Chapter 2

Lexical Analysis

Syllabus

- ✧ Lexical Analysis.
- ✧ Review of lexical analysis.
- ✧ Alphabet.
- ✧ Token.
- ✧ Lexical error.
- ✧ Block schematic of lexical analyzer.
- ✧ Automatic construction of lexical analyzer (LEX).
- ✧ LEX specification and features.

2.1 Role of the Lexical Analyzer :

The lexical analyzer is the first phase of a compiler.

Definition :

- Lexical analysis is the operation of reading the input program and breaking it into a sequence of lexemes (tokens).
- The syntax analyzer uses these tokens to produce parse tree .
- Each token is a sequence of characters that represents a unit of information in the source program.
- The interaction between lexical analyzer and parser is well defined. The parser calls a single lexical analyzer subroutine every time as it needs a new token and then subroutine (i.e. Lexical Analyzer) reads input characters until it can identify the next token and returns it to the parser. This relationship is shown in Fig. 2.1.1.

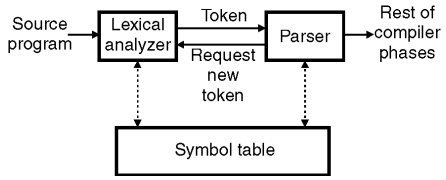


Fig. 2.1.1

- In addition the lexical analyzer also performs certain secondary tasks like **removing the comments and white spaces** (blank; tab and new line characters) from the source program. It may also be given the responsibility of making a copy of the source program with the **error messages** marked in it. Each error message may also be associated with a **line number**.
- The analyzer may keep track of the line number by tracking the number of new line characters seen while reading source program a character by character.
- There are several reasons of separating lexical analysis and parsing in analysis phase :
  (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 if separate lexical analyzer allows us to construct a specialized and potentially more efficient parsers.
  (3) A large amount of time is spent in reading the source program and partitioning it into tokens.
  (3) Compiler portability is enhanced. Input alphabet peculiarities and other device specific anomalies can be restricted to the lexical analyzer.
- Some terms with meaning that are used in lexical analyzer :
  - **Lexemes** : Smallest logical units (words) of a program such as A, B, 10, if, + etc.
  - **Tokens :** Classes of similar lexemes such as identifier, number, constant, operator etc.
  - **Pattern :** Formal or informal description of a token such as an identifier can have at most 8 characters in which first character must be an alphabet and the successive characters can be either digits or alphabets. Pattern is rule that describes a token.
  - The pattern serves the two purposes :
    (1) Matching each string which satisfies the description of the token specified by it.
    (2) Generating the lexical analyzer automatically by using this description.

For example, function definition in C

```
Mult_three (float num1, float num2, float num3)
{
    float ans;
    ans = num1 * num2 * num3;
    return (ans);
}
```

- List of tokens in the function definition and their corresponding lexemes and pattern.

Token	Lexeme	Pattern
Keyword	return, float	return, float
Identifier	num1, num2, num3, ans	Letter followed by letter(s) and/or digit(s)
Delimiter	(, ; ,), { , }	(or; or) or { or }or ,
Operators	=, +	= or +

- Lexical analyzer must also pass additional information along with the token. These items of information are called attributes for tokens.
- Generally a pattern matches more than one lexeme. Therefore the lexical analyzer must provide additional information about the particular lexeme that matches the pattern.

2.2 Lexical Error :

- It is difficult to find out error at the lexical level because a lexical analyzer has a very localized view of a source program.
- Example : If a string wilhe appear in a C program in the following context :

  Wilhe (x > y)

- It is not possible for the lexical analyzer to tell whether wilhe is an undertaken function identifier or a misspelling keyword while. In this case lexical analyzer simply returns a token identifier for wilhe.
- If program uses variable names just differing in one or two characters, there is probability of occurrence of errors due to mistyping.
- Spelling errors situations – error recovery actions
  (1) **Extraneous character** : Deleting an Extra character.
  (2) **A missing character** : Inserting a missing character.
  (3) **A incorrect character** : Replacing an incorrect character by a correct character.
  (4) **Two adjacent transposed characters** : transposing two adjacent character.
- Suppose a situation arises in which lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of the remaining input. The simplest recovery strategy is "**panic mode**" recovery.
- In situation of extraneous character the simpler strategy is to see whether a prefix of

remaining input can be transferred into valid lexeme by just single error transformation but these techniques are not always useful.

- Lexical analyzer in some compilers make a copy of the source program with the error messages marked on it.
- It also takes care that there is no duplication of Error messages.

2.3 Block Schematic of Lexical Analyzer :

äää [University Exam : Dec. 2004 !!!]

