LAB SESSION 8: YACC- VALIDATION OF LOOP STRUCTURES

AIM: To implement a YACC program to validate basic and nested loops.

PROBLEM DEFINITION: Write a YACC program to validate syntax of a basic for loop as well as a nested for loop with assignment statements for a given input.

THEORY: YACC PARSER: Flex generates code for a lexical analyzer, or scanner. It uses patterns that match strings in the input and converts the strings to tokens. Tokens are numerical representations of strings, to simplify processing. This is illustrated in Figure 6.1. As Lex finds identifiers in the input stream, it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of the variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

Yacc generates C code for a syntax analyzer, or parser. It uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree. A syntax tree imposes a hierarchical structure of tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code.

HOW THE PARSER WORKS

Yacc turns the specification file into a C program, which parses the input according to The grammar rule) specification given. 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 look-ahead 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 O, the stack contains only state O, and no look-ahead 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:

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

Using the current state, and the look-ahead 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 of the stack, and in the look-ahead 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 look-ahead token.

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. Sometimes, it may be necessary to consult the look ahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a "") is often a reduce action.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification.

The input tokens it has seen, together with the look-ahead token, cannot be followed by anything that would result in a valid input. The parser reports an error, and attempts to recover the situation to resume parsing: the error recovery (as opposed to the detection of error).

YACC LIBRARY

Yacc tools come with a number of helpful subroutines in the library files. We can include those library files by giving the -ly flag at the end of the compilation command line cc, on Linux or Unix operating systems. Commonly available library functions and keywords are listed below.

- **main() Function**

All versions of yacc implementations come with main() function, which could be reproduced as

```
main (ac, av)
{
        yyparse () ;
        return 0;
}
```

- **yyerror() Function**

```
yyerror (clr * errmsg)
{
        fprint (stderr, "&s \n", err msg) ;
}
```

This default version of yyerror() function in the yacc library merely prints the error message on the standard output. This function could be modified in such a way that it gives at least the line number where the error is found, and the most recent tolen matched by the lexer.

```
yyerror (clear * msg)
{
printf ("%d", "&s" at "%s", yylineno, msg,yytext) ;
}
```

Here yylineno is the current line number where the error is found.

The yyerror() function is very important because whenever a yacc parser detects a syntax error, it calls yyerror() to report the error to the user by passing an error message, which will ultimately help the programmer to debug it.

- **yyerror**

There are instances where you can write the yacc action code in such a way that it detects the context sensitive syntax errors that the parser itself cannot. In such an instance, we can call the macro YYERROR, which will invoke the parser to call yyerror() function and go into the error recovery mode, looking for a state where it can shift an error token.

- **yyerrork**

After yacc detects and reports a syntax error, it normally refrains from reporting another error until it has shifted three consecutive tokens without another error. To a certain extent, this will eliminate the problem of multiple error message, resulting from a single mistake as the parser gets resynchronized.

The errors are reported normally only after the parser is back in synchronized mode (ie. in the normal state). The macro yyerrork tells the parser to return to tie normal state in such cases.

- **yyclearin**

The macro yyclearin in an action discards a look-ahead token if one had been read. This macro is very useful in the case of error recovery in the interactive parser to put back into the known state after an error.

- **yyrecovering()**

After yacc detects and reports a syntax error, it normally enters into an error recovery mode in which it refrains from reporting another error until it had shifted three tokens without error. To a certain extent this will eliminate the problem of multiple error messages resulting from a single error as the parser gets re-synchronized. The macro YYRECOVERING() will return a non zero if the parser is currently in the error recovery mode and zero if it is not.

- **yyparse()**

The execution of the yacc generated parser yyparse() begins when it is called from the main() function. The parser will attempt to parse the input stream as soon as it is called.

It returns zero if the parser succeeds parsing, and non-zero if not.

Unlike yylex() (i.e. generated lexical analyzer), every time we call yyparse(), to start parsing, forgetting where it was (i.e. state) the last time it returned.

- **yyabort**

This special statement YYABORT in a yacc action code makes the parser function yyparse() to return immediately with a non-zero value, indicating it is not able to parse the input stream successfully. This is useful when an action routine detects an unexpected error, where there is no port in continuing.

- **yyaccept**

This special statement YYACCEPT in the yacc actions makes the parser function yyparse() to return immediately with a value zero, indicating that it is able to parse input stream successfully.

