Chapter 2

Lexical Analysis

2.1 Role of the Lexical Analyzer

- Groups input characters into tokens. This is an expensive phase of the compiler, and efficiency can be important.
- Different ways to construct:
  - Scanner generator (e.g., LEX)
  - Written by hand in a high level language
  - Written by hand in assembly language (efficiency)
- The lexical analyzer is usually set up to be a function (procedure, coroutine) of the syntax analyzer (parser) and returns a token and other needed information when called
- Features of a lexical analyzer
  - Return a token to syntax analyzer
  - Strips white space (blanks, tabs, newlines)
  - Keep track of line numbers (for errors, etc.)
  - Generate output listing with errors marked, if required
  - Delete comments
  - Expands macros, if the language has them.
  - Convert numbers to internal form
- Describe the difference between *token*, *lexemes*, and *patterns*.
- If a lexeme is different from the token, then the lexical analyzer must also return the lexeme.
- Decomposition of grammar
  - Necessary to determine what the lexical analyzer will recognize vs. what the syntax analyzer will recognize
  - Delimiters, identifiers, constants are termed *basic symbols*

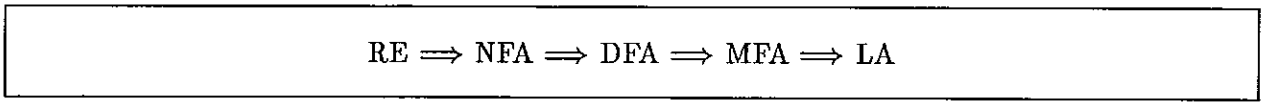


Figure 2.1: Automatic lexical analyzer generator process

  - Their structure may be arbitrarily changed without altering the power of the language
  - Structure of basic symbols can generally be described with regular expressions

- Describe the difference between reserved words and keywords. Describe the difference in implementation.

- Representation of tokens

  - Integers (LEX)
  - Enumerated type
  - Each keyword is a separate token
  - Identifier is a token – name of identifier also returned
  - Constants are tokens – value of constant is also returned

2.2 Automatic Generation of Lexical Analyzers

- The process shown in Figure 2.1 can be done automatically given a regular expression as input and the lexical analyzer as output. The output may be an actual program or just a set of tables representing the finite state machine.

2.3 Specification of Tokens

- We need a way to give a concise description of a token. *Regular expressions* give us an effective way to describe tokens.

- Definition of regular expressions over the alphabet Σ.

  1. $\emptyset$ is a regular expression that denotes the empty set.
  2. ϵ is a regular expression denoting the set that contains only the empty string.
  3. If $a \in \Sigma$, then a is a regular expression that denotes a.
  4. Suppose r and s are regular expressions denoting the languages $L(r)$ and $L(s)$. Then,
     (a) $(r)|(s)$ is a regular expression denoting $L(r) \cup L(s)$ – union.
     (b) $(r)(s)$ is a regular expression denoting $L(r)L(s)$ – concatenation.
     (c) $(r)^*$ is a regular expression denoting $(L(r))^*$ – Kleene's closure.
     (d) (r) is a regular expression denoting $L(r)$.

- A languages denoted by a regular expression is said to be a *regular set.*

- Additional notation used for convenience:

1. r^+ denotes all string consisting of *one* or more strings in r concatenated together.
2. For zero or one instances of something, the operator ? is used. Thus $r?$ denotes zero or one instances of r.
3. If k is a constant, the set A^k represents all strings formed by concatenating k strings from A. That is, $A^k = (AAA...)$ (k copies).
4. It is also possible to have character classes. Consider $[A-Za-z][A-Za-z0-9]^*$.

- Unnecessary parentheses can be avoided in regular expressions if we adopt the conventions that:

  1. The unary operator * has the highest precedence and is left associative,
  2. Concatenation has the second highest precedence and is left associative,
  3. Union has the lowest precedence and is left associative.

- Some examples of regular expressions:

  1. a^* – string of 0 or more a's.
  2. $(a|b)^*$ – all strings of a's and b's.
  3. $(\epsilon|a|b)(a|b)(a|b)(a|b)(a|b)^*$ – strings of length 3 or more consisting of a's and b's.
  4. BEGIN | END | IF | THEN | ELSE – keywords.
  5. $letter(letter|digit)^*$ – identifiers in Pascal.
  6. $(digit)(digit)^*$ – constants.
  7. $A|B|C|...|Y|Z$ – letters.
  8. 0|1|...|9 – digits

2.4 LEX

- Consider Figure 2.2 to see the process of creating a lexical analyzer using LEX.
- Consider Figure 2.3 that shows the basic format of a LEX file.
- Notes:
  - *Definition* – any combination of
    - definitions – *name space translation*
    - included code – *space code*
    - included code –
      ```
      %{
      code
      %}
      ```
  - *Rules* – any number of rules that have the form *expression* { *action* }.
    - The expression is a regular expression that describes that token to be recognized (it is essentially a pattern for the token).
    - The action is the C code to be executed when the pattern is match. If it is more than a single statement, it should be enclosed in braces.