- Lexical analyzer reads the source program character by character from the secondary storage but it is costly. Therefore, a block of data is first read into a buffer and then scanned by the lexical analyzer.
- It also reduces the amount of time in the lexical analyzer phase.
- Many source languages take time when the lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before announcing a match.
- As large amount of time is consumed in moving characters, specialized **buffering techniques** have been developed to reduce the amount of overhead required to process us input character. Many buffering scheme can be used like one buffering scheme, two buffer scheme etc. The one buffering scheme has some problems.
- Another technique is two buffer scheme. In two buffer scheme two buffers are scanned alternately. Each buffer is N character long, where N is the number of character on the block. It read N input character into each half of the buffer using one system read command. When one reaches the end of the current buffer, the buffer is filled.
- To maintain input buffer, lexical analyzer uses two pointers : a **lexeme beginning pointer** and **a forward pointer** to keep track of the portion of the input string scanned. The string of characters between the two pointers is the current lexeme.
- Initially both pointers point to the beginning of the lexeme. Once the next lexeme is determined, the forward pointer is set to the character at its right end. After the lexeme is processed, both pointers are set to the character immediately after the lexeme. Using this scheme, it can treat comments and white spaces as patterns that yield no token. This operation is shown in Fig. 2.3.1.

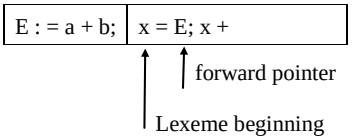


Fig. 2.3.1

- If the forward pointer is about to move fast the half way mark, the right half is filled with N new input character. If the forward pointer is about to move fast 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.

Example 2.3.1 : Code of advance fp

```
If (fp = = eof (buffer)) // end of first half
  {
      reload buffer 2;
      fp : = fp + 1;
  }
  else if (fp = eof (buffer)) // end of second
  {
      reload buffer;
      fp : = address of buffer /; // starting of buffer 1
  }
  else
  {
      fp = fp + 1;
  }
  fp = forward pointer
```

- This buffering scheme shown in code of Ex. 2.3.1 works quite well most of the time, but the amount of look ahead is limited and this limited look ahead may make it impossible to recognize token in situation where the distance traveled by forward pointer is more than the length of the buffer.
- For e.g.

  DECLARE (ARG1, ARG2, … ARGn) in PL/I program
- Using code from Ex. 2.3.1 requires two tests for each advance of the forward pointer. We can reduce the tests to one if we extend buffer half to hold a "sentinel" (eof) character at the end shown in Ex. 2.3.2.
- Using sentinels, we can write the code that performs only one test to see whether forward pointer points to the sentinel eof. If it reaches to end of buffer or end of the file the code performs move tests. As shown in Ex. 2.3.2.

Example 2.3.2 :

```
fp = fp + 1
 if (fp = = eof)
 {
     if (fp : : eob1)
     {
           reload buffer2;
           fp = fp + 1;
     }
     else if (fp = = eob2)
     {
           reload buffer1;
           fp : fp – 1 /;
     }                              *fp = forward pointer
     else
     terminate scanning
   }
```

Comment [a1]: Please change figure number according to figure sequence of the this second chapter

Comment [a2]: Please change figure number according to figure sequence of the this second chapter

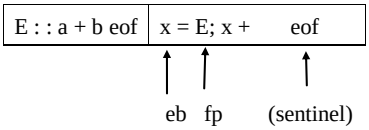


Fig. 2.3.2

2.4 Token Specification :

2.4.1 Alphabet :

A string ever some alphabet :

- The term alphabet denotes any finite set of symbols.
  e.g. {0, 1} are binary alphabet.
- Before declaration of token it checks for pattern match. Regular expressions are an important notation for specifying pattern. They represent pattern of strings of character.
- A string like an alphabet is a finite sequence of legal symbols drawn from that alphabet e.g. Compiler, Ulman etc. are strings over alphabet containing letter. We can define length of string as a number of symbols in the string.

2.4.2 Regular Expression :

- Regular expressions represent patterns of strings of characters. A regular expression may completely be defined by the set of strings. A regular expression over an alphabet is defined by following rules.
  1. ε (read as epsilon) is an regular expression.
  2. If a symbol 'u' is in alphabet, then u is a regular expression.
     /* 1 and 2 defines Simple Regular Expressions as ε and u where as more complex regular expressions are defined by applying unary or binary operations */
  3. If r and s are regular expressions over the alphabet then following more complex regular expressions can be obtained as
     (a) **r/s** or **r+s** is a regular expression. (Operation applied is **Union**)
     (b) **rs** is a regular expression. (Operation applied is **Concatenation**)
  4. If r is regular expression then
     (a) **r*** is regular expression. (Operation applied is **Kleen Closure**)
     (b) **(r)** is regular expression.
- Every regular expressions denote, language like {ε }, {a} by Regular Expressions **ε** and **a** (from above 1 and 2)
- Language L(r) U L(S) by Regular Expression **r/s** or **r+s**
- Language L(r)* by Regular Expression **r*.**
- Using definition of regular expression we may define a regular expression for identifier.
  letter (letter) digit) *

Regular Expression	Language of the Regular Expressoin
ε	{ ε }
A	{ a }
a+b	{a,b}
Ab	{ab}
A*	{ε,a,aa,aaa,aaaa,……}

2.4.3 Regular Definitions :

- For convenience, we can give names to regular expressions and define regular expressions using these names as if they are symbols. If Σ is as alphabet of basic symbol then regular definition is a sequence of definition of the form

