EXPERIMENT NO 4

Date of performance :
Date of submission :

Aim: To implement a simple parser using Lex YACC

Software Used: c language

Theory:

A Parser for a Grammar is a program which takes in the Language string as it's input and produces either a corresponding Parse tree or an Error. When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the Language. The Lexical analysis can store all the recognized tokens in an intermediate file and give it to the Parser as an input. However it is more convenient to have the lexical Analyzer as a coroutine or a subroutine which the Parser calls whenever it requires a token. Parse trees are the Graphical representation of the grammar which filters out the choice for replacement order of the Production rules.

The Lex utility generates a 'C' code which is nothing but a yylex () function which can be used as an interface to YACC. A good amount of details on Lex can be obtained from the Man Pages itself. A Practical approach to certain fundamentals are given here. The General Format of a Lex File consists of three sections:

1. Definitions
2. Rules
3. User Subroutines

Definitions consists of any external 'C' definitions used in the lex actions or subroutines. e.g. all preprocessor directives like #include, #define macros etc. These are simply copied to the lex.yy.c file. The other type of definitions are Lex definitions which are essentially the lex substitution strings, lex start states and lex table size declarations.The Rules is the basic part which specifies the regular expressions and their corresponding actions. The User Subroutines are the function definitions of the functions that are used in the Lex actions.

Yacc is the Utility which generates the function 'yyparse' which is indeed the Parser. Yacc describes a context free, LALR (1) grammar and supports both bottom-up and top-down parsing. The general format for the YACC file is very similar to that of the Lex file.

1. Declarations
2. Grammar Rules
3. Subroutines

In Declarations apart from the legal 'C' declarations there are few Yacc specific declarations which begins with a %sign.

Algorithm:

Step 1. Identify the Terminal and Non-Terminal Symbols from the BNF and Lex.
Step 2. Try coding all the grammar rules in yacc with empty actions Compile,link it to Lex and check for conflicts. This is an easy way of validating the BNF for reduce/reduce and shift/reduce conflicts.
Step 3. Search for any reduce/reduce conflict. Resolve it in Lex.
Step 4. Resolve any shift/reduce conflicts. Details on resolving it given later.
Step 5. Write rules for all possible syntax errors.Details on error handling are given later.
Step 6. Code the yyerror function in subroutine section.

Step 6. Design the Data Structure which can be easily integrated with the grammar rules for syntax directed translation.
Step 7. From the Data Structures and Lex needs ,formulate the correct Stack.The stack must have pointers for all the data structures.
Step 8. Do the appropiate type binding to all tokens and yacc variables (non-terminals).
Step 9. Write all the data structures in a seperate file and inlude it in yacc.
Step 10.Code all the actions.
Step 11.Restrict the actions in case of error, i.e no data structure should be built but parsing should continue to get more errors.

Eliminating shift/reduce errors

1. use -d switch of yacc to create debug file(y.output). This file will contain the full transition diagram description and the points at which any conflict arises.
2. Try assigning precedence and associativity to tokens and literals by using %left %right %noassoc %prec. Note that precedence level in the same line is same and down the line increases.
3. In majority of the cases shift/reduce conflict is always in the vicinity of left/right recursions. These might not be due to associativity or precedence relations.e.g consider the rules

s->XabY
a->E|aXAY E=empty transition
b->E|bXBY

The syntax of these rules says that there is block XY which can have zero or more blocks of type A & B. These rules have shift/reduce conflict on the symbol X since in s->AabY for making a transition from literal a to b with input X it has no way to tell if it should reduce or shift another token .

These rules can be rewritten as following

s->XaY
a->E|aXbY
b->A|B

Syntax checking and error recovery.It is one of the toughest part of parsing. There are many functions like yyerrok etc. However not all yacc versions supports them. The simplest method is just to use the pseudo literal 'error'. In a rule whenever there is an error, yacc pushes a pseudo literal error and takes in next input. On identifying the rule it pops the stack and takes proper actions. Thus in this way the file pointer will always point to the right location and next rule can be looked for correctly. In our example code error class {yyerror("Missing Relation"); } says that if only class exists then error must be flaged. the literal error is pushed on the stack if relation is missing and then class is pushed. On reduction it calls yyerror with msg string.

Program :

```
//lex prgram
%{ #include “y.tab.h”
Extern int yylval
%}
%%
[0-9]+{yylval=atoi(yytext);
Return NUM;
}
Return yytext[0];
\n return 0;
%%
Int yywrap()
{
Return1;
}

// yacc program<x1.y>
%{
Include<stdio.h>
%}
```

```
% token A NUM
%%
State: A'='E | E        {printf("\n the result=%d\N",$1);}
;
E:E'+'Nunm{$$=$1+$3;}
| NUM  { $$=$1;}
;
%%
Extern FILE yyin;
Main()
{
Do
{
Yyparse();
}
While(!feof(yyin))
}
Yyerror(char *s)
{
Fprintf(stderr,"%s\n",s);
}
```

Output

```
[root@app root]$ lex x1.l
[root@app root]$ yacc –d x1.y
[root@app root]$ cc lex.yy.c y.tab.c
[root@app root]$ ./a.out
4+5
The result=10
```

Conclusion: Hence we have generated implemented parser using Lex YACC.

SIGN AND REMARK

DATE

R1	R2	R3	R4	R5	Total (15 Marks)	Signature