Unit III : LEX Tool

Course : Language Processor and Compiler Construction

Manisha Mali
manisha.mali@viit.ac.in

BRACT'S, Vishwakarma Institute of Information Technology, Pune-48
(An Autonomous Institute affiliated to Savitribai Phule Pune University)
(NBA and NAAC accredited, ISO 9001:2015 certified)

Department of Computer Engineering

Contents

- **Phase structure of Compiler and entire compilation process.**
- **Lexical Analyser**
- **The Role of the Lexical Analyzer**
- **Input Buffering**
- **Specification of Tokens**
- **Recognition Tokens**
- **Design of Lexical Analyzer using Uniform Symbol Table, Lexical Errors**
- **LEX: LEX Specification, Generation of Lexical Analyser by LEX.**

Outline

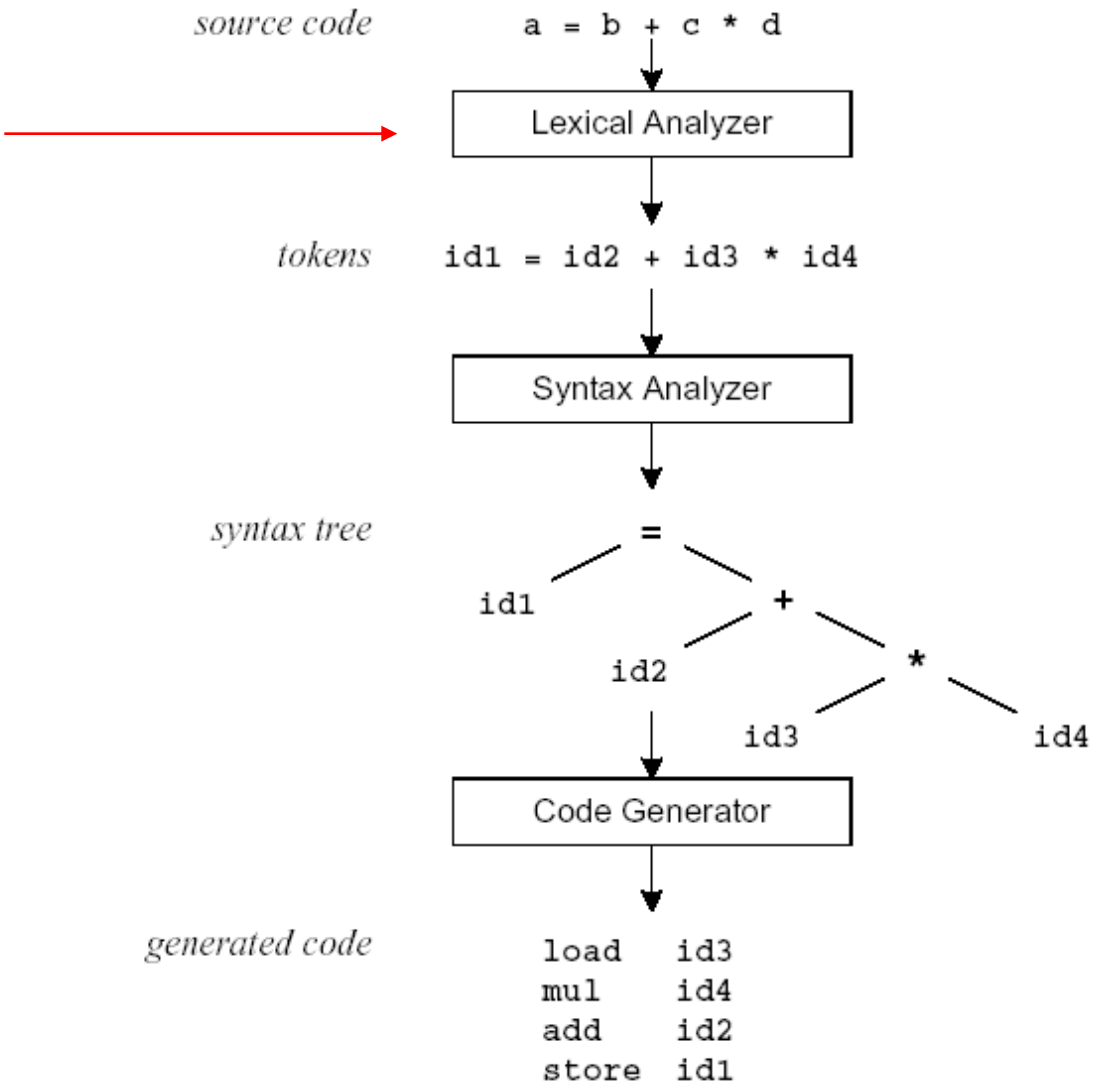
- References.
- Lex:
  - Theory.
  - Execution.
  - Example.
- Yacc:
  - Theory.
  - Description.
  - Example.
- Lex & Yacc linking.

