SYSTEM SOFTWARE AND COMPILERS 18CS61

MODULE-4

Dr.Sanchari Saha
Assistant Professor
Dept. of CSE,
CMRIT, Bangalore

Text Books

1. System Software by Leland. L. Beck, D Manjula, 3rd edition, 2012
Module 1

2. Alfred V Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman , Compilers-Principles, Techniques and Tools, Pearson, 2nd edition, 2007 .
Module 2, 3, 5

3. Doug Brown, John Levine, Tony Mason, lex & yacc, O'Reilly Media, October 2012.
Module 4

Course Learning Objectives:

This course (18CS61) will enable students to:

- Define System Software.
- Familiarize with source file, object file and executable file structures and libraries
- Describe the front-end and back-end phases of compiler and their importance to students

Course Outcome and CO-PO Mapping

CO-PO and CO-PSO Mapping																		
Course Outcomes		Modules covered	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	Explain system software	1	3	3	2	2	-	-	-	-	-	-	-	-	-	-	-	-
CO2	Design and develop lexical analyzers, parsers and code generators	2,3,4,5	3	3	3	3	3	2	-	-	-	-	-	-	-	-	2	-
CO3	Utilize lex and yacc tools for implementing different concepts of system software	4	3	3	3	3	3	2	-	-	-	-	-	-	-	-	3	-

Module 4- syllabus

Lex and Yacc –The Simplest Lex Program, Grammars, Parser-Lexer Communication, A YACC Parser, The Rules Section, Running LEX and YACC, LEX and Hand- Written Lexers, Using LEX - Regular Expression, Examples of Regular Expressions, A Word Counting Program

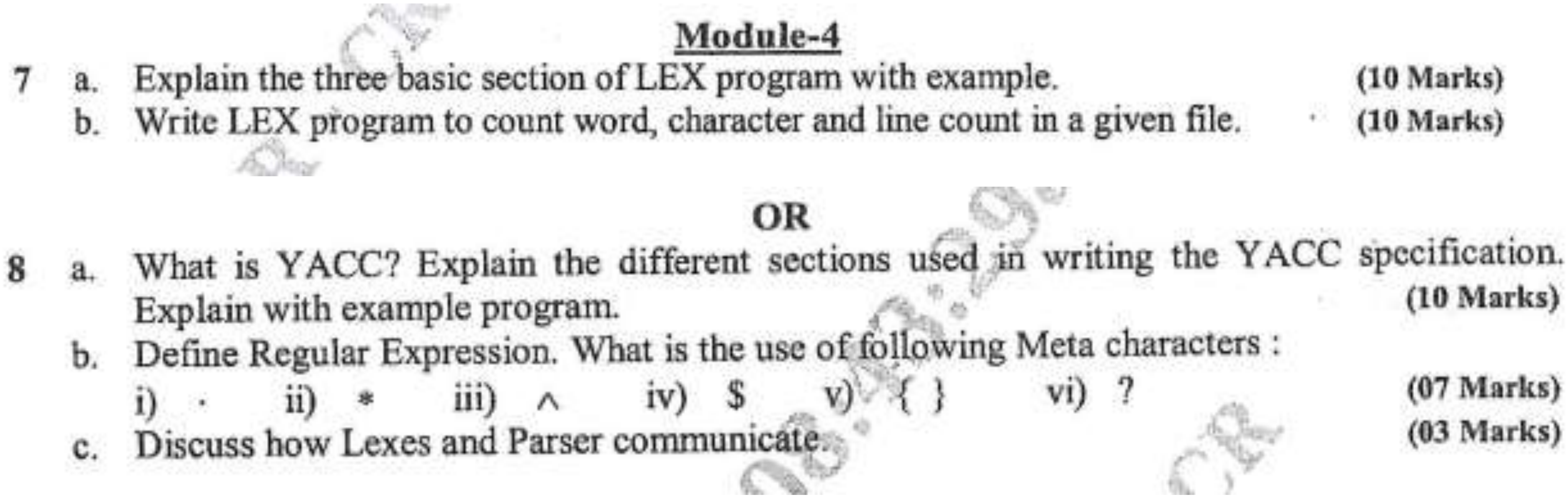
Using YACC – Grammars, Recursive Rules, Shift/Reduce Parsing, What YACC Cannot Parse, A YACC Parser - The Definition Section, The Rules Section, The LEXER, Compiling and Running a Simple Parser, Arithmetic Expressions and Ambiguity.

Text book 3: Chapter 1,2 and 3.

RBT: L1, L2, L3

VTU QP

Aug -2022



Module-4

7 a. Explain the three basic section of LEX program with example. **(10 Marks)**

b. Write LEX program to count word, character and line count in a given file. **(10 Marks)**

OR

8 a. What is YACC? Explain the different sections used in writing the YACC specification. Explain with example program. **(10 Marks)**

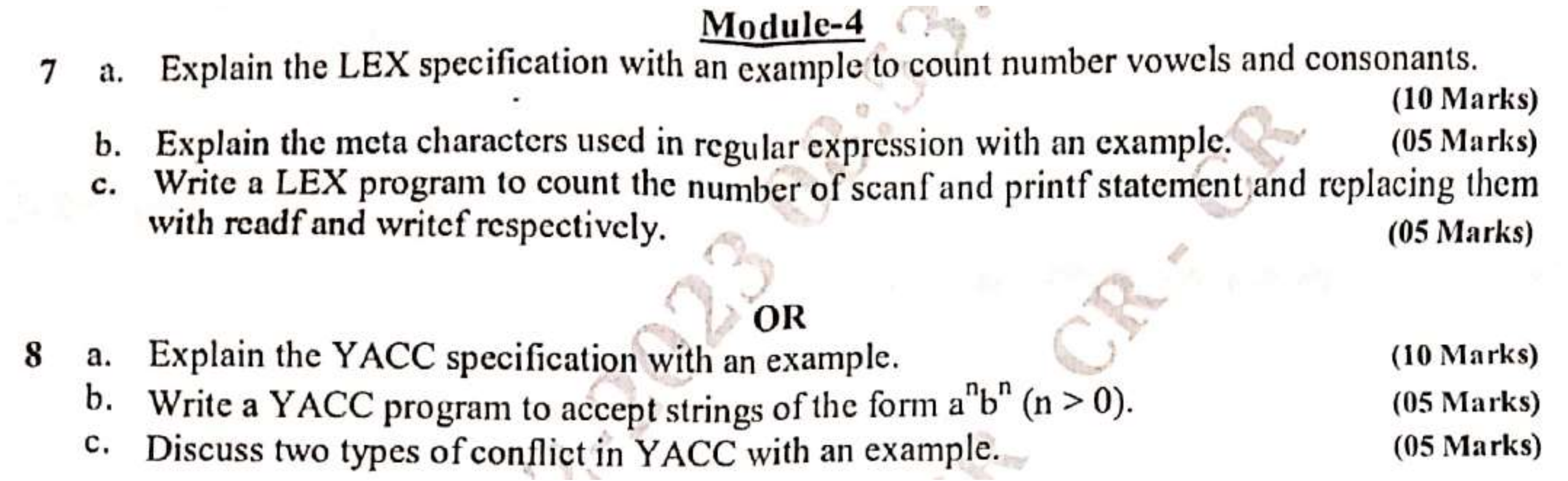
b. Define Regular Expression. What is the use of following Meta characters :

i) . ii) * iii) ^ iv) $ v) { } vi) ? **(07 Marks)**

c. Discuss how Lexes and Parser communicate. **(03 Marks)**

VTU QP

Jan -2023



Module-4

7 a. Explain the LEX specification with an example to count number vowels and consonants. **(10 Marks)**

b. Explain the meta characters used in regular expression with an example. **(05 Marks)**

c. Write a LEX program to count the number of scanf and printf statement and replacing them with readf and writef respectively. **(05 Marks)**

OR

8 a. Explain the YACC specification with an example. **(10 Marks)**

b. Write a YACC program to accept strings of the form $a^n b^n$ $(n > 0)$. **(05 Marks)**

c. Discuss two types of conflict in YACC with an example. **(05 Marks)**

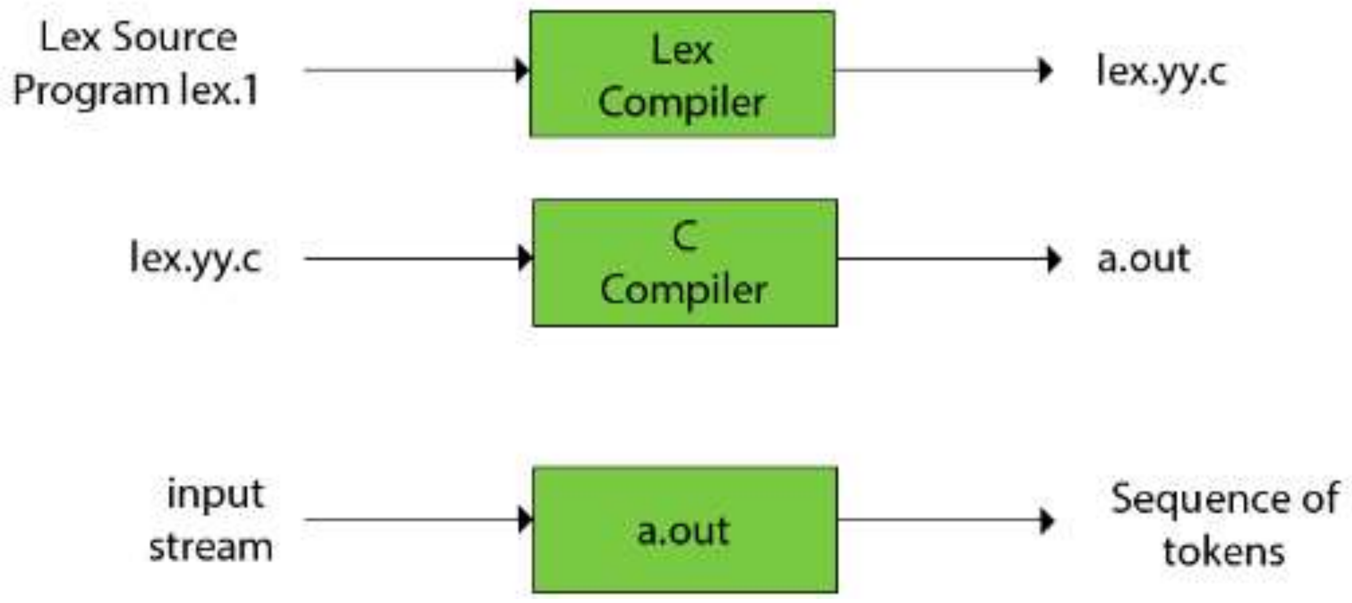
Chapter 1: Lex and Yacc

The Simplest Lex Program

This lex program copies its standard input to its standard output:

```
%%
  . | \n    ECHO;
  %%
```

Lex automatically generates the actual C program code needed to handle reading the input file and sometimes, as in this case, writing the output as well.



LEX

•Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
•The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
•It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

•Firstly lexical analyzer creates a program lex.l in the Lex language. Then Lex compiler runs the lex.l program and produces a C program lex.yy.c.
•Finally C compiler runs the lex.yy.c program and produces an object program a.out.
•a.out is lexical analyzer that transforms an input stream into a sequence of tokens.

Lex file format

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

```
%{
definitions
%}

%%

{ rules
}
%%
{ user subroutines
}
```

Definitions It is enclosed within ' %{ ' and ' %} ' . It is generally used to declare functions, include header files, or define global variables and constants.

Rules Rules in a LEX program consists of two parts :

1. The pattern to be matched
2. The corresponding action to be executed

User subroutines are auxiliary procedures needed by the actions. LEX generates C code for the rules specified in the Rules section and places this code into a single function called yylex(). In addition to this LEX generated code, the programmer may wish to add his own code to the lex.yy.c file.

Lex Libraries

yylex() is a function of return type int.

- LEX automatically defines yylex() in lex.yy.c but does not call it.
- The programmer must call yylex() in the Auxiliary functions section of the LEX program.
- LEX generates code for the definition of yylex() according to the rules specified in the Rules section.

LEX declares the function **yywrap()** of return-type **int** in the file lex.yy.c .

- **yylex()** makes a call to **yywrap()** when it encounters the end of input.
- If yywrap() returns zero (indicating false) yylex() assumes there is more input and it continues scanning from the location pointed to by yyin.
- If yywrap() returns a non-zero value (indicating true), yylex() terminates the scanning process and returns 0 (i.e. “wraps up”).

lex is a tool for recognizing tokens in a program. we have to describe the tokens using regular expressions. Along with regular expression we have to provide the file which needs to be scanned or tokenized. This file is pointed by **yyin**.

yylex() is the function which we have to invoke to start the process. when we invoke yylex(), lex will take the file which is pointed by **yyin** and scan stream of characters and try to match the characters with the regular expression you provided.

It keeps the matched string in **yytext** and the length of the matched string in **yyleng**. For each token it generates a token value which is kept in **yylval**.

yyleng and **yytext** will keep on changing on each matching of pattern.

