Module-4
Lex and Yacc

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.

Lex and Yacc help us to write programs that transform structured input.Applications

- Simple text search program
- Compiler

Lex is a tool for building lexical analyzers or lexers i.e. yylex(). A lexer takes an arbitrary input stream and tokenizes it., divides it up into lexical tokens. This tokenized output can then be processed further, usually by yacc, or it can be "end product".

When we write a lex specification, we create a set of patterns which lex matches against the input. Each time one of the patterns matches, the lex program invokes C Code that you provide which does something with the matched text. In this way a lex program divides the input into strings which we call tokens. Lex itself doesn't produce an executable program; instead it translates the lex specification into a file containing a C routine called yylex(). Your program calls yylex() to run the lexer.

Structure of a Lex file

Lex file is divided into three sections, separated by lines that contain only two percent signs, asfollows:

```
        Definitions
%%
        Rules
%%
        User subroutines
```

red :	required
blue :	optional

- The **definition** section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

  - Literal Block: C code bracketed by lines *"%{"* and *"%}"*
  - Definitions: **NAME** <*Expression*>

    DIG [0-9]

- The **rules** section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
  - Pattern {action}
- The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in therules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

Example of a Lex file

The following is an example Lex file for the flex version of Lex. It recognizes strings of numbers(positive integers) in the input, and simply prints them out.

```
/*** Definition section ***/
%{
/* C code to be copied verbatim */
#include <stdio.h>
%}

/* This tells flex to read only one input file */
% option noyywrap

%%
   /*** Rules section ***/

   /* [0-9]+ matches a string of one or more digits */
   [0-9]+  {/* yytext is a string containing the matched text. */
                printf("Saw an integer: %s\n", yytext);}

    .|\n      {  /* Ignore all other characters. */   }

%%
/*** C Code section ***/
int main(void)
{
   /* Call the lexer, then quit. */
   yylex();
   return 0;
}
```

If this input is given to flex, it will be converted into a C file, lex.yy.c. This can be compiled into an executable which matches and outputs strings of integers. For example, given the input:

```
abc123z.!&*2gj6
```

the program will print:

```
Saw an integer: 123
Saw an integer: 2
Saw an integer: 6
```

Regular expressions in lex are composed of *metacharacters* (Table 2-1). Pattern matching examples are shown in Table 2-2. Within a character class, normal operators lose their meaning. Two operators allowed in a character class are the hyphen (“`-`”) and circumflex (“`^`”). When used between two characters, the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

Metacharacter	*Matches*
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of preceding expression
`+`	one or more copies of preceding expression
`?`	zero or one copy of preceeding 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 2-1: Pattern Matching Primitives

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

Table 2-2: Pattern Matching Examples

<u>LEX REGULAR EXPRESSIONS.</u> A LEX regular expression is a word made of

- text characters (letters of the alphabet, digits, ...)
- operators : " \ { } [ ] ^ $ < > ? . * + | () /Moreover

- An operator can be used as a text character if it is precedes with the escape operator(backslash).
- The quotation marks indicate that whatever is contained between a pair of quotes is to betaken as text characters. For instance

  xyz"++"

  matches the string xyz++.

<u>A CHARACTER CLASS</u> is a class of characters specified using the operator pair [ ].

The expression [ab] matches the string a or b.

Within square brackets most operators are ignored except the three special characters \ - ^ are which used as follows

(a) the escape character \ as above,

(b) the minus character - which is used for ranges like indigit [0-9]

(c) the *hat* character ^ as first character after the opening square bracket, it is used for complemented matches like in

NOTabc [^abc]

OPTIONAL EXPRESSIONS. The ? operator indicates an optional element of an expression. For instance

ab?c matches either ac or abc.

REPEATED EXPRESSIONS. Repetitions of patterns are indicated by the operators * and +.

- The pattern a* matches any number of consecutive a characters (including zero).
- The pattern [a-z]+ is any positive number of consecutive lower-case alphabeticcharacters.

Hence we can recognize identifiers in a typical computer language with[A-Za-z][A-Za-z0-9]*
Repetitions can also be obtained with the pair operator {}.

- If {} encloses numbers, it specifies repetitions. For instance a{1,5} matches 1 to 5repetitions of a.
- **Note** that if {} encloses a name, this name should be defined in the definition section.Then LEX substitutes the definition for the name.

ALTERNATING. The operator | indicates alternation.

For instance(ab|cd) matches the language consisting of both words ab and cd.

GROUPING. Parentheses are used for grouping (when not clear).

For instance(ab|cd+)?(ef)*
denotes the language of the words that are either empty or

- optionally starts with
  - ab or
  - c followed by any positive number of d
- and continues with any number of repetition of ef

Another example: an expression specifying a real number

-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
where \. denotes a literal period.