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\vdots$$
$$\vdots$$
$$d_n \rightarrow r_p$$

When d_1 is a distinct name and each r_1 is regular expression over the symbols in $\Sigma \cup \{d_1, d_2 \ldots d_{i-1}\}$ that Σ is basic symbols and $d_1 \ldots d_{i-1}$ are defined names.

e.g. : In defining identifier using regular expression

letter → A | B | … | Z | a | b | … | z |

digit → 0 | 1 | … 19

id → letter (letter/digit)*

2.4.4 Notation Shorthand :

- To construct regular expressions some notation shorthand as follows :
  1. One or more instance : The unary postfix + operator
  2. Zero or one instance : The unary posfix ? operator
  3. Character classes : [A – Za – Z], [abc]

2.4.5 Construction of Lexical Analyzer :

1. Automatic generation of lexical analyzer :

- Lexical analyzers can be constructed in two ways.
- First method involves writing a program to do the lexical analysis.
- Another method uses automatic generation of lexical program which is faster.
- But with coding lexical analyzer is more efficient.
- For coding, lexical analyzer needs tokens and grammar using that tokens as

Example 2.4.1 :

stmt → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term | term

term → id | num
if → if
then → then
else → else
relop → < | < = | = | < > | > | > =
id → letter (letter/digit)*

Fig. 2.4.1

- Here the terminals are if, then, else
- Set of strings are defined by regular definition in Ex. 2.4.1.
- In programming tokens (which) are going to be declared matched by several different regular expressions e.g. if, else, while either which may lead to ambiguity. To resolve this, we must give preference to reserve word, if string is matched by reserve word by an identifier.
- In addition to this lexemes are separated by delimiters like white space, task, newlines. We have to define white space and if it match lexical analyzer will ignore that token and not return is to parse which strip out the while spaces. We can define white space as follows :

  delim → blank | tab | newline
  ws → delim + (this is rule for white spaces)

Comment [a3]: change this number accordingly

2.5 Transition Diagram :

- We want to construct a lexical analyzer that will identify the lexeme for the next tokens in the input buffer and produce a token and its attributes value. Before doing that we draw a translation diagram corresponding to each token as an intermediate step in the construction of lexical analyzer.
- Translation diagram is a directed graph with nodes representing states and edges representing translations on input symbols. A state is a representation of a portion of input seen so far. For each transition diagram, there is a start state signifying anticipation of the corresponding token and a final state signifying the end of the token.
- A transition diagram is useful in two ways. It serves as precise specification of token. It also keep track of information about characters that are seen as forward pointer fp scans the input. A state is a representation of the portion of input seen so far. Each edge leaving a state S has label which indicates the input characters that can next appear after the transition diagram has reached that state S. An edge labeled by character that is not indicated by any of the other edges leaving from state S.
- On reaching a state S, we advance the forward pointer and read the next input character. If this input character matches, the label of an edge from the current state, a transition is made between these two states. If it does not matches, the label of any of the edges from the current state, the transition diagram indicates a failure. Above transition diagram are

determine Fig. 2.5.1 shows transition diagram for identifier.

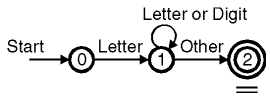


Fig. 2.5.1 (a)

Comment [a4]: change this number accordingly

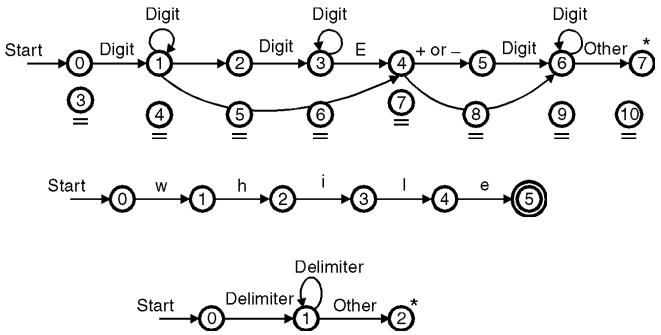


Fig. 2.5.1 (b) : Regular definition for unsigned number and its transition diagram

2.6 Converting Transition Diagram into Code :

- The lexical analyzer coding required by following Fig. 2.6.1. with reference to Fig2.5.1 (a) & (b)

```
nexttoken ( )
{
while(1)
{
        while(1)
        {    state =0; start= 0;
                switch (state)
                {
                Case '0' :
                    c = nextchar ( );
                    if (c = = ' ' || c = = '\t' || c = =  ' \n')
                    {
                            state = 0;
                            lexeme begin ++;
                    }
                    else if letter(c)
                    {
                            state = 1;
                    }
                    else state = fail;
                    {
                            break;
                    }
            Case 1 :  c= nextchar ( );
                    If (letter (c) or digit (c))
                            {       state = 1;
                            }
                            else
                            {       state = 2;
                            }
                            break;
```

