Module4

Lex and Yacc:

Lexis a computer program or tool that generates lexical analysers/scanners/lexers.

Lex helps you by taking a set of possible tokens and producing a C routine (lexical analyser) that can identify those tokens.

Lexical Analysis is the process of converting a source programs into a sequence of tokens.

A program or function that performs lexical analysis is called a **lexical analyser**, lexer or scanner.

The set of descriptions given to lex is called **lex specification**.

The token descriptions that lex uses are known as **regular expressions.**

The lex lexer is almost always faster than a lexer that you might write in C by hand.

As the input is divided into tokens, a program often needs to establish the relationship among the tokens. This task is known as parsing and the list of rules that define the relationships that the program understands is a grammar.

The simplest Lex Program:

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

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

Structure of a Lex file: (Recognizing words with lex)

```
%{
/* simple word recognition
A verb/not a verb */
%}


%%
[\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 ()
{
        yylex ();
}
```

The first section, the ***definition section***, introduces any initial C program code we want copied into the final program. This C code is surrounded with the special delimiters "%{ "and "%}". Lex copies the material between "%{ "and "%}" directly to the generated C file.

In this example, the only thing in the definition section is some C comments.

This %% marks the end of this section.

The next section is the ***rules section***. Each rule is made up of two parts: a pattern and an action, separated by whitespace. The lexer that lex generates will execute the action when it recognizes the pattern. These patterns are UNIX-style regular expressions.

The first rule in the above example:

```
[\t ]+                    ;    /* ignore white space */
```

The square brackets, "[]", indicate that any one of the characters within the brackets matches the pattern. Either accept "\t" or " ". The "+" means that the pattern matches one or more consecutive copies of the sub pattern that precedes the plus. Thus the pattern describes white space.

The second part of the rule, the action, is simply a semicolon, a do-nothing C statement. Its effect is to ignore the input.

The next set of rules uses the "|" action. This is a special action that means to use the same action as the next pattern, so all of the verbs use the action specified for the last one.

The end of the rules section is delimited by another %%.

The final section is the user subroutine section, which can consist of any legal C code. Lex copies it to the C file after the end of the lex generated code.

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

The lexer produced by lex is a C routine called yylex (). Unless the actions contain explicit return statements, yylex () won't return until it has processed the entire input.

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:

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:

When a lex scanner and a yacc parser is used together, the parser is the higher level routine.

The parser 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 the tokens are of interest to the parser. For example in most programming languages the parser doesn't want comments and whitespace. For these ignores tokens, the lexer doesn't return do that it can continue on to the next token.

The lexer and the parser have to agree what the token codes are. Yacc will define the token codes.

The 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

Token always returned for the logical end of the input.

Yacc can optionally write a C header file containing all of the token definitions. This file is called as y.tab.h and can be included on UNIX/MS-DOS systems.

The parts of speech lexer:

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

```
        }
};
%%
struct word {
        char *word_name;
        int word_type;
        struct word *next;
};
struct word *word_list;
int add_word (int type, char *word)
{
        struct word *wp;
        if (lookup_word (word)! = LOOKUP) {
                printf (“word already exists\n”);
                return 0;
        }
        wp= (struct word *) malloc (sizeof(struct word));
        wp->next=word_list;
        wp->word_name= (char *)malloc(strlen(word)+1);
        wp->word_type=type;
        word_list=wp;
        return 1;
}
int lookup_word(char *word)
{
        struct word *wp=word_list;
        for (; wp; wp=wp->next) {
                if (strcmp(wp->word_name,word)==0)
                        return wp->word_type;
        }
        return LOOKUP;
}
```

A Yacc Parser:

```
%{
/*
*A lexer for the basic grammar to use for recognizing English sentences.
*/
#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);
}
```

Example:

verb is am are was were be being been do

is

is: verb

noun chew eat lick

verb chew eat lick

verb run stand sleep

dog run

dog: noun

run: verb

chew eat sleep cow horse

chew: verb

eat: verb

sleep: verb

cow: noun

horse: noun

verb talk

talk: verb

Structure of a Yacc Parser:

