UNIT – 5

Structure of Lex Program

1) Definition section
2) Rules "
3) User - subroutine section.

Simple LEX pgm

```
% %                    → file.l
    . | \n  ECHO ;
% %
```

└ Prints whatever is given as input

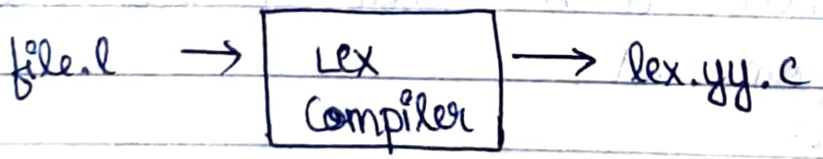
Compilation & Execution of LEX pgm.

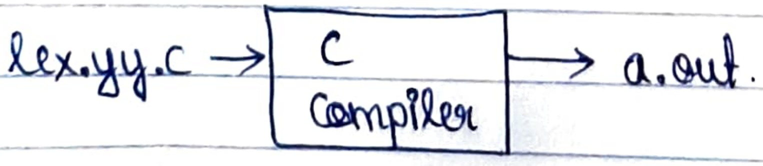
file.l

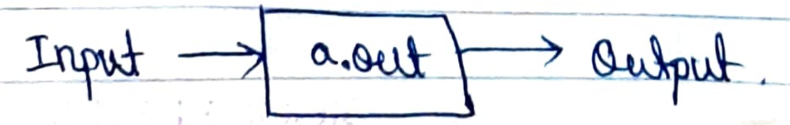
```
$$ lex file.l ↵
$$ cc lex.yy.c -ll ↵
$$ ./a.out ↵
Belagavi ↵
Belagavi
```

→ Command line argument

→ Output







Eg:
```
%{

%}
%%
    is|am|are   { printf("%s is a verb\n", yytext); }
    [a-z]+      { printf(" %s is nota verb\n", yytext); }
%%

main()
{
      yylex()
}
```

<u>Regular Expression</u>

^ → if used outside the square brackets, it starts matching from the beginning.

[^0-9] → Inside [] it acts as negation i.e except 0-9 other nos.

$ → similar to ^ outside [] but starts matching from end.

| → or

~~1) Write a~~

Imp Q) Write a lex pgm to count the no. of words, characters & lines. from a given input file.

→
```
%{
  #include<stdio.h>
    int w=0, c=0, l=0;
%}
%%
    [\n]     {l++; c++;}
    [^ \t\n]+    {w++; c= c+yyleng;}
    .          {c++;}
%%
```

Space

Built in variable that contains length of the word

From a file

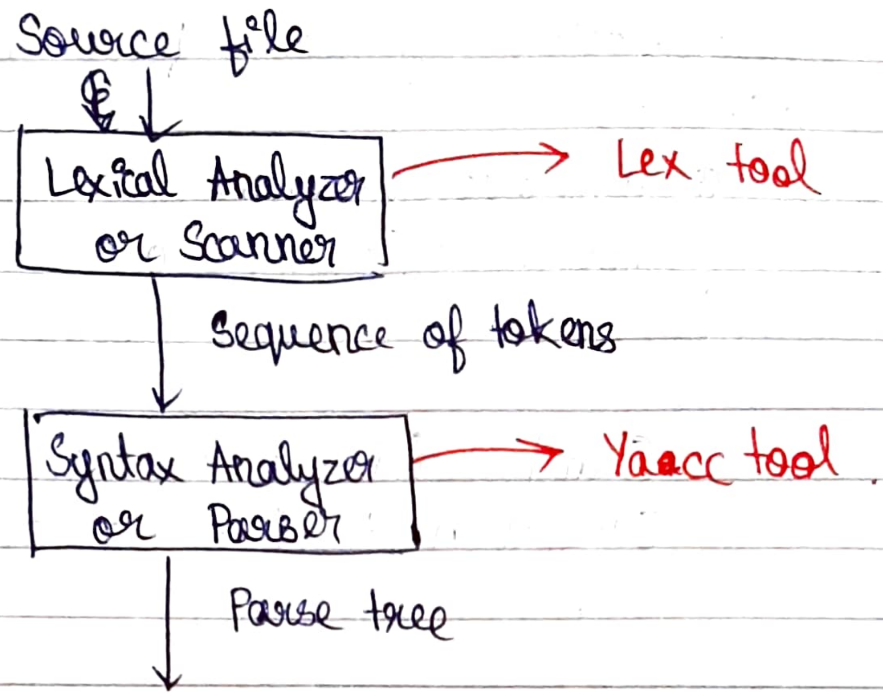
```
main(int argc, char *argv[])
{
    FILE *fp;
    ~~printf~~ fp = fopen(argv[1], "r");
    yyin = fp;
    yylex();
    printf("No. of words are : %d \n", w);
      "    ("  "  lines          "       ", l);
      "    ("  "  characters     "       ", c);
}
```

or write filename.