CONTEXT SENSITIVITY. LEX provides some support for contextual grammatical rules.

- If ^ is the first character in an expression, then this expression will only be matched at the beginning of the line.
    - ^a {flag='a'; ECHO;}
- If $ is the last character in an expression, then this expression will only be matched at the end of a line.
- If *r* and *s* are two LEX regular expressions then *r*/*s* is another LEX regular expression.
    - It matches *r* if and only if it is followed by an *s*.
    - It is called a *trailing context*.
    - After use in this context, *s* is then returned to the input before the action isexecuted. So the action only sees the text matched by *r*
    - *Left context* is handled by means of *start conditions*

Lex Program Start Conditions

A rule may be associated with any start condition. However, the **lex** program recognizes the rule only when in that associated start condition. You can change the current start condition at any time.

Define start conditions in the *definitions* section of the specification file by using a line in the following form:

%Start name1 name2

where name1 and name2 define names that represent conditions. There is no limit to the number of conditions, and they can appear in any order. You can also shorten the word Start to s or S.

When using a start condition in the rules section of the specification file, enclose the name of the start condition in <> (less than, greater than) symbols at the beginning of the rule. The following example defines a rule, expression, that the **lex** program recognizes only when the **lex** program isin start condition name1:

<name1> expression

To put the **lex** program in a particular start condition, execute the action statement in the action part of a rule; for instance, BEGIN in the following line:

BEGIN name1;

This statement changes the start condition to name1.

To resume the normal state, enter:

BEGIN 0;

or

BEGIN INITIAL;

where INITIAL is defined to be 0 by the **lex** program. BEGIN 0; resets the **lex** program to its initial condition.

The **lex** program also supports exclusive start conditions specified with %**x** (percent sign, lowercase x) or %**X** (percent sign, uppercase X) operator followed by a list of exclusive start names in the same format as regular start conditions. Exclusive start conditions differ from regular start conditions in that rules that do not begin with a start condition are not active when the lexical analyzer is in an exclusive start state. For example:

```
%s      one
%x      two
%%
abc     {printf("matched "); ECHO;BEGIN one;}
<one>def        {printf("matched "); ECHO;BEGIN two;}
<two>ghi        {printf("matched "); ECHO;BEGIN INITIAL;}
```

In start state one in the preceding example, both abc and def can be matched. In start state two, only ghi can be matched.

Parser-Lexer Communication

When you use a lex scanner and a yacc parser together, the parser is the higher level routine. It calls the lexer **yylex()** whenever it needs a token from the input. The lexer then scans through theinput recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token’s code as the value of **yylex()**.

Not all tokens are of interest to the parser—in most programming languages the parser doesn’t want to hear about comments and whitespace, for example. For these ignored tokens, the lexer doesn’t return so that it can continue on to the next token without bothering the parser.

The lexer and the parser have to agree what the token codes are. We solve this problem by lettingyacc define the token codes.

Grammar

%token A B

%%

str: S

  ;

S: A S B

  |

  ;

%%

The tokens in our grammar are:A and B . Yacc defines each of these as a small integer using apreprocessor *#define*. Here are the definitions it used in this example:

```
# define A
251# define
B 252
```

Token code zero is always returned for the logical end of the input. Yacc doesn't define a symbolfor it, but you can yourself if you want.
Yacc can optionally write a C header file containing all of the token definitions. You include thisfile, called *y.tab.h* on UNIX systems, in the lexer *and use the preprocessor symbols in your lexeraction code.*

LEX Programs

1. **Program to copy its standard input to its standard output.**

```
%%
%%

(or)

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

2. **Program to count the number of lines in its standard input.**

```
%{
     #include<stdio.h>
      int count=0;
%}

%%

\n   count++;
.    ;

%%

void main( )
{
     yylex( );
     printf("The total number of  lines = %d\n", count);
}
```

3. **/*lex program to count number of words*/**

```
%{
#include<stdio.h>
#include<string.h>
int i = 0;
%}

/* Rules Section*/
%%
([a-zA-Z0-9])* {i++;} /* Rule for counting
                             number of words*/

"\n" {printf("%d\n", i); i = 0;}
%%

int yywrap(void){}

int main()
{
    // The function that starts the analysis
    yylex();

    return 0;
}
```

4. **/*lex code to count words that are less than 10 and greater than 5 */**

```
%{
int len=0, counter=0;
%}

%%
[a-zA-Z]+ { len=strlen(yytext);
             if(len<10 && len>5)
             {counter++;} }
%%

int yywrap (void )
{
   return 1;
}

int main()
{
printf("Enter the string:");
yylex();
printf("\n %d", counter);
return 0;
}
```

5. **/* Lex program to Identify and Count Positive and Negative Numbers */**