Reference Books

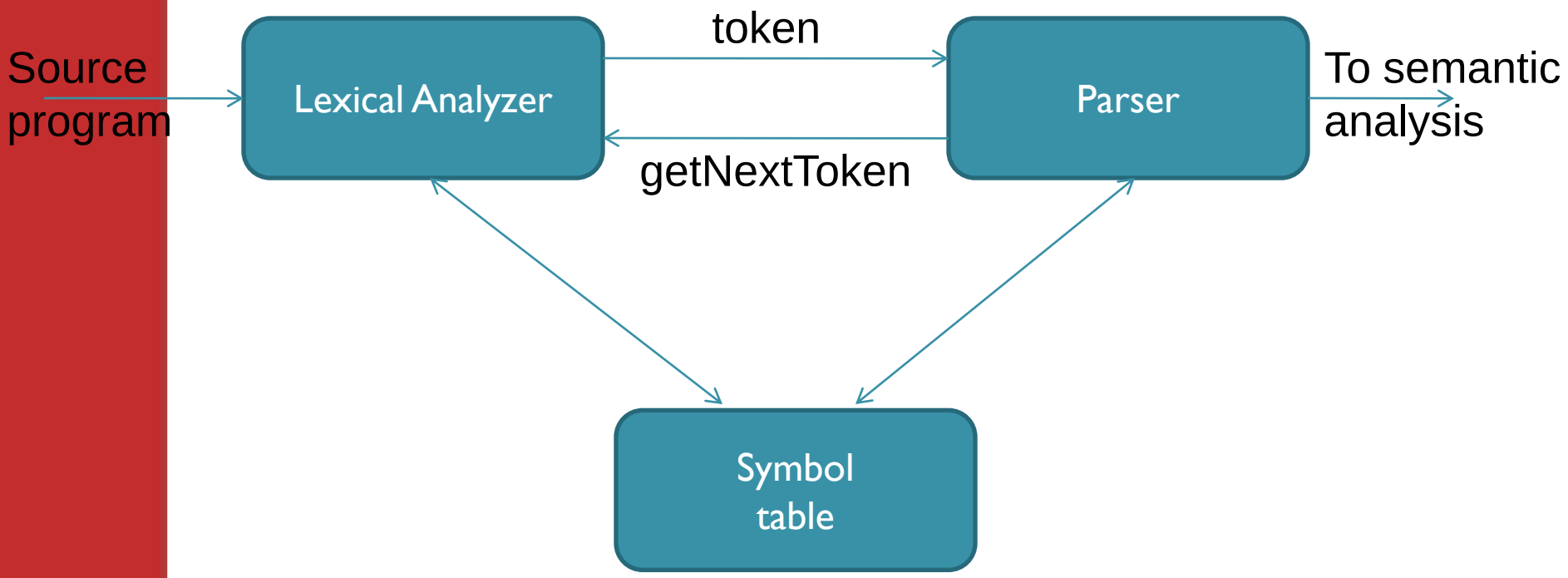
- lex & yacc, 2nd Edition
  - by John R.Levine, Tony Mason & Doug Brown
  - O'Reilly
  - ISBN: 1-56592-000-7

- Mastering Regular Expressions
  - by Jeffrey E.F. Friedl
  - O'Reilly
  - ISBN: 1-56592-257-3

Compilation Sequence



The role of lexical analyzer



Lex

- lex is a program (generator) that generates lexical analyzers, (widely used on Unix & Linux).
- It is mostly used with Yacc parser generator.
- Written by Eric Schmidt and Mike Lesk.
- It reads the input stream (specifying the lexical analyzer) and outputs source code implementing the lexical analyzer in the C programming language.
- Lex will read patterns (regular expressions); then produces C code for a lexical analyzer that scans for identifiers.