This statement is useful in the situation where the lexical analyzer cannot identify when the input data ends, but the parser can.

PROGRAM:

Lex file:

```
%{
#include "for.tab.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* yylval is a union; set the string field */
extern int yylineno;
%}


%%
"for"           { return FOR; }
"while"         { return WHILE; }
"do"            { return DO; }
"###"           { return SEP; }
"//"[^\n]*      { /* skip single-line comment */ }
"/*"            {
        /* skip C-style comment manually so we can count newlines */
        int c;
        while ((c = input()) != 0) {
                if (c == '\n') ++yylineno;
                if (c == '*') {
                        int d = input();
                        if (d == '/') break;
                        if (d == 0) break;
                        unput(d);
                }
        }
}
"int"           { yylval.s = strdup(yytext); return TYPE; }
"++"            { return INC; }
```

```
"--"            { return DEC; }
"<="            { return LE; }
">="            { return GE; }
"=="            { return EQ; }
"!="            { return NE; }
";"             { return ';'; }
"("             { return '('; }
")"             { return ')'; }
"{"             { return '{'; }
"}"             { return '}'; }
"="             { return '='; }
"+"             { return '+'; }
"-"             { return '-'; }
[ \t\r]+        { /* skip spaces/tabs */ }
"\n"            { yylineno++; }
[0-9]+          { yylval.s = strdup(yytext); return NUM; }
[A-Za-z_][A-Za-z0-9_]* { yylval.s = strdup(yytext); return ID; }
.               { return yytext[0]; }

%%

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

YACC file:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern int yylex(void);
extern int yylineno;
void yyerror(const char *s);
int had_error = 0; /* flag for current program */
%}

%union {
  char *s;
}

%token <s> ID NUM
%token INC DEC
%token LE GE EQ NE
```

```
%token FOR WHILE DO
%token TYPE
%token SEP

%left EQ NE
%left '<' '>' LE GE
%left '+' '-'
%left '*' '/'
%right UMINUS

%start program_list

%%

program_list:
   /* empty */
 | program_list program
 ;

program:
    stmt_list SEP    { if (!had_error) printf("Program: syntactically correct.\n"); else printf("Program: has syntax errors.\n"); had_error = 0; }
  | stmt_list        { /* last program may not have trailing SEP */ if (!had_error) printf("Program: syntactically correct.\n"); else printf("Program: has syntax errors.\n"); had_error = 0; }
 | error SEP      { printf("Program: has syntax errors.\n"); yyclearin; had_error = 0; }
 ;

stmt_list:
   /* empty */
 | stmt_list stmt
 ;

stmt:
   assignment ';'
 | decl_stmt
 | for_stmt
 | while_stmt
 | do_while_stmt
 | block
 ;
```

```
block:
   '{' stmt_list '}'
 ;

assignment:
   ID '=' expr     { /* assignment accepted */ }
 ;

decl_stmt:
   TYPE ID ';'
 | TYPE ID '=' expr ';'
 ;

expr:
   expr '<' expr
 | expr '>' expr
 | expr LE expr
 | expr GE expr
 | expr EQ expr
 | expr NE expr
 | expr '+' expr
 | expr '-' expr
 | expr '*' expr
 | expr '/' expr
 | '-' expr %prec UMINUS
 | '(' expr ')'
 | ID
 | NUM
 ;

for_stmt:
   FOR '(' for_init ';' for_cond ';' for_inc ')' stmt
 ;

while_stmt:
   WHILE '(' expr ')' stmt
 ;

do_while_stmt:
   DO stmt WHILE '(' expr ')' ';'
 ;
```

```
for_init:
   assignment
 ;

for_cond:
   expr
 ;

for_inc:
   ID INC
 | ID DEC
 | assignment
 ;

%%

void yyerror(const char *s) {
 fprintf(stderr, "Syntax error at line %d: %s\n", yylineno, s);
 had_error = 1;
}

int main(void) {
 /* parse input and rely on per-program messages; return 0 */
 yyparse();
 return 0;
}
```