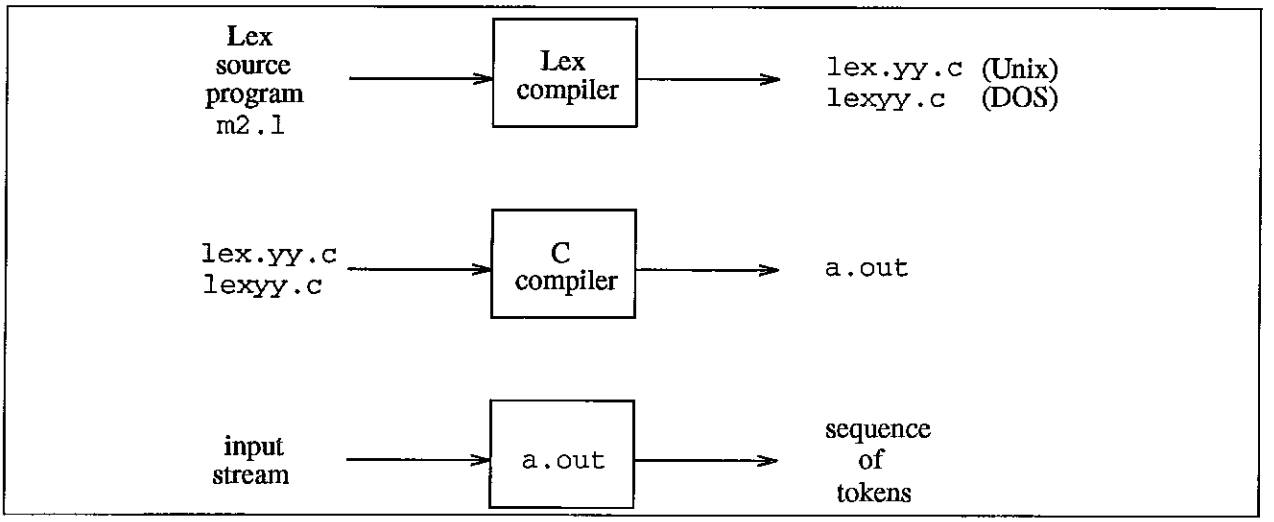


Figure 2.2: Creating a lexical analyzer with LEX

```
  { definitions }
%%
  { rules }
%%
  { programmer subroutines }
```

Figure 2.3: LEX file format

```
%{
#include <string.h>
#include "utility.h"
#include "pascal.tab.h"
%}
letter          [a-zA-Z]
digit           [0-9]
lord            [a-zA-Z0-9]
%%
BEGIN             {return(BEGINSY);}                                    10
END               {return(ENDSY);}
WHILE             {return(WHILESY);}
...
{letter}({lord})* {yylval.name_ptr = strdup(yytext); return(IDENTSY);}
({digit})+        {yylval.int_val = intnum(); return(CONSTANTSY);}
":="              {return(ASSIGNSY);}
":"               {return(COLONSY);}
...
.                 {error("Illegal character");}
%%                                                                      20
int intnum ()
  /* convert character string into an integer */
{
  ...
};  /* intnum */
```

Figure 2.4: Input to LEX for Pascal

  - `yytext` - variable where the lexeme is kept. This is a character string and is reused for every time a new token is recognized.
  - `yyleng` - the length of the string representing the lexeme.
  - `yylval` - a variable in which the lexeme can be returned. Describe the structure of the variable.
  - `yywrap` is a function that is called when EOF is encountered. When it is called, it needs to return a 1. Therefore, put in the definition `#define yywrap() 1`. This is already done in FLEX.
- *Programmer subroutines* are any C code needed by the lexical analyzer. This might include code to translate character strings into integer, for example.
- LEX creates a file called `lex.yy.c` that contains a function called `yylex`.

- As an example consider Figure 2.4 which shows the input into LEX for a small subset of Pascal tokens.
  - The prototypes for the function `error` are in the file `utility.h`.
  - The file `pascal.tab.h` is created by BISON.

```
%{
#include <string.h>
#include "utility.h"
#include "pascal.tab.h"
%}
letter          [a-zA-Z]
digit           [0-9]
lord            [a-zA-Z0-9]
%%
BEGIN             {return(BEGINSY);}
END               {return(ENDSY);}
WHILE             {return(WHILESY);}
...
{letter}({lord})* {yylval.name_ptr = strdup(yytext); return(IDENTSY);}
({digit})+        {yylval.int_val = intnum(); return(CONSTANTSY);}
":="              {return(ASSIGNSY);}
":"               {return(COLONSY);}
...
.                 {error("Illegal character");}
%%
int intnum ()
  /* convert character string into an integer */
{
  ...
};  /* intnum */
```

2.5 Hand Coded Lexical Analyzer

- Lexical analyzers may be hand written because of the lack of a tool or because of efficiency.
- Consider Figure 2.5 that shows an outline of a lexical analyzer for a subset of Pascal.
- Notes:
  - Handling reserved words
    * Use symbol table - when an identifier is recognized, the symbol table is searched to see if it is a reserved word. The symbol table would have to be initialized with all the reserved words and their token representation. If the identifier is found to be a reserved word, the appropriate token is returned. This mechanism works especially well if keywords are used instead of reserved words, then same mechanism used for scope can be used to handle this.
    * Use a reserved word table - initialized with all reserved words and their token representation. When an identifier is recognized, the table is searched to see if it is a reserved word. If it is, the appropriate token is returned. This is good use for a perfect hash table.