After scanning the whole file it needs to print the output in a file..
yyout is the pointer to a file where it has to keep the output.

As soon as **yyparse()** encounters input that does not match any known grammatical construction, it calls the **yyerror()** function. In this case, the argument that it passes to yyerror() is:

"Syntax error"

If you are using the default version of yyerror(), it simply displays this message to stderr; however, **you can supply your own yyerror() function if you want to do other processing.**

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

Table 1: Pattern Matching Primitives

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

%%
```

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`

Table 2: Pattern Matching Examples

Sample LEX program to recognize integer numbers

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

%%
[0-9]+ { printf(“Saw an integer: %s\n”, yytext); }
. { ;}

%%
main( )
{
printf(“Enter some input that consists of an integer
number\n”);
yylex();
}
int yywrap()
{
return 1;
}
```

Running Lex program:

```
[student@localhost ~]$ lex 1a.l

[student@localhost ~]$ cc lex.yy.c

[student@localhost ~]$ ./a.out

Enter some input that consists of an integer
number

hello 2345

Saw an integer: 2345
```

Explanation:

- First-line runs lex over the lex specification & generates a file, lex.yy.c which contains C code for the lexer.
- The second line compiles the C file.
- The third line executes the C file.

lex code to determine whether input is a valid identifier or not

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

% %
  // reg expression for valid identifiers
    ^[a - z A - Z ][a - z A - Z 0 - 9 ] *  {printf("Valid Identifier");}

 // reg expression for invalid identifiers
^[^a - z A - Z ]  {printf("Invalid Identifier");}
. { ; }
% %
```

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

Source Code to recognize and display comments, numbers, identifiers, and strings

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

%%
[\t ]+ { ; }

[0-9]+|[0-9]*\.[0-9]+ { printf("\n%s is NUMBER", yytext);}

#.* { printf("\n%s is COMMENT", yytext);}

[a-zA-Z][a-zA-Z0-9]+ { printf("\n%s is IDENTIFIER", yytext);}

\"[^ \"\n]*\" { printf("\n%s is STRING", yytext);}

\n { ECHO;}
%%
```

```
int main()
{
printf("enter input");
 yylex();
}

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

***word count Program in Lex: -> to count the number of lines, words, spaces and characters present in an input file (without considering command line arguments)

Logic:

Read each character from the text file :

Is it a capital letter in English? [A-Z] : increment capital letter count by 1.
Is it a small letter in English? [a-z] : increment small letter count by 1
Is it [0-9]? increment digit count by 1.
All other characters (like '!', '@','&') are counted as special characters

How to count the number of lines? we simply count the encounters of '\n' <newline> character.that's all!!

To count the number of words we count white spaces and tab character(of course, newline characters too..)

Lex Code

```
%{
#include<stdio.h>
int lines=0,
words=0,s_letters=0,c_letters=0,
num=0, spl_char=0,total=0;
%}

%%
\n { lines++; words++;}
[\t ' '] words++;
[A-Z] c_letters++;
[a-z] s_letters++;
[0-9] num++;
. spl_char++;
%%
```

```
void main( )
{
yyin= fopen("myfile.txt","r");
yylex();
total=s_letters+c_letters+num+spl_char;
printf(" This File contains ...");
printf("\n\t%d lines",  lines);
printf("\n\t%d words", words);
printf("\n\t%d small letters",  s_letters);
printf("\n\t%d capital letters", c_letters);
printf("\n\t%d digits",  num);
printf("\n\t%d special characters", spl_char);
printf("\n\tIn total %d characters.\n", total);
}

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

Write a Lex Program to recognize verbs in Speech

```
%{
#include<stdio.h>
%}
%%
[\t ]+   {; }
/* ignore white space */
is | am | are | were | was | be | being | been | do | does | go| have| can | could | will | would | should
{printf (“%s: is a verb\n”, yytext) ;}
[a-zA-Z]+            {printf (“%s: is not a verb\n”, yytext) ;}
.|\n         {ECHO ;}
%%
 main ()
{
printf(“enter input”);
yylex ();   }
```

***Write a Lex Program to identify multiple parts of Speech

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

%%
[\t ]+   { ; }  /* ignore whitespace */
;
is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go      { printf("%s: is a verb\n", yytext); }
```

```
very |
simply |
gently |
quietly |
calmly |
angrily   { printf("%s: is an adverb\n",
yytext); }

to |
from |
behind |
above |
below |
between
below     { printf("%s: is a
preposition\n", yytext); }
```

```
if |
then |
and |
but |
or        { printf("%s: is a conjunction\n", yytext); }

their |
my |
your |
his |
her |
its     { printf("%s: is a possessive adjective\n",
yytext); }

tall |
short |
big|
small |
good |
bad      { printf("%s: is a adjective\n", yytext); }
```

```
I |
you |
he |
she  |
we  |
they       { printf("%s: is a pronoun\n", yytext); }


[a-zA-Z]+   {printf("%s:  don't recognize, might be a noun\n", yytext); }
.|\n       { ECHO;/* normal default anyway */ }

%%

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

***Lex program to recognize tokens (Id, keywords, ws, relop, number) and return the tokens found

```
%{
/* 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}+)?

%%
{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) j}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}
%%

//auxiliary functions
int installID() {
    /* function to install the lexeme,
    whose first character is pointed to by yytext,
    a d whose length is yyleng,
    into the symbol table and
    return a pointer thereto */
}
int installNum() {
    /* similar to installID, but puts numerical
    constants into a separate table */
}
main()
{
yylex();
}
```

A few observations about this code will introduce us to many of the important features of Lex.

- In our example, we have listed in a comment the names of the manifest constants, LT, IF, and so on.
- Also there is a sequence of regular definitions in the code.These use the extended notation for regular expressions.
- Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces '{ }'.
- In the definition of id and number, parentheses are used as grouping meta symbols.
- In the auxiliary-function section, we see two such functions, **installID()** and **installNum().**

Function **installID()** is called to place the lexeme found in the symbol table.
This function returns a pointer to the symbol table, which is placed in global variable yylval, where it can be used by the parser or a later component of the compiler.
Note that installID() has available to it two variables that are set automatically by the lexical analyzer that Lex generates :

(a) **yytext** is a pointer to the beginning of the lexeme

(b) **yyleng** is the length of the lexeme found.

The token name ID is returned to the parser.

Lexer with Symbol Table

The table of words is a simple symbol table, a common structure in lex and yacc applications. A C compiler, for example, stores the variable and structure names, labels, enumeration tags, and all other names used in the program in its symbol table. Each name is stored along with information describing the name. In a C compiler the information is the type of symbol, declaration scope, variable type, etc. In our current example, the information is the part of speech.

We have 2 symbol table maintenance routines:

- **add_word(),** which puts a new word into the symbol table
- **lookup_word( ),** which looks up a word which should already be entered.

Lexer with symbol table / [Word recognizer with a symbol table]

Definition Section

```
%{

enum {
      LOOKUP =0, /* default - looking rather than defining. */
      VERB,
      ADJ,
      ADV,
      NOUN,
      PREP,
      PRON,
      CONJ
};

int state;

int add_word(int type, char *word);
int lookup_word(char *word);
%}
```

Rule Section

```
%%
\n    { state = LOOKUP; }   /* end of line, return to
default state */

 /* whenever a line starts with a reserved part of
speech name , start defining words of that type */
^verb { state = VERB; }
^adj  { state = ADJ; }
^adv  { state = ADV; }
^noun { state = NOUN; }
^prep { state = PREP; }
^pron { state = PRON; }
^conj { state = CONJ; }

[a-zA-Z]+ {
                /* a normal word, define it or look it up */

if(state != LOOKUP) {
                /* define the current word */

 add_word(state, yytext);
             }
else {
                switch(lookup_word(yytext))
{
case VERB: printf("%s: verb\n", yytext); break;
case ADJ: printf("%s: adjective\n", yytext);
break;
case ADV: printf("%s: adverb\n", yytext); break;
case NOUN: printf("%s: noun\n", yytext); break;
case PREP: printf("%s: preposition\n", yytext);
break;
case PRON: printf("%s: pronoun\n", yytext);
break;
case CONJ: printf("%s: conjunction\n", yytext);
break;
default:  printf("%s: don't recognize\n", yytext);
break; }
          }
         }
.    { ; }

%%
```

- The caret, “^”, at the beginning of the pattern makes the pattern match only at the beginning of an input line.

- We reset the state to LOOKUP at the beginning of each line so that after we add new words interactively we can test our table of words to determine if it is working correctly.

- If the state is LOOKUP when the pattern “[a-zA-Z]+” matches, we look up the word, using lookup_word(), and if found print out its type.

- If we’re in any other state, we define the word with add_word().

Subroutine/ Auxiliary Function Section

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

/* define a linked list of words and types */
struct word {
      char *word_name;
      int word_type;
      struct word *next;
};

struct word *word_list;   /* first element in word
list */
extern void *malloc() ;
int add_word(int type, char *word)
{
      struct word *wp;
      if(lookup_word(word) != LOOKUP)
{
 printf("!!! warning: word %s already defined \n",
word);
            return 0;
      }
/* word not there, allocate a new entry and link
it on the list */
      wp = (struct word *) malloc(sizeof(struct word));
      wp->next = word_list;

      /* have to copy the word itself as well */
      wp->word_name = (char *)
malloc(strlen(word)+1);
      strcpy(wp->word_name, word);
      wp->word_type = type;
      word_list = wp;
      return 1;  /* it worked *
}
int lookup_word(char *word)
{
      struct word *wp = word_list;
      /* search down the list looking for the word */
      for(; wp; wp = wp->next) {
      if(strcmp(wp->word_name, word) == 0)
            return wp->word_type;
      }
      return LOOKUP;       /* not found */
}
```

Grammars

- We need to recognize specific sequences of tokens and perform appropriate actions.
- A description of such a set of actions is known as a **grammar.**

Examples:

Simple sentence types: noun verb (or) noun verb noun
Subject -> noun | pronoun

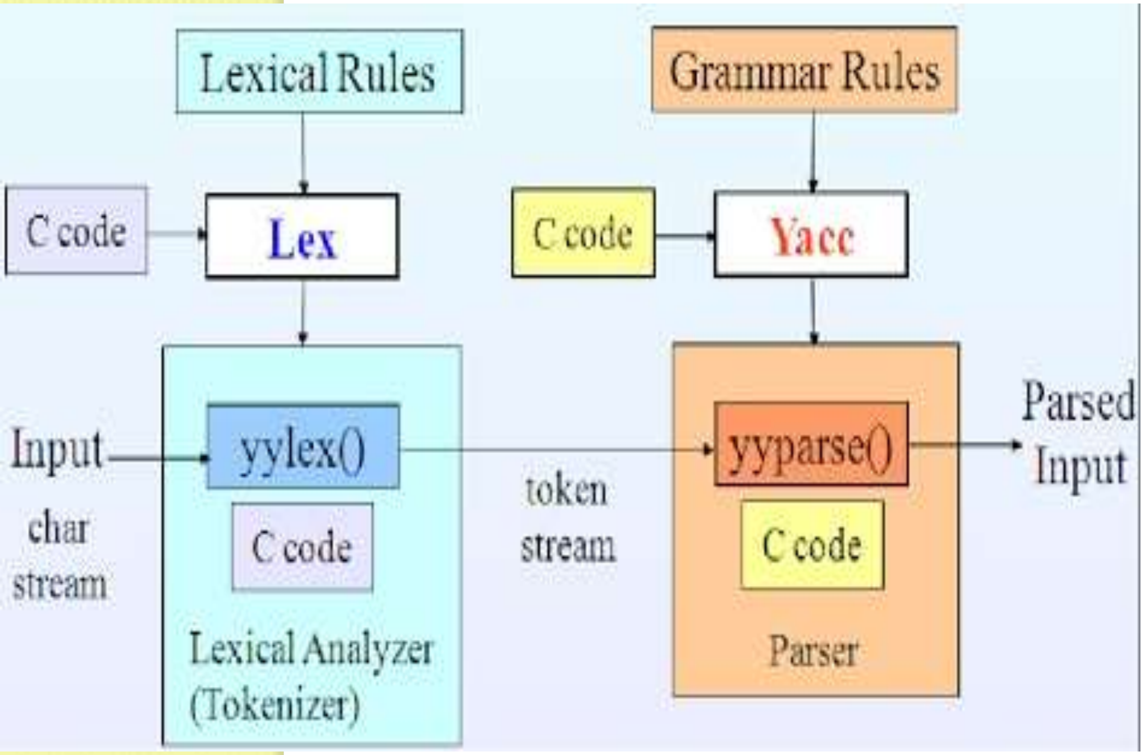
This would indicate that the new symbol subject is either a noun or a pronoun.

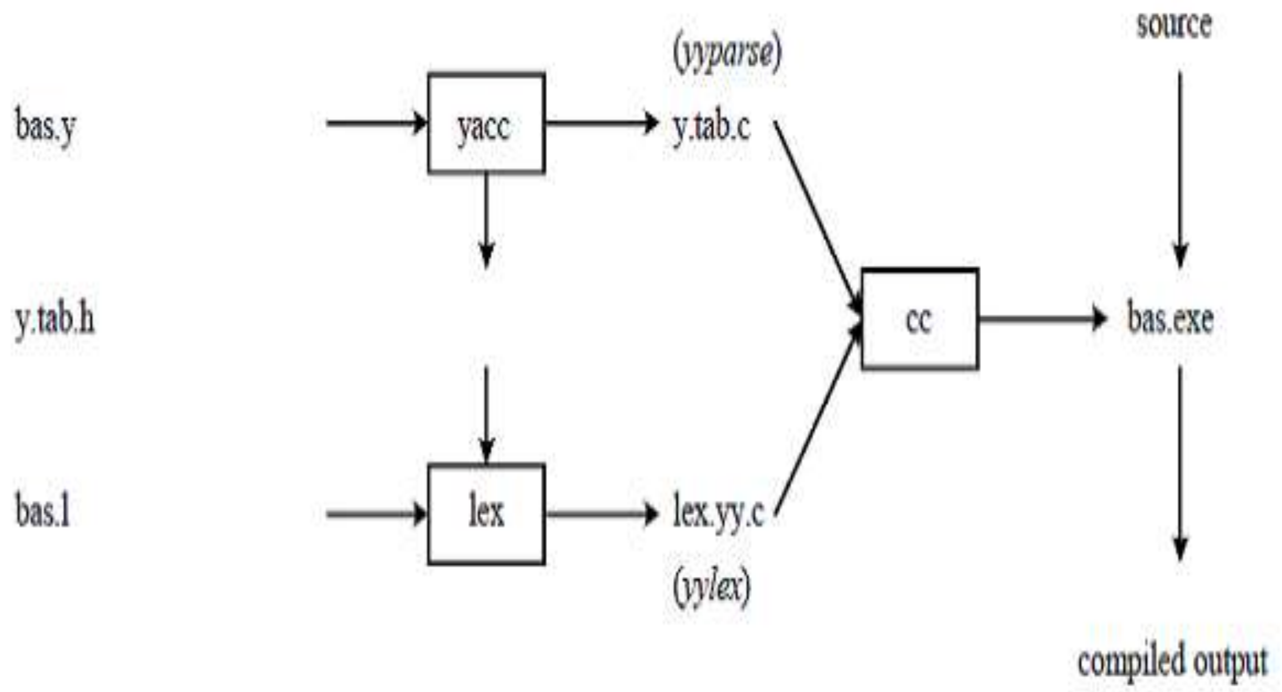
Object -> noun
Sentence -> subject verb object

The lexical analyser must be modified in order to return values useful to parser.

Parser- Lexer Communication

- In the process of compilation, the lexical analyzer and parser work together.
- When the parser requires a string of tokens, it invokes a lexical analyzer, in turn, the lexical analyzer supplies tokens to the parser.





Parser-Lexer Communication

- When we use LEX and YACC together the **YACC becomes a high-level routine**. It calls the yylex() function of the lexer in order to identify and collect the tokens. These generated tokens can then be demanded by the YACC parser for further arrangement.

- The parser collects a sufficient number of tokens and builds the parse tree. Not all the tokens are useful to the parser. Some tokens can be ignored for an efficient compilation process.

- Generally, **any token can be agreed upon by LEX and YACC using a token code**. The token code can be defined using a macro **#define**.

- YACC writes the token codes in a separate header file called **y.tab.h** as follows:

```
#define NAME 259
```

The token code 0 represents the logical end of the input.

Few more Example of tokens are NOUN, PRONOUN, VERB, ADVERB, ADJECTIVE, PREPOSITION and CONJUNCTION.

Yacc defines each of these as a small integer using pre-processor #define.

```
#define NOUN 257
#define PRONOUN 258
#define VERB 259
#define ADVERB 260
#define ADJECTIVE 261
#define PREPOSITION 262
#define CONJUNCTION 263
```

A Yacc Parser

Structure of Yacc Program

declarations
%%
rules
%%
routines

Input to yacc is divided into three sections. The definitions section consists of token declarations and C code bracketed by “%{“ and “%}”. The grammar is placed in the rules section and user subroutines are added in the routines section.

Ex of token declaration: **%token INTEGER**

THE DECLARATIONS SECTION may contain the following items.

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword %token.
- Declaration of the start symbol using the keyword %start
- C declarations: included files, global variables, types.
- C code between %{ and %}.

RULES SECTION.

A rule has the form:

```
nonterminal : sentential form
          | sentential form
          .................
          | sentential form
          ;
```

Actions may be associated with rules and are executed when the associated sentential form is matched.

Lexer to be called from the Parser *(Parts of speech program):* Speech Parser

```
%{
#include "y.tab.h"
#define LOOKUP 0
int state;
%}
%%
\n          {state = LOOKUP ;}
\.\n        {state=LOOKUP; return 0 ;}
```

/ the backslash in front of the period, quotes the period. So, this rule matches a period followed by a new line.*/*

```
^verb       {state=VERB ;}
^adj        {state=ADJECTIVE ;}
^adv        {state=ADVERB ;}
^noun       {state=NOUN ;}
^prep       {state=PREPOSITION ;}
^pron       {state=PRONOUN ;}
^conj       {state=CONJUNCTION ;}
[a-zA-Z]+   { if (state! =LOOKUP) {
              add_word (state, yytext);
              }
else {
switch (lookup_word (yytext)) {
Case VERB:  return (VERB);
Case ADJECTIVE: return (ADJECTIVE);
Case ADVERB: return (ADVERB);
Case NOUN: return (NOUN);
Case PREPOSITION: return (PREPOSITION);
Case PRONOUN:   return (PRONOUN);
Case CONJUNCTION: return (CONJUNCTION);
default:    printf ("%s: don't recognize\n", yytext);  } } }
```

- {;} %%

- The caret, “^”, at the beginning of the pattern makes the pattern match only at the beginning of an input line.

- We reset the state to LOOKUP at the beginning of each line so that after we add new words interactively we can test our table of words to determine if it is working correctly.

- If the state is LOOKUP when the pattern “[a-zA-Z]+” matches, we look up the word, using lookup_word(), and if found print out its type.

- If we’re in any other state, we define the word with add_word().

Yacc sentence parser (Introducing to the first cut of yacc grammar

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

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%
sentence: subject VERB object   { printf("Sentence is valid.\n"); }
      ;

subject:    NOUN | PRONOUN
      ;

object:         NOUN


      ;

%%
```

```
extern FILE *yyin;
main()
{
    do
     {
         yyparse();
    }
      while (!feof(yyin));
}

yyerror(s)
char *s;
{
   fprintf(stderr, "%s\n", s);
}
```

The most important subroutine is **main()** which repeatedly calls yyparse() until the lexer's input file runs out.

The routine yyparse() is the parser generated by yacc, so our main program repeatedly tries to parse sentences until the input runs out. (The lexer returns a zero token whenever it sees a period at the end of a line; that's the signal to the parser that the input for the current parse is complete.)

Extended version of the yacc sentence parser: We have expanded our sentence rule by introducing simple sentence and compound sentence grammar

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

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%

sentence: simple_sentence  { printf("Parsed a simple sentence.\n"); }
      | compound_sentence { printf("Parsed a compound sentence.\n"); }
      ;

simple_sentence: subject verb object
      |    subject verb object prep_phrase
      ;

compound_sentence: simple_sentence CONJUNCTION simple_sentence
      |    compound_sentence CONJUNCTION simple_sentence
      ;
```

```
subject:   NOUN
      |    PRONOUN
      |    ADJECTIVE subject
      ;

verb:      VERB
      |    ADVERB VERB
      |    verb VERB
      ;

object:          NOUN
      |    ADJECTIVE object
      ;

prep_phrase:    PREPOSITION NOUN
      ;
%%
```

```
extern FILE *yyin;

main()
{
while(!feof(yyin));
{
yyparse();
}
}

yyerror(s)
char *s;
{
    fprintf(stderr, "%s\n", s);
}
```

Running LEX

```
 $ lex filename.l
$ cc  lex.yy.c  -ll
 $ ./a.out
```

Running YACC

```
$ lex filename.l
$ yacc -d filename.y
$ cc  lex.yy.c  y.tab.c -ll
$ ./a.out
```

Lexer written LEX vs Hand- Written Lexers

A lexer written in C :

```
#include <stdio.h>
#include <ctype.h>
#define NUMBER 400
#define COMMENT 401
#define TEXT 402
#define COMMAND 403

    main(int argc,char *argv[ ])
    {
    int val;
    while (val = lexer() )
    printf("value is %d\n",val);
    }
```

a lexer written in C suitable for a simple command language that handles commands, numbers, strings, and new lines, ignoring white space and comments.

lexer()

```
lexer()
    {
    int c;

  while ((c=getchar()) == ' ' || c == '\t’) ;
   if (c == EOF)
   return 0;
   if (c == '.' || isdigit(c))
{   /* number */
    while ((c = getchar()) != EOF && isdigit (c)) ;
    if (c == '.’)
    while ((c = getchar()) != EOF && isdigit (c) ;
     ungetc(c, stdin);
       return NUMBER;
        }
        if ( c =='#’)
{ /* comment */
      while ((c = getchar()) != EOF && c != '\n’) ;
 ungetc(c,stdin);
        return COMMENT;
        }
```

```
if ( c ==‘ “ ’)
{  /* literal text */
while ((c = getchar()) != EOF &&  c != ‘ “ ' && c != '\n’);
          if(c == '\n’)
          ungetc(c,stdin);
           return TEXT;
       }
       if ( isalpha(c))
{  /* check to see if it is a command */

while ((c = getchar()) != EOF && isalnum(c)) ;
           ungetc(c, stdin);
           return COMMAND;
       }
       return c;
    }
```

// Note: The C library function **int ungetc (int char, FILE *stream)** *pushes the character char (an unsigned char) onto the specified stream so that the this is available for the next read operation.*

- **char** − This is the character to be put back. This is passed as its int promotion.
- **stream** − This is the pointer to a FILE object that identifies an input stream.

Note: There's no statement between the while and the closing semicolon because nothing else needs to be done.
c= getchar() is reading the next character from the file, and (c= getchar()) != '\n' is comparing that read character to the newline character.
There's another test for the special EOF value, to make sure it doesn't keep trying to read once you reach the end of the file

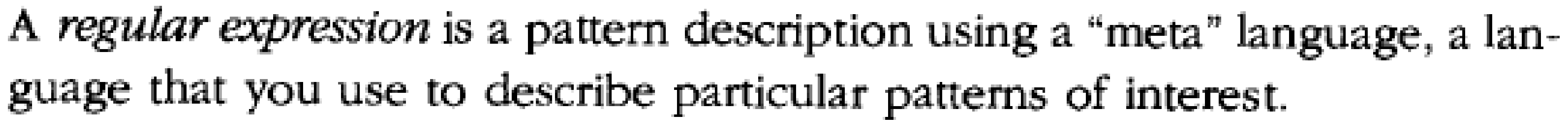
The same lexer written in lex

```
%{
#include <stdio.h>
  #define NUMBER 400
  #define COMMENT 401
  #define TEXT 402
  #define COMMAND 403
  %}

  %%
  [ \t]+            { ; }

  [0-9]+ |[0-9]+\.[0-9]+ | \.[0-9]+ { return NUMBER; }

   #.*                   { return COMMENT; }

   \"[^\"\n]*\"          { return TEXT;   }

  [a-zA-Z][a-zA-Z0-9]+    { return COMMAND;}

   \n                    { return '\n'; }
  %%
```

```
main(int argc,char *argv[ ])
      {
   int val;
   while(val = yylex())
printf ("value is %d\n",val);
   }
```

Lex handles some subtle situations in a natural way that are difficult to get right in a hand written lexer. For example, assume that you're skipping a C language comment. To find the end of the comment, you look for a "*", then check to see that the next character is a "/". If it is, you're done, if not you keep scanning. A very common bug in C lexers is not to consider the case that the next character is itself a star, and the slash might follow that. In practice, this means that some comments fail.

Using LEX - Regular Expression


A *regular expression* is a pattern description using a “meta” language, a language that you use to describe particular patterns of interest.

Meta character Description

\ Marks the next character as either a special character or a literal. For example, n matches the character n, whereas \n matches a newline character. The sequence \\\\ matches \\ and \\(matches (.

^ Matches the beginning of input.

$ Matches the end of input.

***** Matches the preceding character zero or more times. For example, zo* matches either z or zoo.

+ Matches the preceding character one or more times. For example, zo+ matches zoo but not z.

? Matches the preceding character zero or one time. For example, a?ve? matches the ve in never.

- Matches any single character except a newline character.

[xyz] A character set. Matches any one of the enclosed characters. For example, [abc] matches the a in plain.

"[Tt]he" → The cat sat on the mat.

"^[Tt]he" → The cat sat on the mat.

[^xyz] A negative character set. Matches any character that is not enclosed. For example, [^abc] matches the p in plain. "[^c]at" → The cat sat on the mat .

[a-z] A range of characters. Matches any character in the specified range. For example, [a-z] matches any lowercase alphabetic character in the English alphabet.

[^m-z] A negative range of characters. Matches any character that is not in the specified range. For example, [m-z] matches any character that is not in the range m through z.

\d	Matches a digit character.
\s	It is used to match a one white space character.
\0	It is used to match a NULL character.
\n	It helps a user to match a new line.

| Matches either the preceding regular expression or the following regular expression. For example:

```
cow|pig|sheep
```

matches any of the three words.

"..." Interprets everything within the quotation marks literally—metacharacters other than C escape sequences lose their meaning.

/ Matches the preceding regular expression but only if followed by the following regular expression. For example:

```
0/1
```

matches "0" in the string "01" but would not match anything in the strings "0" or "02". The material matched by the pattern following the slash is not "consumed" and remains to be turned into subsequent tokens. Only one slash is permitted per pattern.

() Groups a series of regular expressions together into a new regular expression. For example:

```
(01)
```

represents the character sequence 01. Parentheses are useful when building up complex patterns with *, +, and |.

{} Indicates how many times the previous pattern is allowed to match when containing one or two numbers. For example:

```
A{1,3}
```

matches one to three occurrences of the letter A. If they contain a name, they refer to a substitution by that name.

\ Used to escape metacharacters, and as part of the usual C escape sequences, e.g., "\n" is a newline character, while "*" is a literal asterisk.

(pattern) Matches a pattern and remembers the match. The matched substring can be retrieved from the resulting matches collection by using this code: Item [0]...[n]. To match parentheses characters ( ), use \(or \).

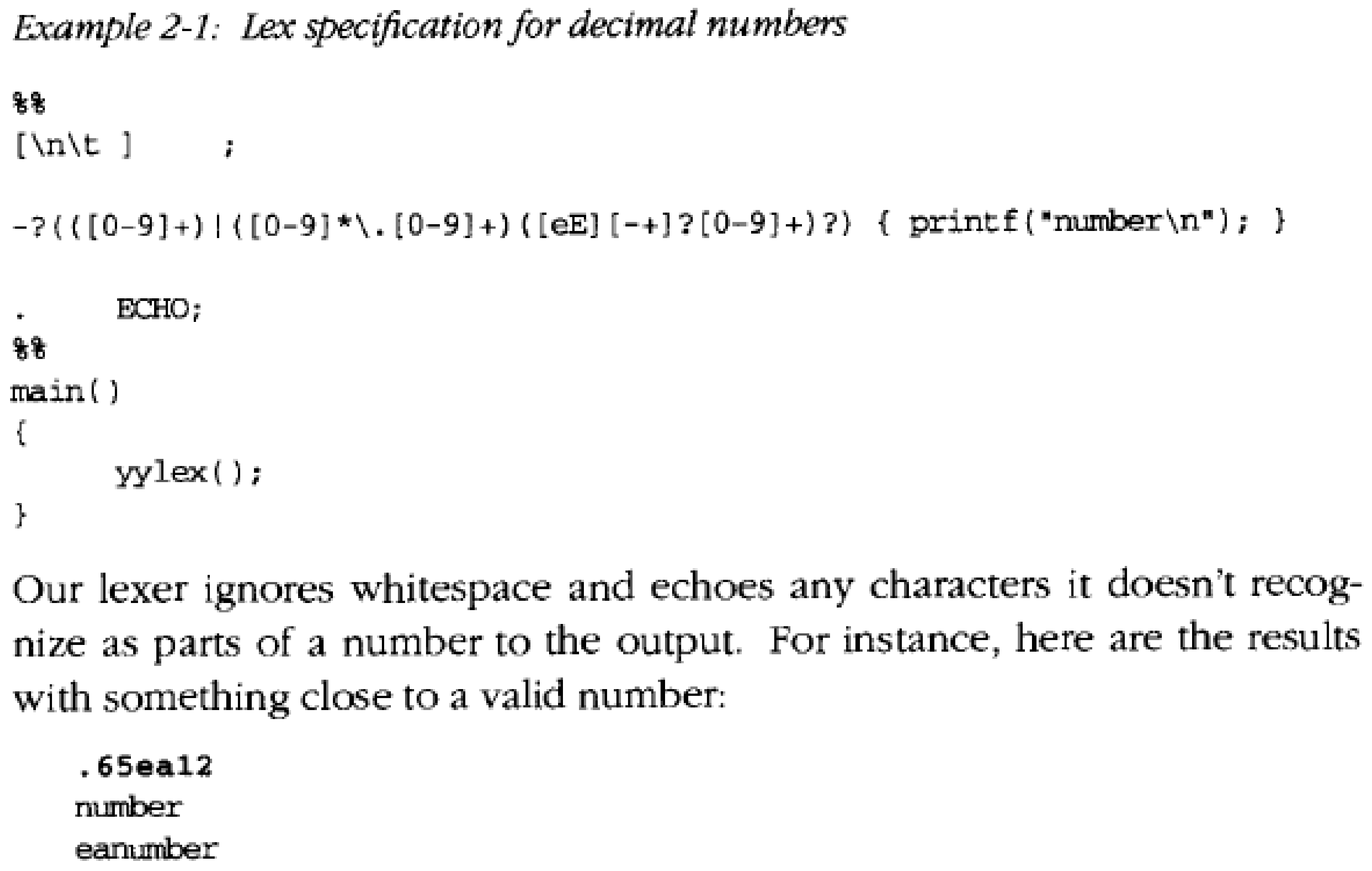
x|y Matches either x or y. For example, z|wood matches z or wood. (z|w)oo matches zoo or wood.

{n} n is a non-negative integer. Matches exactly n times. For example, o{2} does not match the o in Bob, but matches the first two os in foooood.

{n,} In this expression, n is a non-negative integer. Matches the preceding character at least n times. For example, o{2,} does not match the o in Bob and matches all the os in foooood. The o{1,} expression is equivalent to o+ and o{0,} is equivalent to o*.

{n,m} The m and n variables are non-negative integers. Matches the preceding character at least n and at most m times. For example, o{1,3} matches the first three os in fooooood. The o{0,1} expression is equivalent to o?.

Examples of Regular Expressions in Lex Code



Example 2-1: Lex specification for decimal numbers

```
%%
[\n\t ]      ;

-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) { printf("number\n"); }

.      ECHO;
%%
main()
{
       yylex();
}
```

Our lexer ignores whitespace and echoes any characters it doesn't recognize as parts of a number to the output. For instance, here are the results with something close to a valid number:

```
    .65ea12
    number
    eanumber
```

Another common regular expression is one used by many scripts and simple configuration files, an expression that matches a comment starting with a sharp sign, "#".* We can build this regular expression as:

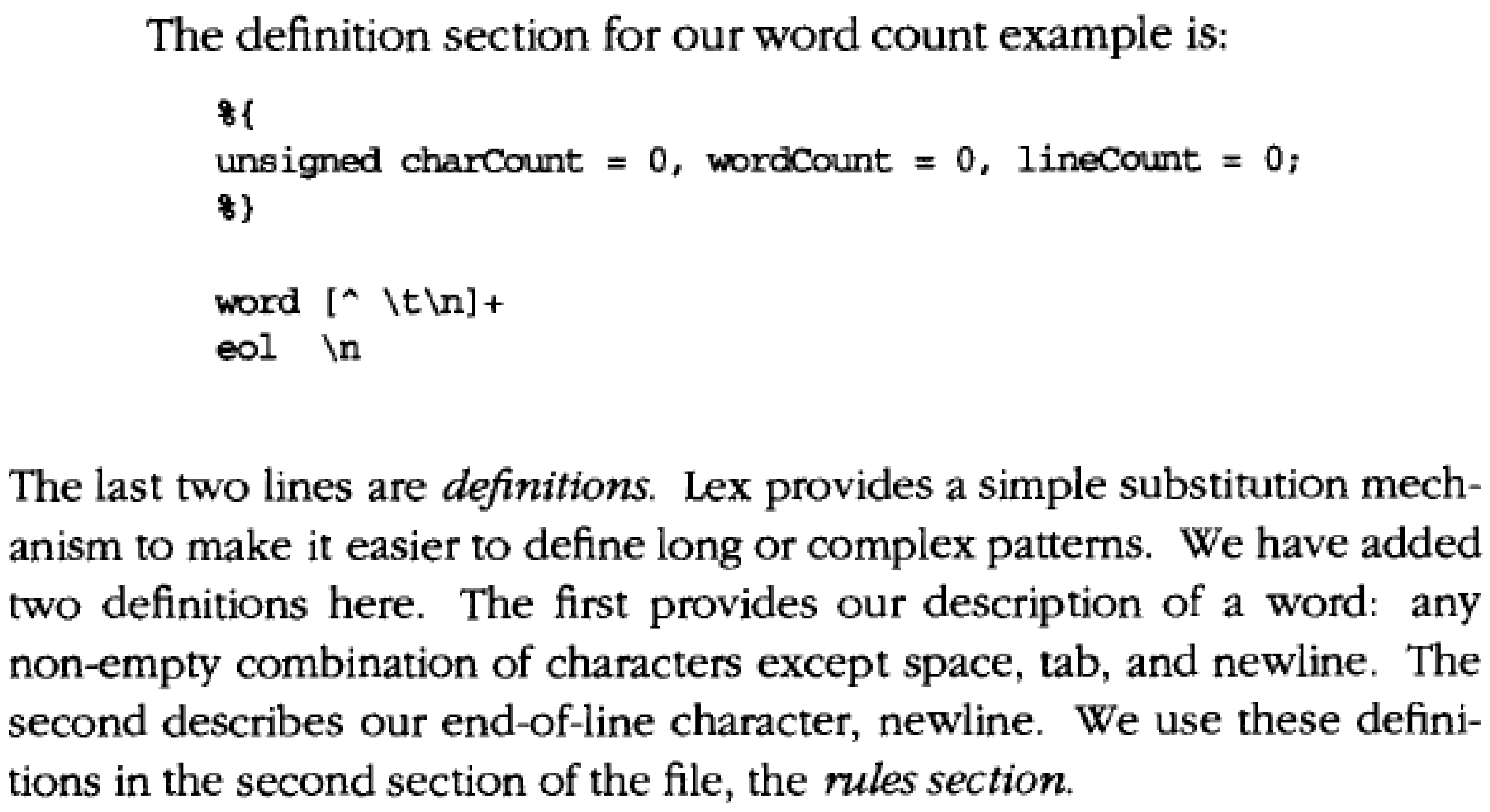
```
#.*
```

The "." matches any character except newline and the "*" means match zero or more of the preceding expression. This expression matches anything on the comment line up to the newline which marks the end of the line.

Finally, here is a regular expression for matching quoted strings:

```
\"[^"\n]*["\n]
```

A Word Counting Program



The definition section for our word count example is:

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol  \n
```

The last two lines are *definitions*. Lex provides a simple substitution mechanism to make it easier to define long or complex patterns. We have added two definitions here. The first provides our description of a word: any non-empty combination of characters except space, tab, and newline. The second describes our end-of-line character, newline. We use these definitions in the second section of the file, the *rules section*.

The rules section contains the patterns and actions that specify the lexer. Here is our sample word count's rules section:

```
%%
{word}      { wordCount++; charCount += yyleng; }
{eol} { charCount++; lineCount++; }
.      charCount++;
```

The beginning of the rules section is marked by a "%%". In a pattern, lex replaces the name inside the braces {} with *substitution*, the actual regular expression in the definition section. Our example increments the number of words and characters after the lexer has recognized a complete word.

The actions which consist of more than one statement are enclosed in braces to make a C language compound statement.

Our sample also uses the lex internal variable **yyleng** which contains the length of the string our lexer recognized. If it matched well-being, **yyleng** would be 10.

When our lexer recognizes a newline, it will increment both the character count and the line count. Similarly, if it recognizes any other character it increments the character count. For this lexer, the only "other characters" it could recognize would be space or tab; anything else would match the first regular expression and be counted as a word.

The lexer always tries to match the longest possible string, but when there are two possible rules that match the same length, the lexer uses the earlier rule in the lex specification. Thus, the word "I" would be matched by the {word} rule, not by the . rule.

