Don Bosco Institute of Technology, Kulra(W)
Department of Computer Engineering

CSL601: System Programming and Compiler Construction Lab2022-23

Name : Ronak Surve
Roll No : 64
Subject : SPCC Lab
Batch : C

Experiment No.:	07
Experiment Title:	Intermediate code Generator Task: Implement Intermediate Code Generation and convert infix expression to postfix expression.(Yacc parser)
Student Name	Ronak Surve
Roll No.	64
Objectives :	1) To understand parser generator tool : YACC and Intermediate code generation phase of compiler

Theory /Algorithm :	Intermediate code generation is a phase in the compiler design process where the compiler takes the source code and converts it into an intermediate representation that is more suitable for further analysis and optimization. The intermediate code generated is not executable, but it is easier to work with and is more machine-independent than the source code. The process of intermediate code generation involves several steps, including lexical analysis, syntax analysis, and semantic analysis. The output of these steps is a parse tree, which is then transformed into an abstract syntax tree (AST). The AST is a simplified representation of the parse tree that eliminates irrelevant details and provides a more concise and understandable view of the code structure. After the AST is generated, the compiler can start the intermediate code generation phase. This phase involves converting the AST into a more machine-independent representation, typically using three-address code or quadruples. Three-address code is a low-level language that represents expressions in terms of three operands and an operator. For example, the expression "a = b + c" would be represented as "t1 = b + c" and "a = t1". Quadruples are similar to three-address code but use four fields to represent the operator, two operands, and a result. The intermediate code generation phase involves several tasks, including expression evaluation, control-flow analysis, and code optimization. Expression evaluation involves converting expressions into three-address code or quadruples. Control-flow analysis involves identifying basic blocks and constructing control-flow graphs to represent the program's

control flow. Code optimization involves transforming the intermediate code to improve performance or reduce code size.

Once the intermediate code is generated, it can be further optimized using techniques such as constant folding, strength reduction, and loop optimization. Constant folding involves replacing expressions with their constant values at compile-time. Strength reduction involves replacing expensive operations with cheaper ones, such as replacing multiplication with addition. Loop optimization involves optimizing loops to reduce the number of iterations or eliminate redundant code.

In conclusion, intermediate code generation is an essential phase in the compiler design process. It involves converting the AST into a more machine-independent representation using three-address code or quadruples. The intermediate code can then be further optimized to improve performance or reduce code size, making it easier for the compiler to generate efficient machine code.

Program Code:

scan.l

```
%{
#include"y.tab.h"
extern int yylval;
%}
%%
[0-9]+  {yylval=atoi(yytext); return NUMBER;}
\n      return 0;
[ \t]   ;
.       return *yytext;
%%

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

postfix.y

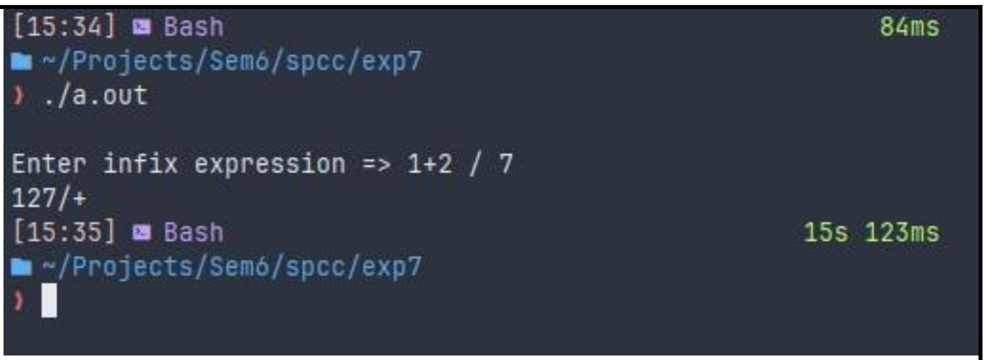
```
%{
#include<stdio.h>
%}
%token NUMBER
```

```
%left '+' '-'

%left '*' '/'

%right NEGATIVE

%%


S:  E {printf("\n");}

    ;

E:  E '+' E {printf("+");}

    |   E '*' E {printf("*");}

    |   E '-' E {printf("-");}

    |   E '/' E {printf("/");}

    |   '(' E ')'

    |   '-' E %prec NEGATIVE {printf("-");}

    |   NUMBER     {printf("%d", yylval);}

    ;

%%


int main(){

    printf("\nEnter infix expression => ");

    yyparse();

}


int yyerror (char *msg) {

    return printf ("Error: %s\n", msg);

