NIKIT GOKHE

Class – SY Comp D1

Roll No.224024

GR No. 21810522

ASSIGNMENT NO :4

AIM:

Write a program using YACC specifications to implement syntax analysis phase of compiler to validate type and syntax of variable declaration in Java

THEORY:

Yacc takes a concise description of a grammar and produces a C routine that can parse that grammar, a parser. The yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a syntax error whenever its input doesn't match any of the rules. A yacc parser is generally not as fast as a parser you could write by hand, but the ease in writing and modifying the parser is invariably worth any speed loss. The amount of time a program spends in a parser is rarely enough to be an issue anyway.

The structure of a yacc parser is, not by accident, similar to that of a lex lexer. Our first section, the definition section, has a literal code block, enclosed in "comments belong inside C code blocks, at least within the definition sec- tion) and a single include file.

Then come definitions of all the tokens we expect to receive from the lexi- cal analyzer. In this example, they correspond to the eight parts of speech. The name of a token does not have any intrinsic meaning to yacc, although well-chosen token names tell the reader what they represent. Although yacc lets you use any valid C identifier name for a yacc symbol, universal custom dic- tates that token names be all uppercase and other names in the parser mostly or entirely lowercase.

The first indicates the end of the rules and the beginning of the user subroutines sec- tion. The most important subroutine is main() which repeatedly calls yyparse() until the lexer's input file runs out. The routine yyparse() is the parser generated by yacc, so our main program repeatedly tries to parse sentences until the input runs out. (The lexer returns a zero token when- ever it sees a period at the end of a line; that's the signal to the parser that the input for the current
parse is complete.)

The rules section describes the actual grammar as a set of production rules or simply rules. (Some people also call them productions.) Each rule con- sists of a single name on the left-hand side of the ":" operator, a list of sym- bols and action code on the right-hand side, and a semicolon indicating the end of the rule. By default, the first rule is the highest-level rule. That is, the parser attempts to find a list of tokens which match this initial rule, or more commonly, rules found from the initial rule. The expression on the right-hand side of the rule is a list of zero or more names.

Running YACC:

>yacc -d filename.y

>lex filename.l

>gcc y.tab.c lex.yy.c

>./a.out

The first line runs lex over the lex specification and generates a file, lex.yy.c, which contains C code for the lexer. In the second line, we use yacc to generate both y.tab.c and y.tub.h (the latter is the file of token defi- nitions created by the -d switch.) The next two lines compile each of the two C files. The final line links them together and uses the routines in the lex library libl.a, normally in /usr/Zib/libl.a on most UNrX systems. If you are not using AT andT lex and yacc, but one of the other implementations, you may be able to simply substitute the command names and little else will change. (In particular, Berkeley yacc and flex will work merely by changing the lex and yacc commands to byacc and Jlex, and removing the -22 linker flag.) However, we know of far too many differences to assure the reader that this is true. For example, if we use the GNU replacement bison instead of yacc, it would generate two files called chl-M.tab.c and ch1 -M.tab.h. On systems with more restrictive naming, such as MS-DOS, these names will change (typically ytab.c and ytab.h.)

Algorithm:

1. Start
2. Declare the libraries in the declaration section of yacc file
3. Specify the tokens TYPE , ID , SC
4. Specify the rule in rule section for type checking the rule will be start:TYPE ID SC
5. After checking rule perform action printf(”valid”).
6. In main() call function yyparse()and yywrap()
7. Declare the libraries in the declaration section of lex file
8. In rule section specify the regular expressions for the syntax and return appropriate action
9. End

Expected Input: int a ;

Expected Output: Valid Variable declaration

Expected Input: a int;

Expected Output: Invalid variable declaration

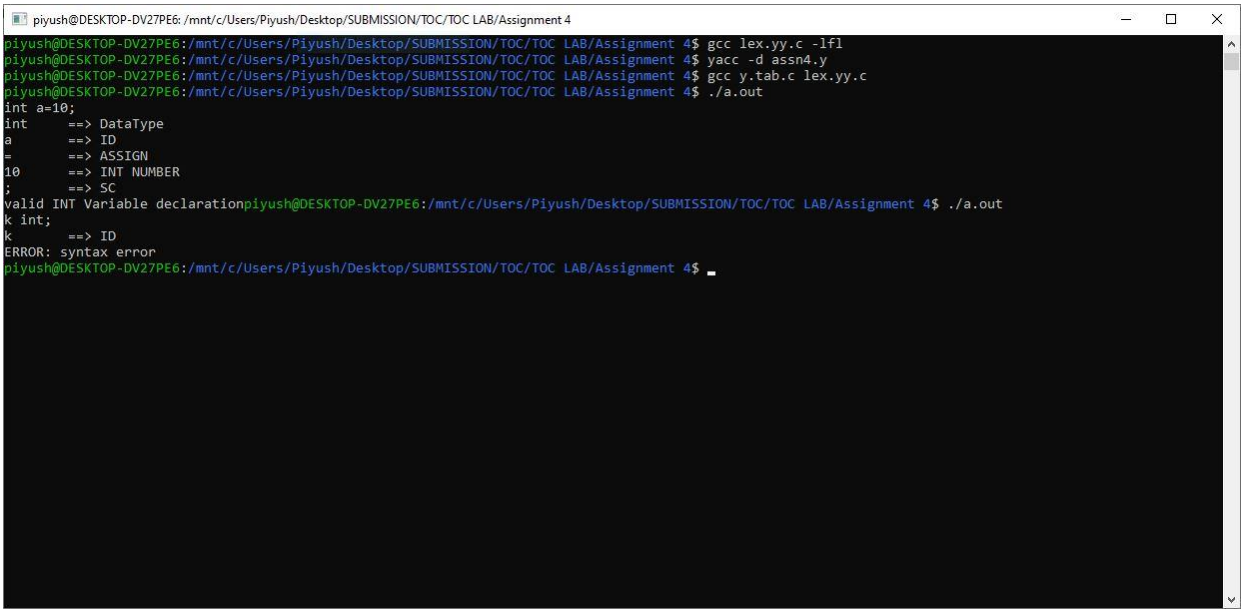
LEX CODE: (ASSN4.l)

