Compiler Design
Subject Code : CS3501

Compilers – Principles, Techniques and Tools, Second Edition by Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman

Dr. C. Dhaya

Professor

Department of Computer Science & Engineering

Adhiparasakthi Engineering College

Melmaruvathur - 603319

Unit - II
Lexical Analysis

1. Need & Role of Lexical analyzer
2. Lexical Errors
3. Expressing tokens by Regular Expressions
4. Converting Regular Expressions to DFA
5. Minimization of DFA
6. Language for Specifying Lexical Analyzers - LEX
7. Design of the Lexical Analyzer for the Sample Language

Lexical analysis

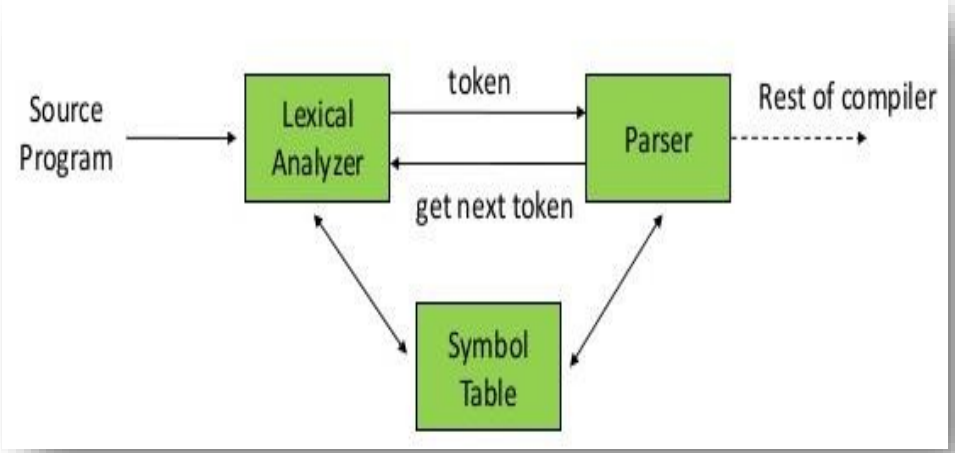
Used in Pattern Matching algorithms

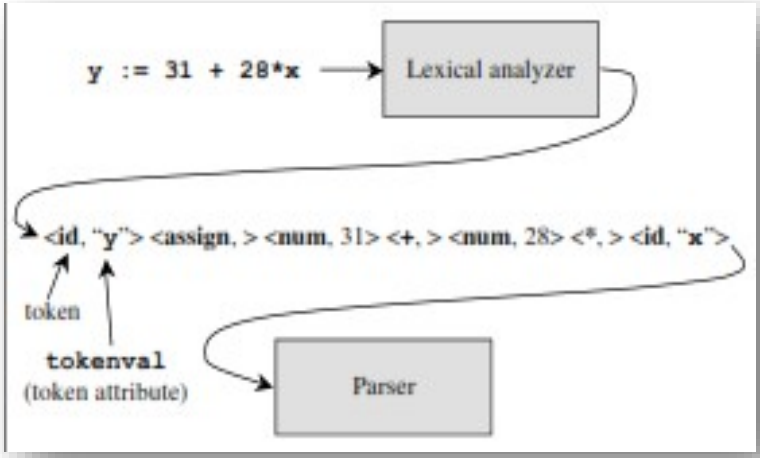
- Lexical analysis is the first phase of compiler which reads the source character one at a time and converts into sequence of tokens.



- The program which perform lexical analysis → Lexical Analyzer (or) LEXER (or) SCANNER (or) TOKENIZER

The Role of Lexical Analyzer





Lexical Analyzer

- Scans the source program to recognize the tokens

It works in 2 parts

1st task : scan

2nd task : generates the series of tokens

Functions of Lexical analyzer

- Produces stream of tokens
- Eliminates comments, white spaces, new line character, tabs etc.
- Associates error messages with source program
- Generates symbol table which stores the information about identifiers, constants encountered
- Keeps track of the number of lines

Issues in Lexical analysis

Why separating Lexical and parsing?

- Simpler design
- Compiler efficiency is improved
  - More time is spend for recognition of tokens
- Compiler portability is enhanced
  - Some non standard symbols ↑ in pascal can be isolated by lexical analyzer

Tokens, Patterns, Lexemes

Token → Sequence of characters having a collective meaning, such as keywords, operators, identifiers etc.
Lexeme → Instance of the Token
Pattern → Describes the rule that the lexeme takes

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
relation	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"core dumped"

Attributes for Tokens

- A token represent more than 1 lexemes, some additional information are needed – Attributes of the token
- **Example:** E = M * C ** 2

  <id, pointer to symbol-table entry for E**>**

  <assign_op, >

  <id, pointer to symbol-table entry for M**>**

  <mult_op, >

  <id, pointer to symbol-table entry for C**>**

  <exp_op, >

  <num, integer value 2**>**

Lexical errors

- Some errors are out of power of lexical analyzer to recognize:
  - **fi** (a == f(x)) …
    - Lexical analyzer cannot tell its misspelled keyword
- However it may be able to recognize errors like:
  - d = 2r
    - Such errors are recognized when no pattern for tokens matches a character sequence

Errors in different Phases

- Lexical analysis → when they do not form token
- Syntactical analysis → syntax errors
- Semantic analysis → type mismatch

- Necessity of Error Handler in a Parser
  - It should report the presence of errors clearly and accurately
  - It should recover from each error quickly
  - It should not significantly slow down the processing of the programs

Error Recovery Strategies

- Panic Mode Recovery
  - Discard input until a token in a set of designated synchronizing tokens is found
- Phrase-level Recovery
  - Perform local corrections to repair the program
- Error Productions
  - Augment grammar with productions for erroneous constructs
- Global Corrections
  - Choose a minimal sequence of changes to obtain global least cost correction.

Panic Mode

- The parser discovers an error. It then discards input symbols till a designated set of synchronizing token is found.