2.6 Finite Automata

- A finite state automaton can recognize any token specified by a regular expression.
- Consider the example in Figure 2.6
  - Show the different parts of the FSA: states, start state, final states, transitions, labels on transitions
  - Begin in start state and if we terminate in a final state, we have a valid token; if not, an error.
  - The FSA in Figure 2.6 is *incompletely specified* since some of the states do not have transitions on some elements of the alphabet.
  - The lexical analyzer could be thought of as a set of FSA's, one for each possible token.
- Formal definition of FSA: A deterministic finite–state automaton, or DFA, is a five–tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:
  1. Q is a finite set of *states.*
  2. Σ is a finite set of permissible input tokens, i.e., the *alphabet* of the machine.
  3. δ is a partial function that maps a state and an input token to another state, called the *state transition function.*
  4. $q_0 \in Q$ is a designated state called the *initial* or *start state* of the FSA.
  5. $F \subset Q$ are the final states (has to be at least one).
- For the example in Figure 2.6, we have the following:
  1. $Q = \{0, 1, 2, 3, 4, 5\}$

```
yylex ()
{ /* appropriate declarations */
  while (ch != EOF)
  { ch = skip_blanks();
    switch (ch)
    { case 'a': case 'b': ... case 'z':
      case 'A': case 'B': ... case 'Z':
        return(get_symbol());
        break;
      case '0': case '1': ... case '9':                    10
          return(get_number());
          break;
        case '''':
          return(get_string());
          break;
        case '*' :
       ch = next_character();
       return(MULTSY);
       break;
      case '+' :                                           20
          ch = next_character();
          return(PLUSSY);
          break;
        case ':' :
       ch = next_character();
       if (ch == '=')
       {
         ch = next_character();
         return(ASSIGNSY);
       } /* then */                                       30
       else
         return(COLONSY);
         break;
      ...
      default :
        if (ch = EOF)
          return(EOFSY);
        else error("Invalid character");
        ch = next_character;
        break;                                             40
    } /* switch */
  } /* while */
} /* yylex */
```

Figure 2.5: Hand coded lexical analyzer for a subset of Pascal

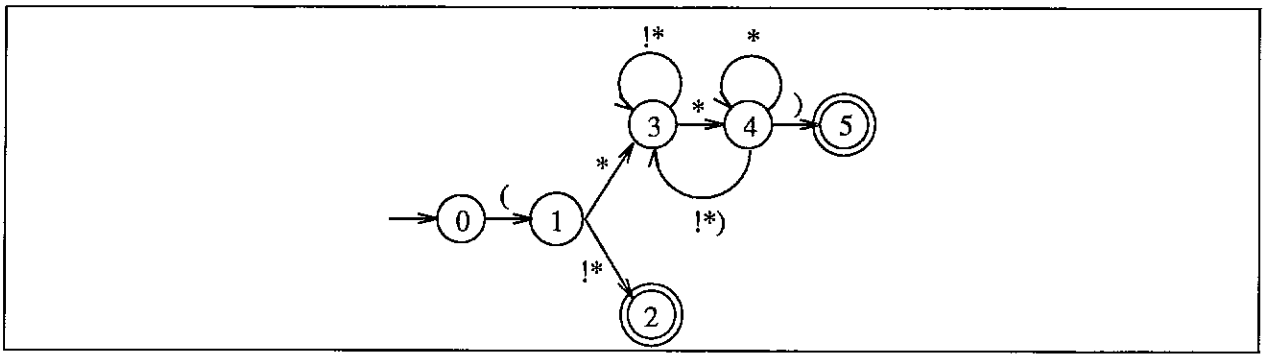


Figure 2.6: Example DFA for a Pascal comment

δ	(	*	)	Other
0	1			
1	2	3	2	2
2				
3	3	4	3	3
4	3	4	5	3
5				

Figure 2.7: Transition table

2. Σ = {any valid character}
3. The mapping function is given as $\delta(0,() = 1$ and $\delta(1,*) = 3$ etc., or can be shown in a table as shown in Figure 2.7. This is called a transition table.
4. $q_0 = 0$
5. $F = \{2,5\}$

- *Configuration* is designated (q,ω), where q is a state and ω is the string remaining to be scanned.
  - (q_0,ω) – initial configuration.
  - (q,ϵ) – final configuration if $q \in F$.
  - $\vdash$ designates a move.
- A move is made such that the following is true:

$$(q,a\omega) \vdash (q',\omega) \text{ iff } a \in \Sigma, \omega \in \Sigma^*, \text{ and } q' \in \delta(q,a).$$