The third and final section of the lex specification is the user subroutines section. Once again, it is separated from the previous section by "%%". The user subroutines section can contain any valid C code. It is copied verbatim into the generated lexer. Typically this section contains support routines. For this example our "support" code is the main routine:

```
%%
main()
{
        yylex();
        printf("%d %d %d\n",lineCount, wordCount, charCount);
}
```

(// without considering command line arguments)

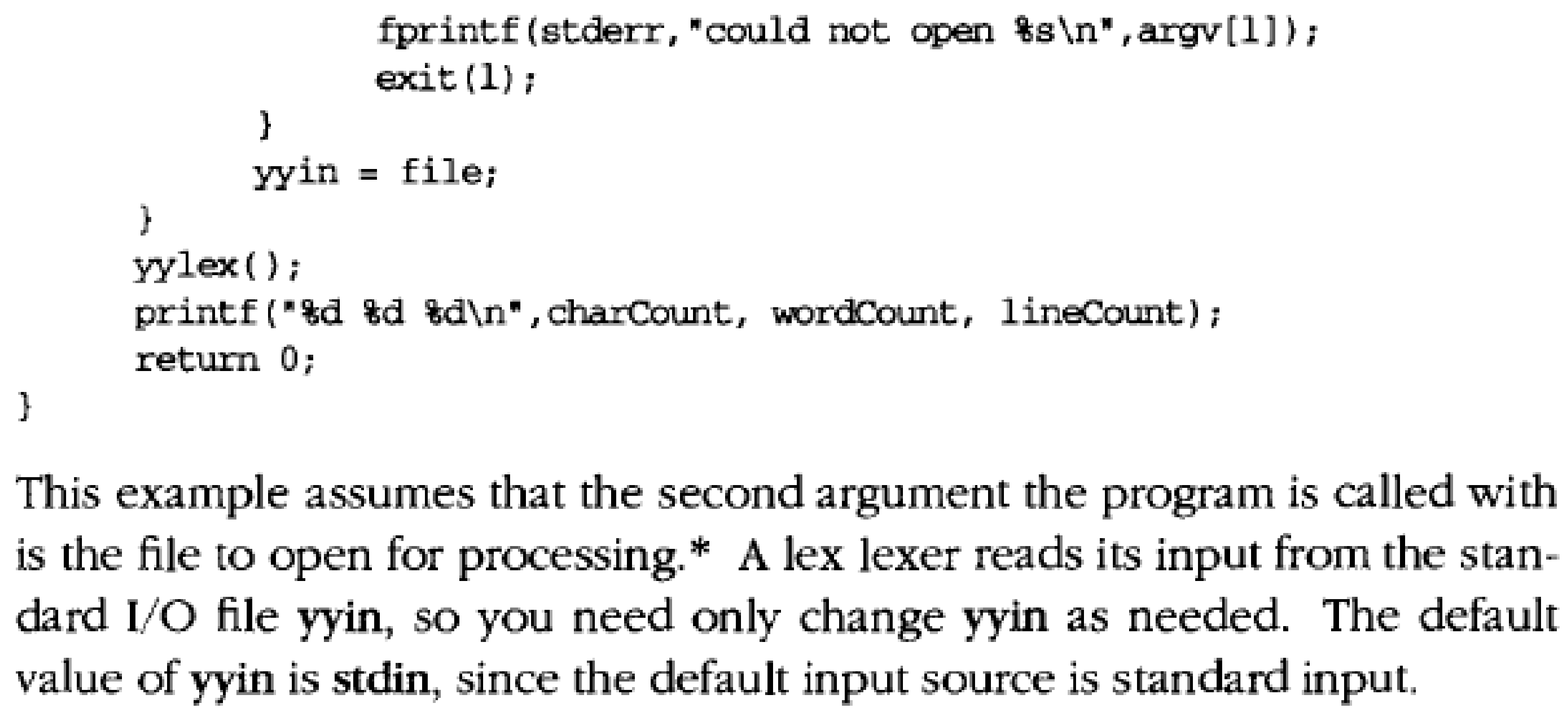
Note that our sample doesn't do anything fancy; it doesn't accept command-line arguments, doesn't open any files, but uses the lex default to read from the standard input.

However, it is worthwhile to look at one way to reconnect lex's input stream, as shown in Example 2-2.

// User Subroutine section to accept input from file using command line arguments

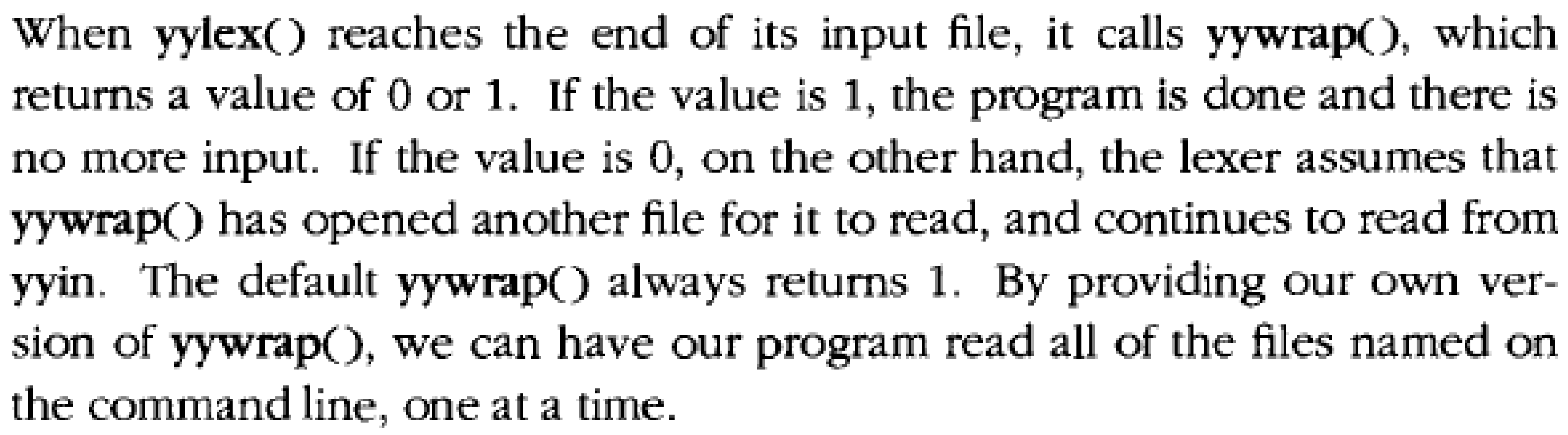
Example 2-2: User subroutines for word count program ch2-02.l

```
main(argc,argv)
int argc;
char **argv;
{
        if (argc > 1) {
                FILE *file;

                file = fopen(argv[1], "r");
                if (!file) {
```



```
                        fprintf(stderr,"could not open %s\n",argv[1]);
                        exit(1);
                }
                yyin = file;
        }
        yylex();
        printf("%d %d %d\n",charCount, wordCount, lineCount);
        return 0;
}
```

This example assumes that the second argument the program is called with is the file to open for processing.* A lex lexer reads its input from the standard I/O file **yyin**, so you need only change **yyin** as needed. The default value of **yyin** is **stdin**, since the default input source is standard input.



When **yylex()** reaches the end of its input file, it calls **yywrap()**, which returns a value of 0 or 1. If the value is 1, the program is done and there is no more input. If the value is 0, on the other hand, the lexer assumes that **yywrap()** has opened another file for it to read, and continues to read from **yyin**. The default **yywrap()** always returns 1. By providing our own version of **yywrap()**, we can have our program read all of the files named on the command line, one at a time.

What is the difference between Lex and Yacc

- Lex is a tool for writing lexical analyzers/scanner. Yacc is Tool for constructing syntax analyzers/parser
- Lex reads a specification file containing regular expressions and generates a C routine that performs lexical analysis. Matches sequences that identify tokens. Yacc reads a specification file that codifies the grammar of a language and generates a parsing routine

Format of a lex specification file:
definitions
%%
regular expressions and associated actions (rules)
%%
user routines

Format of a yacc specification file:
declarations
%%
grammar rules and associated actions
%%
C programs

Yacc specification describes a CFG, that can be used to generate a parser.
Elements of a CFG:
1. Terminals: tokens and literal characters,
2. Variables (nonterminals): syntactical elements,
3. Production rules, and
4. Start variable.

Format of a production rule:
symbol: definition
{action} ;

Chapter 2: Using Yacc

Grammars

- Grammars for yacc are described using a variant of Backus Naur Form (BNF).
- A BNF grammar can be used to express context-free languages.
- Most constructs in modern programming languages can be represented in BNF.

For example, the grammar for an expression that multiplies and adds numbers is

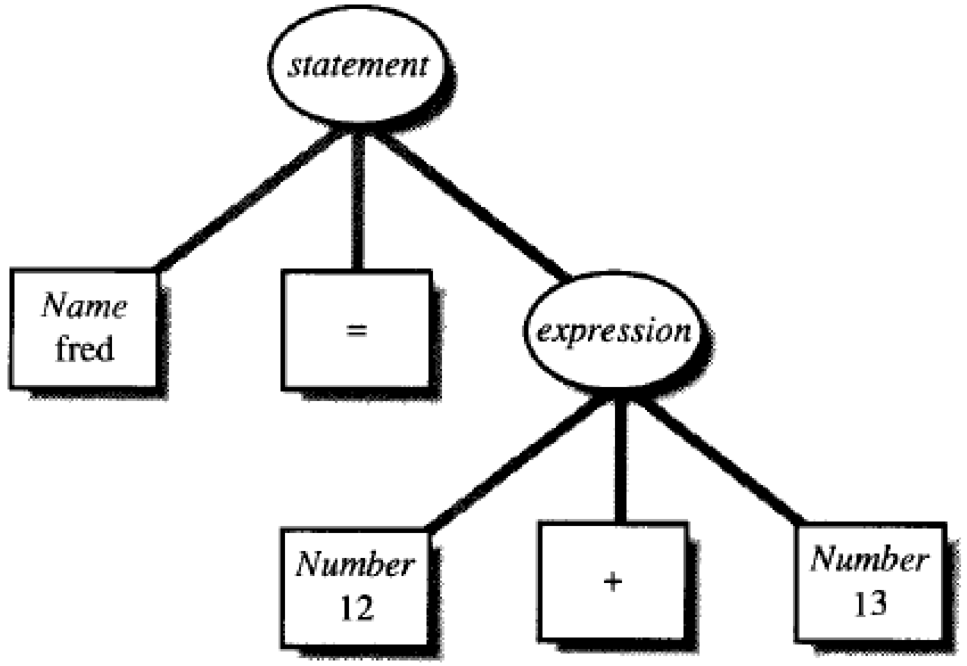
```
1      E -> E + E
2      E -> E * E
3      E -> id
```

Three productions have been specified. **Terms that appear on the left-hand side (lhs) of a production**, such as E (expression) are **nonterminals**. Terms such as id (identifier) are **terminals** (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

```
E -> E * E              (r2)
  -> E * z              (r3)
  -> E + E * z          (r1)
  -> E + y * z          (r3)
  -> x + y * z          (r3)
```

The usual way to represent a parsed sentence is as a tree. For example, if we parsed the input "fred = 12 + 13" with this grammar, the tree would look like Figure 3-1. "12 + 13" is an *expression*, and "fred = *expression*" is a *statement*. A yacc parser doesn't actually create this tree as a data structure, although it is not hard to do so yourself.

A Parse Tree

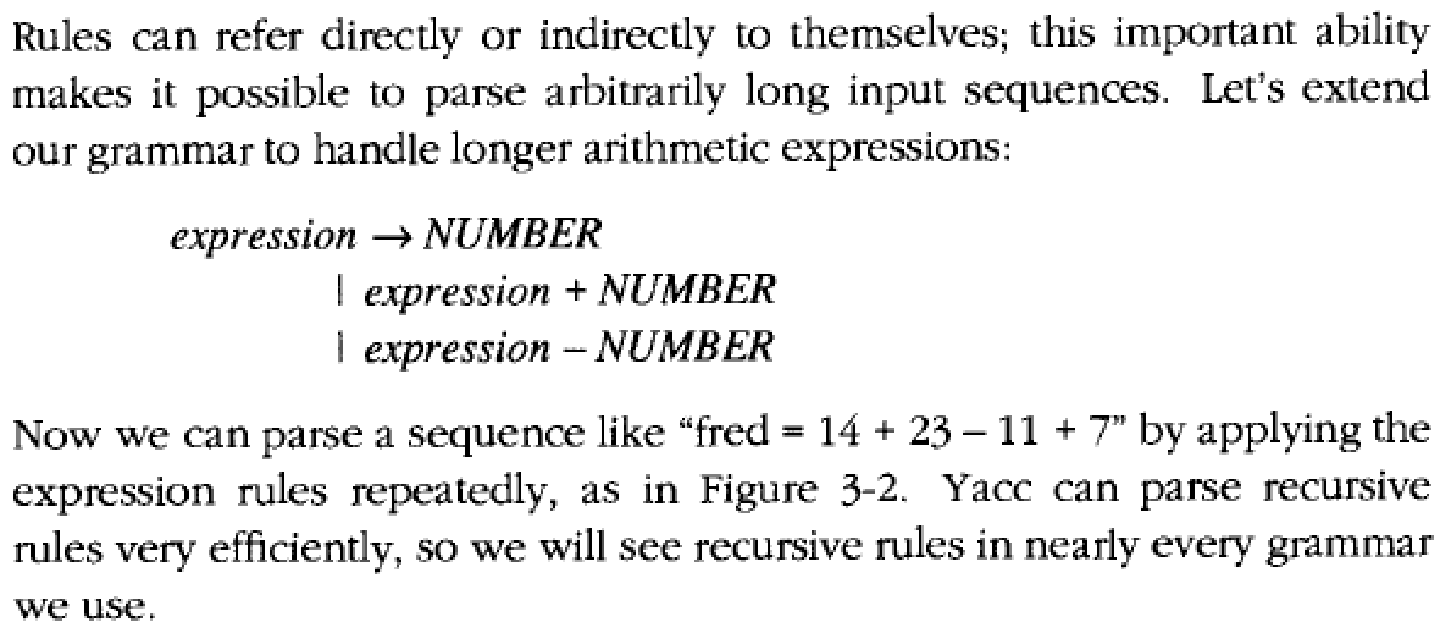


Every grammar includes a *start* symbol, the one that has to be at the root of the parse tree. In this grammar, *statement* is the start symbol.

