahead makes it impossible to recognize tokens in situations Where the distance, forward pointer must travel is more than the length of buffer.

Algebraic properties of regular expressions: The algebraic law obeyed by regular expressions are called algebraic properties of regular expression. The algebraic properties are used to check equivalence of two regular expressions.

S.No	Properties	Meaning
1	r1|r2=r2|r1	| is commutative
2	r1|(r1|r3)=(r1|r2)|r3	| is associative
3	(r1 r2)r3=r1(r2 r3)	Concatenation is associative

4	r1(r2|r3)=r1 r2|r1 r3 (r2|r3) r1=r2 r1|r3 r1	Concatenation is distributive over |
5	Є r = r є = r	Є is identity
6	r*=(r|є)*	Relation between є and *
7	r**=r*	*is idempotent

Operations on languages.

- **Union -** L U M ={s | s is in L or s is in M}
- **Concatenation** – LM ={st | s is in L and t is in M}
- **Kleene Closure –** L* (zero or more concatenations of L)
- **Positive Closure –** L+ (one or more concatenations of L)

Specification of Tokens

An alphabet or a character class is a finite set of symbols. Typical examples of symbols are letters and characters.

The set {0, 1} is the binary alphabet. ASCII and EBCDIC are two examples of computer alphabets.

Strings

A string over some alphabet is a finite sequence of symbol taken from that alphabet.

For example, banana is a sequence of six symbols (i.e., string of length six) taken from ASCII computer alphabet. The empty string denoted by $\in$, is a special string with zero symbols (i.e., string length is 0).

If x and y are two strings, then the concatenation of x and y, written xy, is the string formed by appending y to x.

For example, If x = dog and y = house, then xy = doghouse. For empty string, $\in$, we have S$\in$ = $\in$S = S.

String exponentiation concatenates a string with itself a given number of times:

S^2 = SS or S.S

S^3 = SSS or S.S.S

S^4 = SSSS or S.S.S.S and so on

By definition S^0 is an empty string, $\in$, and S` = S. For example, if x =ba and na then xy^2 = banana.

Languages

A language is a set of strings over some fixed alphabet. The language may contain a finite or an infinite number of strings.

Let L and M be two languages where L = {dog, ba, na} and M = {house, ba} then

- Union: LUM = {dog, ba, na, house}
- Concatenation: LM = {doghouse, dogba, bahouse, baba, nahouse, naba}
- Expontentiation: L2 = LL
- By definition: L^0 ={ $\in$} and L` = L

The kleene closure of language L, denoted by L*, is "zero or more Concatenation of" L.

$L^* = L^0 \cup L` \cup L^2 \cup L^3 \ldots \cup L^n \ldots$

For example, If L = {a, b}, then

L* = { $\in$, a, b, aa, ab, ab, ba, bb, aaa, aba, baa, . . . }

The positive closure of Language L, denoted by L^+, is "one or more Concatenation of" L.

$L^+ = L` \cup L^2 \cup L^3 \ldots \cup L^n \ldots$

For example, If L = {a, b}, then
L^+ = {a, b, aa, ba, bb, aaa, aba, . . . }

A regular expression is built up out of simpler regular expressions using a set of defining rules.

- Each regular expression r denotes a language L(r).
  - The rules that define the regular expressions over alphabet ∑.
- (Associated with each rule is a specification of the language denoted by the regular expression being defined)
  - 1. *ε* is a regular expression that denotes {ε}, i.e. the set containing the empty string.
  - 2. If *a* is a symbol in ∑, then *a* is a regular expression that denotes {a}, i.e. the set containing the string *a*.
  - 3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then
- a) (r) | (s) is a regular expression denoting the languages L(r) U L(s).
- b) (r)(s) is a regular expression denoting the languages L(r)L(s).
- c) (r)* is a regular expression denoting the languages (L(r))*.
- d) (r) is a regular expression denoting the languages L(r).
  - A language denoted by a regular expression is said to be a regular set.
  - The specification of a regular expression is an example of a recursive definition.
  - *Rule (1) and (2)* form the basis of the definition.
  - *Rule (3)* provides the inductive step.

Regular expressions: Regular expression is used to define precisely the statements and expressions in the source language. For e.g. in Pascal the identifiers is denotes in the form of regular expression as letter **letter(letter|digit)*.**

Recognizer: It is a part of LEX analyzer that identifies the presence of a token on the input is a recognizer for the language defining that token. It is the program, which automatically recognizes the tokens. A recognizer for a language L is a program that takes an input string "x" and response "yes" if "x" is sentence of L and "no" otherwise.

Rules for constructing regular expressions:

The rules are divided into two major classifications(i) Basic rules (ii) Induction rules

Basic rules:

i) **ε** is a regular expression that denotes {**ε**} (i.e.) the language contains only an empty string.

ii) For each a in ∑, then a is a regular expression denoting {a} , the language with only one string , which consists of a single symbol a.