What is Lex?

- The main job of a *lexical analyzer (scanner)* is to break up an input stream into more usable elements (*tokens*)

```
a = b + c * d;
```

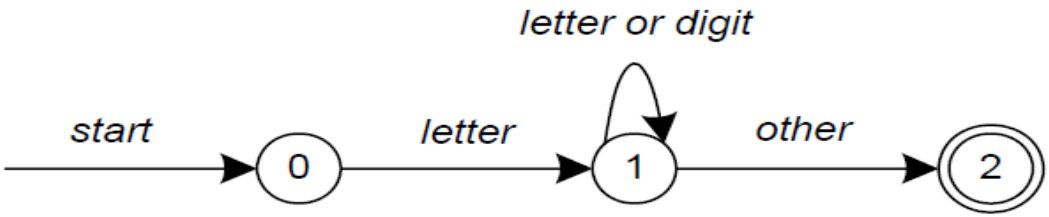
ID ASSIGN ID PLUS ID MULT ID SEMI

- Lex is an utility to help you rapidly generate your scanners

Lex

- A simple pattern: **letter(letter|digit)***

- Regular expressions are translated by lex to a computer program that mimics an FSA.
- This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits.

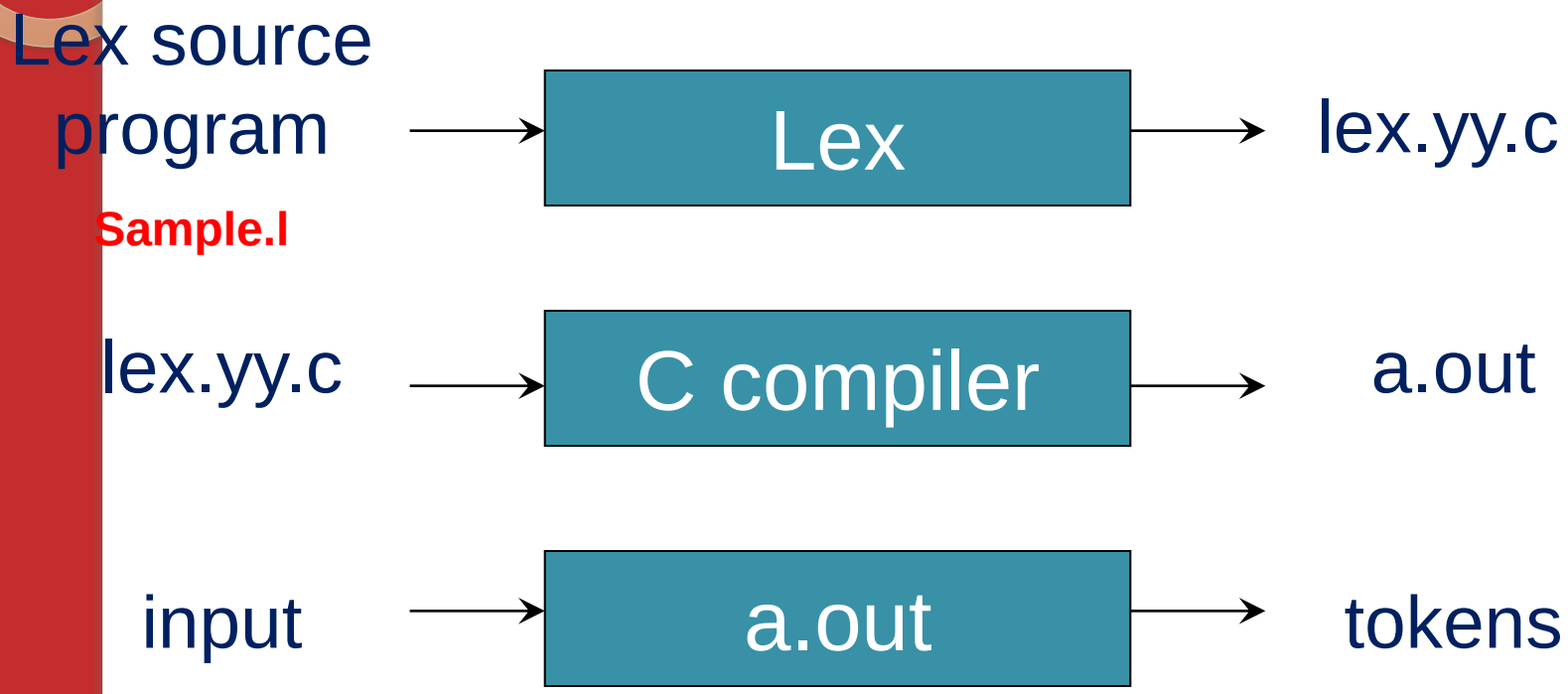


Lex

```
start:   goto state0

state0:  read c
         if c = letter goto state1
         goto state0

state1:  read c
         if c = letter goto state1
         if c = digit goto state1
         goto state2

state2:  accept string
```

- Some limitations, Lex cannot be used to recognize nested structures such as parentheses, since it only has states and transitions between states.
- So, Lex is good at pattern matching, while Yacc is for more challenging tasks.

An Overview of Lex



Usage

- To run Lex on a source file, type
  `lex scanner.l`
- It produces a file named lex.yy.c which is a C program for the lexical analyzer.
- To compile lex.yy.c, type
  `cc lex.yy.c -ll`
- To run the lexical analyzer program, type
  `./a.out < inputfile`

Lex

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Pattern Matching Primitives

Lex

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\-z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a|b]	one of: a, |, b
a|b	one of: a, b

- Pattern Matching examples.

Arbitrary Character .

- To match almost character, the operator character . is the class of all characters except newline

- `[\40-\176]` matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde~)

Precedence of Operators

- Level of precedence
  - Kleene closure (*), ?, +
  - concatenation
  - alternation (|)
- All operators are left associative.
- Ex: `a*b|cd*` = `((a*)b)|(c(d*))`

Lex

……..Definitions section……

%%

……Rules section……..

%%

…….C code section (subroutines)……..

- **The input structure to Lex.**

```
%%
      /* match everything except newline */
.     ECHO;
      /* match newline */
\n    ECHO;

%%

int yywrap(void)  {
      return 1;
}

int main(void)  {
      yylex();
      return 0;
}
```

•Echo is an action and predefined macro in lex that writes code matched by the pattern.

Definitions Section