- Figure 2.8 shows a program that uses the transition table. Show how this would work with the comment `(* a *)`.
- We can now describe the language, $L(M)$, recognized by a FSA M:

$$L(M) = \{\omega \in \Sigma^* | (q_0,\omega) \vdash^* (q,\epsilon) \text{ for some } q \in F\}$$

```
Read(CurrChar);
State = InitialState;
while (TRUE)
{
  NextState = T[State][CurrChar];
  if ((NextState == Error) || (CurrChar == Eof))
    break;
  State = NextState;
  Read(CurrChar);
} /* while */                                                    10
if (State in FinalStates)
/* return or process valid token */
else
  LexicalError();
```

Figure 2.8: Program for transition table

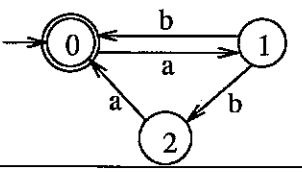


Figure 2.9: FSA for M_1

- Two machines M and M' are said to be syntactically equivalent if they recognize the same language, i.e., $L(M) = L(M')$. The machines need not have the same number of states.
- Consider the FSA in Figure 2.9 for M_1. Describe $L(M_1)$. $((ab|aba)^*)$
- What is different between the FSA in Figure 2.6 and the one in Figure 2.9?
  - Deterministic FSA – no choice provided in any of its moves.
  - Nondeterministic FSA – some arbitrary choices are permitted in some of its transitions.
- Differences between NFA and DFA.
  - $\delta(q, a) = \{\text{some set of states}\}$ – for a given terminal symbol, there may be a choice.
  - $\delta(q, \epsilon) = \{\text{some set of states}\}$ – there may be empty moves.
- Consider the Figures 2.10 and 2.11 and describe the languages. $(L(M_2) = L(M_3) = (ab|aba)^*)$ What is important about M_3? It is deterministic. We want to know how to transform NFA into DFA. Why? Easy to program and simulate (no backtracking).

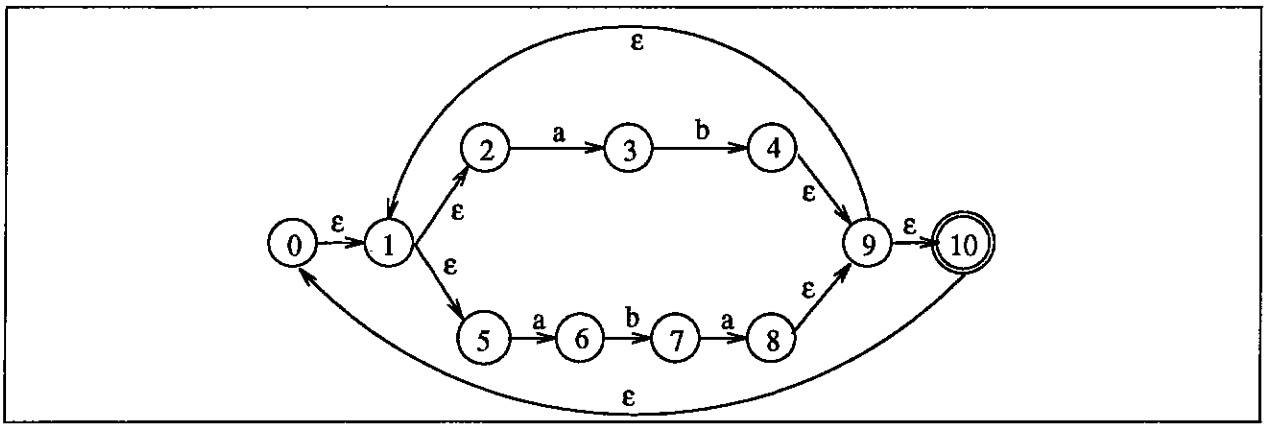


Figure 2.10: NFA for M_2

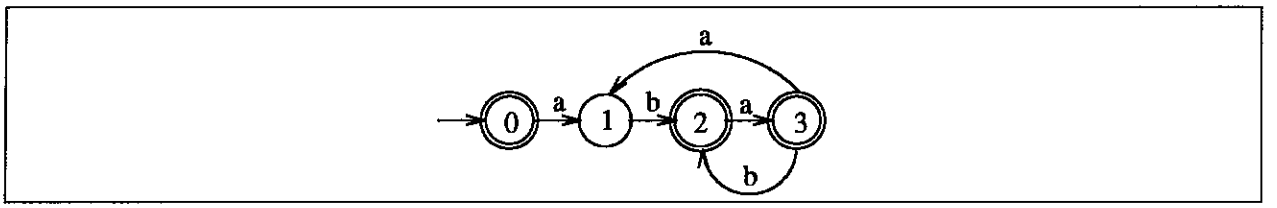


Figure 2.11: DFA for M_3

2.7 Translating a RE Into a Minimal FSA

2.7.1 RE to NFA

- Algorithm for translating a regular expression into a nondeterministic FSA is shown below:
  - For ϵ, see Figure 2.12.
  - For $a \in \Sigma$, see Figure 2.13.
  - For $A|B$, see Figure 2.14.
  - For AB, see Figure 2.15.
  - For A^*, see Figure 2.16.
- Consider Figure 2.17 for an example of the algorithm used to construct the NFA for $(ab|aba)^*$.