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
DIGIT [0-9]
REAL {DIGIT}+[.]{DIGIT}*
LETTER [A-Za-z]
ASSIGN =
%%
[\t ] ;
int {printf("%s\t==> DataType\n",yytext);return (INT);}
float {printf("%s\t==> DataType\n",yytext);return (FLOAT);}
char {printf("%s\t==> DataType\n",yytext);return (CHAR);}
boolean {printf("%s\t==> DataType\n",yytext);return (BL);}
true|false { printf("%s\t==> BOOLEAN VAL\n",yytext);return BLVAL;}
['][^\t\n]['] { printf("%s\t==> CHAR VALUE\n",yytext);return CHVAL;}
[a-zA-z]+[a-zA-z0-9_]*  {printf("%s\t==> ID\n",yytext);return ID;}
{REAL} { printf("%s\t==> REAL NUMBER\n",yytext);return REAL;}
{DIGIT}+ { printf("%s\t==> INT NUMBER\n",yytext);return NUM;}
"," {printf("%s\t==> COMMA\n",yytext);return COMMA;}
";" {printf("%s\t==> SC\n",yytext);;return SC;}
{ASSIGN} {printf("%s\t==> ASSIGN\n",yytext);return AS;}
\n return NL;
. ;
%%
int yywrap()
{
return 1;
}
```

YACC CODE: (ASSN4.Y)

```
%{
#include<stdio.h>
void yyerror(char*);
int yylex();
//file* yyin;
%}
%token id sc int char float bl blval chval real as num comma nl
%%
s: type1|type2|type3|type4
;
type1:int varlist sc nl { printf("valid int variable declaration"); return 0;}
/// for "int a" test case(without sc ;) nl is added at end otherwise it waits for input
;
type2:float varlist2 sc nl{ printf("valid float variable declaration");return 0;}
;
type3:char varlist3 sc nl{ printf("valid char variable declaration");return 0;}
;
type4:bl varlist4 sc nl{ printf("valid boolean variable declaration");return 0;}
;
varlist:  id | id comma varlist | id as num |id as num comma varlist | //this is for epsilon case (empty)
;
varlist2:  id | id comma varlist2 | id as real |id as real comma varlist2 |
;
varlist3: id | id comma varlist3 | id as chval | id as chval comma varlist3 |
;
varlist4: id | id comma varlist4 | id as blval | as blval comma varlist4 |
;
%%
void yyerror(char *s )
{
fprintf(stderr, "error: %s\n",s);
}
int main()
{
//yyin=fopen("input.txt","r");
yyparse();
//fclose(yyin);
        return 0;
}
```

OUTPUT:


piyush@DESKTOP-DV27PE6: /mnt/c/Users/Piyush/Desktop/SUBMISSION/TOC/TOC LAB/Assignment 4
piyush@DESKTOP-DV27PE6:/mnt/c/Users/Piyush/Desktop/SUBMISSION/TOC/TOC LAB/Assignment 4$ gcc lex.yy.c -lfl
piyush@DESKTOP-DV27PE6:/mnt/c/Users/Piyush/Desktop/SUBMISSION/TOC/TOC LAB/Assignment 4$ yacc -d assn4.y
piyush@DESKTOP-DV27PE6:/mnt/c/Users/Piyush/Desktop/SUBMISSION/TOC/TOC LAB/Assignment 4$ gcc y.tab.c lex.yy.c
piyush@DESKTOP-DV27PE6:/mnt/c/Users/Piyush/Desktop/SUBMISSION/TOC/TOC LAB/Assignment 4$ ./a.out
int a=10;
int ==> DataType
a ==> ID
= ==> ASSIGN
10 ==> INT NUMBER
; ==> SC
valid INT Variable declarationpiyush@DESKTOP-DV27PE6:/mnt/c/Users/Piyush/Desktop/SUBMISSION/TOC/TOC LAB/Assignment 4$ ./a.out
k int;
k ==> ID
ERROR: syntax error
piyush@DESKTOP-DV27PE6:/mnt/c/Users/Piyush/Desktop/SUBMISSION/TOC/TOC LAB/Assignment 4$