  In case of an error like:

  ```
  a = b + c      // no semi-colon
  d = e + f ;
  ```

  The compiler will discard all subsequent tokens till a semi-colon (; : etc. are synchronizing token) is encountered.

Drawback: Skips considerable amount of input

Phrase-level Recovery

- Local Correction by parser on remaining input, by some string which allows parser to continue.
  - Replacing comma by semicolon, inserting extra semicolon etc.

- Drawbacks: Improper replacement might lead to infinite loops.

Error Productions

- A method of anticipating common errors that might be encountered.
  - Works for only common mistakes which was earlier identified.
- Augmenting the grammar with productions that generate erroneous constructs.

Global Corrections

- Ideally, we would like the compiler to make as few change as possible in processing incorrect inputs.
- We have to globally analyze the input to find the error.

- Drawback: Too costly to implement in terms of both time and space, and only theoretical.

Lexical error handling approches

- Successive characters are ignored until a token is framed
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Input buffering

- The lexical analyzer scans the input string from left to right.
- It uses 2 pointers
  - **begin_ptr(bp)**
  - **forward_ptr(fp)**

  - To keep track of the input scanned.
  - Initially both the pointers, point to the 1st character of the i/p string.

Contd…

- Forward_ptr moves ahead character by character. As soon as blank space is encountered, it indicates end of lexeme.

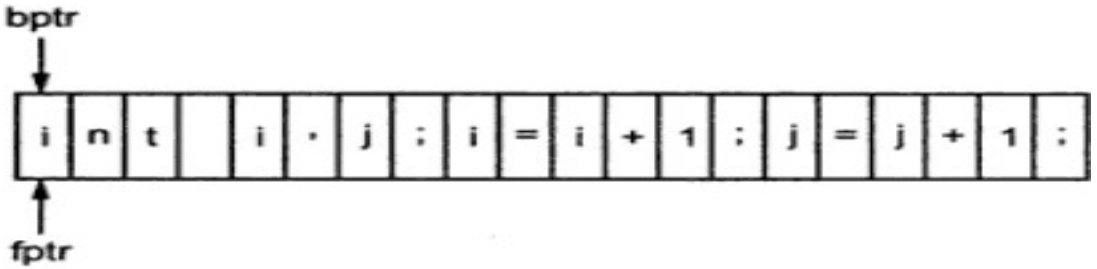


Fig. 2.2 Initial configuration

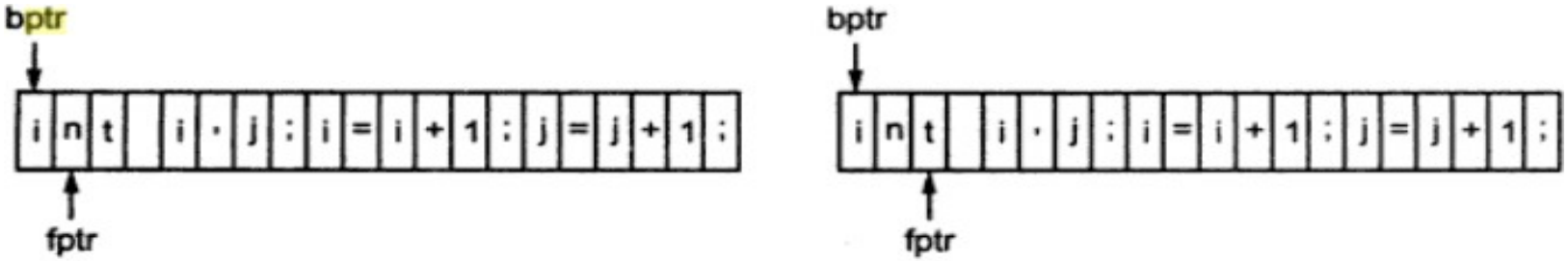


Fig. 2.3 Input buffering

Contd…

- Then both begin_ptr and forward_ptr is set to next token.

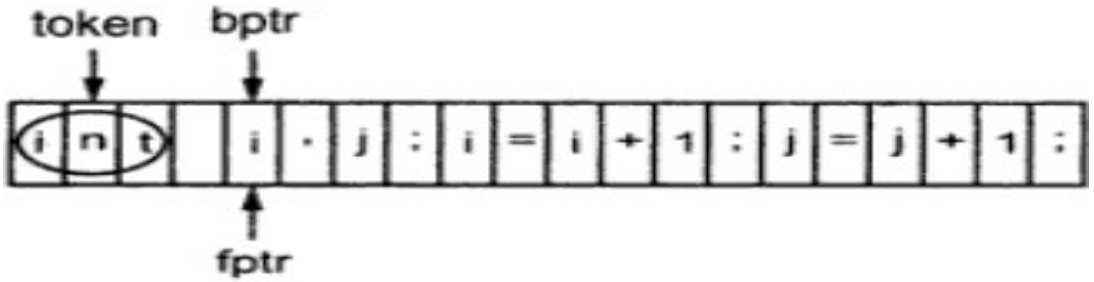


Fig. 2.4 Input buffering

- Input characters are read from secondary storage. But it is too costly. So, buffering schemes are used
  - One buffer scheme
    - Only 1 buffer is used to store the input string. But when string is too long, overwriting occurs.
  - Two buffer scheme
    - 2 buffers are used. First buffer and second buffer.

Why buffering?

- Initially forward_ptr at the beginning of first half. When it moves to the end of first half, then second half is filled with i/p characters.

- Then, forward_ptr is moved to right end of the second half, the forward_ptr wraps to the beginning of the first half.