Induction rules:

i) If R and S are regular expressions denoting languages LR and LS respectively then,

ii)(R)/(S) is a regular expression denoting LR U LS

iii)(R).(S) is a regular expression denoting LR . LS

iv)(R)* is a regular expression denoting LR*.

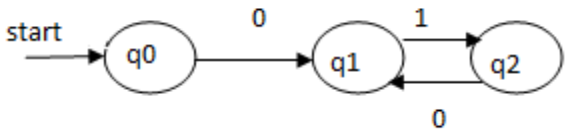
To Recognize the tokens

Finite Automata

- A 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.
- One can compile any regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton.
- A finite automation can be deterministic means that more than one transition out of a state may be possible on a same input symbol.
- Both automata are capable of recognizing what regular expression can denote.

A finite automate is a mathematical model that consists of-
i)Input set Σ
ii)A finite set of states S
iii)A transition function to move from one state to other.
iv)A special state called start state.
v)A special state called accept state.
The finite automate can be diagrammatically represented by a directed graph called transition graph.

-

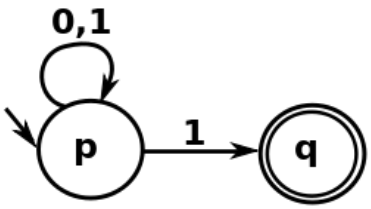


Nondeterministic Finite Automata (NFA)

A nondeterministic finite automaton 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 transition function move that maps state-symbol pairs to sets of states.
4. a state so called the initial or the start state.
5. a set of states F called the accepting or final state.

- An NFA can be described by a transition graph (labeled graph) where the nodes are states and the edges shows the transition function.
- The labeled on each edge is either a symbol in the set of alphabet, ∑, or denoting empty string.
- This automation is nondeterministic because when it is in state-0 and the input symbol is a, it can either go to state-1 or stay in state-0.
- The advantage of transition table is that it provides fast access to the transitions of states and the disadvantage is that it can take up a lot of soace.
- In general, more than one sequence of moves can lead to an accepting state. If at least one such move ended up in a final state. For instance
- The language defined by an NFA is the set of input strings that particular NFA accepts.

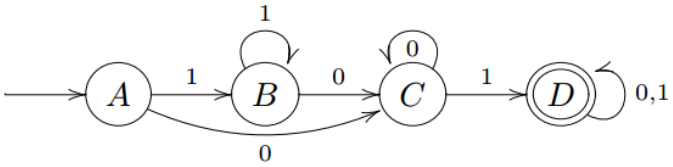


Deterministic Finite Automata (DFA)

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

1. no state has an $\in$-transition
2. for each state s and input symbol a, there is at most one edge labeled a leaving s.

- A DFA has st most one transition from each state on any input. It means that each entry on any input. It means that each entry in the transition table is a single state (as oppose to set of states in NFA).
- Because of single transition attached to each state, it is vary to determine whether a DFA accepts a given input string.



Conditions to satisfy for NFA:

(i) NFA should have one start state.
(ii) NFA may have one or more accepting states.
(iii) ε-Transitions may present on NFA.
(iv) There can be more than transitions, on same input symbol from any one state.
(v) In NFA, from any state "S"

A. There can be at most 2 outgoing e-transitions.
B.There can be only one transition on real input symbol.
C.On real input transition it should reach the new state only.

Difference between Deterministic FA and Non Deterministic FA.

S.No	NFA	DFA
1.	E-Transition is present	No E-Transition
2.	More than one transition for real variable	Only one transition for real symbol
3.	It's not easy implemented	It's easily implemented
4.	Compare to DFA not fast	Can load to Faster recognizer.

Time and space complexity of NFA and DFA.

Automation	Space	Time
NFA	O(|r|)	O(|r|*|r|)
DFA	O(2|r|)	O(|x|)

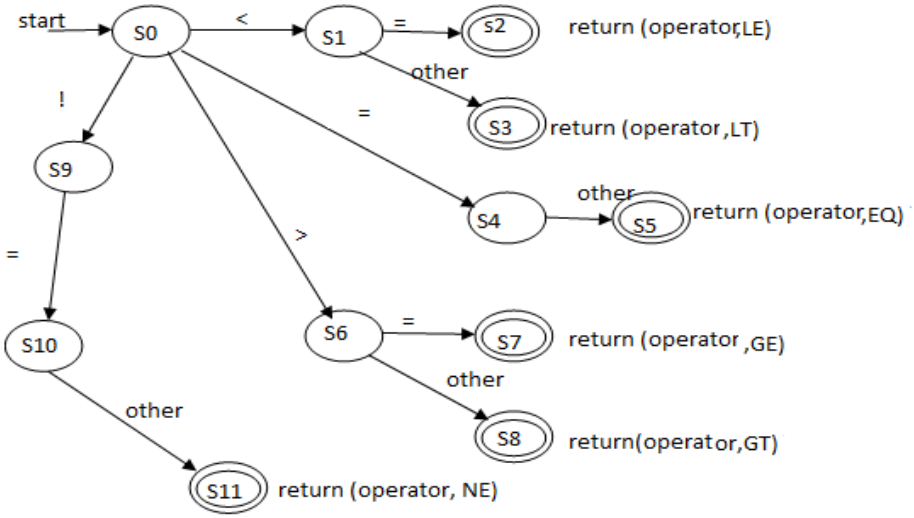
Regular definition to represent date and time in the following format:

MONTH DAY YEAR

MONTH--> [Jan-Dec]
DAY-->[1-31]
YEAR-->[1900-2025]

To recognize tokens :
Consider the following grammar and try to construct an analyzer that will return <token, attribute> pairs.
relop ⟶ < | = | = | <> | = | >
id ⟶ letter (letter | digit)*
num ⟶ digit+ ('.' digit+)? (E ('+' | '-')? digit+)?
delim ⟶ blank | tab | newline
ws ⟶ delim+

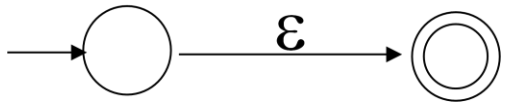
Transition diagram to represent relational operators.
The relational operators are:<,<=,>,>=,=,!=



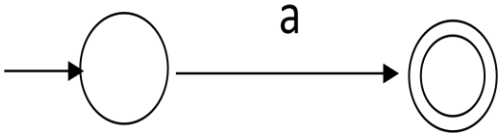
Construction of an NFA from a Regular Expression

We now give an algorithm for converting any regular expression to an NFA that defines the same language. The algorithm is syntax-directed, in the sense that it works recursively up the parse tree for the regular expression. For each subexpression the algorithm constructs an NFA with a single accepting state.
The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA.