The structure of a yacc parser is similar to that of a lex lexer.

```
%{
/*
*A lexer for the basic grammar to use for recognizing English sentences.
*/
#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 first section, **<u>the definition section</u>**, has a literal code block, enclosed in “%{ “and “%}”.

Here it is used for C comment lines and a single include file.

Then come definitions of all tokens that is expected to receive from the lexical analyser.

The first %% indicates the beginning of the rules section.

The **<u>rules section</u>**describes the actual grammar as a set of production rules or simply rules. Each rule consists of a single name on the left hand side of the “:” operator, a list of symbols and action code on the right hand side, and a semicolon indicating the end of the file. By default, the first rule is the highest level rule.

The action part of a rule consists of a C block, beginning with “{“and ending with “}”. The parser executes an action at the end of a rule as soon as the rule matches.

In ***sentence*** rule, the action reports that we've successfully parsed a sentence. Since ***sentence*** is the top-level symbol, the entire input must match a ***sentence***. The parser returns to its caller, in this case the main program, when the lexer reports the end of the input. Subsequent calls to yyparse () reset the state and begin processing again.

If the rule is not matched the parser calls yyerror (), which is provided in the user subroutines section, and then recognizes the special rule error. You can provide error recovery code that tries to get the parser back into a state where it can continue parsing. If the recovery fails or, as is the case here, there is no error recovery code, yyparse () returns to the caller after it finds an error.

The third and final section, the user subroutines section, begins after the second %%. The **user subroutine section**can contain any C code and is copied into the resulting parser. In our example, minimal set of functions necessary for a yacc generated parser is provided using a lex-generated lexer to compile: main () and yyerror (). The main routine keeps calling the parser until it reaches the end-of-file on yyin, the lex input file. The only other necessary routine is yylex () which is provided by the lexer.

Regular Expressions:

A regular expression is a pattern description using a "meta" language, a language that is used to describe particular patterns of interest.

The characters used in this Meta language are part of the standard ASCII character set used in UNIX and MS-DOS.

The characters that form regular expressions are:

.	Matches any single character except the new line character (\n).
*	Matches zero or more copies of the preceding expression. Example a* means zero or more copies of a.
[ ]	A character class which matches any character within the brackets. If the first characteris ^ it changes the meaning to match any character except the ones within the brackets.A dash inside the square brackets indicates a character range. Example [0-9] means thesame thing [0123456789]. Other meta characters have no special meaning within square brackets except that C escape sequences starting with "\" are recognized.
^	Matches the beginning of a line as the first character of a regular expression. Also used for negation within square brackets. Example 1: ^(int) beginning of a line should be int. Example 2: [^ \t] means any non-empty combination of characters except space, tab and newline.
$	Matches the end of a line as the last character of a regular expression.
{ }	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.

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

+	Matches one or more occurrence of the preceding regular expression. For example: [0-9]+ matches "1", "111", or "123456" but not an empty string.

?	Matches zero or one occurrence of the preceding regular expression. For example: -?[0-9]+ matches a signed number including an optional leading minus.

|	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". 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 |.

Examples of Regular Expressions:

Digit:	[0-9]

Integer:	[0-9] +

Optional unary minus:

-? [0-9] +

Decimal numbers:

[0-9] * \. [0-9] +	matches 0.0, 4.5, or .31458

To match 0 or 2 along with decimal number:

([0-9] +) | ([0-9] * \. [0-9] +)

Now add unary minus also:

-? (([0-9] +) | ([0-9] * \. [0-9] +))

Float-style exponent:

[eE] [-+]? [0-9]+	matches E-3 or e12

Real number specification:

-? **(**([0-9] +) | ([0-9] * \. [0-9] +) ([eE] [-+]? [0-9] +)? **)**

This expression makes the exponent part optional.

A Word Counting Program:

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}
word                    [^ \t\n]+
eol                     \n
%%
{word}                  {wordCount++; charCount+=yyleng ;}
{eol}                   {charCount++; lineCount ;}
.                       charCount;
%%
main ()
{
        yylex ();
        printf (“%d %d %d\n”, lineCount, wordCount, charCount);
}
```

Using file:

```
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 ("%u %u %u\n", charCount, wordCount, lineCount);
        return 0;
}
```

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 there is no more input. If the value is 0 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.

→Write a lex program to find the number of vowels and consonants.

```
%{
/* to count vowels and consonents*/
int vowels = 0;
int consonents = 0;
%}

%%

[ \t\n]+                                        ;
[aeiouAEIOU]                                    vowels++;
[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]    consonents++;
.                                               ECHO;

%%
main()
{
        yylex ();
        printf ("The number of vowels=%d\n",vowels);
        printf ("The number of consonents=%d\n",consonents);
}
```

Using Yacc

Lex divides the input stream into tokens and then yacc takes these tokens and groups them together logically. Yacc takes a grammar that is specified and writes a parser that recognizes valid sentences in that grammar. A grammar is a series of rules that the parser uses to recognize syntactically valid input.