```
%{
int positive_no = 0, negative_no = 0;
%}

/* Rules for identifying and counting
positive and negative numbers*/
%%
^[-][0-9]+ {negative_no++;
             printf("negative number = %s\n",
                  yytext);} // negative number

[0-9]+ {positive_no++;
        printf("positive number = %s\n",
                  yytext);} // positive number
%%

/*** use code section ***/
```

```
int yywrap(){}
int main()
{

yylex();
printf ("number of positive numbers = %d,"
        "number of negative numbers = %d\n",
                  positive_no, negative_no);

return 0;
}
```

6. LEX program to count the number of vowels and consonants in a given string

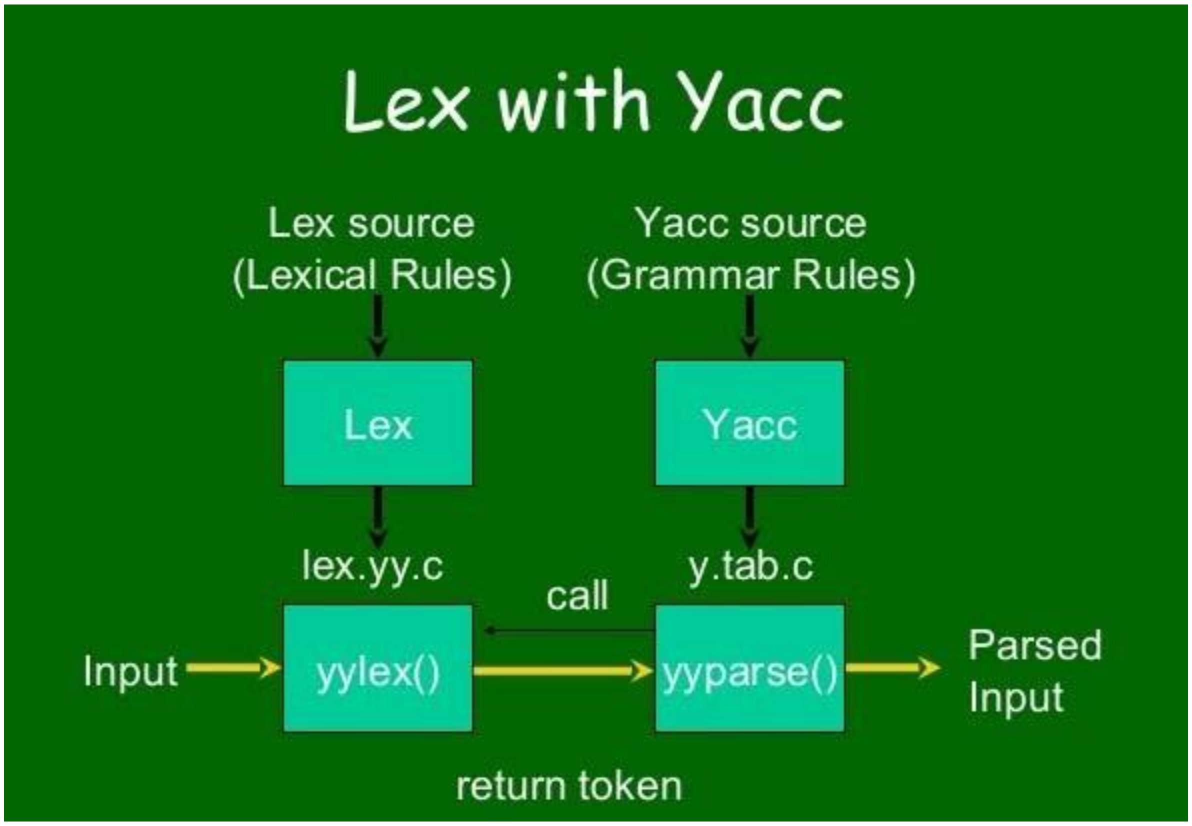
```
%{
 int vow_count=0;
 int const_count =0;
%}