Two buffer Algorithm

```
if (forward_ptr at end of first half) then
{       load second half;
        forward_ptr = forward_ptr + 1;
}
```

```
else if (forward_ptr at end of second half) then
{       reload first half;
        move forward_ptr to beginning of first_half;
}
```

```
else
        forward_ptr = forward_ptr + 1;
```

Sentinels - Special character that cannot be part of source program usually eof




```
forward_ptr = forward_ptr + 1;
If (forward_ptr != eof )then
begin
        if (forward_ptr at end of first half) then
        {
                load second half;
                forward_ptr = forward_ptr + 1;
        }
        else if (forward_ptr at end of second half) then
        {
                reload first half;
                move forward_ptr to beginning of first_half;
        }
     end.
```

e

buffer marks the end of input */

terminate lexical analysis;

Specification of tokens

- To specify tokens, regular expressions are used. When a pattern is matched by some regular expression, then token can be recognized.
  - Alphabet or character class
  - String
  - Language

Alphabet or character class

- Denotes finite set of symbols
  - { 0, 1} → binary alphabet
  - ASCII and EBCDIC → computer alphabets

String

- Finite number of alphabets (or) letters. Its also called as words.
  - Length of the string |S|
  - Empty string ε
  - Empty set of strings ϕ

String – Contd…

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**.

Example

Terms for parts of a string (examples)

Let us take this string: banana

prefix:

ε, b, ba, ban, ..., banana

suffix:

ε, a, na, ana, ..., banana

substring:

ε, b, a, n, ba, an, na, ..., banana

subsequence:

ε, b, a, n, ba, bn, an, aa, na, nn, ..., banana

Operations on Languages

- Union: $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
- Concatenation: $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
- Kleene closure:
  - $L^* = \bigcup_{i=0}^{\infty} L^i$
  - Zero or more concatenations
- Positive closure:
  - $L^+ = \bigcup_{i=1}^{\infty} L^i$

- One or more concatenations

Language

- Language is a collection of strings. Several operations that is performed on languages are,

Operation	Description
Union of two languages L1 and L2.	L1U L2 = {set of strings in L1 and strings in L2}.
Concatenation of two languages L1 and L2.	L1.L2 = {set of strings in L1 followed by set of strings in L2}.
Kleen closure of L.	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes zero or more concatenations of L
Positive closure of L.	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes one or more concatenations of L

Example

- L = { A,B, C…Z, a,b,c…z} D = { 0,1,2,3…9}
- Various operations,
  - LUD → set of letters and digits
  - LD → set of strings consisting of letters followed by digits
  - L^5 → set of strings having length =5
  - L^* → set of all strings including Ɛ
  - L^+ → set of all strings except Ɛ
  - L(LUD) → set of all strings beginning with a letter
  - D^+ → set of all strings of 1 or more digits

Regular Expressions

- RE are mathematical symbols which describe set of strings of specific language. It is easy way of representing Tokens.

Basic Rules of Regular expression:

Given an alphabet Σ,

(1) ε is a regular expression that denote $\{\varepsilon\}$, the set that contains the empty string.

(2) For each a $\varepsilon \Sigma$, a is a regular expression denote {a}, the set containing the string a.

(3) r and s are regular expressions denoting the language (set) L(r) and L(s). Then

- (r) | (s) is a regular expression denoting L(r) U L(s)
- (r) (s) is a regular expression denoting L(r) L (s)
- (r)* is a regular expression denoting (L (r)) *

RE Examples

- L(**001**) = {001}
- L(**0+10***) = { 0, 1, 10, 100, 1000, 10000, … }
- L(**0*10***) = {1, 01, 10, 010, 0010, …} i.e. {w | w has exactly a single 1}
- L(**ΣΣ**)* = {w | w is a string of even length}
- L((**0(0+1)**)*) = { ε, 00, 01, 0000, 0001, 0100, 0101, …}
- L((**0+ε)(1+ ε**)) = {ε, 0, 1, 01}
- L(1Ø) = Ø ; concatenating the empty set to any set yields the empty set.
- Rε = R
- R+Ø = R

- Note that R+ε may or may not equal R (we are adding ε to the language)
- Note that RØ will only equal R if R itself is the empty set.

Some definitions:

- Regular set
  - Language denoted by regular expression
- Equivalent regular expression
  - If 2 regular expressions r and s, denote the same language, then r = s.
- Precedence of operators
  1. * - highest precedence and left associative
  2. Concatenation – left associative
  3. | - lowest precedence and left associative

Contd… - Algebric laws

Axiom	Description
r| s = s| r	| is commutative
r | (s| t) = (r | s) | t	| is associative
(rs)t = r(st)	Concatenation is associative
r(s|t) = rs|rt and (r|s)t = rt|st	Concatenation distributes over |
ε r = r ε = r	ε is an identity element for concatenation
r* = (r|ε)*	Relation between r and ε

r = r***	***** Is idempotent

Regular definition

- Gives names to regular expressions to construct more complicate regular expressions.

  d1 -> r1

  d2 ->r2

  …

  dn ->rn

- example:

  letter -> A | B | C | … | Z | a | b | …. | z

  digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  identifier -> letter (letter | digit) *

Notations used to represent Regular Expression

- One or more instances
  - Unary postfix operator + is used
    - If ‘r’ is a RE denoting L(r)
    - $(r)^+$ is a RE denoting $(L(r))^+$
    - Example:
      - L(r) ={a} then $(L(r))^+$ = {a,aa,aaa…}
- Zero or more instances
  - Unary postfix operator ? is used
    - If ‘r’ is a RE denoting L(r)
    - (r)? is a RE denoting (L(r))U {**ε**}
- Character classes
  - If a,b,c are symbols of an alphabet, the notation [abc] denotes the RE a|b|c.
    - [a-z] denotes the RE a|b|….|z

Non-regular language

- Some languages cannot be described by regular expressions.
- REs cannot in
  - Balanced (or)Nested constructs
  - Repeated strings
  - Arbitrary numbers cannot be compared
  - Palindrome cannot be checked

Problem 1

- Let Σ= {a,b}. Give the regular set for the following regular expression

Regular Expression	Regular set
a | b	{a,b}
(a|b) (b|a)	{ab,aa,ba,bb}
a*	{ε, a,aa,aaa….}
a|ab	{a,b,aab,ab…}
(a|b)*	{ε, a,b,ab,abb,bb…}

Problem 2

- Operations on languages

Operation	Notation
Union	LUM = { S | S is in L (or) S is in M}
Concatenation	LM = { S | S is in L (and) S is in M}
Kleene Closure	$L^* = L^0 \cup L^1 \cup L^2$
Positive closure	$L^+ = L^1 \cup L^2 \cup L^3$ …

Problem 3

L = {00,10} and M ={01,11}

Find,

- LUM = {00,10,01,11}
- LM = {0001,0011,1001,1011}
- L^0 = {ε}
- L^1 = {00,10}
- L^2 = {0000,0010,1000,1010} --- a^2,ab,ba,b^2
- M^2 = {0101,0111,1101,1111}
- L^* = {ε ,00,10,0000,0010,...}
- L^+ = {00,10,0000,0010,...}

Problem 4

- Represent the following using regular expressions:

1. set of all strings of 0's and 1's ending with 01.