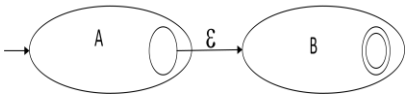
NFA for €



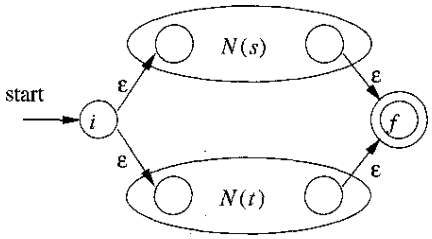
NFA for a



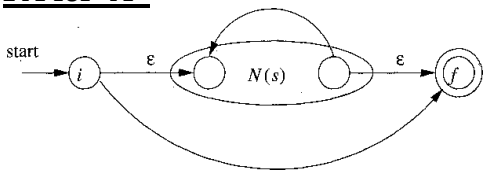
NFA for AB



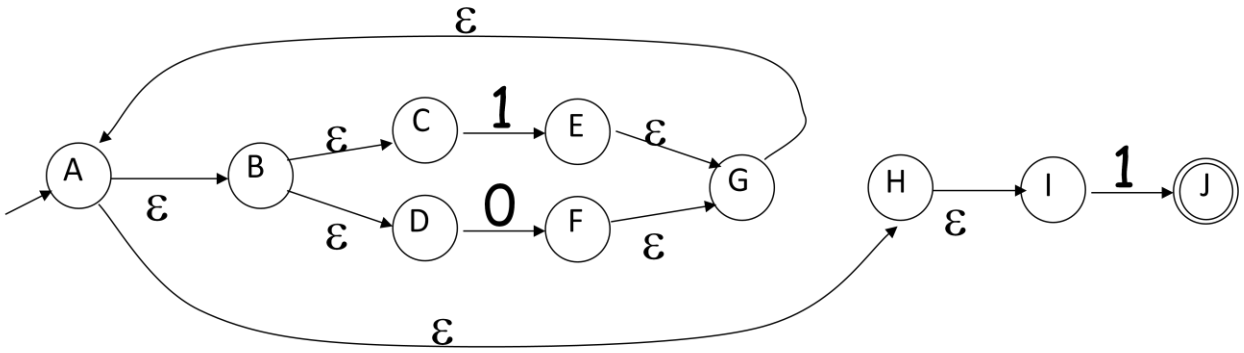
NFA for A/B



FA for A*



NFA and DFA for (1 | 0)*1



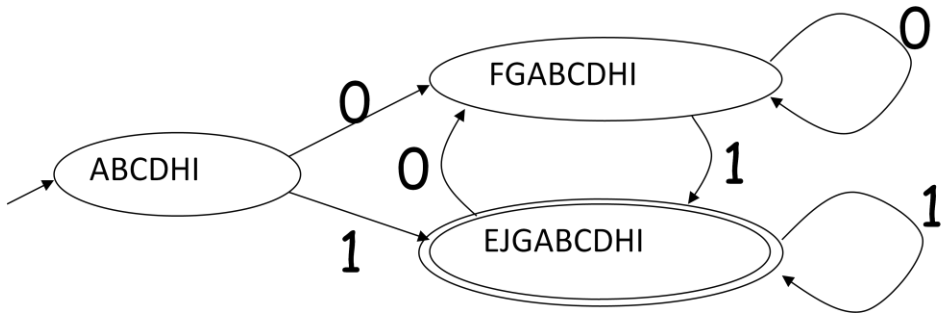
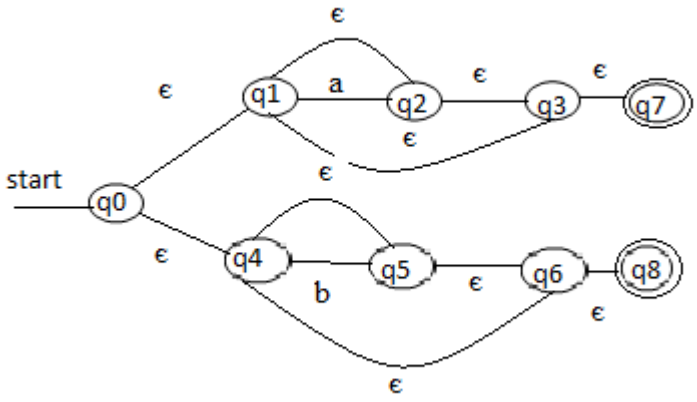


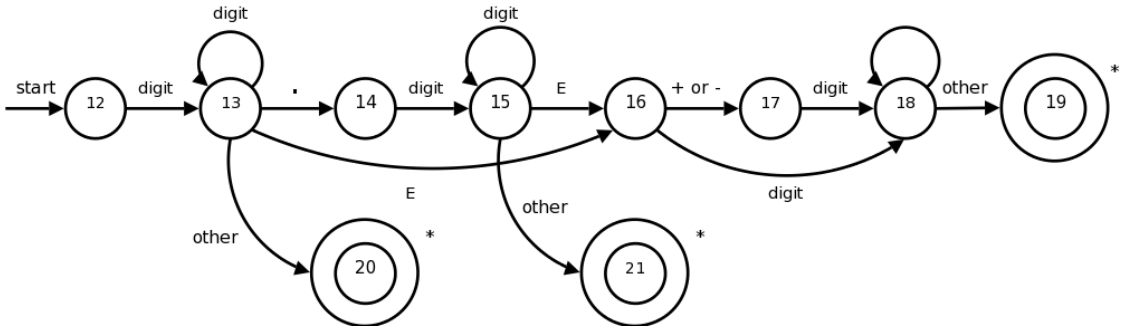
Table Implementation of a DFA

	0	1
S	**T**	**U**
T	**T**	**U**
U	**T**	**U**

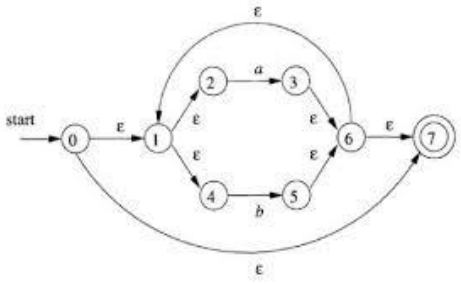
NFA for a*|b*



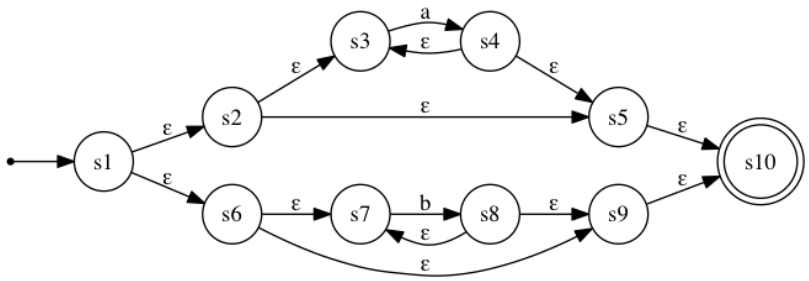
Transition diagram for unsigned numbers:



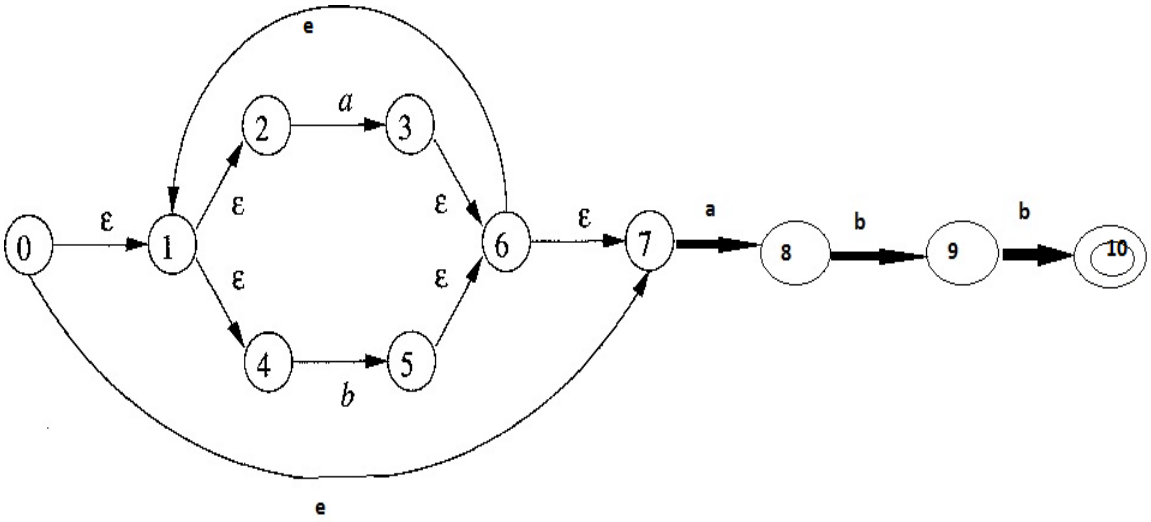
NFA for (a/b)*



NFA for (a*/b*)



NFA for (a*/b*) abb



Conversion of an NFA to a DFA

OPERATION	DESCRIPTION
e-closure(s)	Set of NFA states reachable from NFA state *s* on e-transitions alone.
e-closure(T)	Set of NFA states reachable from some NFA state *s* in set *T* on e-transitions alone; = Us in T *e-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.*

subset construction:

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

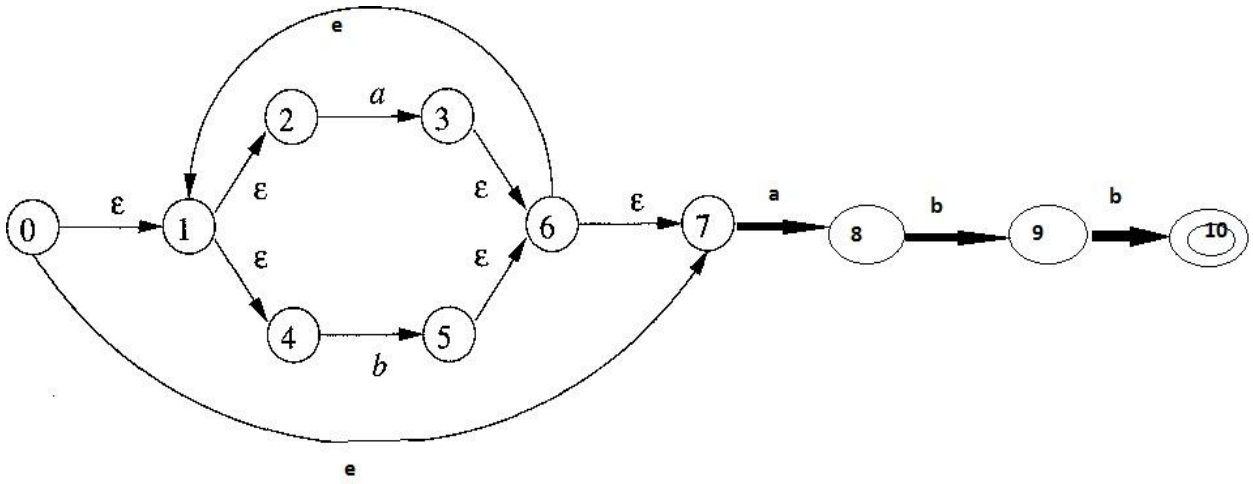
Computing *e-closure(T)*

push all states of *T* onto *stack;*

initialize *e~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 e)
if (*u* is not in *e-closure(T)) {*
add *u* to *e-closure(T);*
push *u* onto *stack;*
}
}

Construct the minimized DFA for the expression: (a/b)*abb. Construct the NFA using Thompson's rule.

NFA for **(a|b)*abb**



Procedure :

€ closure (0) = { 0,1,2,4,7} - **A**
DTrans[A,a] = {3,8} - **B**
DTrans[A,b] = {5} - **C**

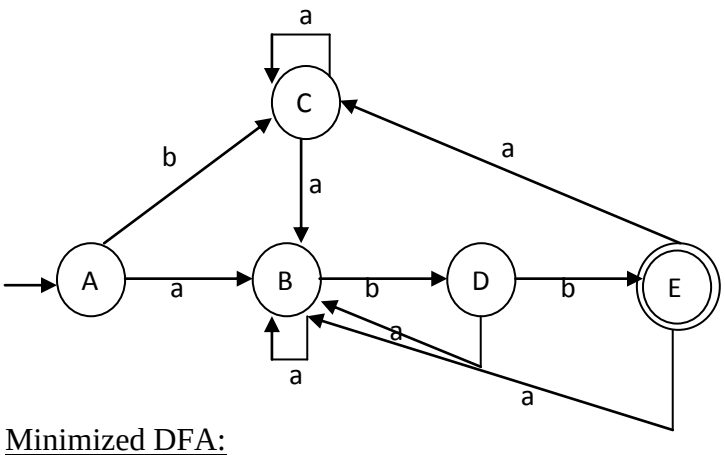
€ closure B (3,8) = { 3,6,7,1,2,4,8}
= { 1,2,3,4,6,7,8}
DTrans[B,a] = {3,8} - **B**
DTrans[B,b] = {5,9} - **D**