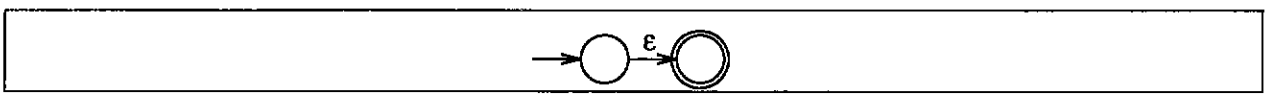


Figure 2.12: FSA for ϵ

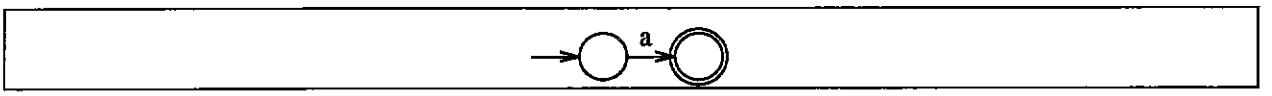


Figure 2.13: FSA for a

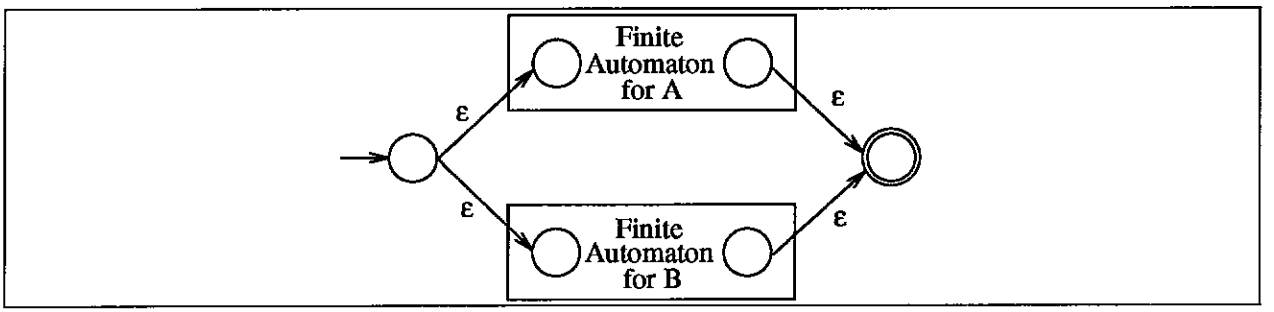


Figure 2.14: FSA for $A|B$

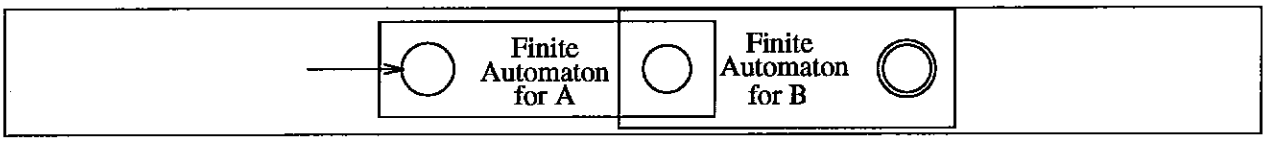


Figure 2.15: FSA for AB

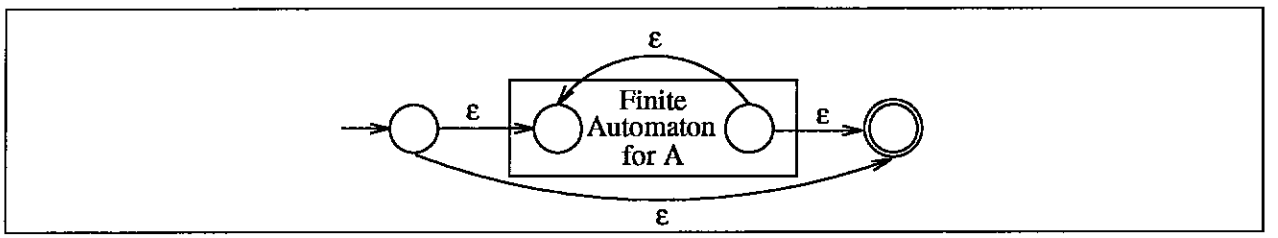


Figure 2.16: FSA for A^*

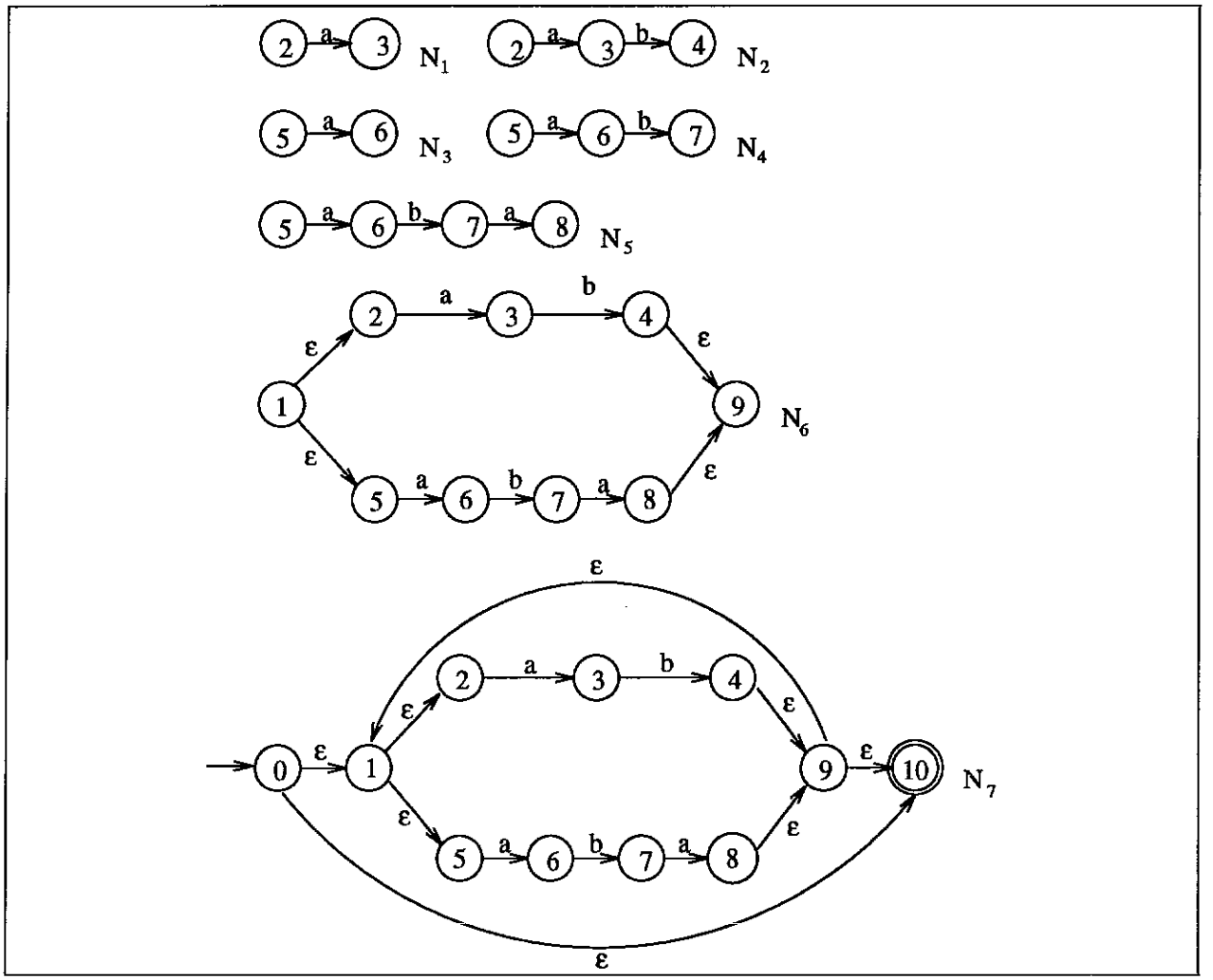


Figure 2.17: NFA for the regular expression $(ab|aba)^*$

2.7.2 NFA To DFA

- Algorithm for the construction of the DFA is shown in Figure 2.18.
- See Figure 2.19 for an example from Figure 2.17 making it deterministic.

2.7.3 Minimizing DFA

- FSA with minimum number of states is unique.
- Assume only two states, nonfinal states and final states.
- Call algorithm in Figure 2.20 with the set of nonfinal states.
- Set of states returned is the minimized DFA.
- See Figure 2.21 that shows the minimization of Figure 2.19.

2.7.4 Example 2

- Assume we have the regular expression $(a|b)(a|b)^*b$.
- Figure 2.22 shows the NFA constructed using the algorithm.
- Figure 2.23 shows the DFA constructed using the algorithm.
- Figure 2.24 shows the MFA constructed using the algorithm.

2.7.5 Example 3

- Assume we have the regular expression $(a|b)^*abb$.