Fig. 2.6.1 contd..

```
Case 2 :    forwardpointer = forwardpointer – 1 /* retract */
            return ( identifier, input [lexembegin forward pointer])
            break;
Case 3 : /* cases 3-10 here */
Case 11 :
            C = nextchar( );
            if (c == '=') state = 12;
            else if (c == '<') state = 13;
            else if (c == '>') state = 14;
            break;
Case 12 :
            return (rop, EQ);
            break;
Case 13 :
            C=nextchar ( );
            if (c == '=') state = 15;
            else if (c == '>') state = 16;
            else state = 17;
            break;

Case 14 :   c = nextchar ( );
            if (c== '=') state 19
            else    state 18;
            break;
Case 15 :   return (rop, LE);
            break;

Case 16 :    forwardpointer = forwardpointer-1;
             return (rop, NE);
            break ;
Case 17 :   forwardpointer = forwardpointer – 1
             return (rop, LT);
             break;
Case 18 :   forwardpointer = forwardpointer – 1;
……..       return (rop, GT);
            break;
```

Fig. 2.6.1

- The lexical analyzer is represented by the function nexttoken().
- This function returns the next token and its attributes.
- It traverses transition diagrams successively starting with the diagram beginning in state=0. The code reads a character from the input buffer and advances forward pointer using a function nextchar( ) if there is edge leaving a state. The control is then transferred to the code for the state pointed to by that edge i.e. next state.
- If current state indicates a token, the program returns that token. If it does not match with next state, it invokes fail function before return.

fail( )

```
{
        forwardpointer = lexeme beginning;
        Switch (start)
        case 0 : start = 3;
        case 3 : start = 11;
        case 11 : recover error ( );
        default : error message ( );
}
```

- It backtracks to the beginning of the lexeme and tries next diagram specified if the current transition diagram fails.

2.7 Automatic Generation of Lexical Analyzer :

- To generate a lexical analyzer two important things are needed. Firstly it will need a precise **specification of the tokens** of the language. Secondly it will need a specification of the **action to be performed on identifying each token**. For this operation several tools have been built which uses regular expression as an output, it generates a lexical analyzer. The particular tool of UNIX called **Lex** that has been widely used to specify lexical analyzer for a variety of languages.
- Lex is generally used as shown in Fig. 2.7.1.
- First, a specification of a lexical analyzer is prepared. This specification used to write program which is having extension ***.l*** (e.g. **first.*l*, or ex.*l***). This program is written in lex language and run through the Lex compiler to produce C code in lex.yy.c. The program lex.yy.c basically consists of a transition diagram constructed from the regular expressions of first *.l* or lex.*l*. (Finally lex.yy.c is run through the C compiler to produce object program a.out, and lexical analyzer transforms an input streams into a sequence of tokens.).

Comment [a5]: number

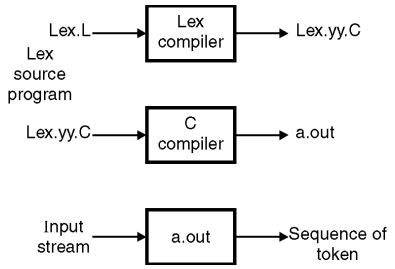


Fig. 2.7.1

2.7.1 Lex Specifications :

- A lex program consists of three parts :

  {declaration}

  % %

  {translation rules}

  % %

  {programmer subroutines}

- The first section of declaration includes declaration of variable, constants and regular definitions.
- Second section is for translation rules which consists of regular expression and action with respect to it. If regular definition is declared in declaration section, it uses that for regular expression. Every section end with % % symbol.
- The translation rules of a Lex program are statements of the form as follows:

  r {action;}

- Here, each r is a regular expression and action is a program fragment describing what action to be taken when pattern matches.
- The actions are written in C language.
- The third section holds whatever program subroutines are needed by the actions. These procedures can be compiled separately and loaded later with the lexical analyzer.

2.7.2 Lex Pattern :

Lex pattern are standard UNIX regular expressions using standard symbol which check the input stream (lexeme).

Standard regular expression	Matches	Example
c	Single character not operator	x
\c	Any character following the \ less its meaning and take literally	\ *
"S"	String S literally	"**"
.	Any single character except a newline (\n)	a . *h
^	Beginning of line	^abc
$	End of line	abc$
[S]	Any character in S	[abc], [A – Z]
[^S]	Any character except from S	[^abc]
r*	Zero or more occurrence	a*
r+	One or more occurrence	a+
r?	Zero or one r	0?
r {m, n}	m to n occurrence of r	a{1,5}
r1 r2	r1 then r2	ab
r1 | r2	r1 or r2	a : b
(r)	r	(a : b)
r1/r2	r1 when followed by r2	Abc/123

2.7.3 Lex Actions :

- Lex actions are C statements which are executed or actions are performed when regular expression pattern matches with lexeme. An action may be of single line C statement or multiple statement enclosed in {…} C brackets.

  For example :

```
%%
"india"
{
    printf ("India is great");
}
%%
```

- Here India is a string, if matches with lexeme action take place and print "India is great".
- Some important variables :

yylval : Global variable which returns more information about lexeme to parser with the value in lexeme.

yytext : The variable yytext carries point to the variable that we have been calling lexeme beginning that is a pointer to the first character of the lexeme.