%%
[aeiouAEIOU] {vow_count++;}
[a-zA-Z] {const_count++;}
%%
int yywrap(){}
int main()
{
 printf("Enter the string of vowels and consonents:");
 yylex();
 printf("Number of vowels are: %d\n", vow_count);
 printf("Number of consonants are: %d\n", const_count);
 return 0;
}
```

YACC

Yacc (for "**yet another compiler compiler**.") is the standard parser(yyparse()) generator for the Unix operating system. An open source program, **yacc** generates code for the parser in the C programming language.

Lex divides the input stream into pieces (tokens) and then yacc takes these pieces and groups them together logically.

Parser Lexer Communication



When you use a lex scanner and a yacc parser together, the parser(i.e.yyparse( )) is the higher level routine. It calls the lexer **yylex()** whenever it needs a token from the input. The lexer then scans through the input recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's code as the value of **yylex()**.

Not all tokens are of interest to the parser—in most programming languages the parser doesn't want to hear about comments and whitespace, for example. For these ignored tokens, the lexer doesn't return so that it can continue on to the next token without bothering the parser.

The lexer and the parser have to agree what the token codes are. We solve this problem by letting yacc define the token codes.

Grammar

%token A B

```
%%

str: S

  ;

S: A S B

  |

  ;

%%
```

The tokens in our grammar are:A and B . Yacc defines each of these as a small integer using a preprocessor *#define*. Here are the definitions it used in this example:

```
    # define A 251
    # define B 252
```

Token code zero is always returned for the logical end of the input. Yacc doesn't define a symbol for it, but you can yourself if you want.

Yacc can optionally write a C header file containing all of the token definitions. You include this file, called *y.tab.h* on UNIX systems, in the lexer *and use the preprocessor symbols in your lexer action code.*

Grammar

A grammar is a series of *rules* that the parser uses to recognize syntactically valid input. For example, here is a version of the grammar which is used to build a calculator.

statement → *NAME* = *expression*

expression → *NUMBER* + *NUMBER* | *NUMBER* − *NUMBER*

The vertical bar, “|”, means there are two possibilities for the same symbol, i.e., an *expression* can be either an addition or a subtraction. The symbol to the left of the → is known as the *left- hand side* of the rule, often abbreviated LHS, and the symbols to the right are the *right-handside*, usually abbreviated RHS. Several rules may have the same left-hand side; the vertical bar isjust a short hand for this. Symbols that actually appear in the input and are returned by the lexer are *terminal* symbols or *tokens*, while those that appear on the left-hand side of some rule are *non-terminal* symbols or non-terminals. Terminal and non-terminal symbols must be different; it is an error to write a rule with a token on the left side.

A Yacc Parser

A yacc grammar has the same three-part structure as a lex specification. (Lex copied its structure from yacc.) The first section, the definition section, handles control information for the yacc-generated parser (from here on we will call it the parser), and generally sets up the execution environment in which the parser will operate. The second section contains the rules for the parser, and the third section is C code copied verbatim into the generated C program.

The 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, C code enclosed in %{ %} lines. We start our first parser by declaring two symbolic tokens.

```
%token NAME NUMBER
```

We can use single quoted characters as tokens without declaring them, so we don’t need to declare “=”, “+”, or “–”.

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
```

rules

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The 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
    ;
```

Unlike lex, yacc pays no attention to line boundaries in the rules section, and you will find that a lot of whitespace makes grammars easier to read. We've added one new rule to the parser: a statement can be a plain expression as well as an assignment. If the user enters a plain expression, we'll print out its result.

The symbol on the left-hand side of the first rule in the grammar is normally the start symbol, though we can use a %**start** declaration in the definition section to override that.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side.
In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A   :   B  C  D   ;
A   :   E  F  ;
A   :   G  ;
```

can be given to Yacc as

```
A    :   B C D
     |   E F
     |   G
;
```

- It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.
- If a nonterminal symbol matches the empty string **(ε)**, this can be indicated in the obvious way:
- **empty : ;**
- Names representing tokens must be declared; this is most simply done by writing
- **%token name1, name2 . . .**

in the declarations section. Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

- Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section.

- It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the % start keyword:
- %start symbol

- The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.
- It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as ``end- of-file'' or ``end-of-record''.

SYMBOLS AND ACTIONS:

A literal consists of a character enclosed in single quotes ``'''. As in C, the backslash ``\'' is an escape character within literals, and all the C escapes are recognized. Thus

```
%token NAME NUMBER
%%
statement: NAME '=' expression
         | expression { printf(“= %d\n”, $1);}
;
expression: expression '+' NUMBER { $$ = $1 + $3; }
          | expression ‘-' NUMBER { $$ = $1 - $3; }
          | NUMBER { $$ = $1; }
      ;
```

The rules that build an expression compute the appropriate values, and the rule that recognizes an expression as a statement prints out the result. In the expression building rules, the first and second numbers' values are $1 and $3, respectively. The operator's value would be $2, although in this grammar the operators do not have interesting values. The action on the last rule is not strictly necessary, since the default action that yacc performs after every reduction, before running any explicit action code, assigns the value $1 to $$.

Actions:

- With each grammar rule, the user may associate actions to be Yacc: Yet Another Compiler-Compiler performed each time the rule is recognized in the input process.