Example for a grammar:

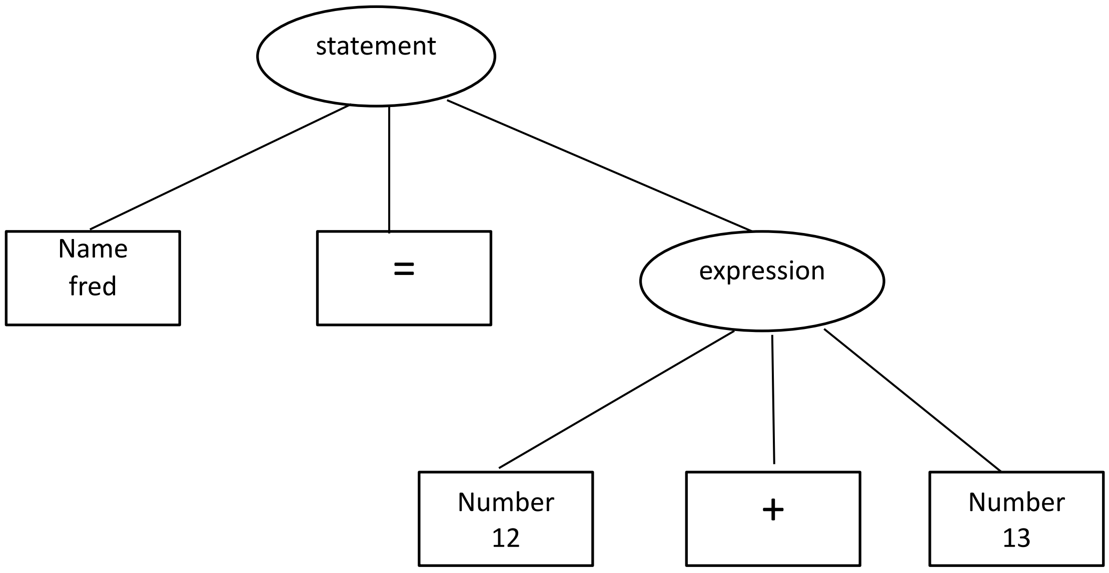
statement -> NAME = exp

exp -> NUMBER + NUMBER | NUMBER – NUMBER

The vertical bar "| "means there are two possibilities for the same symbol, ie 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 (LHS), and the symbols to the right are the *right-hand side* (RHS).

Symbols that actually appear in the input and are returned by the lexer are terminals symbols or tokens, while those that appear on the left-hand side of some rule are non-terminals 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.

The usual way to represent a parsed sentence is as a tree. For example, if the parsed input "fred = 12 + 13" with this grammar, the tree would look like fig. "12 + 13" is an expression, and fred=expression" is a statement.



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:

Rules can refer directly or indirectly to themselves.