Recursive Rules

A rule is called **recursive** when its result nonterminal appears also on its right hand side. Nearly all Bison grammars need to use recursion, because that is the only way to define a sequence of any number of a particular thing.

Any kind of sequence can be defined using either left recursion or right recursion, but you should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space. Right recursion uses up space on the Bison stack in proportion to the number of elements in the sequence, because all the elements must be shifted onto the stack before the rule can be applied even once.

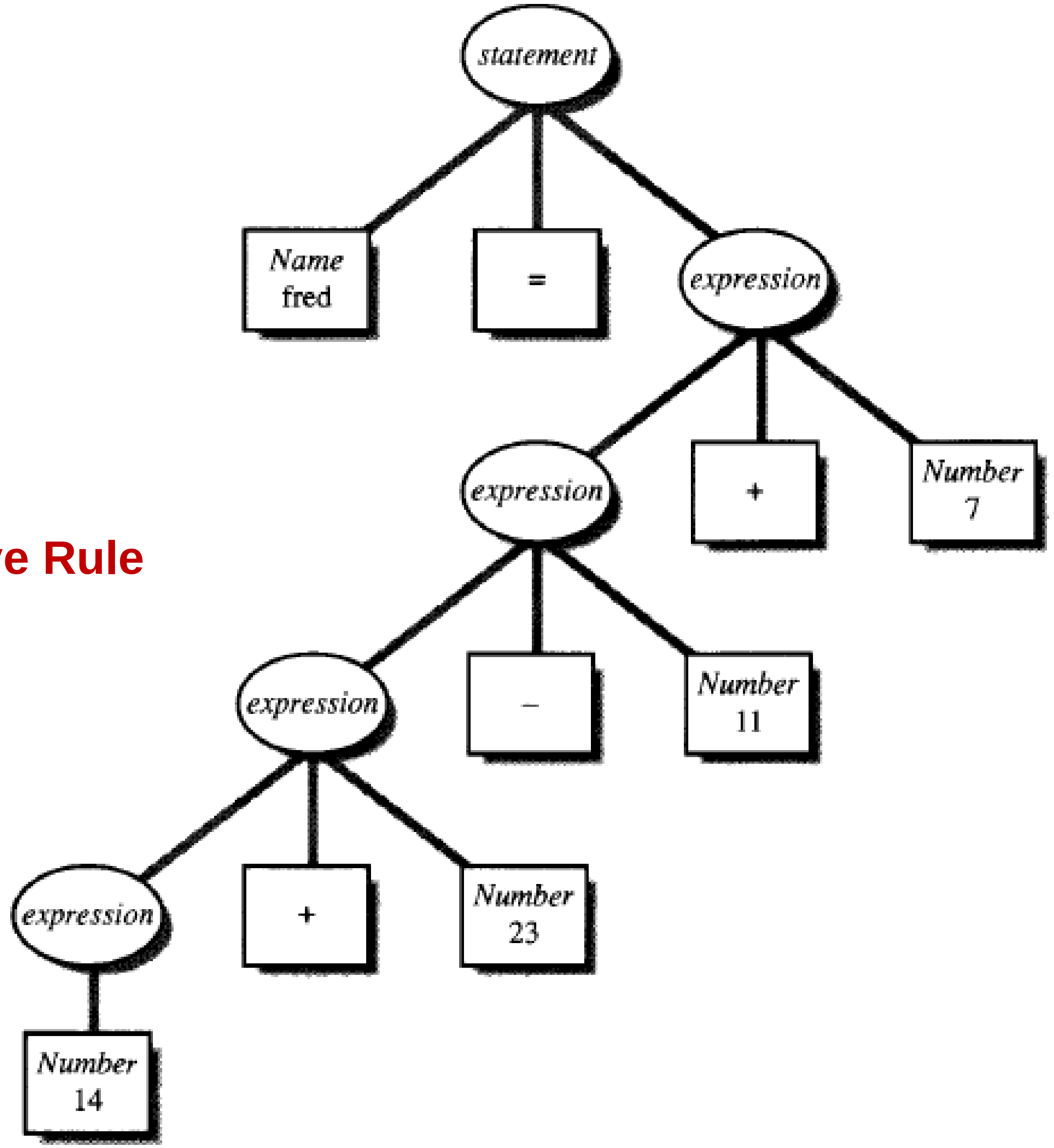


Rules can refer directly or indirectly to themselves; this important ability makes it possible to parse arbitrarily long input sequences. Let's extend our grammar to handle longer arithmetic expressions:

$$\begin{aligned} \mathit{expression} &\rightarrow \mathit{NUMBER} \\ &\mid \mathit{expression} + \mathit{NUMBER} \\ &\mid \mathit{expression} - \mathit{NUMBER} \end{aligned}$$

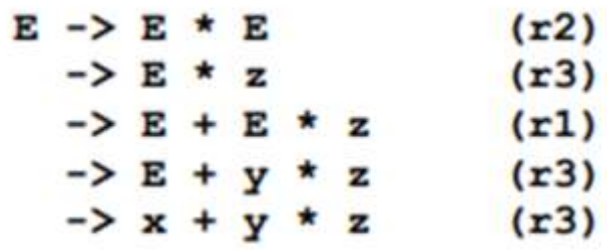
Now we can parse a sequence like "fred = 14 + 23 – 11 + 7" by applying the expression rules repeatedly, as in Figure 3-2. Yacc can parse recursive rules very efficiently, so we will see recursive rules in nearly every grammar we use.

A Parse Tree using Recursive Rule

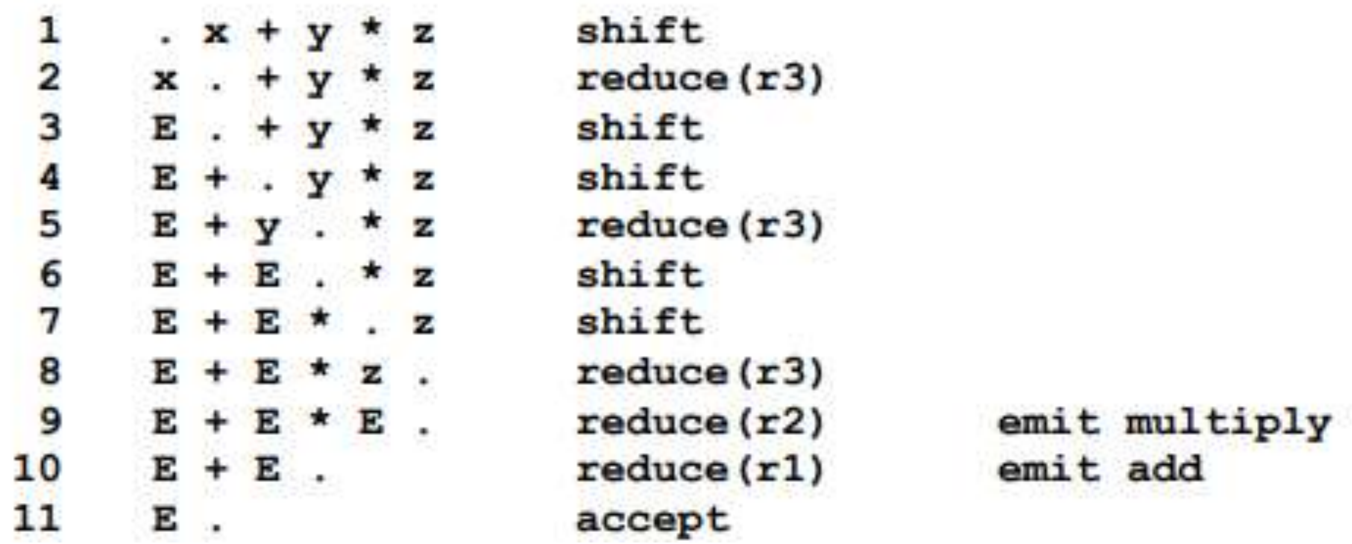


Shift/Reduce Parsing

At each step we expanded a term, replacing the **lhs** of a production with the corresponding **rhs**. The numbers on the right indicate which rule applied. To parse an expression, we actually need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar, we need to reduce an expression to a single nonterminal. This is known as **bottom-up or shift-reduce parsing**, and uses a stack for storing terms. Here is the same derivation, but in reverse order:



```
E -> E * E              (r2)
  -> E * z              (r3)
  -> E + E * z          (r1)
  -> E + y * z          (r3)
  -> x + y * z          (r3)
```



```
 1    . x + y * z     shift
 2    x . + y * z     reduce(r3)
 3    E . + y * z     shift
 4    E + . y * z     shift
 5    E + y . * z     reduce(r3)
 6    E + E . * z     shift
 7    E + E * . z     shift
 8    E + E * z .     reduce(r3)
 9    E + E * E .     reduce(r2)     emit multiply
10    E + E .         reduce(r1)     emit add
11    E .             accept
```

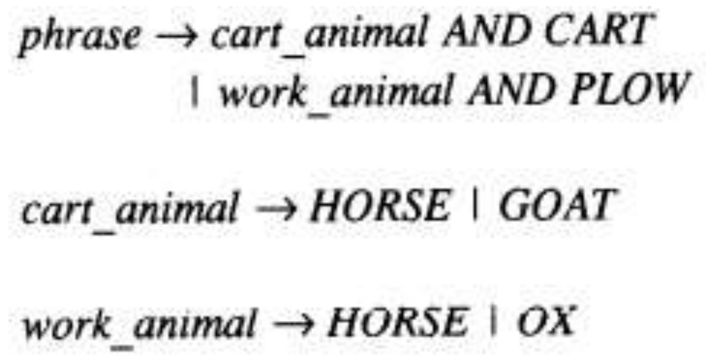
Terms to the left of the dot are on the stack, while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production, we replace the matched tokens on the stack with the lhs of the production. Conceptually, the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a handle, and we are reducing the handle to the lhs of the production.

This process continues until we have shifted all input to the stack, and only the starting nonterminal remains on the stack. In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack, changing x to E.

We continue shifting and reducing, until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly, the add instruction is emitted in step 10. Thus, multiply has a higher precedence than addition

What YACC Cannot Parse

Although yacc's parsing technique is general, you can write grammars which yacc cannot handle. It cannot deal with ambiguous grammars, ones in which the same input can match more than one parse tree.* It also cannot deal with grammars that need more than one token of lookahead to tell whether it has matched a rule. Consider this extremely contrived example



```
phrase → cart_animal AND CART
       | work_animal AND PLOW

cart_animal → HORSE | GOAT

work_animal → HORSE | OX
```

This grammar isn't ambiguous, since there is only one possible parse tree for any valid input, but yacc can't handle it because it requires two symbols of lookahead. In particular, in the input "HORSE AND CART" it cannot tell whether HORSE is a cart-animal or a work-animal until it sees CART, and yacc cannot look that far ahead.

If we changed the first rule to this:

$$\begin{aligned} \textit{phrase} &\rightarrow \textit{cart\_animal CART} \\ &\mid \textit{work\_animal PLOW} \end{aligned}$$

yacc would have no trouble, since it can look one token ahead to see whether an input of HORSE is followed by CART, in which case the horse is a cart-animal or by PLOW in which case it is a work-animal.

A YACC Parser - The Definition Section, The Rules Section

A yacc grammar has the same three-part structure as a lex specification.

- The first section, the definition section, handles control information for the yacc-generated parser and generally sets up the execution environment in which the parser will operate.
- The second section contains the rules for the parser,
- The third section is C code

Definition Section

The definition section includes declarations of the tokens used in the grammar, the types of values used on the parser stack, and other odds and ends. It can also include a literal block.

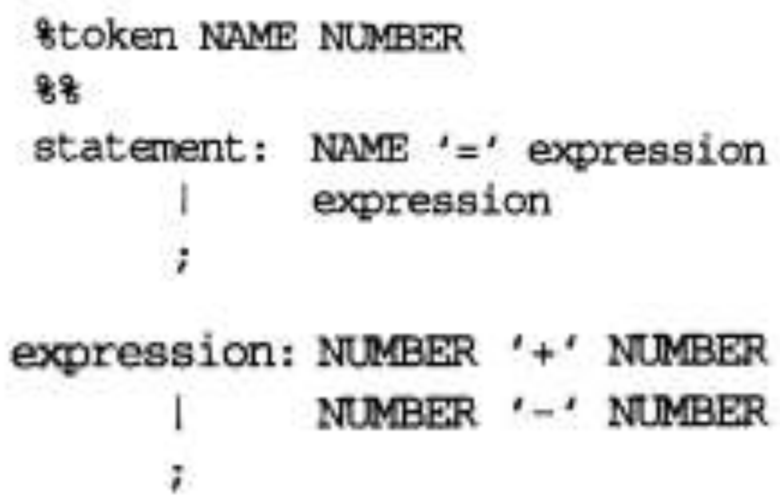
We start our first parser by declaring two symbolic tokens.

```
%token NAME  NUMBER
```

You can use single quoted characters as tokens without declaring them, so we don't need to declare "=", "+", or "-".