- In declarations surrounded by % {and %} are declarations for manifest constants. A manifest constant is an identifier that is declared to represent a constant. Anything appearing between these brackets is copied directly into the lexical analyzer lex.yy.c and is not treated as part of the regular definitions or the translation rules.
- In third subroutine section we can write a user subroutines its option to user e.g. yy/ex is a function automatically get called by compiler at compilation and execution of lex program or we can call that function from the subroutine section.

2.7.4 Some Small Program Using Lex :

Program 1 : Remove white spaces from input stream :

```
%%
[ \t];
%%
save by whitespace./
Syntax to execute the program is
$ lex whitespace.l
$ cc lex.yy.c –/ /                    (–/ /  link lex library)
$ a.out                               (provide input string to this file a.out)
Input is :
India is great.
```

Output is :

```
Indiaisgreat.
```

Regular expression [\t] matches tabs and space. If it does not match, no action performed and lexeme beginning pointer is incremented to read next lexeme.

Program 2 : Find and replace :

```
%{    # include <stdio.h>
      # include <string.h>
      char find [10];
      char replace [10];
      File *fout;
%}
%%
[a-z A-Z]([a-ZA-z][0-9])*          {
      if (strcmp(yytext, find) = = 0)
            fprintf(fout, “%S”, replace)
      else
            fprintf(fout, “%S”, yytext)
}
%%
main(int argc, char **argv)
```

```
{
        if(argc>1)
        {
                FILE *fin;
                fin = fopen(argv[1], "r");
                if (!fin)
                        printf (" I/P File can not open");
                printf("Enter string to find");
                scanf("%S", find);
                printf("Enter string to replace by");
                scanf("%S", replace);

                fout = fopen("output.c", "w");
                if(!fout)
                        printf("Output file not created");
                yyin = fin;
                yylex( );
        }
}
```

- File saved with the filename findreplace.l.
- The main function uses command line arguments, to read name of input file of run time and save new replaced string output in "output.c".
- yyin is special pointer variable, which points to input stream. Input is entered through keyboard. That time it point to standard input output variable. In this program input read from a separate file i.e. yyin = fin.
- In second section regular expression matches string and perform function [a – z] [A – Za – Z.0-9]. It finds string and replaces it by required string.

lex.yy.c.

```
$ lex findreplace./
$ cc lex.yy.c -//
$ a.out input.c                    // input file as command line argument
```

Output :

```
Input  : input.c          //input file
Output : output.C         // output file
        Contents of Files
        // input.c
        We are going to Pune.
        $ a.out input.c
        Enter the string to find : Pune
        Enter the string to replace by : Nasik

        $ Vi out.c
        We are going to Nasik.
```

Program 3 : Change case using regular definition in declaration section and use that declaration in defining RE.

```
%{    # include<stdio.h>
      # include<string.h>

      int buff = 0;
      int len = 0;
      int choice= 0;
      int i = 0;
      int case=0;
%}
LET [a-zA-Z]
DIG [0-9]
%%
{LET} ({LET}{DEG})*     {
                        switch(choice)
                        {
                        case 1 :
                                len = strlen(yytext);
                                for(i = 0; i<len; i++)
                                {
                                        buff = yytext / [i];
                                        if(buff <= 122 && buff > = 97)
                                        else
                                        {       buff = buff + 32;
                                                yytext[i] = buff;
                                        }
                                }
                                break;
                                case 2 :
                                        for(i = 0; i < len; i++)
                                        {
                                                buff = yytext [i];
                                                if (buff < = 90 && buff > = 65)
                                                else
                                                {       buff = buff – 32;
                                                yytext[i] = buff;
                                                }
                                        }
                                break;
                                case 3 :
                                                buff = yytext(0);
                                                if(buff <= 122 && buff > = 97)
                                                {
                                                buff = buff – 32;
                                                yytext[0] = buff
                                                }
```

```
                                        break;
                                }
                                printf("%S", yytext);

%%
int main(int argc char **argv)
{
        FILE *file;
         if(argc > 1)
        {       file = fopen(argv [1], "r")
                printf("1.Lower, 2.Upper, 3.Title");
                scanf("%d", &case)
                yyin = file;
                yylex( );
        }
 return(0);
}
```

Program 4 : Lexical Analyzer

```
 %{     # include <stdio.h>
%}
LET [a-z A-Z]
DEG [0-9]
%%
[ \t]+ ;
int/float/char/if/else/do/while/break/continue/double/signed/unsigned/long/for/switch     {
                                        printf("Keyword Found; %s", yytext)                 }
main"("")"/exit/getch"("")"             {       printf("Function is Found %s", yytext)      }
{LET}[{LET} {DEG}]*                     {       printf("Identifier found : %s", yytext);    }
%[sdf]                                  {       printf("For mat symbol of identifier : %s yytext");  }
{DIG}+                                  {       printf("Constant is %s", yytext);           }
"/*".* "/*/"                            {       printf("Commentt : %s", yytext);            }
"{" / "}" / "(" / ")" / "[" / "]" / ";" / "," {       printf("Delimiters : %s", yytext);    }
"+" / "–" / "/" / "*"                   {       printf ("opertor : %s", yytext);            }
" ="                                     {      printf("Assignment operator : %s", yytext);  }
"<" / ">" / "< =" / "> =" / "= ="       {       printf ("Relational operator : % S", yytext);}

%%
main(int argc, char ** argv)
 if(argc > 1)
{
        FILE *fp;
        fp = fopen (argv[1], "r");
        if (!fp)
                { printf("file error"); }
```

```
        yyin = fp;
        yylex( );
}
```

