1. INTRODUCTION

1.1. Project Details

Grammar Name: String, Char and Int Operations.

Grammar Rules:

1.1.1. x=p+q Concatenates the p with q and stores result in x
1.1.2. x=p-q Removes the last "q" symbol from p and store answer back in x.
1.1.3. x=p<->q Replaces all occurrence of p variable with q variable in x string.
1.1.4. pos=x?p Find and return the first position in X where q string was found

- Variables can be of type: "string" or "char"
- Variables can be assigned value using "=" symbol.
- Variable names should start with "small case character" & can't contain integer.
- The maximum size of name can be 3.

Regular Expression:

letter	-->	[a-z A-Z]
sletter	-->	[a-z]
assop	-->	=
relop	-->	+ | - | <-> | ?
singlequote	-->	'
doublequote	-->	"
string	-->	string
char	-->	char
int	-->	int
id	-->	sletter(letter)?(letter)?
whitespace	-->	(space|tab|newline)*

Token Table:

Regular Expression	Token	Attribute-Value
ws	-	-
id	id	pointer to table entry
string	string	-
char	char	-
int	Int	-
=	assop	assign
+	relop	concate
-	relop	remove
<->	relop	replace
?	relop	find
‘	singlequote	quotes
“	doublequote	quotes

Valid Input:

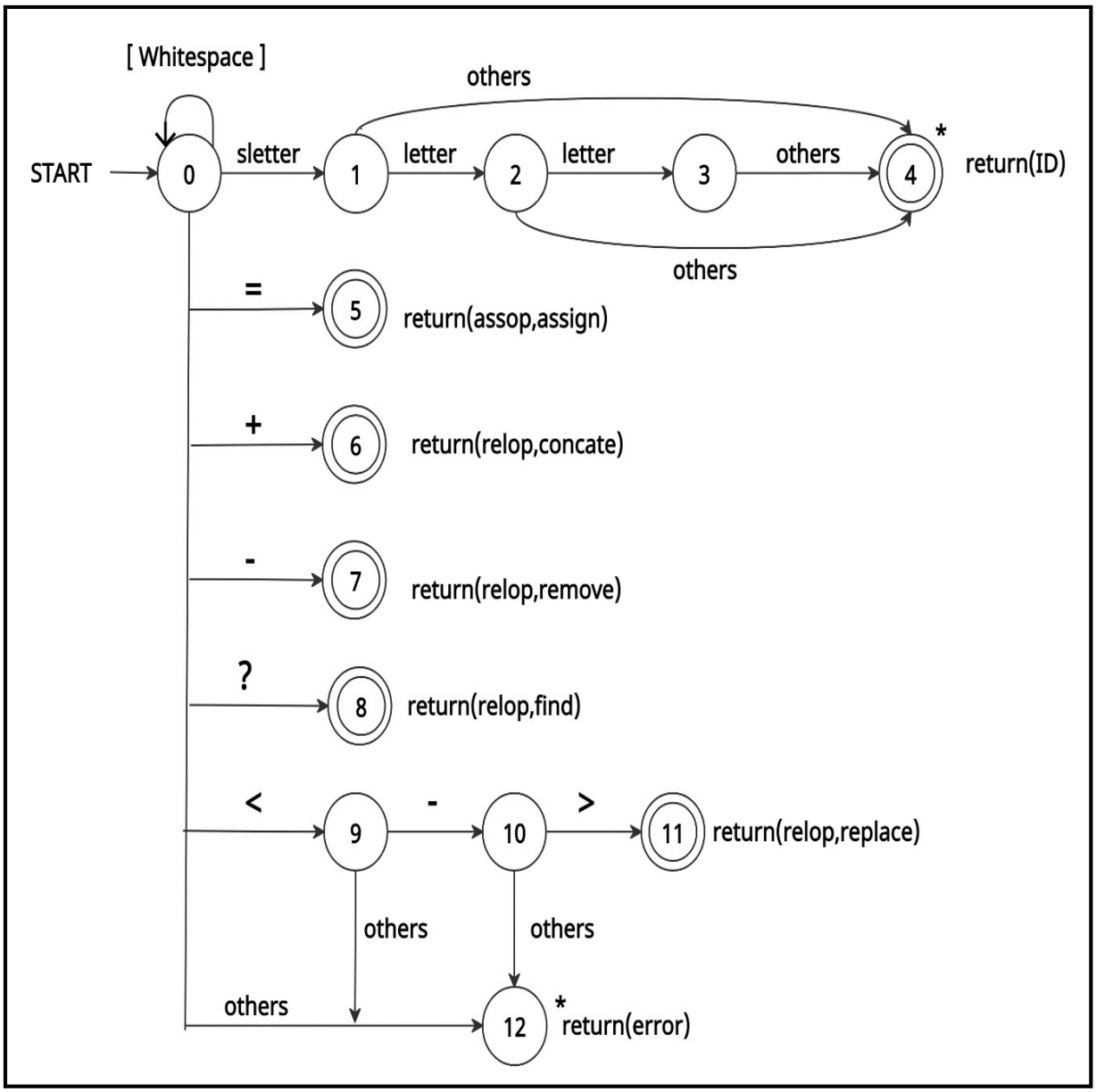
- **String can be declared in double quote only.**
- **Char can be declared in single quote only.**
- **Position operator (?) value return only int value so only (int pos = pqr ? xyz) is valid**