Rules Section

The rules section simply consists of a list of grammar rules in much the same format as we used above. Since ASCII keyboards don't have a → key, we use a colon (:) between the left- and right-hand sides of a rule, and we put a semicolon at the end of each rule

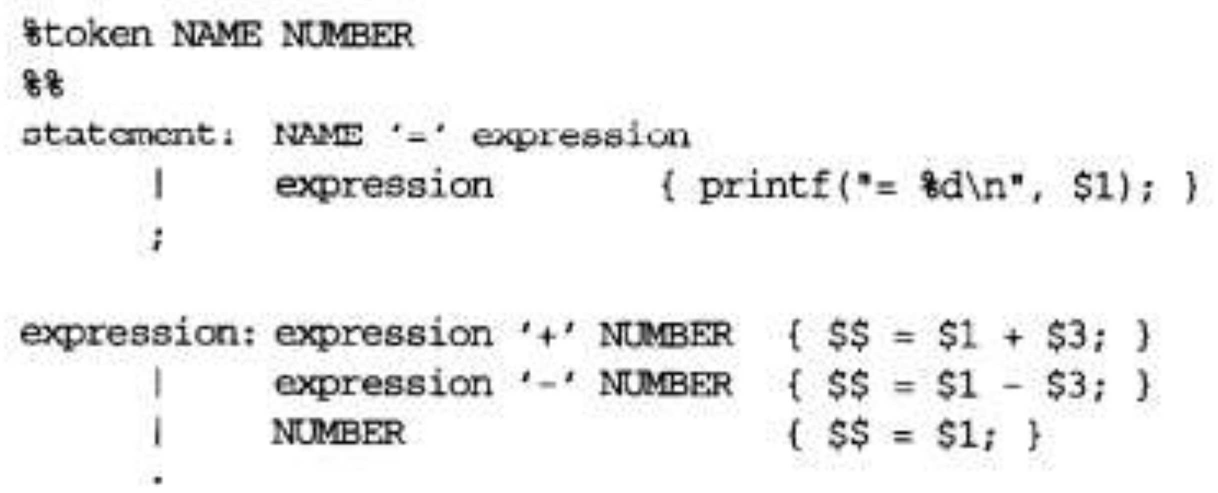


```
%token NAME NUMBER
%%
statement:  NAME '=' expression
       |    expression
       ;

expression: NUMBER '+' NUMBER
       |    NUMBER '-' NUMBER
       ;
```

Symbol Values and Actions

Every symbol in a yacc parser has a value. The value gives additional information about a particular instance of a symbol. If a symbol represents a number, the value would be the particular number. If it represents a literal text string, the value would probably be a pointer to a copy of the string. If it represents a variable in a program, the value would be a pointer to a symbol table entry describing the variable

Whenever the parser reduces a rule, it executes user C code associated with the rule, known as the rule's action. The action appears in braces after the end of the rule, before the semicolon or vertical bar. The action code can refer to the values of the right-hand side symbols as $1, $2, . . . , and can set the value of the left-hand side by setting $$. I

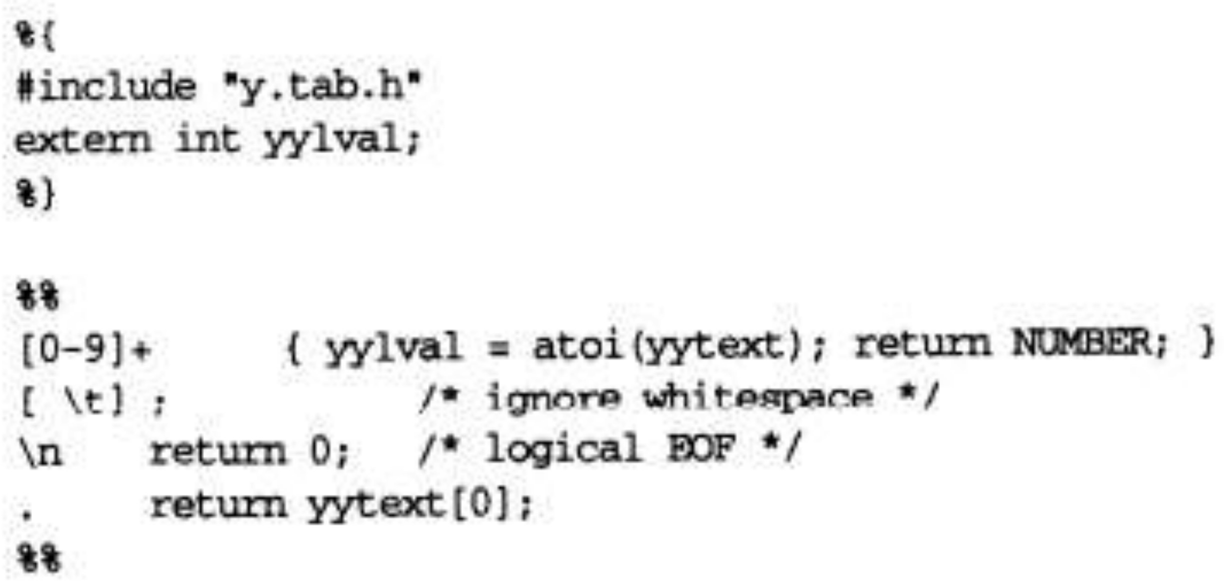


```
%token NAME NUMBER
%%
statement:  NAME '=' expression
        |   expression          { printf("= %d\n", $1); }
        ;

expression: expression '+' NUMBER  { $$ = $1 + $3; }
        |   expression '-' NUMBER  { $$ = $1 - $3; }
        |   NUMBER                 { $$ = $1; }
        ;
```

The LEXER

- To try out our parser, we need a lexer to feed it tokens. As we mentioned earlier, the parser is the higher level routine, and calls the lexer **yylex()** whenever it needs a token from the input.
- As soon as the lexer finds a token of interest to the parser, it returns to the parser, returning the token code as the value.
- Yacc defines the token names in the parser as C preprocessor names in y.tab.h so the lexer can use them.

Whenever the lexer returns a token to the parser, if the token has an associated value, the lexer must store the value in yylval before returning. In this first example, we explicitly declare yylval. In more coniplex parsers, yacc defines yylval as a union and puts the definition in y.tab.h



```
%{
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+        { yylval = atoi(yytext); return NUMBER; }
[ \t] ;                 /* ignore whitespace */
\n      return 0;       /* logical EOF */
.       return yytext[0];
%%
```

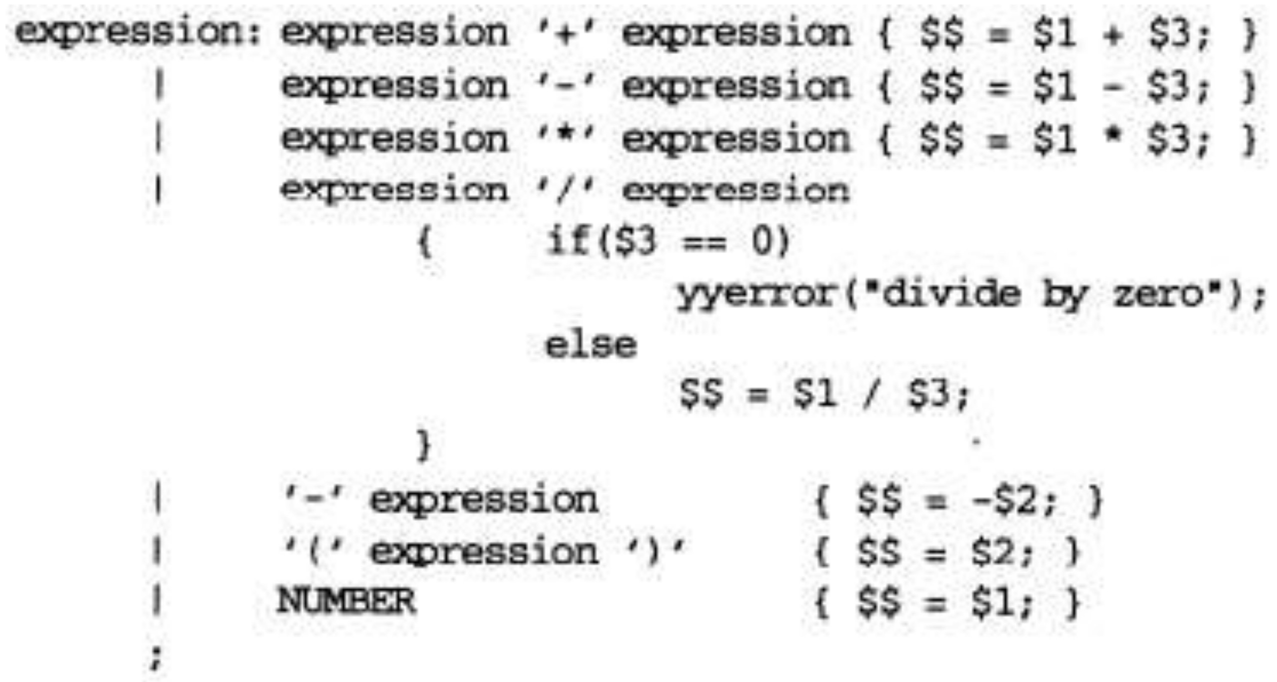
Here is a simple lexer to provide tokens for our parser:

Compiling and Running a Simple Parser

```
$ lex filename.l
$ yacc -d filename.y
$ cc  lex.yy.c  y.tab.c -ll
$ ./a.out
```

Arithmetic Expressions and Ambiguity.

Let's make the arithmetic expressions more general and realistic, extending the expression rules to handle multiplication and division, unary negation, and parenthesized expressions:



```
expression: expression '+' expression { $$ = $1 + $3; }
          |     expression '-' expression { $$ = $1 - $3; }
          |     expression '*' expression { $$ = $1 * $3; }
          |     expression '/' expression
                    {     if($3 == 0)
                                yyerror("divide by zero");
                          else
                                $$ = $1 / $3;
                    }
          |     '-' expression            { $$ = -$2; }
          |     '(' expression ')'        { $$ = $2; }
          |     NUMBER                    { $$ = $1; }
          ;
```

The action for division checks for division by zero, since in many implementations of C a zero divide will crash the program.
It calls **yyerror(),** the standard yacc error routine, to report the error.

But this grammar has a problem: it is extremely ambiguous.

For example, the input 2+3*4 might mean (2+3)*4 or 2+(3*4), and the input 3-4-5-6 might mean 3-(4-(5-6)) or (3-4)-(5-6) or any of a lot of other possibilities.

Figure 3-3 shows the two possible parses for 2+3’4.

If you compile this grammar as it stands, yacc will tell you that there are 16 shift/reduce conflicts, states where it cannot tell whether it should shift the token on the stack or reduce a rule first.

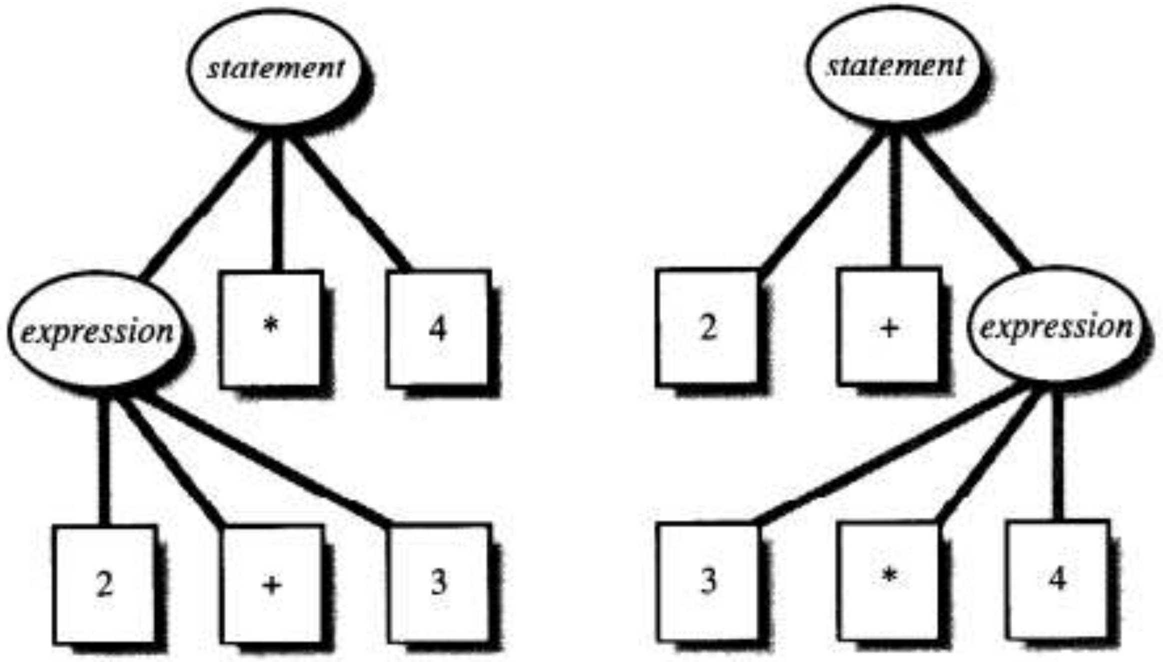
For example, when parsing "2+3*4", the parser goes through these steps (we abbreviate expression as E here):

2	shift NUMBER
E	reduce E → NUMBER
E +	shift +
E + 3	shift NUMBER
E + *E*	reduce E → NUMBER

At this point, the parser looks at the "*", and could either reduce "2+3" using:
expression: expression '+' expression
to an expression,

or shift the "*" expecting to be able to reduce:
expression: expression '*' expression
later on.