2.8 Finite Automata :

- In compilation of regular expression it constructs generalized transition diagram called finite automaton. A finite automaton can be deterministic or non-deterministic.
- "Non-Deterministic Finite Automaton" have more than one transition out of state may be possible or the same input symbol.
- Both finite automata accepts a sequence of input characters and performs a sequence of action if the input string is a valid sentence in the language.
- Thus both can recognize exactly what regular expressions can denote. However, there is a time space trade off; while deterministic finite automata can lead to faster recognizers than non-deterministic automata.
- Regular Expressions can be converted in to DFA (Deterministic Finite Automata) and NFA (Non-deterministic Finite Automata) by using algorithms .
- First let us see conversion of regular expression into a non-deterministic automation.
- For example, regular expression (a/b)*abb consisting of the set of all strings of a's and b's ending in abb.

2.8.1 Non-deterministic Finite Automata :

- NFA consist of
  1. a set of states S
  2. a set of input symbols Σ
  3. a transition function between states δ
  4. a state So that is distinguished as the starting state
  5. a subset of set of states F distinguished as accepting final states.
- An NFA can be represented diagrammatically by a labeled directed graph, called a transition graph, in which the nodes are the state and the labeled edges represent the transition function. This graph looks like a transition diagram, but the same character can label two or more transitions out of one state. For (a/b)*abb.a transition graph is shown in Fig. 2.8.1.

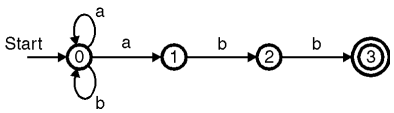


Fig. 2.8.1

In above NFA, transition graph :

1. {0, 1, 2, 3} are set of states.

2. {a, b} are input symbol.
3. This transition function can be implementing transition table that is a row for each state and column for each input symbol and a transition table for the NFA of Fig. 2.8.1 is shown in Fig. 2.8.2.

State	a	B
0	{0, 1}	{0}
1	--	{2}
2	--	{3}
3	--	--

Fig. 2.8.2 : Input symbol

- An NFA accepts an input string y if and only if there is some path in the transition graph from start state to accepting state.

2.8.2 DFA (Deterministic Finite Automata) :

1. No state has an ε transitions.
2. For each state S and input symbol 'a' there is at most one edge labeled as 'a' coming out of S.
   - When DFA represented by transition diagram all outgoing edges are labeled with an input character and no two edges leaving a given state have the same input symbol.
   - We can use transition table to represent transition diagram like NFA.
   - Each entry in transition table has only single state. If the string is valid according to language of finite automata there exists unique path from starting state to final state of Automata for that string. Therefore it is very easy to decide whether given input string is valid or not I in Deterministic Finite Automata than that of Non-deterministic finite automata .
   - In Fig. 2.8.3 we have seen NFA for (a/b)*abb, lets see a transition diagram of DFA for the same regular expression.

Comment [a6]: number

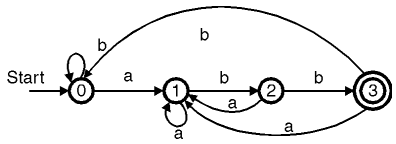


Fig. 2.8.3

   - In NFA, it has two transitions from state 0 on input 'a' that is it may go to state 0 to 1. But in DFA there is one state 1 which have transition on symbol 'a'.
   - NFA can have many more states than the equivalent DFA and it is difficult to decide which path to follow in the NFA. Because there can be more than one output state for same input character and current state which makes it difficult to do decide which state should be selected as next state.
   - In case of NFA to decide that a particular string is not accepted by NFA we have to ensure that none of all possible paths from starting state can reach to final state of NFA for the given string.

- Thus to summarise
  - NFAs are easy to obtain from RE
  - NFAs are costlier programs than DFA (as NFA may have more states than that of its equivalent minimized DFA)
  - DFAs can easily and quickly decide whether given string is valid or not according to language.
  - Therefore RE are converted to NFA which are then converted to their equivalent minimized DFA.
  - Note: For every conversion an algorithm is used.