€ closure C (5) = { 5,6,7,1,2,4}
= { 1,2,3,4,5,6,7}
DTrans[C,a] = {3,8} - **B**
DTrans[C,b] = {5} - **C**
€ closure D (5,9) = { 5,6,7,1,2,4,9}
= { 1,2,4,5,6,7,9}
DTrans[D,a] = {3,8} - **B** -
DTrans[D,b] = {5,10} - **E**

DFA TABLE :

States	a	b
A	**B**	**C**
B	**B**	**D**
C	**B**	**C**
D	**B**	**E**
E	**B**	**C**

DFA:



Minimized DFA:

A = C = E = [B C]

RULES:

1. Middle states can be combined with starting state or end state.
2. Two or more middle states can be combined.
3. Starting state and end state should not be combined.

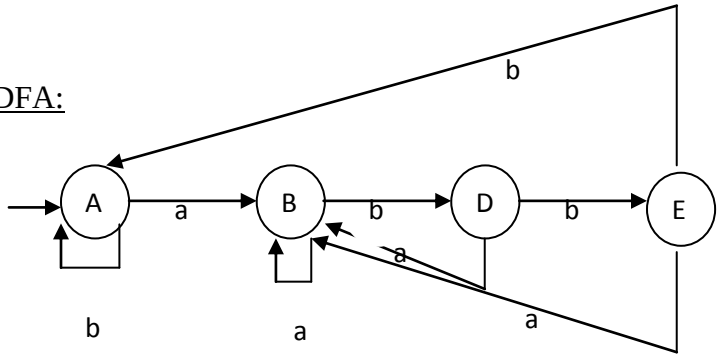
Therefore A= C or C = E.

Let us consider A = C. The resultant table is

Minimized table:

States	a	b
A	**B**	**A**
B	**B**	**D**
D	**B**	**E**
E	**B**	**A**

Minimized DFA:



Conversion of Regular Expression to DFA Directly using syntax tree

- The "*important states"* of an NFA are those without an ε-transition, that is if *move*({*s*},*a*) ≠ ∅ for some *a* then *s* is an important state
- The subset construction algorithm uses only the important states when it determines ε-*closure*(*move*(*T*,*a*))
- 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*

  nullable(*n*): the sub tree 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 sub tree at node *n*

  lastpos(*n*): the set of positions that can match the last symbol of a string generated be the sub tree at node *n*

  followpos(*i*): the set of positions that can follow position *i* in the tree.

Rules for followpos:

There are only two ways that a position of a regular expression can be made to follow another.

i). I f n i s a cat-node with left child *c* \ and right child **C 2 ,** then for every position *i* in *lastpos(ci*), all positions in *firstpos(c2)* are in *followpos(i).*

Left child's last postion follows right childs first position

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

First position follows last position

Routine for followpos()

{

initialize *Dstates* to contain only the unmarked state *firstpos(no),* where no is the root of syntax tree T for (r) #;

while (there is an unmarked state *S* in *Dstates) {*

mark *S;*

for (each input symbol *a) {*

let *U* be the union of *followpos(p)* for all *p*

in *S* that correspond to *a;*

if (*U* is not in *Dstates)*
add *U* as an unmarked state to *Dstates;*
Dtran[S, a) = U;
}
}
}

Node *n*	*nullable*(*n*)	*firstpos*(*n*)	*lastpos*(*n*)
Leaf ε	true	∅	∅
Leaf *i*	false	{*i*}	{*i*}
| / \ 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) **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)