1. string s; //only declaration of string
2. string str = “Shyam”; //string declaration with assignment
3. string str = pqr + xyz; //string operation using variable(pqr,xyz)
4. string str = “Shyam” + “Mahesh”; //string operation using //value(“Shyam”,”Mahesh”)
5. char c; //only declaration of char
6. char c = ‘A’; //char declaration with assignment
7. int pos = pqr ? xyz; //position operation and return type is int

1 to 7 are valid declaration rest of all are syntax error.

2. LEXICAL PHASE DESIGN

2.1. Deterministic Finite Automaton design for Lexer



2.2. Algorithm of Lexer

```
Lexer()
{
   input(c);
   state=0;
   while(c!=EOF)
   {
      switch(state)
      {
         case 0: if(c==' ' || c=='\t' || c=='\n') state=0;
                 else if(c == sletter) state = 1;
                 else if(c == '=') state = 5;
                 else if(c == '+') state = 6;
                 else if(c == '-') state = 7;
                 else if(c == '?') state = 8;
                 else if(c == '<') state = 9;
                 else state = 12;
                 break;

         case 1: input(c);
                 if(c == letter)         state = 2;
                 else state = 4;
                 break;

         case 2: input(c);
                 if(c == letter)     state=3;
                 else state = 4;
                 break;

         case 3: input(c);
                 if(c == other) state=4;
                 break;
```

```
            case 4: state=0;
                    unput(c);
                    return(ID);

            case 5: state = 0;
                    return(assop,assign);

            case 6: state = 0;
                    return(relop,concate);

            case 7: state = 0;
                    return(relop,remove);

            case 8: state = 0;
                    return(relop,find);

            case 9: input(c);
                    if(c == '-')            state = 10;
                    else state=12;
                    break;

            case 10: input(c);
                     if(c == '>')           state = 11;
                     else state=12;
                     break;

            case 11: state = 0;
                     return(relop,replace);

            case 12: unput(c);
                     return(error);
        }
    }
}
```

2.3. Implementation of lexer

Flex Program:

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

ID [a-z][a-zA-Z]?[a-zA-Z]?