- The definitions section contains declarations of simple name definitions to simplify the scanner specification.
- Name definitions have the form:

```
name definition
```

- Example:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

Rules Section

- The rules section of the lex input contains a series of rules of the form:

```
pattern action
```

- Example:

```
{ID} printf( "An identifier: %s\n", yytext );
```

- The *yytext* and *yylength* variable.
- If action is empty, the matched token is discarded.

Action

- If the action contains a '{', the action spans till the balancing '}' is found, as in C.
- An action consisting only of a vertical bar ('|') means "same as the action for the next rule.“
- The *return* statement, as in C.
- In case no rule matches: simply copy the input to the standard output (A default rule).

Precedence Problem

- For example: a “<“ can be matched by “<“ and “<=“.
- The one matching most text has higher precedence.
- If two or more have the same length, the rule listed first in the lex input has higher precedence.

User Code Section

- The user code section is simply copied to *lex.yy.c* exactly.
- The presence of this section is optional; if it is missing, the second %% in the input file may be skipped.
- In the definitions and rules sections, any indented text or text enclosed in %{ and %} is copied exactly to the output (with the %{ }'s removed).

Lex

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Lex predefined variables.

Review of Lex Predefined Variables

Name	Function
`char *yytext`	pointer to matched string
`int yyleng`	length of matched string
`FILE *yyin`	input stream pointer
`FILE *yyout`	output stream pointer
`int yylex(void)`	call to invoke lexer, returns token
`char* yymore(void)`	return the next token
`int yyless(int n)`	retain the first n characters in yytext
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`ECHO`	write matched string
`REJECT`	go to the next alternative rule
`INITAL`	initial start condition
`BEGIN`	condition switch start condition

Lex – Lexical Analyzer

- Lexical analyzers **tokenize** input streams
- Tokens are the **terminals** of a language
  - English
    - words, punctuation marks, …
  - Programming language
    - Identifiers, operators, keywords, …
- Regular expressions define **terminals/tokens**

Lex Source Program

- Lex source is a table of
  - regular expressions and corresponding program fragments