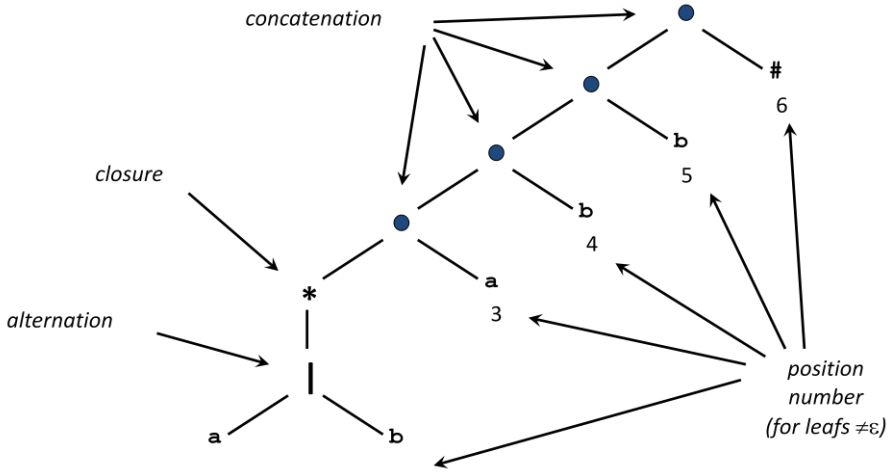
Construction of a DFA from a regular expression r.
INPUT: A regular expression r.
OUTPUT: A DFA *D* that recognizes *L(r).*
METHOD:
1. Construct a syntax tree T from the augmented regular expression (r) #.
2. Compute *nullable, firstpos, lastpos,* and *followpos* for *T,* using the methods
3. Construct *Dstates,* the set of states of DFA *D,* and *Dtran,* the transition function for *D*. The states of *D* are sets of positions in *T*. Initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state of *D* is *firstpos(no),* where node no is the root of T. The accepting states are those containing the position for the endmarker symbol #.

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

Construct the DFA Directly from regular expression using syntax tree for (a|b)*abb#

Syntax tree for (a|b)*abb#



Node	*firstpos*	*lastpos*	*followpos*
1	1	1	{1, 2, 3}
2	2	2	{1, 2, 3}
3	1,2,3	3	{4}
4	1,2,3	4	{5}
5	1,2,3	5	{6}
6	1,2,3	6	-

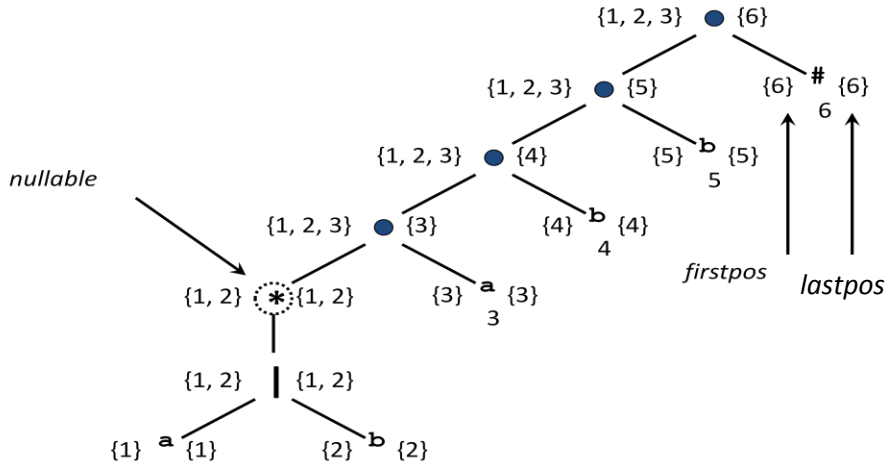
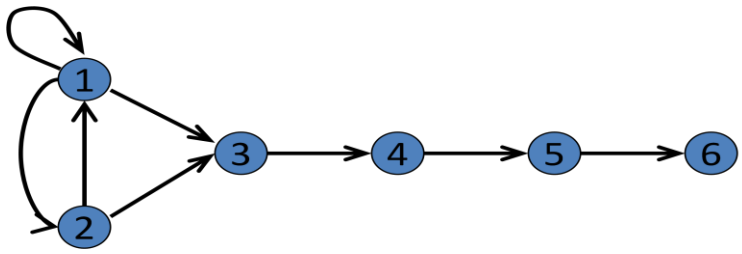


Diagram for followpos:



Procedure :