%%

string printf("< Keyword , string>\n");
char   printf("< Keyword , char>\n");
int     printf("< Keyword , int>\n");
"+"    printf("< Concatenate Op , %s >\n",yytext);
"-"     printf("< Remove Op , - >\n");
"<->" printf("< Replace Op , <-> >\n");
\?      printf("< Position Op , ? >\n");
=       printf("< Assignment Op , = >\n");
\'      printf("< Single Quote , ' >\n");
\"      printf("< Double Quote , \" >\n");
;       printf("< Semicolon , ; >\n");

{ID}                printf("< ID , %s >\n",yytext);
[A-Z]               printf("< CharValue , %s >\n",yytext);
[a-zA-Z0-9]+        printf("< StringValue , %s >\n",yytext);
[ \t]               {}
.                    printf("<Invalid Token , %s >\n",yytext);

%%

int yywrap(){}
int main()
{
        yylex();
        return 0;
}
```

2.4. Execution environment setup

Step by Step Guide to Install FLEX and Run FLEX Program using Command Prompt(cmd)

Step 1:- /*DOWNLOADING FLEX*/

/*For downloading CODEBLOCKS */

- Open your Browser and type in "codeblocks"

- Goto to Code Blocks and go to downloads section

- Click on "Download the binary release"

- Download and Install codeblocks-20.03mingw-setup.exe

/*For downloading FLEX GnuWin32 */

- Open your Browser and type in "download flex gnuwin32"

- Goto to "Download GnuWin from SourceForge.net"

- Downloading will start automatically and the Install

/*SAVE IT INSIDE C FOLDER*/

Step 2:- /*PATH SETUP FOR CODEBLOCKS*/

- After successful installation

Goto program files->CodeBlocks-->MinGW-->Bin

- Copy the address of bin :-

it should somewhat look like this

C:\Program Files (x86)\CodeBlocks\MinGW\bin

- Open Control Panel-->Goto System-->Advance System Settings--
>Environment Variables

- Environment Variables--> Click on Path which is inside System
variables - Click on edit

- Click on New and paste the copied path to it:-

- C:\Program Files (x86)\CodeBlocks\MinGW\bin

- Press Ok!

Step 3:- /*PATH SETUP FOR GnuWin32*/

- After successful installation Goto C folder

- Goto GnuWin32-->Bin

- Copy the address of bin it should somewhat look like this

C:\GnuWin32\bin

- Open Control Panel-->Goto System-->Advance System Settings--
>Environment Variables

- Environment Variables--> Click on Path which is inside System
variables - Click on edit
- Click on New and paste the copied path to it:-

- C:\GnuWin32\bin

- Press Ok!

/*WARNING!!! PLEASE MAKE SURE THAT PATH OF CODEBLOCKS IS BEFORE GNUWIN32---THE ORDER MATTERS*/

Step 4

- Create a folder on Desktop flex_programs or whichever name
you like - Open notepad type in a flex program
- Save it inside the folder like filename.l

-Note :- also include " void yywrap(){} " in the .l file

/*Make sure while saving save it as all files rather than as a text document*/

Step 5 /*To RUN FLEX PROGRAM*/

- Goto to Command Prompt(cmd)

- Goto the directory where you have saved the
program - Type in command :- **flex filename.l**
- Type in command :- **gcc lex.yy.c**

- Execute/Run for windows command promt :- **a.exe**

Step 6
- Finished

2.5. Output screenshots of Lexer

```
string str = abc + pqr;
< Keyword , string>
< ID , str >
< Assignment Op , = >
< ID , abc >
< Concatenate Op , + >
< ID , pqr >
< Semicolon , ; >
```

```
string str = "Shyam" + "Mahesh";
< Keyword , string>
< ID , str >
< Assignment Op , = >
< Double Quote , " >
< StringValue , Shyam >
< Double Quote , " >
< Concatenate Op , + >
< Double Quote , " >
< StringValue , Mahesh >
< Double Quote , " >
< Semicolon , ; >
```

```
char c;
< Keyword , char>
< ID , c >
< Semicolon , ; >
```

```
char c = 'A';
< Keyword , char>
< ID , c >
< Assignment Op , = >
< Single Quote , ' >
< CharValue , A >
< Single Quote , ' >
< Semicolon , ; >
```

```
int pos = abc ? pqr;
< Keyword , int>
< ID , pos >
< Assignment Op , = >
< ID , abc >
< Position Op , ? >
< ID , pqr >
< Semicolon , ; >
```