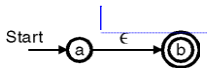
2.8.3 Construction of an NFA from Regular Expression :

- "**Thompson's construction**" is an algorithm used to construct as NFA from a regular expression. The algorithm uses the syntactic structure of the regular expression in to guide the construction process. Here, important think is how to construct automata for expression containing an alternation, concatenation or Kleene closure operator.
  e.g. (1) Alternation (R/S)
  (2) Concatenation (R)S
  (3) Kleen's closure R
- The construction of NFA using Thompson's algorithm for above regular representations are introduces at most two new states, so whatever NFA constructed for a regular expression has at most twice as many states as there are symbols and operator is the regular expression.

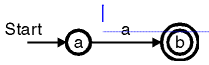
Thompson's construction algorithm :

- **Input** : A regular expression R
- **Output** : An NFA accepting L(R)
- **Method** : First breaks R into its construction sub expressions. Then construct NFA for each basic symbol r. By the syntactic structure of the regular expressions x, we combine these NFA inductively and obtain the NFA for the entire expression.
- NFA has exactly one start state, no edge enter to it, and no edge leaves the final state.
- To construct NFA for each symbol rules :

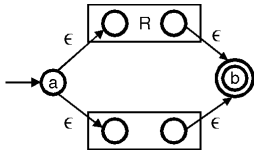
(1) For ε, construct the NFA :



(2) For symbol 'a' of regular expression, the NFA :



(3) For alternation (R)/(S), the NFA for regular expression R and S :

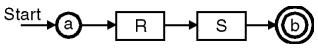


Comment [a7]: letters in circle should be capitalA,B

Comment [a8]: Letters in circle should be capital as A,B

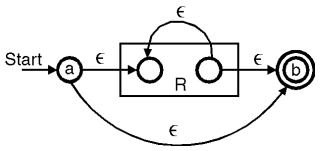
Here there is transition on E from a to one start of R and S. And transition on E to accepting state. This two state are not start or accepting states of N(R/S).

(4) For (R) · (S) Concatenation :



Here R becomes the start state of composite NFA an accepting state of S becomes accepting state of the NFA.

(2) For R*, Kleen's closure :



New start and accepting state a and b. In the NFA, we can travel from a to b directly along with transition of ε, which representing R* zero occurrence of R, we can go a to b with R number of times.

In constructing each state, note that we construct new state that has unique name. Also some important properties like :

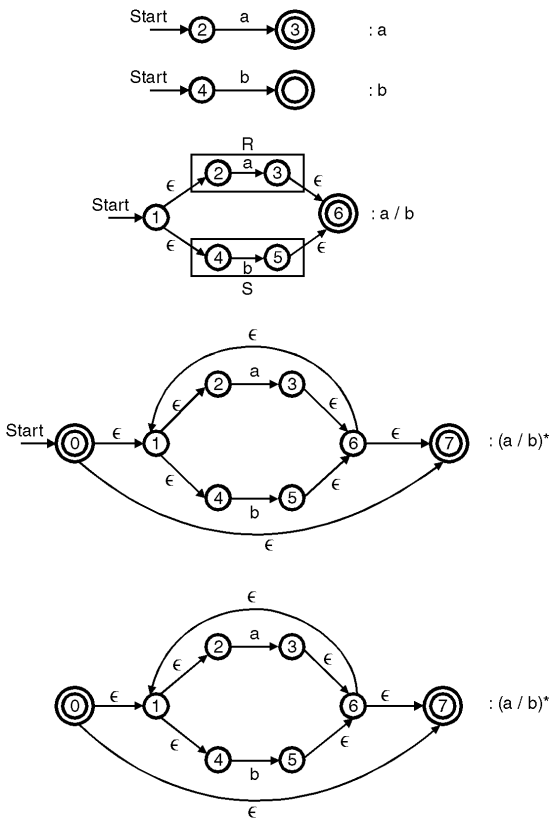
(1) No two states of any component NFA can have the same name.

(2) NFA for any R has exactly one start state and one accepting state.

(3) N(R) has at most twice as may states as the number of symbol and operation in R.

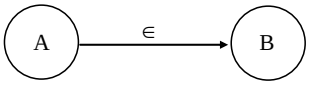
(4) Each state of the NFA for a regular expression R has either one outgoing transition on symbol or at most two outgoing E transitions.

Ex: R=(a/b)* can be converted into epsilon NFA as



2.8.4 Conversion of NFA to DFA :

(1) Find ∈ – Closure of each state of NFA. Now Epsilon transition is as



∈ means empty, thus machine enters state A, it also goes to state B on empty on null input i.e. A and B can be considered as one state.

(2) Identify a group of NFA states to represent the initial state of the DFA. Let us call this state S of DFA, which is the grouping of states $\{S_i\}$ of the NFA.

(3) Identify a group of NFA states; which can be reached from same S_i along a transition marked say 'a'.

(4) Repeat above steps until we reach the stage when no new states can be added to the DFA.

For above example.

(5) $\in$ – closure (0) $\rightarrow$ {1, 2, 4, 7} = A

(A, a) = {3, 6, 7, 1, 2, 4} = B

(A, b) = {5, 6, 7, 1, 2, 4} = C

(B, a) = {3, 6, 7, 1, 2, 4} = B

(B, b) = {5, 6, 7, 1, 2, 4} = C

(C, a) = {3, 6, 7, 1, 2, 4} = B

(C, b) = {5, 6, 7, 1, 2, 4} = C

(6) Starting state of NFA is state 0 and it's $\in$-closure i.e. {1, 2, 4, 7} because on $\in$ symbol from 0 next state transition next states are 2 and 4 and collectively its token state A of DFA.