Starting state A – { 1,2,3} - **A**

DTrans[A,a] = {1,3} = followpos(1) + followpos(3)
= {1,2,3,4} - **B**

DTrans[A,b] = {2} = followpos(2)
= {1,2,3} - **A**

State B {1,2,3,4}

DTrans[B,a] = {1,3} = followpos(1) + followpos(3)
= {1,2,3,4} - **B**

DTrans[B,b] = {2,4} = followpos(2) + followpos(4)
= {1,2,3,5} - **C**

State C {1,2,3,5}

DTrans[C,a] = {1,3} = followpos(1) + followpos(3)
= {1,2,3,4} - **B**

DTrans[C,b] = {2,5} = followpos(2) + followpos(5)
= {1,2,3,6} - **D**

State D {1,2,3,6}

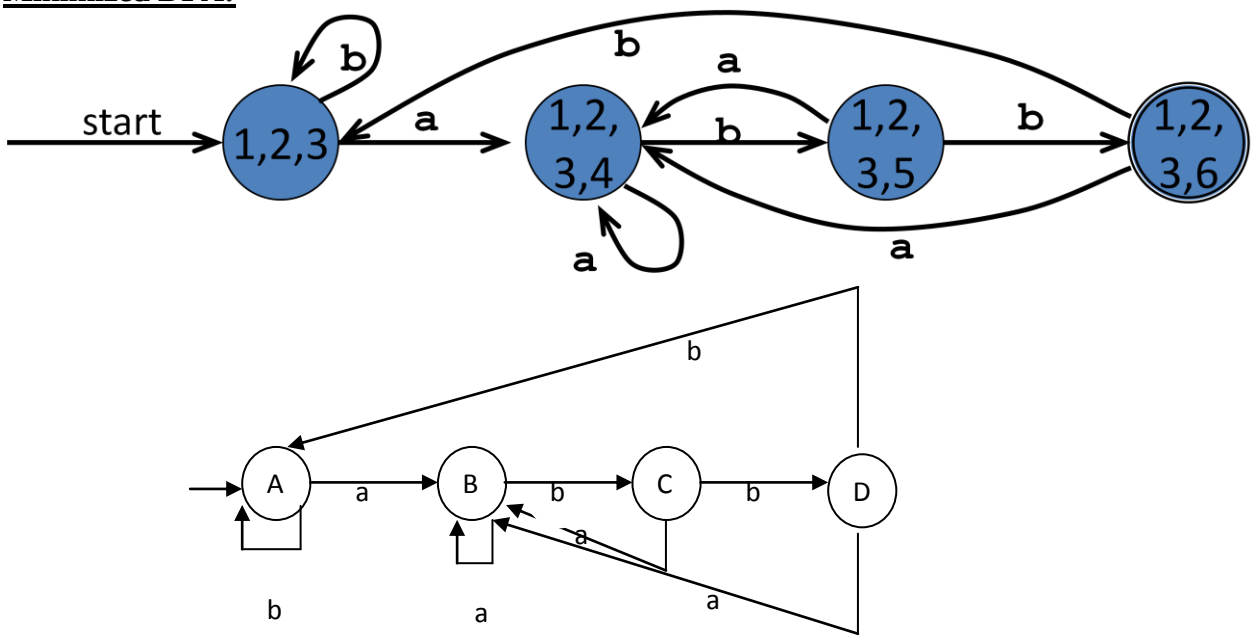
DTrans[D,a] = {1,3} = followpos(1) + followpos(3)
= {1,2,3,4} - **B**

DTrans[D,b] = {2} = followpos(2)
= {1,2,3} - **A**

DFA Table:

States	a	b
A	B	A
B	B	C
C	B	D
D	B	A

Minimized DFA:



Language for specifying lexical Analyzer.

Lex

- The main job of a lexical analyzer (scanner) is to break up an input stream into more usable elements (tokens)
  a = b + c * d;
  ID ASSIGN ID PLUS ID MULT ID SEMI
- Lex is an utility to help you rapidly generate your scanners
- Lexical analyzers tokenize input streams
- Tokens are the terminals of a language
  - English
    - words, punctuation marks, …
  - Programming language
    - Identifiers, operators, keywords, …

- Regular expressions define terminals/tokens
- Lex source is a table of
  - regular expressions and
  - corresponding program fragments

```
digit    [0-9]
letter   [a-zA-Z]
%%
{letter}({letter}|{digit})*        printf("id: %s\n", yytext);
\n                                 printf("new line\n");
%%
main() {
        yylex();
}
```

- Lex source is separated into three sections by %% delimiters
- The general format of Lex source is
- The absolute minimum Lex program is thus

```
{definitions}
%%
{transition rules}
%%
{user subroutines}
```

Lex Predefined Variables

- yytext -- a string containing the lexeme
- yyleng -- the length of the lexeme
- yyin -- the input stream pointer
  - the default input of default main() is stdin
- yyout -- the output stream pointer
  - the default output of default main() is stdout.
- E.g.

```
[a-z]+          printf("%s", yytext);
[a-z]+          ECHO;
[a-zA-Z]+       {words++; chars += yyleng;}
```

- yylex()
  - The default main() contains a call of yylex()
- yymore()
  - return the next token
- yyless(n)
  - retain the first n characters in yytext
- yywarp()
  - is called whenever Lex reaches an end-of-file
  - The default yywarp() always returns 1

Write LEX specifications and necessary C code that reads English words from a text file and response every occurrence of the sub string 'abc' with 'ABC'. The program should also compute number of characters, words and lines read. It should not consider and count any lines(s) that begin with a symbol '#'

```
%{
#include <stdio.h>
#include <stdlib.h>
int cno = 0, wno = 0, lno = 0; /*counts of characters, words and lines */
%}
character [a-z]
digit [0-9]
word ({character}|{digit})+[^({character}|{digit})]
line \n
%%
{line} { lno++; REJECT; }
{word} { wno++; REJECT; }
{character} { cno++; }
%%
void main()
{ yylex();
  fprintf(stderr, "Number of characters: %d; Number of words: %d; Number of lines: %d\n", cno,
wno, lno);
  return;
}
```