- Figure 2.25 shows the NFA constructed using the algorithm.
- Figure 2.26 shows the DFA constructed using the algorithm.
- Figure 2.27 shows the MFA constructed using the algorithm.

Algorithm 2.1 DFA from NFA

{ *Input: An NFS N.* }
{ *Output: A DFA D accepting the same language.* }

{ *Assume: move(T,a) is the set of NFA states to which there is a transition on input symbol a from some NFA state s in T* }

{ *Input: a set of states, T* }

function *ϵ-closure(T)*
    Push all states in T onto *stack*
    Initialize *ϵ-closure(T)* to T
    while (*stack* is not empty) **do**
        Pop t, the top element, off of *stack*
        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 onto *stack*
            end if
        end for
    end while
end function

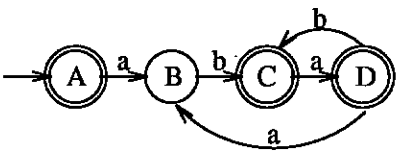
{ *Body of algorithm* }

Initially, *ϵ-closure(s_0)* is the only state in *Dstates* and it is unmarked.
while there is an unmarked state T in *Dstates* **do**
    Mark T
    for each input symbol a **do**
        U = *ϵ-closure(move(T, a))*
        if U is not in *Dstates* **then**
            Add U as an unmarked state to *Dstates*
        end if
        Dtran[T, a] = U
    end for
end while