   (0|1)* 01

1. set of all strings of 0's and 1's having at least 2 occurrences of 00.

   (0|1)*00(0|1)*00(0|1)*

1. set of all strings of 0's and 1's having 2 or more symbols followed by 3 or more zero's.

   (0|1)(0|1)000+

Problem 5

- Give the regular set for the following regular expressions:

1. 0(0|1)*0 – all strings of 0's and 1's, start and end with 0.

   {000,00,010 …)

1. ((ε|0)1*)* - {ε, 0,00,…1,11,111….}
2. (1|10)* - {ε,1,10,110…}
3. (a*b|b*a) – {b,ab,aab,…..a,ba,bba….}

Recognition of tokens

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**



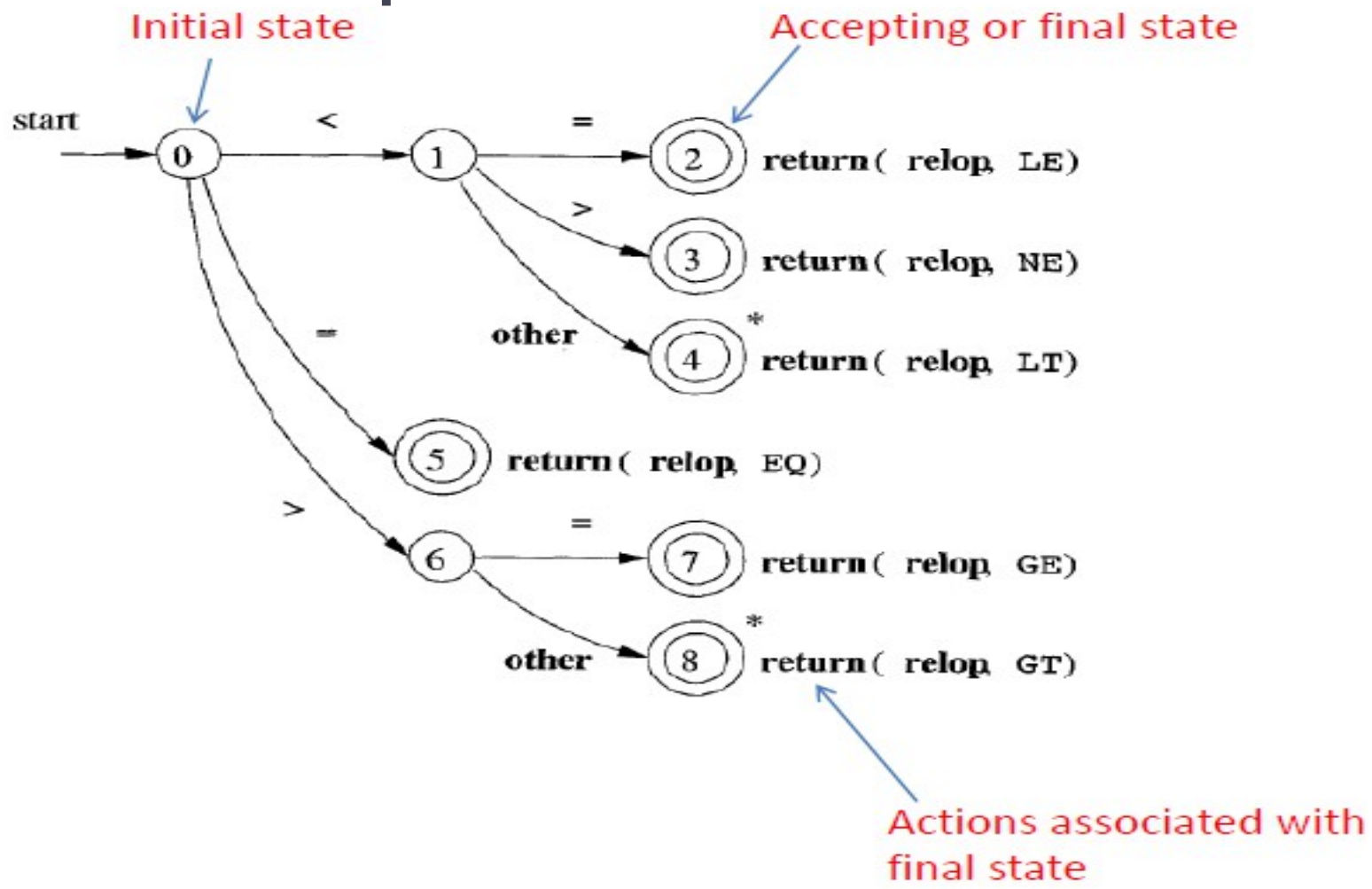
digit	→	`[0-9]`
digits	→	*digit*$^+$
number	→	*digits* (`.` *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`
ws	→	(**blank** | **tab** | **newline**)$^+$

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	LT
`<=`	**relop**	LE
`=`	**relop**	EQ
`<>`	**relop**	NE
`>`	**relop**	GT
`>=`	**relop**	GE