}
```

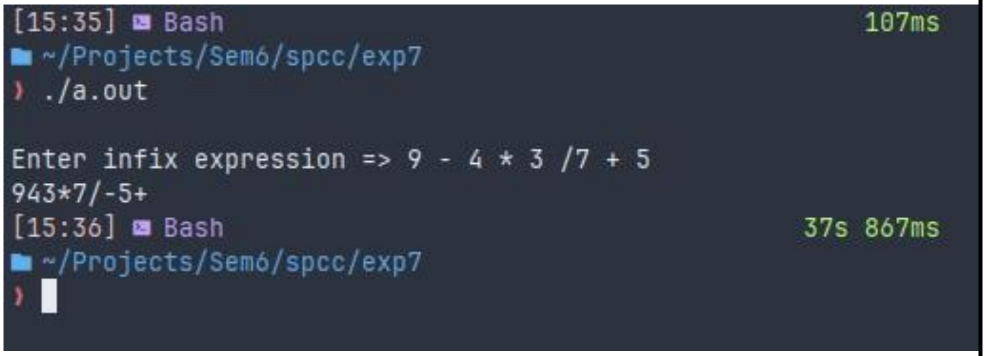
Input to the Program:

1. `1+2 / 7`
2. `9 - 4 * 3 /7 + 5`
3. `1 2 + 3`
4. `12 - 3`
5. `1 + + 2`

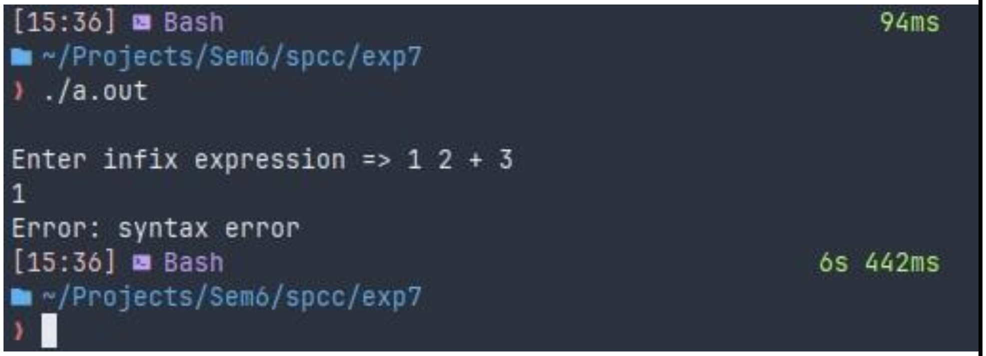
Output of the program:



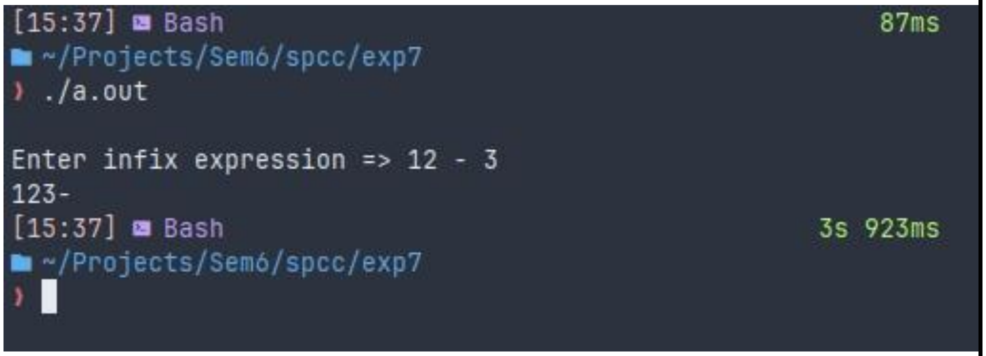
```
[15:34]  Bash                                                84ms
 ~/Projects/Sem6/spcc/exp7
❯ ./a.out

Enter infix expression => 1+2 / 7
127/+
[15:35]  Bash                                            15s 123ms
 ~/Projects/Sem6/spcc/exp7
❯ 
```



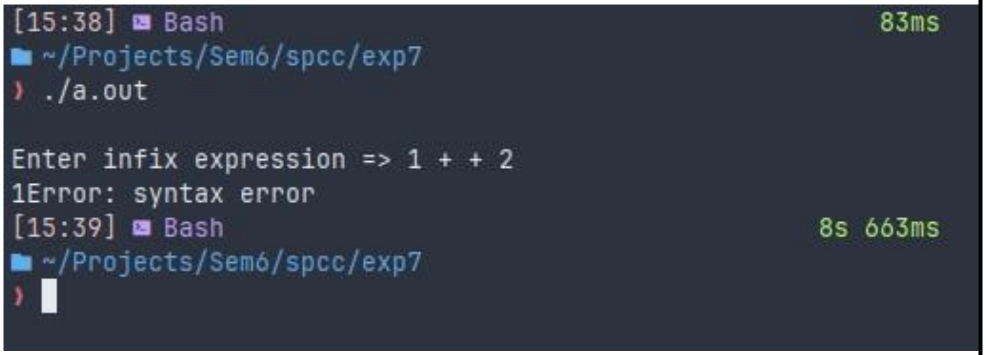
```
[15:35]  Bash                                               107ms
 ~/Projects/Sem6/spcc/exp7
❯ ./a.out

Enter infix expression => 9 - 4 * 3 /7 + 5
943*7/-5+
[15:36]  Bash                                            37s 867ms
 ~/Projects/Sem6/spcc/exp7
❯ 
```



```
[15:36]  Bash                                                94ms
 ~/Projects/Sem6/spcc/exp7
❯ ./a.out

Enter infix expression => 1 2 + 3
1
Error: syntax error
[15:36]  Bash                                             6s 442ms
 ~/Projects/Sem6/spcc/exp7
❯ 
```



```
[15:37]  Bash                                                87ms
 ~/Projects/Sem6/spcc/exp7
❯ ./a.out

Enter infix expression => 12 - 3
123-
[15:37]  Bash                                             3s 923ms
 ~/Projects/Sem6/spcc/exp7
❯ 
```



```
[15:38]  Bash                                                83ms
 ~/Projects/Sem6/spcc/exp7
❯ ./a.out

Enter infix expression => 1 + + 2
1Error: syntax error
[15:39]  Bash                                             8s 663ms
 ~/Projects/Sem6/spcc/exp7
❯ 
```

Outcome of the Experiment:

YACC is a tool used for generating parsers or syntax analyzers for a programming language. It is used with the FLex tool to create a simple compiler for validating source code syntax.

In this experiment these two tools were used to create a compiler to convert arithmetic expressions written in human readable infix form to

	compiler readable postfix form. These expressions should also be valid for any programming language.
Reference:	Class notes https://www.geeksforgeeks.org/