

NATIONAL ENGINEERING COLLEGE, K.R.NAGAR, KOVILPATTI – 628 503
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
19CS62C – PRINCIPLES OF COMPILER DESIGN [EVEN SEM 2024 – 2025]
LABORATORY EXERCISE – INSTRUCTION SHEET

Exercise 6 (a) : - Generate three address code for a simple program using LEX and YACC

Tool Introduction:

YACC stands for **Yet Another Compiler Compiler**. YACC provides a tool to produce a parser for a given grammar. YACC is a program designed to compile a LALR (1) grammar. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.

The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream.

These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

The heart of the input specification is a collection of **grammar rules**. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

date : month_name day ',' year ;

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma `,'` is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

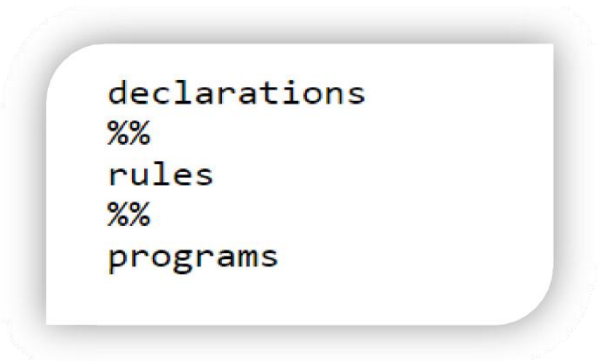
July 4, 1776

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal. To avoid confusion, terminal symbols will usually be referred to as tokens.

Every YACC specification file consists of three sections:

- the declarations,
- (grammar) rules,
- and programs.



```
declarations
%%
rules
%%
programs
```

The rules section is made up of one or more grammar rules. A grammar rule has the form:

`A : BODY ;`

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

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
   | 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, 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 . . .
```

Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called **y.tab.c** on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called **yyparse**; it is an integer valued function. When it is called, it in turn repeatedly calls **yylex**, the lexical analyzer supplied by the user to obtain input tokens. Eventually, an error is detected, in which case (if no error recovery is possible)

yyparse returns the value 1, or the lexical analyzer returns the end marker token and the parser accepts. In this case, yyparse returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called main must be defined, that eventually calls **yyparse**. In addition, a routine called **yyerror** prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of main and yyerror.

Problem Description:

Your task is to implement a simple three address code generator to define an expression grammar rules using YACC specification, subroutine for action and use lex code specification to generate 3 address code from the given input source and provide it to parser for evaluation.

Compilation of YACC Program:

1. Write lex specification in tcode.l file and yacc specification in tcode.y
2. Open Command prompt and Navigate to the Directory where you have saved the files.
3. flex tcode.l
4. yacc tcode.y
5. cc lex.yy.c y.tab.h -ll
6. ./a.out

In Editplus Window Flex Environment:

1. Write lex specification in tcode.l file and yacc specification in tcode.y
2. Choose Tools Menu, Select Compile Lex File
3. Choose Tools Menu, Select Compile YACC File
4. Choose Tools Menu, Select Build Lex+YACC option
5. Open Command Window, navigate to the directory
6. Run a.exe (or) filename.exe

Sample Input and Output:

Enter Arithmetic Expression: a=b+c*d

THREE ADDRESS CODE

t1 = c * d

t2 = b + t1

a = t2

Source Code:

LEX Specification:

tcode.l

```
%{
#include "y.tab.h"
%}

%%

[0-9]+?    {yylval.sym=(char)yytext[0]; return NUMBER;}
[a-zA-Z]+? {yylval.sym=(char)yytext[0];return LETTER;}

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

%%

int yywrap()
{
    return 0;
}
```

YACC Specification:

tcode.y

```
%{
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void ThreeAddressCode();
char AddToTable(char ,char, char);
int ind=0;//count number of lines
char temp = '1'; //for t1,t2,t3.....
struct incod
{
char opd1;
char opd2;
char opr;
```

```
};  
%}
```

```
%union  
{  
  char sym;  
}
```

```
%token <sym> LETTER NUMBER
```

```
%type <sym> expr
```

```
%left '+'
```

```
%left '*' '/'
```

```
%left '-'
```

```
%%
```

```
statement: LETTER '=' expr ';' {AddToTable((char)$1,(char)$3,'=');}  
| expr ';' ;
```

```
expr:
```

```
  expr '+' expr    {$$ = AddToTable((char)$1,(char)$3,'+');}  
| expr '-' expr    {$$ = AddToTable((char)$1,(char)$3,'-');}  
| expr '*' expr    {$$ = AddToTable((char)$1,(char)$3,'*');}  
| expr '/' expr    {$$ = AddToTable((char)$1,(char)$3,'/');}  
| '(' expr ')'     {$$ = (char)$2;}  
| NUMBER           {$$ = (char)$1;}  
| LETTER           {$$ = (char)$1;}  
| '-' expr         {$$ = AddToTable((char)$2,(char)'\t','-' );}  
; 
```

```
%%
```

```
yyerror(char *s)  
{  
  printf("%s",s);  
  exit(0);  
}
```

```
struct incod code[20];
```

```
char AddToTable(char opd1,char opd2,char opr)
{
    code[ind].opd1=opd1;
    code[ind].opd2=opd2;
    code[ind].opr=opr;
    ind++;
    return temp++;
}

void ThreeAddressCode()
{
    int cnt = 0;
    char temp = '1';
    printf("\n\n\t THREE ADDRESS CODE\n\n");
    while(cnt<ind)
    {
        if(code[cnt].opr != '=')
            printf("t%c : = \t",temp++);
        if(isalpha(code[cnt].opd1))
            printf(" %c\t",code[cnt].opd1);
        else if(code[cnt].opd1 >='1' && code[cnt].opd1 <='9')
            printf("t%c\t",code[cnt].opd1);

        printf(" %c\t",code[cnt].opr);
        if(isalpha(code[cnt].opd2))
            printf(" %c\n",code[cnt].opd2);
        else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9')
            printf("t%c\n",code[cnt].opd2);
        cnt++;
    }
}

main()
{
    printf("\n Enter the Expression : ");
    yyparse();
    ThreeAddressCode();
}
```