Transition diagrams

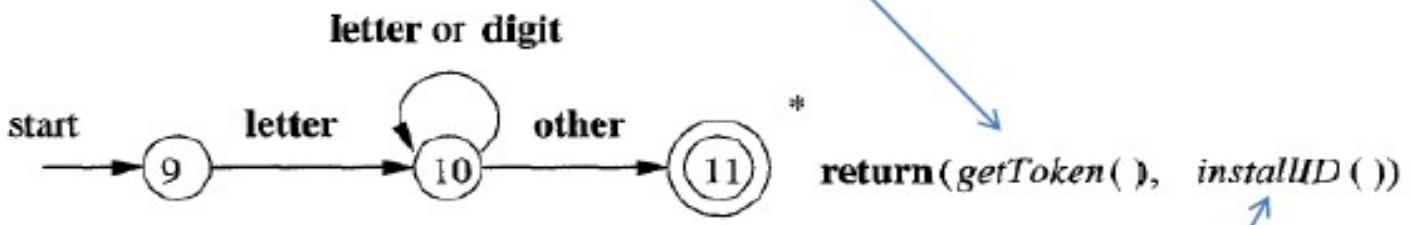
- Intermediate step in constructing lexical analyzer.
  - Similar to flowchart
  - States - represented by circles
  - Edges – states are connected by arrows
  - Start state – initial state
  - Actions – Action state or final state

(i) Transition diagram for relational operator



(ii) Transition diagram for identifier and keywords

- Examine symbol table for the lexeme found
- Returns whatever token name is there

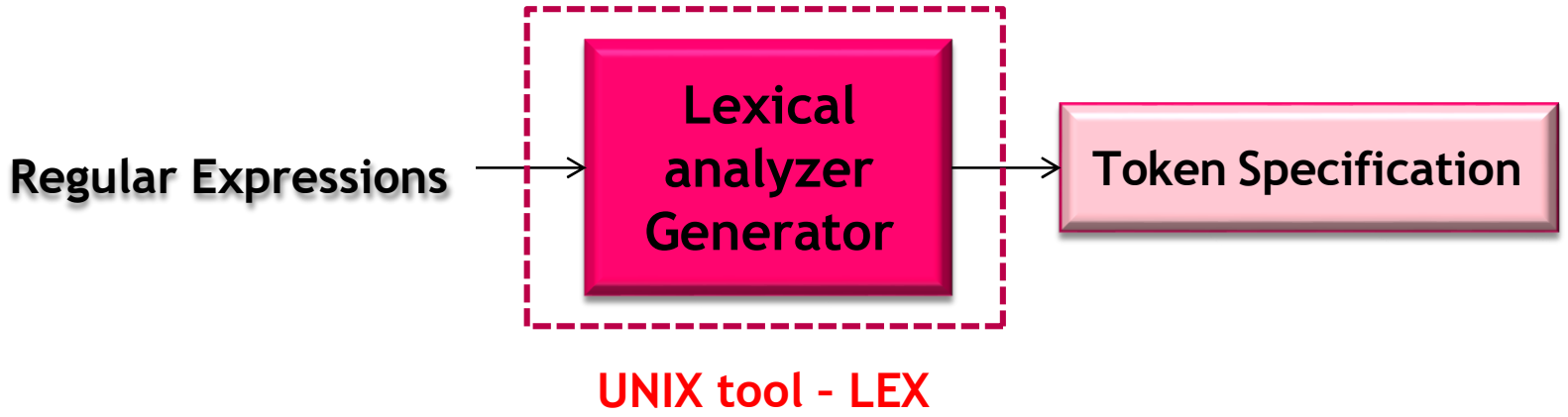


- Places ID in symbol table if not there.
- Returns a pointer to symbol table entry

LEX

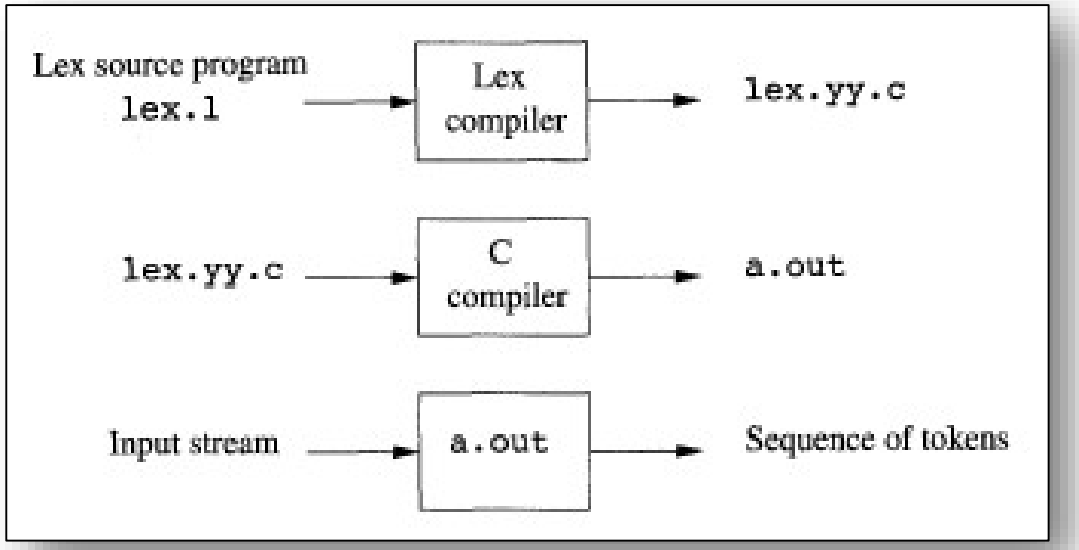
A language for specifying lexical analyzer