- These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.
- An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{" and ``}". For example,

```
A  :  '('  B  ')'
      {  hello( 1, "abc" );  }
```

and

```
XXX  :  YYY  ZZZ
{ printf("a message\n");flag = 25;  }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol “dollar sign" “$" is used as a signal toYacc in this context.

To return a value, the action normally sets the pseudo-variable ``$$" to somevalue. For example, an action that does nothing but return the value 1 is { $$ = 1; }

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus,if the rule is

A : B C D ;

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

expr : '(' expr ')' ;

The value returned by this rule is usually the value of the expr inparentheses. This can be indicated by

expr : '(' expr ')' { $$ = $2 ; }

By default, the value of a rule is the value of the first element in it ($1). Thus,grammar rules of the form

A : B ;

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes,it is desirable to get control before a rule is fully parsed. Yacc permits an actionto be written in the middle of a rule as well as at the end.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{" and ``%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{ int variable = 0; %}
```

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy"; the user should avoid such names.

The User subroutine Section

This section can contain any valid C Code. By default it contains main( ) function which call yyparse( ) function. The yyparse( ) function returns Zero when Stack contains only Start symbol at the end of input otherwise it will return a NON Zero value. If yyparse( ) receives a token which it doesn't know it calls yyerror( ) function.

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer- valued function called yylex.

The user must supply a lexical analyzer to read the input stream and communicate tokens (with

values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define'' mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the

token name DIGIT has been defined in the declarations section of the Yaccspecification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
      extern int yylval;int c;
      . . .
      c = getchar();
      . . .
      switch( c ) {
                              . . .
                    case  '0':
                    case '1':
       . . .
      case '9':
            yylval = c-'0'; return( DIGIT
            );
            . . .

            }
      . . .
```

- The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

- This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser;

How the Parser Works :

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex. however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable

of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.
The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
        IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look ahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ``.'') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
        .    reduce 18
```

refers to grammar rule 18, while the action

```
        IF   shift 34
```

refers to state 34. Suppose the rule being reduced is

```
        A   :    x y z;
```

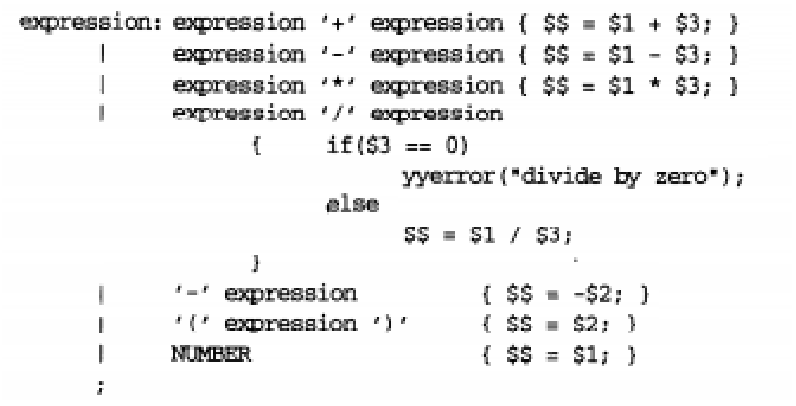
The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule).

In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left sideof the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In   addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the   lexical   analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

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.

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.

Types of Conflicts

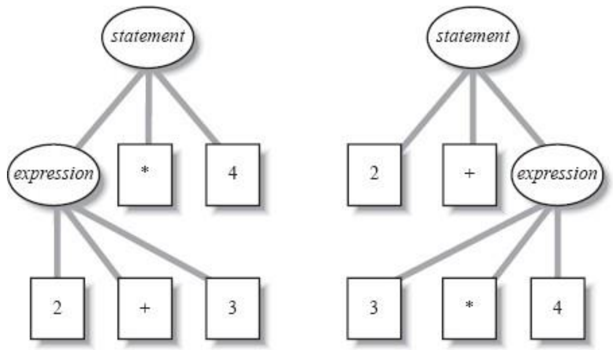
There are two kinds of conflicts that can occur when yacc tries to create a parser: “shift/reduce” and “reduce/reduce.”

Shift/Reduce Conflicts

A *shift/reduce* conflict occurs when there are two possible parses for an input string, and one of the parses completes a rule (the reduce option) and one doesn’t (the shift option). For example, this grammar has shift/reduce conflict:

```
    %%
    Statement: e
             ;
     e: e ‘+‘ e
      | e '-' e
      | e ‘*’ e
      | e ‘/’ e
     | ID
     | NUM
     ;
%%
```

For the input string “2+3*4” there are two possible parses,"(2+3)*4” or “2+(3*4)” So Yacc will tell you that there is an *shift/reduce* conflict.