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

Example

```
digit     [0-9]
letter    [A-Za-z]
%{
     int count;
%}
%%
     /* match identifier */
{letter}({letter}|{digit})*          count++;
%%
int main(void) {
     yylex();
     printf("number of identifiers = %d\n", count);
     return 0;
}
```

- **Whitespace must separate the defining term and the associated expression.**
- **Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with "%{" and "%}" markers.**
- **substitutions in the rules section are surrounded by braces ({letter}) to distinguish them from literals.**

User Subroutines Section

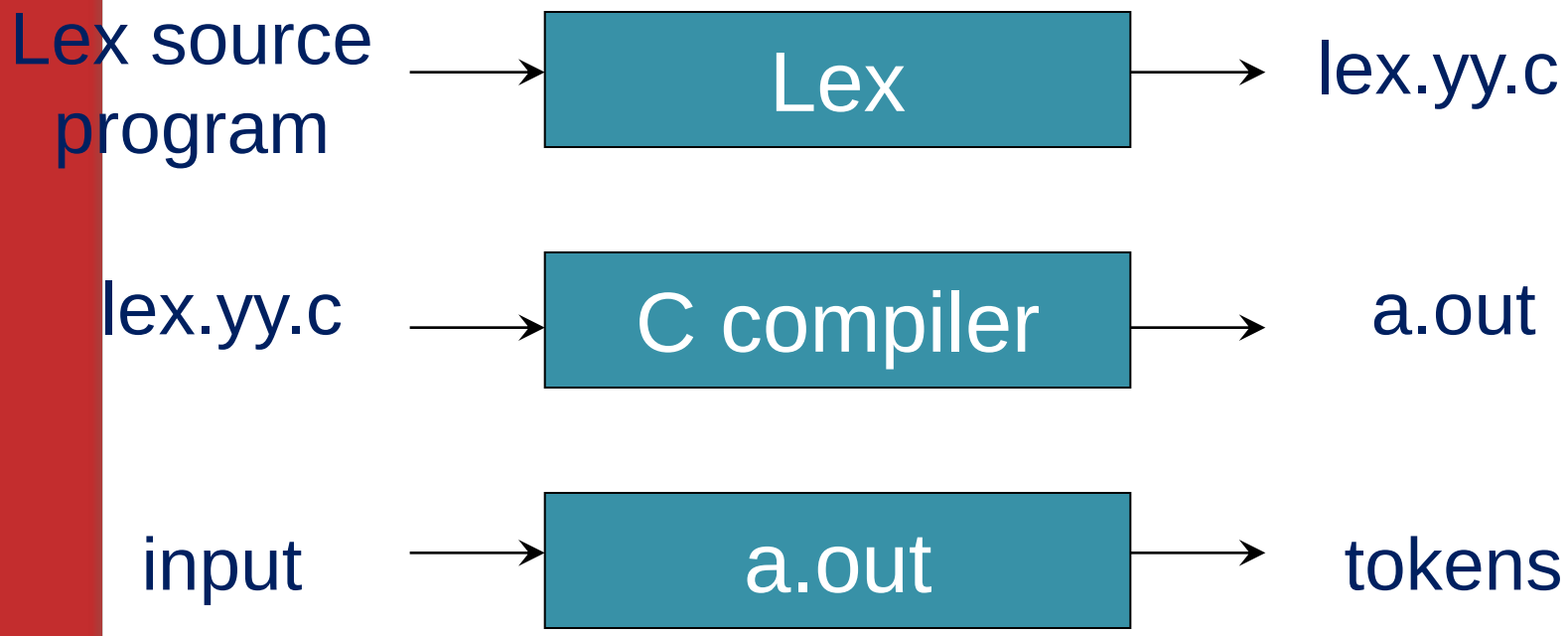
- You can use your Lex routines in the same ways you use routines in other programming languages.

```
%{
  void foo();
%}
letter     [a-zA-Z]
%%
{letter}+ foo();
%%
…
void foo() {
   …
}
```

Lex Source to C Program

- The table is translated to a C program (lex.yy.c) which
  - reads an input stream
  - partitioning the input into strings which match the given expressions and
  - copying it to an output stream if necessary

An Overview of Lex



Usage

- To run Lex on a source file, type
  `lex scanner.l`
- It produces a file named lex.yy.c which is a C program for the lexical analyzer.
- To compile lex.yy.c, type
  `cc lex.yy.c -ll`
- To run the lexical analyzer program, type
  `./a.out < inputfile`