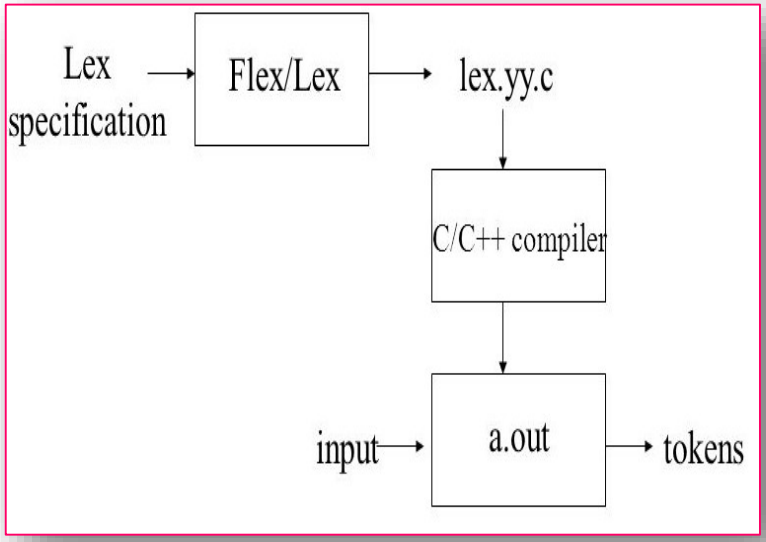
- LEX is a UNIX tool in lexical analysis phase to recognize tokens from regular expressions



Creating a Lexical Analyzer with Lex

Create Specification in lex language – **lex.l**
Run lex.l through compiler to produce **lex.yy.c**
Run lex.yy.c through C compiler to produce **a.out**





Contd…

- ***lex.l*** is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms *lex.l* to a C program known as *lex.yy.c*.
- ***lex.yy.c*** is compiled by the C compiler to a file called *a.out*.
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
- ***yylval*** is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.

Lex Specification – has 3 parts

1. Declarations

   %%

2. Translation rules

   %%

3. Auxiliary procedures

Declarations Section: Variables, manifest constants and regular definitions

Manifest constant – *identifier that is declared to represent a constant*

Translation Rules

p_1	{action1}
p_2	{action 2}
p_3	{action 3}

Actions are written in C language / any other language

Auxiliary Procedures - Contains additional routines needed by the actions

Contd…

- Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found.
- **Once a match is found, the associated action takes place to produce token**.

- The token is then given to parser for further processing.

LEX program for the tokens

1

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

Contd...

2

```
%%

{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum(); return(NUMBER);}
"<"         {yylval = LT; return(RELOP);}
"<="        {yylval = LE; return(RELOP);}
"="         {yylval = EQ; return(RELOP);}
"<>"        {yylval = NE; return(RELOP);}
">"         {yylval = GT; return(RELOP);}
">="        {yylval = GE; return(RELOP);}

%%
```

Contd…

3

```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

install_id

- Looks in symbol table for lexeme matched by pattern id; creates new entry if needed
- Lexeme is extracted using variables **yytext** and **yyleng**

Example Program