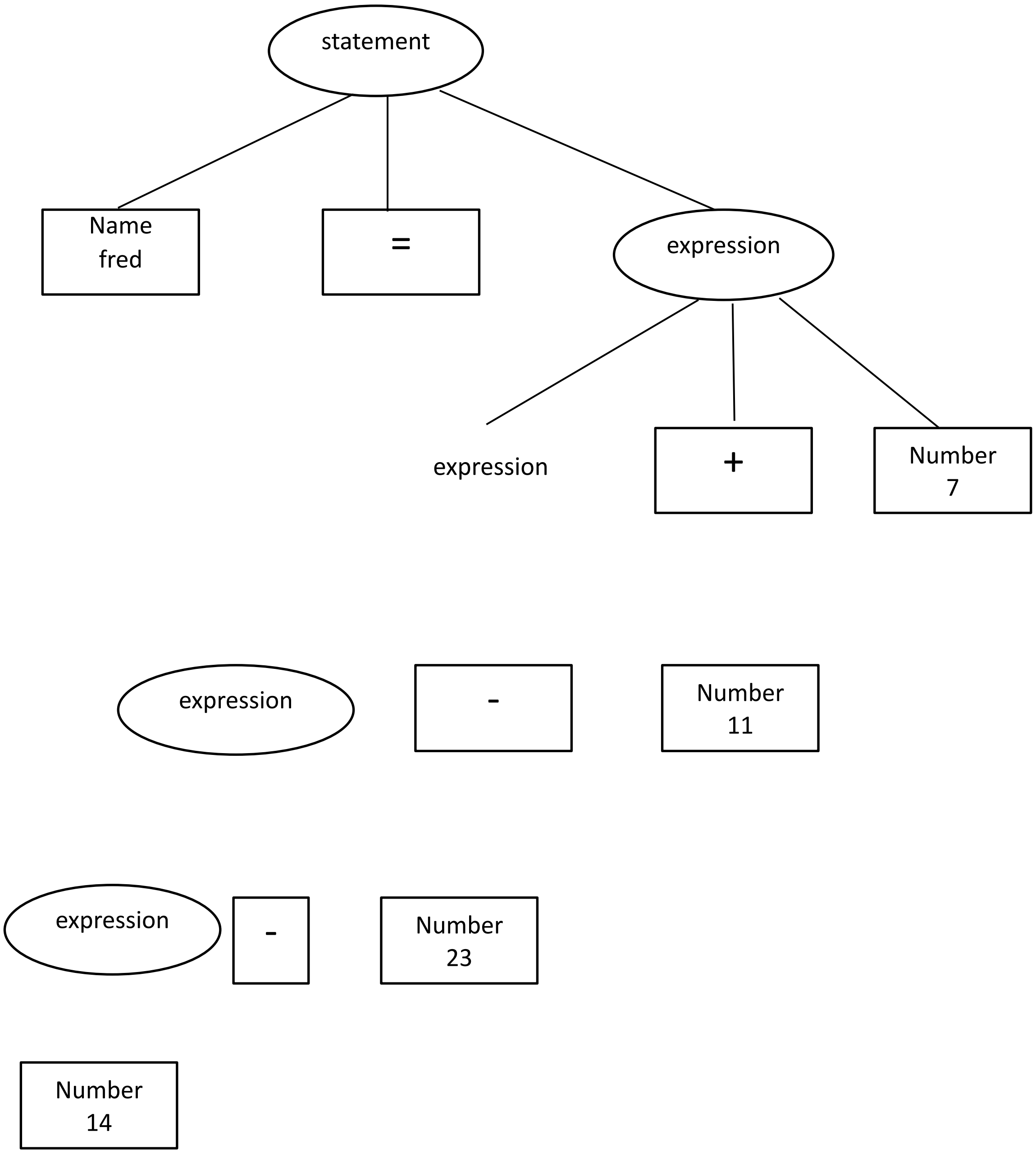
Makes it possible to parse long input sequence.

exp -> NUMBER

| exp + NUMBER

| exp - NUMBER

Now parse a sequence like fred = “14 + 23 – 11 + 7” by applying the expression rules repeatedly, as fig.



Yacc can parse recursive rules very efficiently, so recursive rules are used in almost all grammars.

Shift/Reduce Parsing:

As the parser reads tokens, each time it reads a token that doesn't complete a rule it pushes the token on an internal stack and switches to a new state reflecting the token just read. This action is called a ***shift***.

When it has found all the symbols that constitute the right hand side of a rule, it pops the right hand side symbols off the stack, pushes the left hand side symbol on the stack. This action is called a ***reduction***, since it usually reduces the number of items on the stack.

Whenever yacc reduces a rule, it executes user code associated with the rule.

statement -> NAME = expression

expression -> NUMBER + NUMBER | NUMBER – NUMBER

Input:

"fred = 12 + 13" the parser starts by shifting tokens on to the internal stack one at a time:

fred

fred =

fred = 12

fred = 12 +

fred = 12 + 13

At this point it can reduce the rule "expression -> NUMBER + NUMBER" so it pops the 12, the + and 13 from the stack and replaces them with expression:

fred = expression

now it reduces the rule "statement->NAME= expression", so it pops fred, = and expression and replaces them with statement. We have reached the end of the input and the stack has been reduced to the start symbol, so the input was valid according to the grammar.

A Yacc Parser:

A yacc parser 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 therules for the parser, and the third section is C code copied into thegenerated 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 startour first parser by declaring two symbolic tokens.

```
%token NAME NUMBER
```

The Rules Section:

The rules section simply consists of a list of grammar rules.we use a colon between the left- and right-hand sides of a rule, and we puta semicolon at the end of each rule:

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

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

The final section is the **user subroutine section**, which can consist of any legal C code.