We can avoid *shift/reduce* conflict by telling the yacc about the precedence and associativity of the operators. Precedence Controls which operator to execute first in an expression. Associativity controls the grouping of operators at the same precedence level.

Reduce/Reduce Conflicts

A *reduce/reduce* conflict occurs when the same token could complete two different rules. For example:

```
    %
    %
    Prog            :    proga | progb
                    ;
     Proga          :         'X'
                    ;
     Progb          :         'X'
                    ;
%%
```

An "X" could either be a **proga** or a **progb.** So Yacc will tell you that there is an reduce/reduce conflicts.

Compiling and Running a Simple Parser

For Compiling YACC Program:

- Write lex program in a file file.l and yacc in a file file.y
- Open Terminal and Navigate to the Directory where you have saved the files.
- type lex file.l
- type yacc file.y
- type cc lex.yy.c y.tab.h -ll
- type ./a.out

Example Programs

1. **Write a YACC program which accept strings $a^n b$**

LEX part

```
%{
#include "y.tab.h"
%}
%%
a return A;
b return B;
. return yytext[0];
\n return yytext[0];
%%
```

Yacc part

```
%{
/* Yacc program to recognize the grammar a^nb */
%}
%token A B
%%
str: s '\n'
s : x B
;
x : x A | A
;
%%
main()
{
printf(" Type the String ? \n");
if(!yyparse())
printf(" Valid String\n ");
}
int yyerror()
{
printf(" Invalid String.\n");
exit(0); }
```

2. **YACC program which accept strings that starts and ends with 0 or 1**

```
%{
/* Definition section */
extern int yylval;
%}

/* Rule Section */
%%

0 {yylval = 0; return ZERO;}

1 {yylval = 1; return ONE;}

.|\n {yylval = 2; return 0;}

%%
```

YACC part

```
%{
/* Definition section */
#include<stdio.h>
#include <stdlib.h>
void yyerror(const char *str)
{
printf("\nSequence Rejected\n");
}

%}
%token ZERO ONE

/* Rule Section */
%%
r : s {printf("\nSequence Accepted\n\n");}
;
```

```
s : n
| ZERO a
| ONE b
;
a : n a
| ZERO
;
b : n b
| ONE
;
n : ZERO
| ONE
;
%%

#include"lex.yy.c"
//driver code
int main()
{
 printf("\nEnter Sequence of Zeros and Ones : ");
 yyparse();
 printf("\n");
 return 0;
}
```

3. **Write a YACC program to check whether given string is Palindrome or not.**

```
%{
 /* Definition section */
 #include <stdio.h>
 #include <stdlib.h>
 #include "y.tab.h"
%}

/* %option noyywrap */
/* Rule Section */
%%

[a-zA-Z]+ {yylval.f = yytext; return STR;}
[-+()*/] {return yytext[0];}
[ \t\n] {;}

%%

int yywrap()
{
return -1;
}

YACC part
%{
 /* Definition section */
 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 extern int yylex();
 void yyerror(char *msg);
 int flag;
 int i;
 int k =0;
%}
%union {
 char* f;
}
%token <f> STR
%type <f> E

/* Rule Section */
%%
```

```
S : E {
        flag = 0;
        k = strlen($1) - 1;
        if(k%2==0){

        for (i = 0; i <= k/2; i++) {
        if ($1[i] == $1[k-i]) {
                } else {
                flag = 1;
                }
        }
        if (flag == 1) printf("Not palindrome\n");
        else printf("palindrome\n");
        printf("%s\n", $1);

        }else{
        for (i = 0; i < k/2; i++) {
        if ($1[i] == $1[k-i]) {
        } else {
                flag = 1;
                }
                }
        if (flag == 1) printf("Not palindrome\n");
        else printf("palindrome\n");
        printf("%s\n", $1);


        }
 }
;
E : STR {$$ = $1;}
;
%%
void yyerror(char *msg)
{
 fprintf(stderr, "%s\n", msg);
 exit(1);
}

//driver code
int main()
{
 yyparse();
 return 0;
}
```

4. **Write YACC program for Binary to Decimal Conversion.**

```
%{
/* Definition section */
#include<stdio.h>
#include<stdlib.h>
#include"y.tab.h"
extern int yylval;
%}

/* Rule Section */
%%
0 {yylval=0;return ZERO;}
1 {yylval=1;return ONE;}

[ \t] {;}
\n return 0;
. return yytext[0];
%%

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

```
YACC
%{
/* Definition section */
#include<stdio.h>
#include<stdlib.h>
void yyerror(char *s);
%}
%token ZERO ONE