```
%{
/* a lex program that adds line numbers to lines of text,
printing the new text to the standard output */
#include <stdio.h>
int lineno = 1;
%}
line        .*\n
%%
{line}     {printf("%5d %s", lineno++, yytext); }
%%
main()
{ yylex(); return 0;}
```

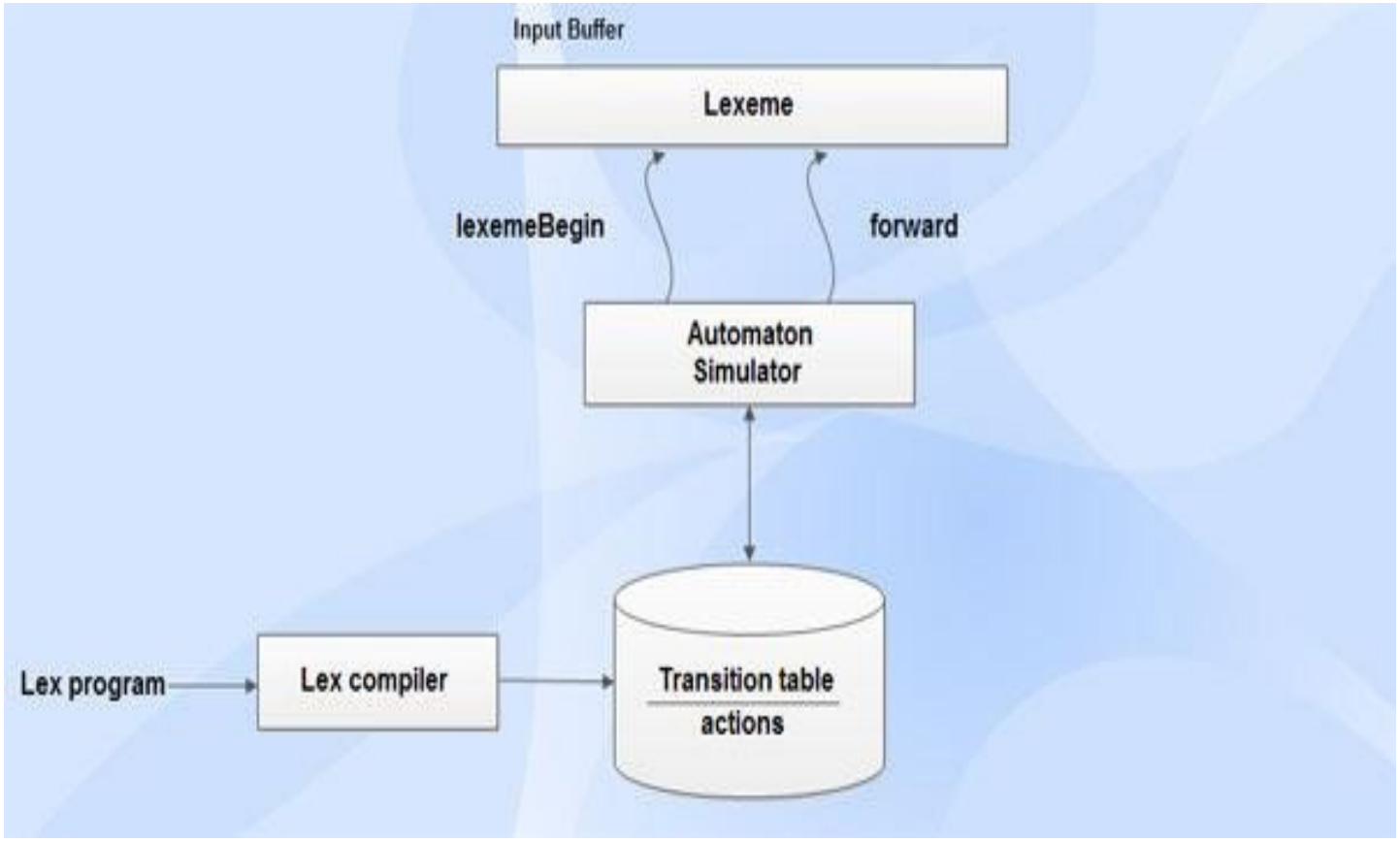
DESIGN OF THE LEXICAL ANALYZER

- Lexical Analyzer can be either generated by NFA or by DFA
  - DFA is preferable in the implementation of LEX

Structure of the Generated Analyzer

Architecture of lexical analyser generated by LEX

- LEX program is turned into transition table & actions, which is used by finite automation simulator


Input Buffer
Lexeme
lexemeBegin
forward
Automaton
Simulator
Lex program
Lex compiler
Transition table
actions

Structure of the Generated Analyzer – Contd…

- Lexical analyser program includes:
  - Program to simulate automata
  - Components created from LEX program are listed as follows:
    - A transition table for automaton.
    - Functions that are passed directly through LEX to the output.
    - Actions from input program (fragments of code) which are invoked by automaton simulator when needed.

Steps to construct automaton

- **Step 1:** Convert each regular expression into NFA either by Thompson's subset construction or Direct Method.
- **Step 2:** Combine all NFAs into one by introducing new start state with s-transitions to each of start states of NFAs N_i for pattern P_i.

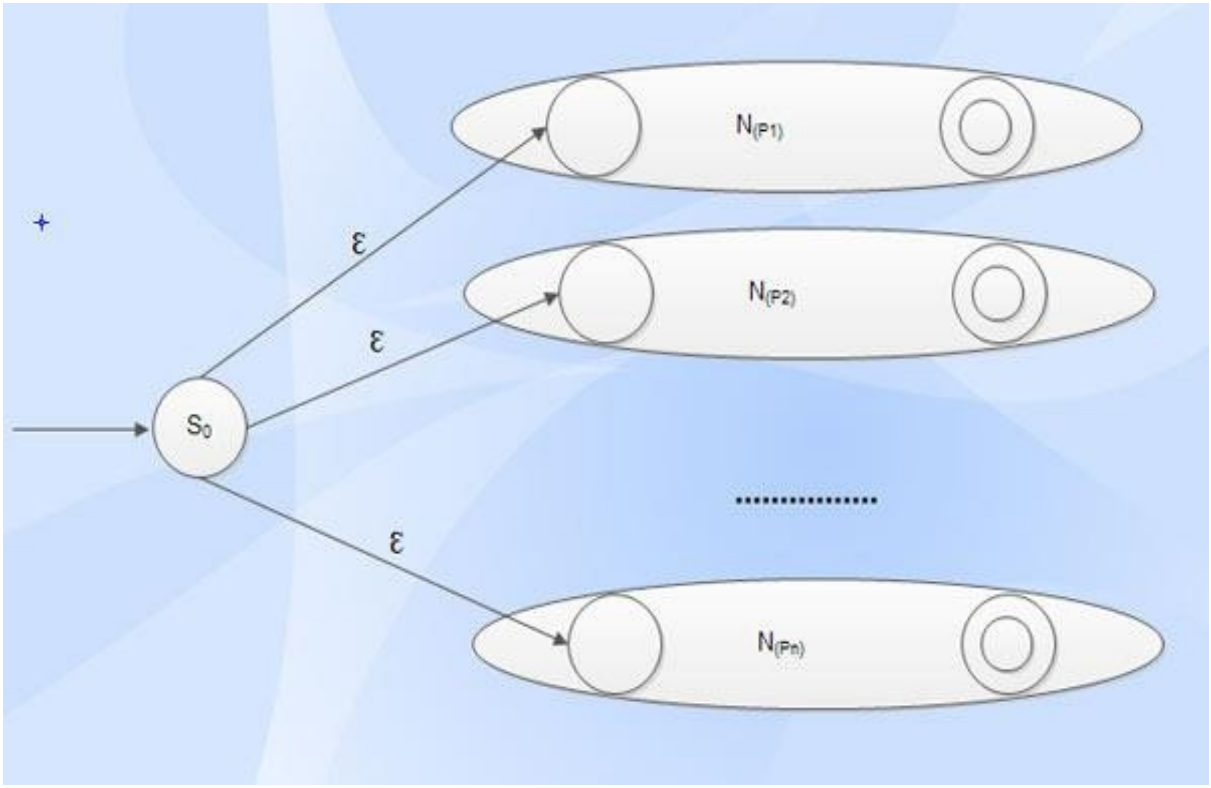
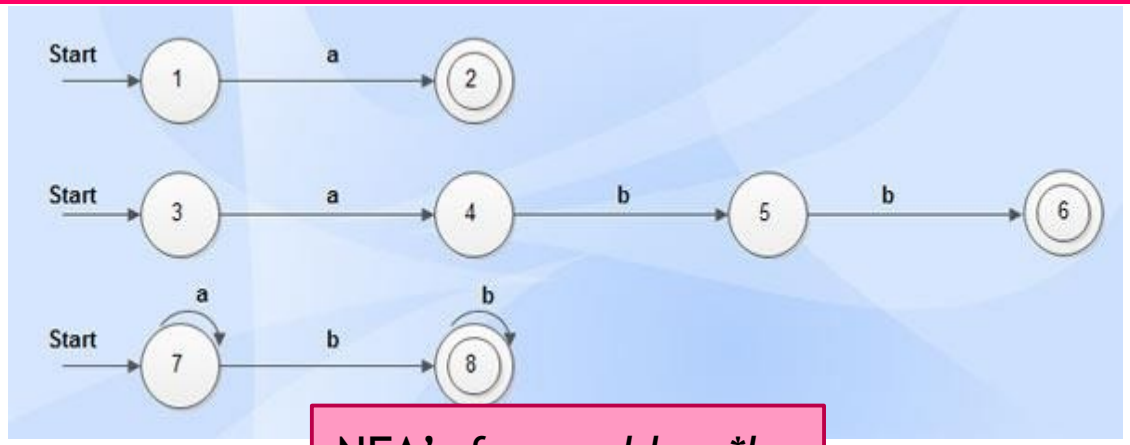


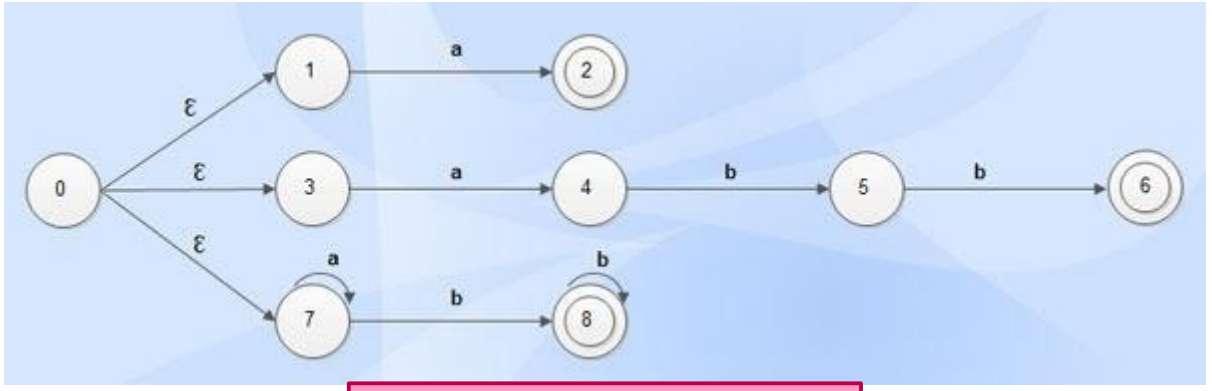
Fig. Construction of NFA from LEX program

Contd…

- (eg.)
  - *a* {action A_1 for pattern P_l}
  - *abb* { action A_2 for pattern P_2 }
  - *a*b+* { action A_3 for pattern P_3 }
- For string *abb*, pattern P_2 and pattern p_3 matches.
  - But the pattern P_2 will be taken into account as it was listed first in lex program.
- For string *aabbb* · · · , matches pattern p_3 as it has many prefixes.



NFA's for *a*, *abb*, *a*b+*



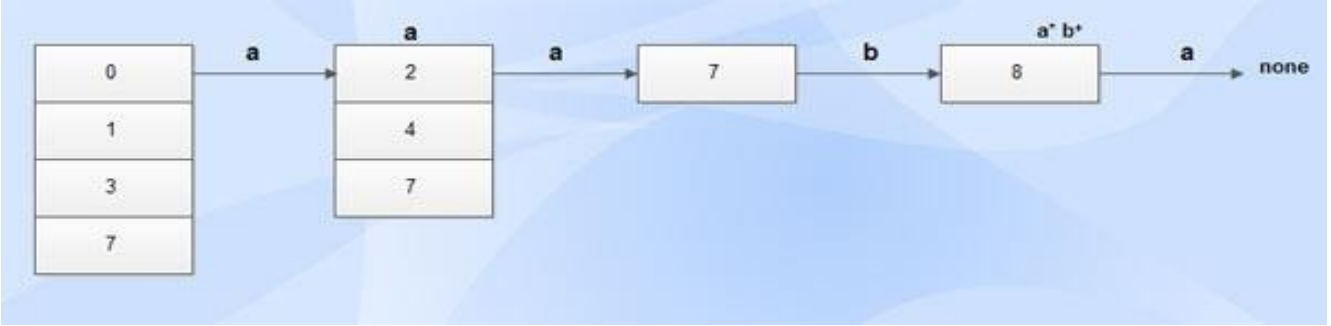
Combined NFA

Pattern Matching Based on NFAs

- Lexical analyser reads input from input buffer from the beginning of lexeme pointed by the pointer *lexemeBegin*.