Ex. LEX program to change case of a given input.

```
%{
%}
 %%
[A-Z]    {printf("%c",yytext[0]+32);}
[a-z]    {printf("%c",yytext[0]-32);}
%%
int main()
{
    yylex();
    return 0;
}
int yywrap()
{
    return 1;
}
```

Input : iNDIA
Output : India

Input : PUNe
Output : punE

Ex1 : LEX program to count number of lines, words, characters and spaces.

```
%{
  int num_lines = 0, num_chars = 0;
%}

%%
\n          ++num_lines;
.           ++num_chars;
%%
main()   {
  yylex();
  printf( "# of lines = %d, # of chars = %d\n",
                num_lines, num_chars );
}
```

Ex 2 : LEX program to count number of lines, words, characters and spaces.

```
%{
int lc,wc,cc,sc;
%}

%%
[\n]    {lc++;}
[a-zA-Z]+       {wc++;}
[ ]     {sc++;}
.       {cc++;}
%%
```

```
int main()
{
        FILE *fp;
        fp=fopen("in.txt","r");
        yyin = fp;
        lc=wc=cc=sc=0;
        yylex();
        printf("\nLine Count : %d\nWord
         Count : %d\nSpace Count : %d
        \nCharacter Count : %d",lc,wc,sc,cc);
        return 0;
}
int yywrap()
{
        return 1;
}
```

Ex 3. LEX program to replace scanf with read, printf with write, also find number of printf and scanf statements.

```
%{
int pc,sc;
FILE *fr,*fw;
%}

%%
"printf" {pc++;
    fprintf(fw,"write");}
"scanf"  {sc++;
    fprintf(fw,"read");}
%%
```

```
int main()
{
        pc=sc=0;
        fr= fopen("ain.txt","r");
        fw= fopen("aout.txt","w");
        yyin=fr;
        yyout=fw;
        yylex();
        printf("Printf Count : %d\nScanf
        Count : %d",pc,sc);
        return 0;
}

int yywrap()
{
        return 1;
}
```

Ex. 3 Contd.

Input File "ain.txt"

```
#include<stdio.h>

int main()
{
    int n;
    printf("Hello! Everybody");

    scanf("%d",&n);
    return 0;
}
```

Output File aout.txt

```
#include<stdio.h>

int main()
{
        int n;
        write("Hello! Everybody ");

        read("%d",&n);

        return 0;
}
```

Ex. 4 LEX program to count number of comment lines in a c program, also eliminate them & copy that program in separate file.

Ex. 4 LEX program to count number of comment lines in a c program, also eliminate them & copy that program in separate file.

```
%{
FILE *fr,*fw;
int cn,f;
%}


%%
"//".*      {cn++;}
"/*".*      {cn++; f=1;}
[\n]        {if(f==1){cn++;} else
    {fprintf(fw,yytext);}}
.*"*/"      {f=0;}
.    {if(f==0){fprintf(fw,yytext);}}
%%
```

```
int main()
{
        f=0;
        fr= fopen("in.c","r");
        fw= fopen("out.c","w");
        yyin=fr;
        yyout=fw;
        yylex();
        printf("\nNo of Comment
            Lines : %d",cn);
        return 0;
}

int yywrap()
{
        return 1;
}
```

Sample programs

- LEX program to convert shorthand English words to longhand English words. And also maintain the case of character at newline or new sentence.
- Implement a scanner for a subset of C language using LEX tool. Implementation should support Error handling