/* Rule Section */
%%
N: L {printf("\n%d", $$);}
L: L B {$$=$1*2+$2;}
| B {$$=$1;}
B:ZERO {$$=$1;}
|ONE {$$=$1;};
%%
//driver code
int main()
{
while(yyparse());
}
yyerror(char *s)
{
fprintf(stdout, "\n%s", s);
}
```

5. **Write a YACC program to evaluate a given arithmetic expression consisting of ‘+’, ‘-‘, ‘*’, ‘/’ including brackets.**

```
%{
   /* Definition section*/
   #include "y.tab.h"
   extern yylval;
}%
```

```
%%
[0-9]+    {yylval = atoi(yytext);
              return NUMBER;}
 [a-zA-Z]+    { return ID; }
[ \t]+         ;  /*For skipping whitespaces*/

\n            { return 0; }
.            { return yytext[0]; }
 %%

%{
    /* Definition section */
  #include <stdio.h>
%}

%token NUMBER ID
// setting the precedence
// and associativity of operators
%left '+' '-'
%left '*' '/'

/* Rule Section */
%%
E : T       {
                printf("Result = %d\n", $$);
                return 0;
            }

T :
    T '+' T { $$ = $1 + $3; }
    | T '-' T { $$ = $1 - $3; }
    | T '*' T { $$ = $1 * $3; }
    | T '/' T { $$ = $1 / $3; }
    | '-' NUMBER { $$ = -$2; }
    | '-' ID { $$ = -$2; }
    | '(' T ')' { $$ = $2; }
    | NUMBER { $$ = $1; }
    | ID { $$ = $1; };
% %

int main() {
    printf("Enter the expression\n");
    yyparse();
}

/* For printing error messages */
int yyerror(char* s) {
    printf("\nExpression is invalid\n");
}
```

Review Questions of Module-4

1. Give the specification of yacc program? Give an example? (8)
2. What is grammar? How does yacc parse a tree? (5)
3. How do you compile a yacc file? (5)
4. Explain the ambiguity occurring in an grammar with an example? (6)
5. Explain shift/reduce and reduce/reduce parsing ? (8)
6. Write a yacc program to test the validity of an arthimetic expressions? (8)
7. Write a yacc program to accept strings of the form anbn , n>0? (8)

Give the Structure of lex program

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching.

The general format of Lex source is:

```
{definitions}
%%
```

{rules}

%%

{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

%%

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executedwhen the expressions are recognized.

Thus an individual rule might appear integer printf ("("found keyword INT"); to look for the string integer in the input stream and print the message “found keyword INT" whenever it appears.

The host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed inbraces.

Explain Regular expression in detail.

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >”

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

xyz"++"

Classes of characters can be specified using the operator pair []. The construction[abc] matches a single character, which may be a, b, or c. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^. The - character indicates ranges. For example,

[a-z0-9<>_]

In character classes, the ^ operator must appear as the first character after the left bracket;it indicates that the resulting string is to be complemented with respect to the computer character set. Thus [^abc]

Optional expressions: The operator ? Indicates an optional element of an expression. Thus

ab?c

matches either ac or abc.

Repetitions of classes are indicated by the operators * and +.

a*: is any number of consecutive a characters, including no character; while a+:

Character	**Meaning**
A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	A character class. Matches *any* character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C.
*****	Match *zero* or more occurrences of the preceding pattern.
+	Matches *one* or more occurrences of the preceding pattern.
?	Matches *zero or one* occurrences of the preceding pattern.
$	Matches end of line as the last character of the pattern.
{ }	Indicates how many times a pattern can be present. Example: A{1,3} implies one or three occurrences of A may be present.
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.

^	Negation.
|	Logical OR between expressions.
"<some symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.

Write a Lex program to count the number of words.

```
 %{
    int wordCount = 0;
    %}
    chars [A-za-z\_\'\.\"]
    numbers ([0-9])+
    delim [" "\n\t]
    whitespace {delim}+
    words {chars}+
    %%

{words} { wordCount++; /*
    increase the word count by one*/ }
    {whitespace} { /* do
    nothing*/ }
    {numbers} { /* one may
    want to add some processing here*/ }
    %%

    void main()
    {
        yylex(); /* start the analysis*/
        printf(" No of words:
```

```
    %d\n", wordCount);
}
int yywrap()
{
    return 1;
}
```

Give Lex structure. Write a Lex Program to count number of vowels and consonants

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching.

The general format of Lex source is:

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

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

%%

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized.

Thus an individual rule might appear integer printf ("("found keyword INT"); tolook for the string integer in the input stream and print the message “found keyword INT" whenever it appears.

The host procedural language is C and the C library function printf is used to printthe string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line;if it is compound, or takes more than a line, it should be enclosed in braces.