Arithmetic Expressions and Ambiguity

Arithmetic expressions may have multiplication and division, unary negation,and parenthesized expressions. For example, consider the grammar,

```
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(), thestandard 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 mightmean 3-(4-(5-6)) or (3-4)-(5-6) or any of a lot of other possibilities. Figure3-3 shows the two possible parses for 2+3*4.

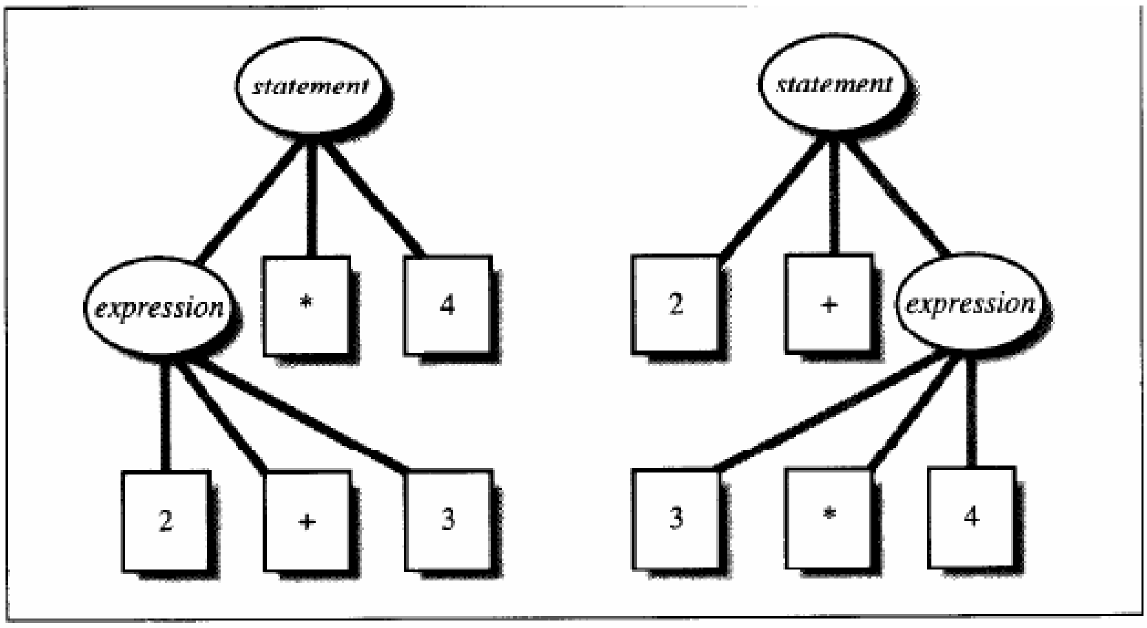


*Figure 3-3: Ambiguous input 2+3*4*

If you compile this grammar as it stands, yacc will tell you that there are 16shift/reduce conflicts, states where it cannot tell whether it should shift thetoken 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+	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.

The problem is that we haven't told yacc about the precedence and associativity of the operators. Precedence controls which operators to execute firstin an expression.

Mathematical and programming tradition (dating backpast the first Fortran compiler in 1956) says that multiplication and divisiontake precedence over addition and subtraction, so a+b*c means a+(b*c) andd/e-f means (d/e)-f. In any expression grammar, operators are groupedinto levels of precedence from lowest to highest. The total number of levels depends on the language. The C language is notorious for having toomany precedence levels, a total of fifteen levels.

Associativity controls the grouping of operators at the same precedencelevel. Operators may group to the left, e.g., a-b-c in C means (a-b)-c, orto the right, e.g., a=b=c in C means a=(b=c). In some cases operators donot group at all.

There are two ways to specify precedence and associativity in a grammar,implicitly and explicitly. To specify them implicitly, rewrite the grammarusing separate non-terminal symbols for each precedence level. Assumingthe usual precedence and left associativity for everything.
But yacc also lets you specify precedences explicitly.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.

```
%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; }
;
%%
```

The rule for negation includes "%prec UMINUS". The only operator this ruleincludes is "-", which has low precedence, but we want unary minus tohave higher precedence than multiplication rather than lower. The %prectells yacc to use the precedence of UMINUS for this rule.