- *Forward* pointer is used to move ahead of input symbols, calculates the set of states it is in at each point.
  - If no next state for some input symbol → then there will be no longer prefix
  - When lexeme is matched with some pattern → that prefix is taken
- Process is repeated until one or more accepting states are reached.
  - If there are several accepting states, then the pattern P_i which appears earliest in the list of LEX program is chosen.

Explanation W= aaba

- Process starts with s-closure of initial state 0. After processing all the input symbols, no state is found as there is no transition out of state 8 on input *a*. Hence, look for accepting state by retracting to previous state. From Fig. state 2 which is an accepting state is reached after reading input symbol *a* and therefore the pattern a has been matched. At state 8, string *aab* has been matched with pattern *abb"*: By Lex rule, the longest matching prefix should be considered. Hence, action A3 corresponds to pattern p3 will be executed for the string *aab*.



DFAs for Lexical Analyzers

- DFAs are also used to represent the output of LEX.
- DFA is constructed from NFA, by converting all the patterns into equivalent DFA using subset construction algorithm.
- If there are one or more accepting NFA states,
  - the first pattern whose accepting state is represented in each DFA state is determined and displayed as output of DFA state.
  - Process of DFA is similar to that of NFA.
  - Simulation of DFA is continued until no next state is found.
  - Then retraction takes place to find the accepting state of DFA.
  - Action associated with the pattern for that state is executed.

Lookahead operator

- Lookahead will typically be important to a lexical analyzer.
- **Tokens are typically read in from left-to-right**, recognized one at a time from the input string.
- It is not always possible to instantly decide if a token is finished without looking ahead at the next character.

- For example…
  - Is “i” a variable, or the first character of “if”?
  - Is “=” an assignment or the beginning of “==”?

UNIT – II Completed