```
%{
/* to find vowels and consonants*/int
   vowels = 0;
   int consonents = 0;
%}
%%
[ \t\n]+
[aeiouAEIOU] vowels++;
[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ] consonants++;
.
%%
main()
{
   yylex();
   printf(" The number of vowels = %d\n", vowels);
   printf(" number of consonents = %d \n", consonents);
return(0);
}
```

structure of Lex and Yacc

Lex is a program generator designed for lexical processing of character input streams.It accepts a high-level, problem oriented specification for character string matching.

The general format of Lex source is:

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

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus %% **(**no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized.

Thus an individual rule might appear integer printf ("("found keyword INT"); tolook for the string integer in the input stream and print the message “found keyword INT" whenever it appears.

The host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the rightside of the line; if it is compound, or takes more than a line, it should be enclosed in braces.

What are conflicts in Yacc?

A set of grammar rules is ambiguous if there is some input string that can be structured intwo or more different ways. For example, the grammar rule

expr : expr '-' expr

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

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. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr - expr - expr

When the parser has read the second expr, the input that it has seen:

expr - expr

matches the right side of the grammar rule above. The parser could reduce the input byapplying this rule; after applying the rule; the input is reduced to expr (the left side of the rule). The parser would then read the final part of the input:

- expr

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

Alternatively, when the parser has seen

expr - expr

Explain shift reducing parsing.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead tokento decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

IF shift 34

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look ahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actionsare appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide

whether to reduce, but usually it is not; in fact, the default action (represented by a ``.")is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules arealso given small integer numbers, leading to some confusion. The action

reduce 18

refers to grammar rule 18, while the action

IF   shift 34

refers to state 34. Suppose the rule being reduced is

A   :   x y z   ;

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off thetop three states from the stack (In general, the number of states popped equals the numberof symbols on the right side of the rule).

In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and theactions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

what are ambiguous grammar? Explain.

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

expr   :   expr '-' expr

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

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. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr - expr - expr

When the parser has read the second expr, the input that it has seen:

expr - expr

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 would then read the final part of the input:

- expr

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

Alternatively, when the parser has seen

expr - expr

What are Yacc tools. What are the two types of conflicts that arises during parsing?

Yacc invokes two **disambiguating** rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

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

Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an ``if-then-else'' construction:

```
stat   :    IF  '('  cond  ')'  stat
       |    IF  '(' cond  ')'  stat  ELSE  stat

       ;
```

In these rules, IF and ELSE are tokens, cond is a nonterminal symbol describing conditional (logical) expressions, and stat is a nonterminal symbol describing statements. The first rule will be called the simple-if rule, and the second the if-else rule.
These two rules form an ambiguous construction, since input of the form

EXAMPLE:

```
IF  ( C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways:

```
IF  ( C1  ) {

      IF  ( C2  )  S1

      }

ELSE  S2
```

or

```
IF  ( C1  ) {

      IF  (  C2  )  S1ELSE
```

S2

}

- The second interpretation is the one given in most programming languages having this construct. Each ELSE is associated with the last preceding ``un- ELSE'd" IF. In this example, consider the situation where the parser has seen

IF (C1) IF (C2) S1

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get

IF (C1) stat

and then read the remaining input,

ELSE S2

and reduce

IF (C1) stat ELSE S2

by the if-else rule. This leads to the first of the above groupings of the input.

- On the other hand, the ELSE may be shifted, S2 read, and then the right hand portion of

IF (C1) IF (C2) S1 ELSE S2

can be reduced by the if-else rule to get

IF (C1) stat

which can be reduced by the simple-if rule.

- Once again the parser can do two valid things - there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.
- This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

IF (C1) IF (C2) S1

- In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputsare characterized by the state of the parser.

stat : IF '(' cond ')' stat

- Once again, notice that the numbers following ``shift" commands refer to other states, while the numbers following ``reduce" commands refer to grammar rule numbers. In the y.output file, the rule numbers are printed after those ruleswhich can be reduced.

What are the usage of YYparse()?

The yacc program gets the tokens from the lex program. Hence a lex program has be written to pass the tokens to the yacc. That means we have to follow different procedure to get the executable file.

i. The lex program <lexfile.l> is fist compiled using lex compiler to get **lex.yy.c.**

ii. The yacc program <yaccfile.y> is compiled using yacc compiler to get **y.tab.c.**

iii. Using c compiler b+oth the lex and yacc intermediate files are compiled withthe lex library function. **cc y.tab.c lex.yy.c –ll.**

iv. If necessary out file name can be included during compiling with –o option.