Assgn. No. 3

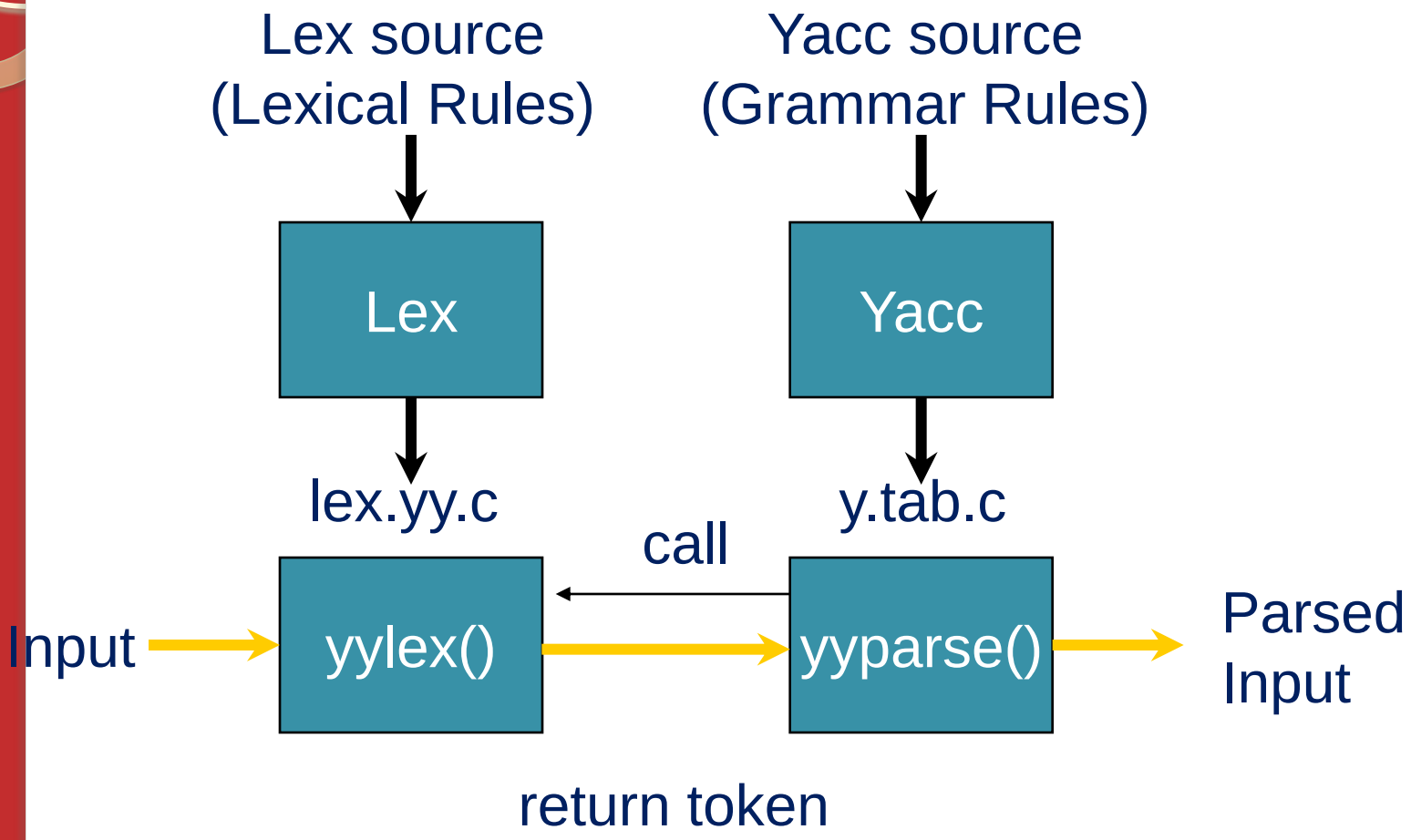
- Lexical analyzer for sample language using LEX.
- a) Lexical Analyzer for C Keywords, identifiers, operators, parenthesis with symbol table.
- b) For a English language paragraph i) count and print no of words, characters, line, and words starting with " b" ii) replace all small case words starting with 'b' with capital case.

Lex vs Yacc

- Lex
  - Lex generates C code for a lexical analyzer, or scanner
  - Lex uses patterns that match strings in the input and converts the strings to tokens

- Yacc
  - Yacc generates C code for syntax analyzer, or parser.
  - Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree.

Lex with Yacc



Availability

- lex, yacc on most UNIX systems
- bison: a yacc replacement from GNU
- flex: *f*ast *lex*ical analyzer
- BSD yacc
- Windows/MS-DOS versions exist

Versions of Lex

- AT&T -- lex
  http://www.combo.org/lex_yacc_page/lex.html
- GNU -- flex
  http://www.gnu.org/manual/flex-2.5.4/flex.html
- a Win32 version of flex :
  http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html
  or Cygwin :
  http://sources.redhat.com/cygwin/

- Lex on different machines is not created equal.

Yacc - Yet Another Compiler-Compiler


Thank you!