sample_inputs.txt:

```
/* Valid basic for loop */
for(i=0;i<10;i++)
 x = x + 1;

###

/* Valid nested for loop */
for(i=0;i<3;i++) {
 for(j=0;j<2;j++)
   k = k + 1;
}
```

```
###

/* Invalid example: missing parenthesis */
for(i=0;i<3;i++ {
  x = 1;
}

###

/* Invalid example: missing semicolon after assignment */
for(a=0;a<5;a++)
  b = b + 1

###

/* Valid while loop */
while(i<10)
  i = i + 1;

###

/* Valid do-while loop */
do {
  x = x + 1;
} while(x < 5);

###

/* Invalid while: missing parentheses */
while i < 10)
  y = 0;

###

/* Invalid do-while: missing semicolon after while */
do
  z = z + 1;
while(z < 3)

###

/* Declaration tests */
```

```
int a;
int b = 10;

###

/* Expression precedence tests */
x = a + b * 2;    /* should parse as a + (b*2) */
 y = (a + b) * 2; /* parentheses change precedence */

###

/* Combined: declaration + loops */
int i = 0;
while(i < 3) {
  i = i + 1;
}

###

/* Invalid declaration: missing semicolon */
int bad = 5

###
```

OUTPUT:

```
PS C:\Users\Joseph\Desktop\compiler design\expt8> cd "c:\Users\Joseph\Desktop\compiler design\expt8"
PS C:\Users\Joseph\Desktop\compiler design\expt8>
PS C:\Users\Joseph\Desktop\compiler design\expt8> bison -d -v .\for.y
.\for.y: warning: 9 shift/reduce conflicts [-Wconflicts-sr]
.\for.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
PS C:\Users\Joseph\Desktop\compiler design\expt8> flex .\for.l
PS C:\Users\Joseph\Desktop\compiler design\expt8>
PS C:\Users\Joseph\Desktop\compiler design\expt8> # On Windows you usually do NOT need -lfl
PS C:\Users\Joseph\Desktop\compiler design\expt8> gcc -o .\for.exe .\for.tab.c .\lex.yy.c
PS C:\Users\Joseph\Desktop\compiler design\expt8> Get-Content .\sample_inputs.txt | .\for.exe
```

```
Program: syntactically correct.
Program: syntactically correct.
Syntax error at line 16: syntax error
Program: has syntax errors.
Syntax error at line 26: syntax error
Program: has syntax errors.
Program: syntactically correct.
Program: syntactically correct.
Syntax error at line 42: syntax error
Program: has syntax errors.
Syntax error at line 52: syntax error
Program: has syntax errors.
Program: syntactically correct.
Program: syntactically correct.
Program: syntactically correct.
Syntax error at line 77: syntax error
Program: has syntax errors.
```

```
PS C:\Users\Joseph\Desktop\compiler design\expt8> Get-Content .\sample_inputs.txt | .\for.exe | Tee-Object -FilePath .\parser_output.txt
Syntax error at line 16: syntax error
Syntax error at line 26: syntax error
Syntax error at line 42: syntax error
Syntax error at line 52: syntax error
Syntax error at line 77: syntax error
Program: syntactically correct.
Program: syntactically correct.
Program: has syntax errors.
Program: has syntax errors.
Program: syntactically correct.
Program: syntactically correct.
Program: has syntax errors.
Program: has syntax errors.
Program: syntactically correct.
Program: syntactically correct.
Program: syntactically correct.
Program: has syntax errors.
```

parser_output.txt:

```
Program: syntactically correct.
Program: syntactically correct.
Program: has syntax errors.
Program: has syntax errors.
Program: syntactically correct.
Program: syntactically correct.
Program: has syntax errors.
Program: has syntax errors.
Program: syntactically correct.
Program: syntactically correct.
Program: syntactically correct.
Program: has syntax errors.
```

CONCLUSION:

In this lab expt , we designed and implemented a syntax validator for basic and nested for-loops with assignment statements using Flex for lexical analysis and Bison/YACC for parsing. We defined tokens for keywords, operators, identifiers, and numbers, and built a C-like grammar that recognizes for-loop structure (initialization, condition, increment) with either single statements or block bodies. The parser reports clear, per-snippet verdicts and line-numbered errors, and we verified behavior using a suite of valid and intentionally invalid examples, separated by a custom snippet delimiter.

Through this expt, we gained hands-on experience with the lexer–parser interface (yylval/headers), operator precedence and associativity for expressions, and practical error handling via yyerror and simple recovery. We also learned to diagnose and reduce grammar conflicts, and prepared a minimal “for-only” variant to match the assignment scope closely. Overall, the lab expt strengthened our understanding of compiler front-end fundamentals and delivered a reproducible Windows build-and-run workflow for validating loop syntax. This expt also works for do while and while too.