COMPILER DESIGN LAB IIIT RANCHI **Posted : April 13, 2024**

Lab Assignment 6 :

Objective : Communication between scanner and parser; simple applications of semantic analysis in the domain of compilers.

General Remarks :

1. Save all your work in this laboratory session in a separate directory, Lab6.
2. Read the supporting document on methods of communication between scanner (generator by lex) and parser (generator by yacc).
3. Generation of scanner and parser, where "file.l" and "file2.y" are the lex and yacc scripts respectively.

```
$ lex file.l
$yacc -dv file2.y
$ gcc lex.yy.c y.tab.c  -ll -o file   (check whether in your installation lex library -ll is required or not).
```

4. Read the Annexure of this document to get the context of the experiments.
5. For each problem, you have to do the assigned task and make observations and write them in a text file. Call one of the faculty to show the text file and get it approved before moving onto the next problem.

P1. The objective of this problem is to understand the communication between a scanner and its partner (the parser) and also to perform simple semantic analysis that is done by a calculator. You have to support 2 operators {+, *} and a token NUM for unsigned integer values. Write an unambiguous grammar for the expressions of interest and a lex script in two files, "calc.l" and "calc.y", based on what you have learnt today and the information given in the attached documents. Generate a parser and run the parser with a few sample inputs of the form, "20 + 3 *5 +1" and verify that the output is as shown below.

Sample Input	Desired Output
20 + 3 *5 +1	Final Value of E : 36

P2. Redo problem P1 but use an ambiguous grammar and disambiguating rules of yacc to create the parser. Note that the specification given below, informs that yacc that both '+' and '*' are left associative operators and also '*' has higher precedence than '+'.

```
%left '+'
%left '*'
```

In case there are several operators, say, α and η which have the same precedence, and both are higher than '*', then we write

```
%left '+'
%left '*'
%left 'α' 'η'
```

Test your parser with the same input of P1 and also other inputs generated by you.

P3. Extend the operator set by using the operators {+, -, *, / %} and generate a parser for expressions for this set. Use an ambiguous grammar for this purpose. Test your parser with the following input and verify the output produced.

Sample Input	Desired Output
20 + 3 *5/1 + 15%2 - 5%2	Final Value of E : 35

ANNEXURE 1

1. You have been informed about attribute values, that accompanies the parser stack. For the rule A → A t B, where A and B are nonterminals and t is a token, the attribute values of the rhs of the rule are denoted by $1, $2,.., while the value of lhs A is denoted by $$. These $symbols can be used in the action part in yacc script to access the values of the grammar symbols.

```
A  →  A  t  B
$$    $1 $2 $3
```

2. For a terminal, say t, lex has to assign the attribute value(s) of t in its action part. The variable yylval is defined for this purpose. The default declaration of yylval is "int yylval = 0" somewhere in the "lex.yy.c file". Hence communicating an integer attribute value from scanner to the parser is enabled by default. To communicate multiple attributes of a token, more work is required to customize the definition of yylval for the intended use.

3. For nonterminals, scanner has no role to play. Yacc permits nonterminal symbols also to have attribute values which in turn can be accessed for semantic analysis in the action part of a rule. In the case of synthesized attributes, we shall use attribute evaluation rules of the form " $$ = f($1, $2, ... $n)", where f() is some function of its arguments. For a production rule of the form, A → X1 X2 ... Xn where $i is an attribute of Xi, place " $$ = f($1, $2, ... $n)" in the action part at the end of the rule. How attribute values are attached to nonterminal symbols is a part of the experimentation for the day.

4. The experiments listed in the problems above are designed to equip you to attach attribute value(s) to terminal and nonterminal symbols as required for semantic analysis to be performed in the action part of CFG rules.. Note that for terminals, the lex script has to do all the work to assign attribute values to the token, while for nonterminals and evaluation of attribute values, the yacc script has the major role to play.

5.The operators in C / C++ with their attributes are given in the following table.

Precedence	Associativity	Arity	Operator	Function of the operator
13	L	binary	*, /, %	multiplicative operators
12	L	binary	+, -	arithmetic operators
11	L	binary	<<, >>	bitwise shift
10	L	binary	<, <=, >, >=	relational operators
9	L	binary	==, !=	equality, inequality
8	L	binary	&	bitwise AND
7	L	binary	^	bitwise XOR
6	L	binary	|	bitwise OR
5	L	binary	&&	logical AND
4	L	binary	||	logical OR
3	L	ternary	?:	arithmetic if
2	R	binary	=, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=	assignment operators,

ANNEXURE 2

Communication between lex script and yacc script

Lex script	Yacc script
"+" {return '+';} "*" {return '*';}	E: E '+' T {$$ = $1 + $3; printf(" Value of E + T: %d\n", $$);} T: T '*' F {$$ = $1 * $3; printf(" Value after T * F : %d\n", $$);}

Lex script	Yacc script
#include "y.tab.h" {digit}+ { yylval = atoi(yytext); return NUM;}	%token NUM F: NUM {$$ = $1; printf(" NUMBER : %d\n", $$);}
#define NUM 258 occurs in the file "y.tab.h"	

Lex script	Yacc script
#include <stdlib.h> {digit}+ { yylval = atoi(yytext); return NUM;}	F: NUM {$$ = $1; printf(" NUMBER : %d\n", $$);} $$ S1
No definition of yylval in the script files; default integer used $1 is assigned the value of yylval (as a global variable)	

- By default, when yylval is not explicitly defined in yacc script, attributes of all grammar symbols are set to an integer value.
- The yylval value sent by the lexer is saved in $1. Therefore if if lexer sets yylval = 100, then after the rule E -> NUM is used, as a side effect, $1 is set to 100. At the end of every rule, yacc supplants the rule $$ =$1; so that the value of $1 is propagated to $$. As a consequence, after the rule has been applied, E carries the value 100 with it.
- This also explains why E -> E + E has the actions stated in the grammar.

```
                    $$   $1  $3
E '+' E  {$$ = $1 + $3; printf(" Value after E + E: %d\n", $$);}
```

End of Experiment 6