Figure 2.18: Algorithm for translating a NFA to a DFA

States	Old States	Input	New State
A	{0,1,2,5,10}	a	B
B	{3,6}	b	C
C	{1,2,4,5,7,9,10}	a	D
D	{1,2,3,5,6,8,9,10}	a	B
		b	C

(a) Mapping of states in N to states in M



(b) Resulting DFA

Figure 2.19: Making nondeterministic FSA deterministic

Algorithm 2.2 DFA Minimization

{ *Input: A DFA M.* }
{ *Output: Minimized DFA.* }

Construct an initial partition Π_{new} of the set of states with two groups: the accepting states F and the nonaccepting states $S - F$

repeat
  $\Pi = \Pi_{new}$
  for each group G of Π **do**
    Partition G into subgroups such that two states s and t of G 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 Π
    { *At worst, a state will be in a subgroup by itself* }
    Replace G in Π_{new} by the set of all subgroups formed
  end for
until $\Pi_{new} = \Pi$
$\Pi_{final} = \Pi$

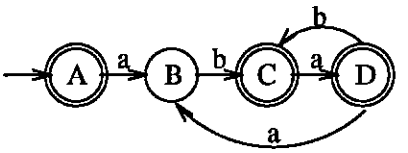
Choose one state in each group of the partition Π_{final} as the *representative* for that group. The representatives will be the states of the reduced DFA M'. Let s be a representative state, and suppose on input a there is a transition of M from s to t. Let r be the representatiave of t's group (r may be t). Then M' has a transition from s to r on a. Let the start state of M' be the representative of the group containing the start state s_0 of M, and let the accepting states of M' be the representatives that are in F. Note that each group of Π_{final} either consists only of state in F or has no states in F.

If M' has a dead state, that is, a state d that is not accepting and that has transitions to itself on all input symbols, then remove d from M'. Also remove any states not reachable from the start state. Any transition to d from other states become undefined.