(7) From this state A on 'a' input symbol transition to state 3, now E closure of 3 is 6, 1, 2, 4, 7. Thus 1, 2, 4, 6, 7 are collectively new state B of DFA.

(8) Similarly {5, 6, 7, 1, 2, 4} from C of DFA of 'A' on after this B on 'a' and B on 'b', C on 'a' and C on 'b' not generated new states i.e. DFA table is prepared is as follows :

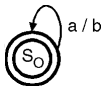
State	Input	
	a	**b**
A	B	C
B	B	C
C	B	C

Next step is to minimize this DFA.

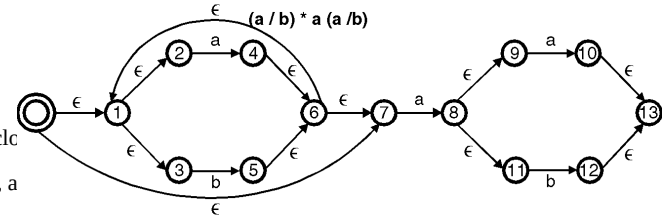
- Find the equivalent states and keep it is one group. The remove one of that and replace over its equivalent state.
- With respect to example and its DFA table B, C are state which is equivalent i.e. its get remove from table and replace by A as no state is remaining for transitions.

State	Input	
	a	**b**
A	A	A

Therefore DFA is as follows :



Example 2.8.1 : Consider following Epsilon NFA for Regular Expression (a/b)*a(a/b)



E clc

(A, a = B

(A, b = C

(B, a) = {4, 6, 7, 1, 2, 3, 8, 9, 10, 11, 13}
= D

(B, b) = {5, 6, 7, 1, 2, 3, 12, 13}
= E

(C, a) = {4, 6, 7, 1, 2, 3, 8, 9, 11}
= B

(C, b) = {5, 6, 7, 8, 1, 2, 3, 9, 11}
= F

(D, a) = {4, 6, 7, 1, 2, 3, 8, 11, 9, 10, 13}
= D

(D, b) = {5, 6, 7, 1, 2, 3, 12, 13}
= E

(E, a) = {4, 6, 7, 9, 8, 1, 2, 3, 11}
= B

(E, b) = {5, 6, 7, 1, 2, 3}
= C

(F, a) = {4, 5, 7, 1, 2, 3, 8, 9, 11, 10, 13}
= D

(F, b) = {5, 6, 7, 1, 2, 3, 12, 13}
= E

	a	**b**	
A	B	C	(AE) (BD) (C) (F)
B	D	E	(A) (B) (C) (F)
C	B	F	
D	D	E	
E	B	C	After reducing
F	D	E	

	a	**b**
A	F	A
C	F	F
F	F	A

Examples :

(3) (a/b)* abb (a/b)*

(4) (a*/b*)*

(5) (a*ba/aba*)

(6) ((a+(b/c)*)*+(a+b*)*)*

2.9 Role of the Finite Automata in the Compiler :

Consider example given for the regular expression of identifier.

- Now regular expressions are used as token describer in lexical Analyzer compiler.
- Lexical Analyzer has to perform major task of Token Identification from the input source program. To identify token lexical analyzer needs token identifier.
- Now as regular expression can be converted into finite automata by using algorithm we can obtain finite automata working as token identifier from regular expression which is token describer.
- Thus finite automata play an important role in lexical analyzer.
- The regular expression can be converted into e-NFA which can be converted into DFA which can be again minimized. This minimized DFA can be used as Token Recognizer in Lexical analyzer.
- Finite Automata is minimized to produce shorter program
- Illustrate RE and FA for identifiers.

Review Questions

Q. 1 What are the advantages of separating lexical analysis form syntax analysis ?

Q. 2 Write a short note on LEX.

Q. 3 Assuming suitable data structure for representing finite automation, write an algorithm (in pseudo 'C') for converting a NFA to its equivalent DFA.

Q. 4 State the importance of lookahead in lexical analysis.

Q. 5 Explain the importance of ordering the lexical rules.

Q. 6 In Lex notation, give regular expression to recognize 'C' comments.

2.10 University Questions and Answers :

May 2004 – Total Marks 8

Q. 1 Discuss the role of finite automata in compiler. **(Section 2.9)** **(8 Marks)**

Dec 2004 – Total Marks 16

Q. 2 Write a Lex rule for " An identifier that starts and ends with digit and maximum length of 20 characters. **(Section 2.7.1)** **(6 Marks)**

Q. 3 State what strategy Lex should adopt if keywords are not reserved words. **(Section 2.7.2)** **(4 Marks)**

Q. 4 Write a short note on input buffer with lexical anlyzer. **(Section 2.3)** **(6 Marks)**

❑❑❑