Without file.

```
main()
{
     yylex();
     printf(-----);
}
```

Structure of Compiler



YA~~A~~CC → Yet Another Compiler Compiler

Structure of YACC pgm

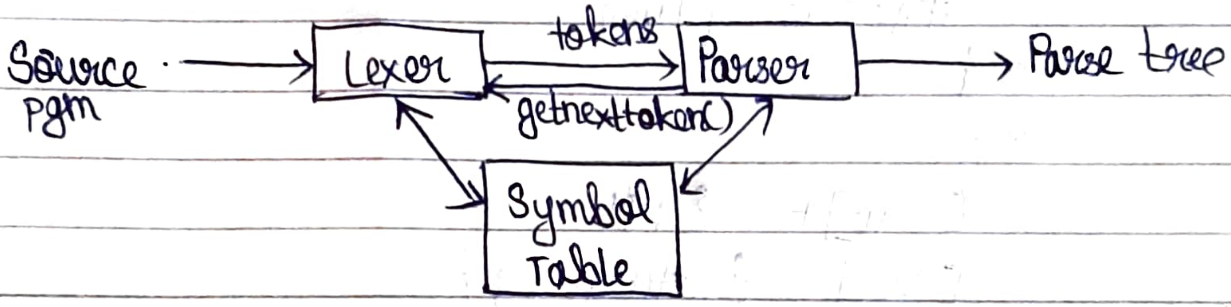
1) Definition section
2) Rules " → rule action
3) Sub-routine "

rule → RE (for Lex)
rule → Grammar (for YACC)

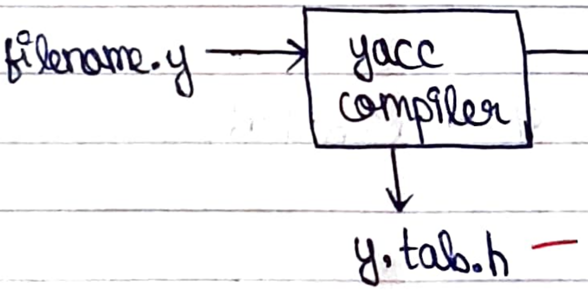
```
%{
%}
% token ---
~~%token~~
%%
%%
```

```
main()
{
        yyparse();
}
```

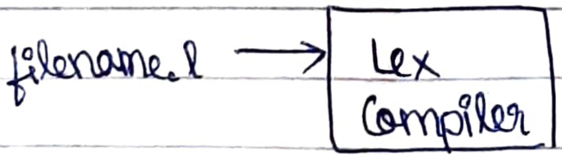
Parser Lexer Communication



whenever a new token is encountered it is added to the symbol table.



— is a header file that contains all the tokens & their def^n.



Tockens are same as terminals.

1) Write a YACC pgm to recognize the language $L = \{ a^n b^n \mid n \geq 1 \}$

→ $S \rightarrow aSb \mid ab$

Lex program ~~yacc program~~

```
%{
    #include<stdio.h>
    #include "y.tab.h"
%}
%%
   [ \t]+   {;}
   [\n]    {return 0;}
   [a]   {return A;}
   [b]  { return B;}
%%
```

yacc program $S \rightarrow aSb \mid ab$

```
%{
%}
% token   A   B
%%
   S : ASB
     | AB
     ;
%%
yyerror()
{
      printf("Invalid\n");
```

```
    exit(0);
}
main()
{
    printf(" Enter the input \n");
    yyparse();  → It will internally call yylex.
    printf(" Valid \n");
}
```

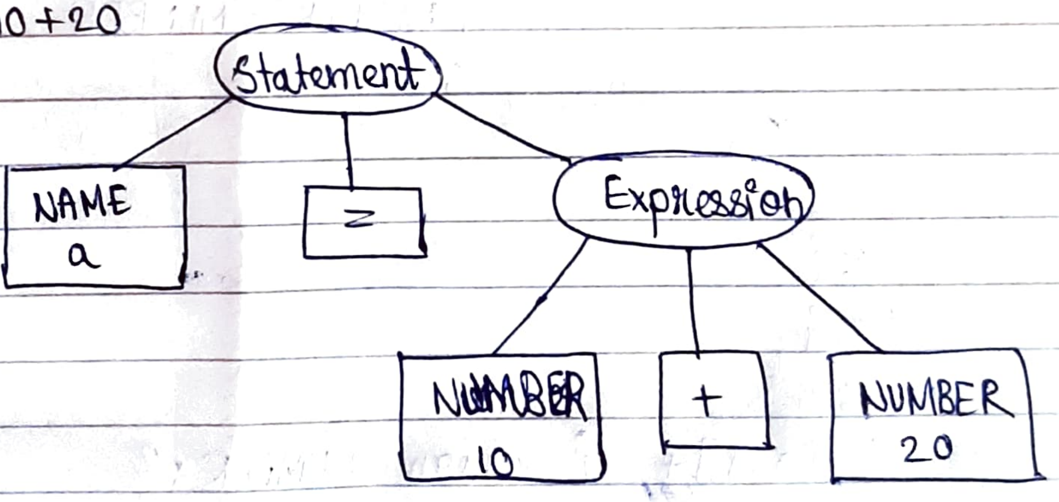
If error occurs then it will call yyerror func^n.

Grammars (CFG)

Statement → NAME = expression

expression → NUMBER + NUMBER | NUMBER − NUMBER.

Eg: a = 10 + 20



Recursive
```
expression → expression + NUMBER
           | expression − NUMBER
           | NUMBER.
```

Imp #

Shift / Reduce Parsing

No. of symbols are reduced.

Eg
```
a = 10+20
a = expression.
```

YACC Parser

% token A B.

Eg:
```
%token   NAME   NUMBER
 %%
    Statement : NAME`='expression
                   ;
    expression : NUMBER`+'NUMBER
                | NUMBER`-'NUMBER
                ;

 %%
```

Tokens if used directly in YACC, they should be in ` '

```
%{
%}
%%
  [0-9]+   { return NUMBER;}
  [a-z]+   { return NAME ;}
  [ \t]+   { ;}
  [ \n }   - - -
    .      { return yytext[0]; }
%%
```