Figure 2.20: Algorithm for DFA–Minimization

$F=A,C,D$
$N=B$

(a) Initial partitioning of states



(b) Final MFA

Figure 2.21: Minimization of deterministic FSA

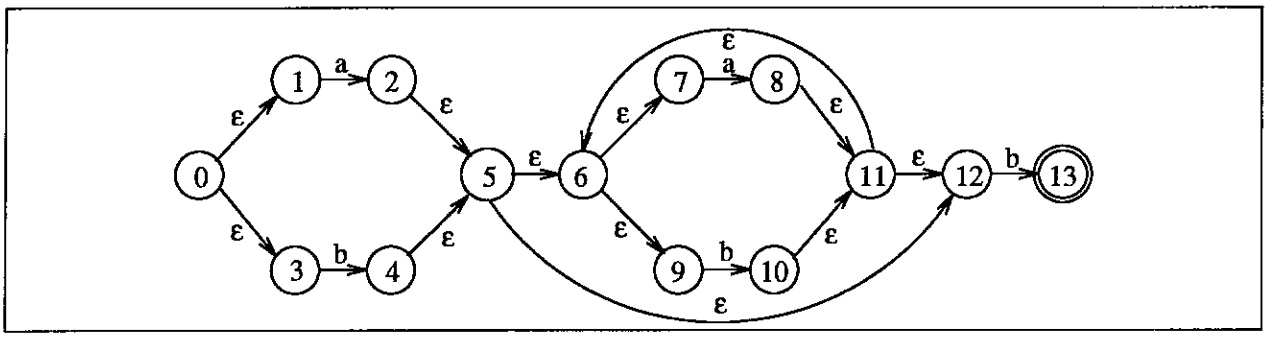


Figure 2.22: NFA for $(a|b)(a|b)^*b$

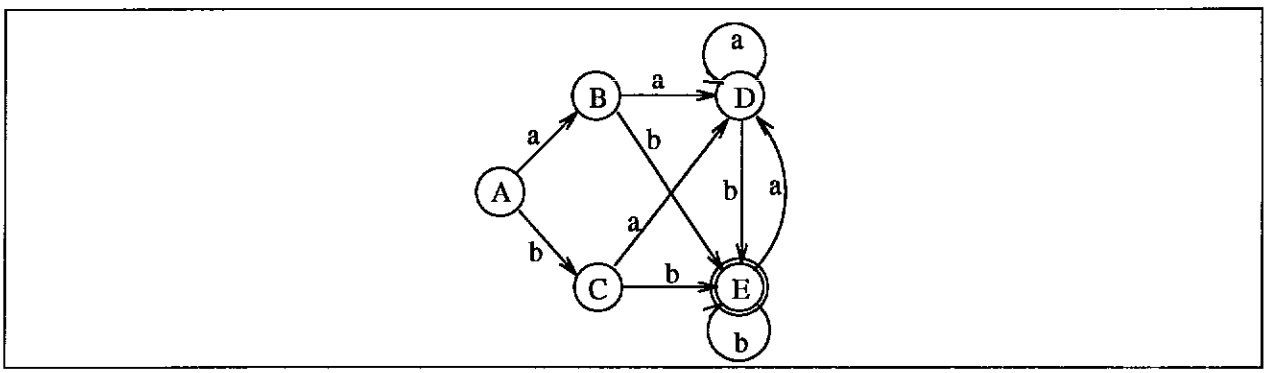


Figure 2.23: DFA for $(a|b)(a|b)^*b$

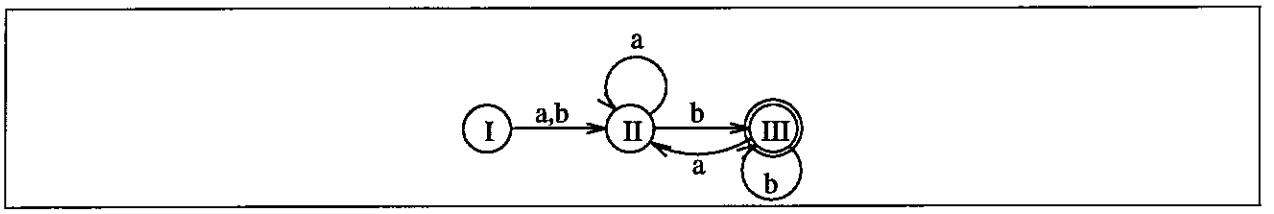


Figure 2.24: MFA for $(a|b)(a|b)^*b$

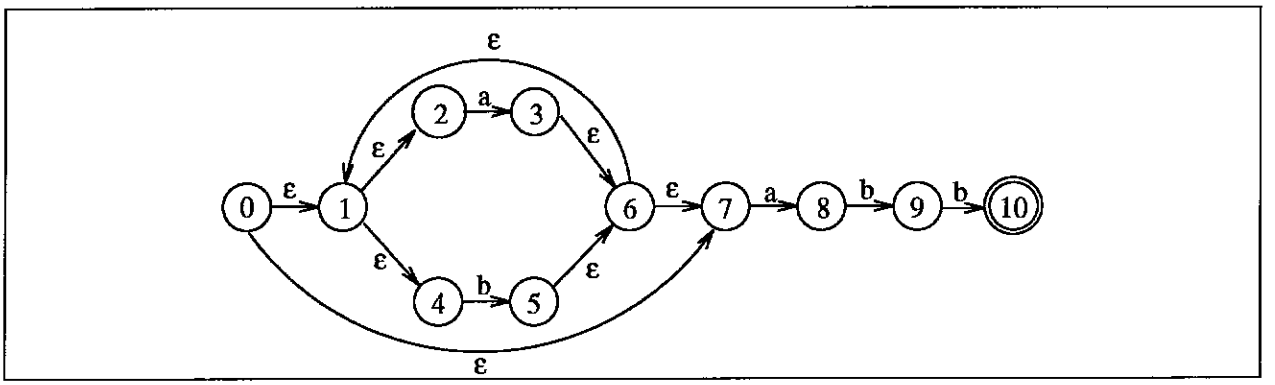


Figure 2.25: NFA for $(a|b)^*abb$

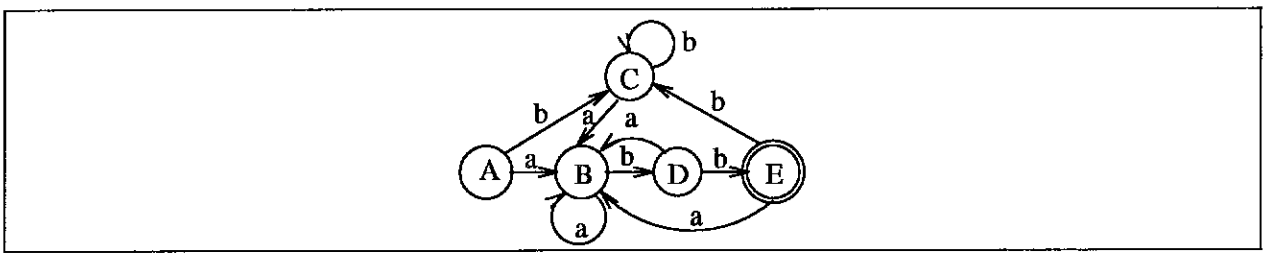


Figure 2.26: DFA for $(a|b)^*abb$

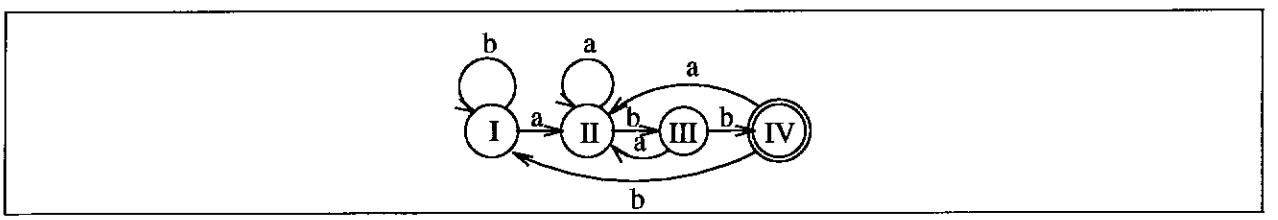


Figure 2.27: MFA for $(a|b)^*abb$