Figure 3-3: Ambiguous input 2+3*4



The problem is that we haven't told yacc about the precedence and associativity of the operators.

Precedence controls which operators to execute first in an expression. Mathematical and programming says that multiplication and division take precedence over addition and subtraction, so a+b*c means a+(b*c) and d/e-f means (d/e)-f. In any expression grammar, operators are grouped into levels of precedence from lowest to highest.

Associativity controls the grouping of operators at the same precedence level. Operators may group to the left, e.g., a-b-c in C means (a-b)-c, or to the right, e.g., a=b=c in C means a=(b=c).

There are two ways to specify precedence and associativity in a grammar, implicitly and explicitly.

To specify them implicitly, rewrite the grammar using separate non-terminal symbols for each precedence level.

Assuming the usual precedence and left associativity for everything, we could rewrite our expression rules this way:

```
expression: expression '+' mulexp
        |   expression '-' mulexp
        |   mulexp
        ;

mulexp:         mulexp '*' primary
        |   mulexp '/' primary
        |   primary
        ;

primary:    '(' expression ')'
        |   '-' primary
        |   NUMBER
        ;
```

This is a perfectly reasonable way to write a grammar, and if yacc didn't have explicit precedence rules, it would be the only way. But yacc also lets you specify precedences explicitly. We can add these lines to the definition section, resulting in the grammar:

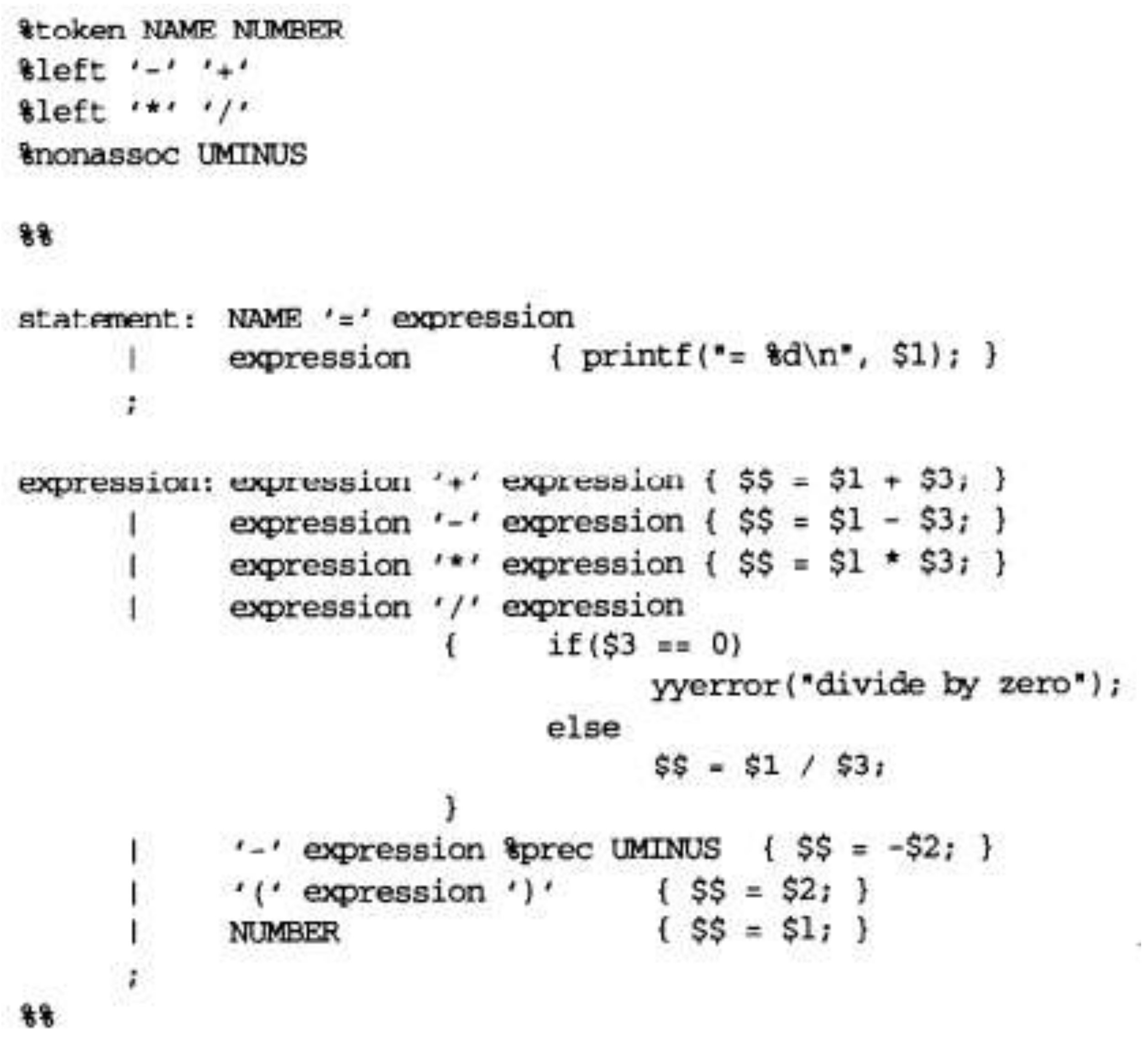
```
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
```

Each of these declarations defines a level of precedence. They tell yacc that "+" and "-" are left associative and at the lowest precedence level, "*" and "/" are left associative and at a higher precedence level, and UMINUS, a pseudo-token standing for unary minus, has no associativity and is at the highest precedence. (We don't have any right associative operators here, but if we did they'd use %right.)

Yacc assigns each rule the precedence of the rightmost token on the right-hand side; if the rule contains no tokens with precedence assigned, the rule has no precedence of its own.

When yacc encounters a shift/reduce conflict due to an ambiguous grammar, it consults the table of precedences, and if all of the rules involved in the conflict include a token which appears in a precedence declaration, it uses precedence to resolve the conflict.

Example : The calculator grammar with expressions and precedence



```
%token NAME NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%%

statement:  NAME '=' expression
        |   expression          { printf("= %d\n", $1); }
        ;

expression: expression '+' expression { $$ = $1 + $3; }
        |   expression '-' expression { $$ = $1 - $3; }
        |   expression '*' expression { $$ = $1 * $3; }
        |   expression '/' expression
                              {     if($3 == 0)
                                            yyerror("divide by zero");
                                    else
                                            $$ = $1 / $3;
                              }
        |   '-' expression %prec UMINUS   { $$ = -$2; }
        |   '(' expression ')'          { $$ = $2; }
        |   NUMBER                      { $$ = $1; }
        ;
%%
```

Ambiguity and Conflicts

A set of grammar rules is ambiguous if some input string can be structured in two or more different ways. For example, the following grammar rule is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them:

expr : expr '-' expr

Notice that this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is the following:

expr - expr - expr

the rule allows this input to be structured as either:

(expr - expr) - expr

or as:

expr - (expr - expr)

The first is called left association, the second right association.

yacc detects such ambiguities when it is attempting to build the parser. Given that the input is as follows, consider the problem that confronts the parser:

expr - expr – expr

When the parser has read the second expr, the input seen is:

expr – expr

It matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to expr (the left side of the rule). The parser then reads the final part of the input (as represented below) and again reduce:

- expr

The effect of this is to take the left associative interpretation.

Alternatively, if the parser sees the following:

expr - expr

it could defer the immediate application of the rule and continue reading the input until the following is seen:

expr - expr – expr

It could then apply the rule to the rightmost three symbols, reducing them to expr, which results in the following being left:

expr – expr

Now the rule can be reduced once more. The effect is to take the right associative interpretation. Thus, having read the following, the parser can do one of two legal things, shift or reduce:

expr – expr

It has no way of deciding between them. This is called a shift-reduce conflict. It might also happen that the parser has a choice of two legal reductions. This is called a reduce-reduce conflict. There are never any shift-shift conflicts.

When there are shift-reduce or reduce-reduce conflicts, yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing the choice to make in a given situation is called a disambiguating rule.

Yacc invokes two default disambiguating rules:

- *In a shift-reduce conflict, the default is to shift.*
- *In a reduce-reduce conflict, the default is to reduce by the earlier grammar rule (in the yacc specification).*

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. **Rule 2** gives the user rather crude control over the behavior of the parser in this situation, but reduce-reduce conflicts should be avoided when possible.

Conflicts can arise because of mistakes in input or logic or because the grammar rules (while consistent) require a more complex parser than yacc can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized.

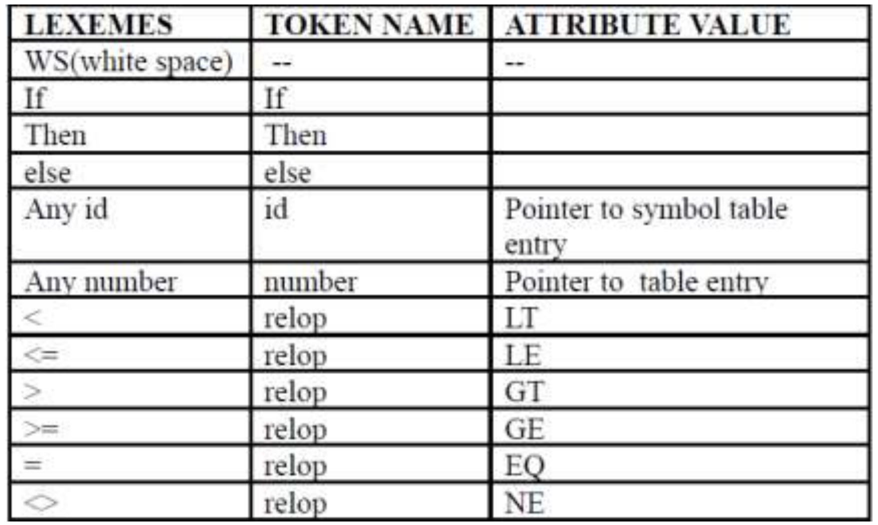
In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, yacc always reports the number of shift-reduce and reduce-reduce conflicts resolved by rules 1 and 2 above.

Question Bank-Module-4

1. Explain the basic sections/ structure of LEX and Yacc program with Example.*(Lex: slide 9; Yacc: slide 54+ any example prog for ex: Lab 1b prog)*
2. Write Lex Program to count words, characters, lines in an input file. *(Ans: slide:52-56)*
3. Write Lex Program to recognize/identify parts of a speech *(Ans: slide:20-22)*
4. With a block diagram explain how Lexer and Parser communicate*(Ans: slide:31—33)*
5. Define Regular Expression. Briefly explain the use of following Meta Characters with suitable example: *(Ans: slide:46--48)*

    i. . ii. * iii. + iv. $ v. ^ vi. ? vii. { } viii. \d

6. Write a lex program to recognize the tokens and return the tokens considering attribute values as given in below table: *(Ans: slide:23)*



LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
WS(white space)	--	--
If	If	
Then	Then	
else	else	
Any id	id	Pointer to symbol table entry
Any number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
>	relop	GT
>=	relop	GE
=	relop	EQ
<>	relop	NE

7) Explain use of following listed Lex libraries: *(Ans: slide:10-12)*
yylex, yyparse, yyin, yytext, yylval, yyout, yywrap
8) Differentiate between Lex and Yacc *(Ans: slide:57)*
9) Define Symbol Table. Explain how symbol table is used in Lexer. *(Ans: slide:25- 27*
10) Explain the difference between Lex and hand written Lexer *(Ans: slide 43-45)*
11) Explain the working of shift reduce parser. *(Ans: slide: 62-63)*
12) Discuss two types of conflicts in YACC with an example. *(Ans: slide: 78-81)*
13) What is ambiguous grammar? Explain with an example how can it be overcome? *(Ans: slide: 71-77)*
14) Explain precedence and associativity. *(Ans: slide: 74)*
15) Explain recursive rules with an example *(Ans: slide:60-61)*
16) Write a YACC program to check whether the given arithmetic expression is valid or not. *(Ans: Lab prog 1b)*