```
string s + - <-> ? ! @ # $ % ^ &
< Keyword , string>
< ID , s >
< Concatenate Op , + >
< Remove Op , - >
< Replace Op , <-> >
< Position Op , ? >
<Invalid Token , ! >
<Invalid Token , @ >
<Invalid Token , # >
<Invalid Token , $ >
<Invalid Token , % >
<Invalid Token , ^ >
<Invalid Token , & >
```

3. SYNTAX ANALYZER DESIGN

3.1 Grammar rules

START -> EXPR SEMICOLON NEWLINE

EXPR -> 'string' VARNAME STR
| 'char' VARNAME CHAR
| 'int' VARNAME INT

STR -> ASSGN STR1 | ^ //string var;
STR1 -> VARNAME SOP VARNAME | "VALUESTR" STR2
//string var = var + var;
//string var = "Shyam";

STR2 -> SOP "VALUESTR" | ^ //string var = "Shyam" + "Mahesh";

CHAR -> ASSGN CHAR1 //char var;
CHAR1 -> 'VALUECHAR' //char var = 'A';

INT -> ASSGN INT1 //int var;
INT1 -> VARNAME IOP VARNAME //int var = var <-> var;

ASSGN -> '='
SOP -> '+' | '-' | '<->'
IOP -> '?'

3.2 Yacc based implementation of syntax analyzer

- **Project.l**

```
%{
   #include <stdio.h>
   #include "y.tab.h"
%}

%%
string {printf("\n--> VALID TOKENS <--\n< Keyword , string>\n");return SDT;}
char    {printf("\n--> VALID TOKENS <--\n< Keyword , char>\n");return CDT;}
int     {printf("\n--> VALID TOKENS <--\n< Keyword , int>\n");return  IDT;}
"+"     {printf("< Relop , + >\n");return SOP;}
"-"     {printf("< Relop , - >\n");return  SOP;}
"<->"  {printf("< Relop , <-> >\n");return SOP;}
\?      {printf("< Relop , ? >\n");return  IOP;}
=       {printf("< Assop , = >\n");return  ASSGN;}
\'      {printf(" < Single Quote , ' >\n");return  SINGLE;}
\"      {printf("< Double Quote , \" >\n");return  DOUBLE;}
;       {return SEMICOLON;}
\n      {return NEWLINE;}

[a-z][a-zA-Z]?[a-zA-Z]? {printf("< ID , %s >\n",yytext);return VAR;}
[A-Z]                   {printf(" < CharValue , %s >\n",yytext);return VALCHAR;}
[a-zA-Z0-9]+            {printf("< StringValue , %s >\n",yytext);return  VALSTR;}
%%

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

- **Project.y**

```
%{
   #include<stdio.h>
   #include<string.h>
   #include<stdlib.h>
   #include "y.tab.h"
   extern int yylex();
   extern int yyparse(void);
   extern int yyerror(char *s);
   int flag=0;
   char s[100]="\n--> 3 ADDRESS CODE <--\n";
%}

%token SEMICOLON NEWLINE SDT CDT IDT SOP IOP ASSGN SINGLE DOUBLE
VAR VALSTR VALCHAR


%%
start : expr SEMICOLON NEWLINE
      {
        printf("\n--> CORRECT SYNTAX <--");
        if(flag==1) printf("\nValid String Declaration\n");
        else if(flag==2) printf("\nValid Char Declaration\n");
        else if(flag==3)printf("\nValid String Operation with Variable\n");
        else if(flag==4) {
            printf("\nValid String Operation with Value\n");
            printf("\n--> 3 Address Code <--\nT1 = \"value1\"\n");
            printf("T2 = \"value2\"\nT3 = T1 operand T2\nvarname = T3\n");
            strcpy(s,"");
        }
        else if(flag==5) printf("\nValid String Assignment\n");
        else if(flag==6) printf("\nValid Char Assignment\n");
        else if(flag==6) printf("\nValid Char Assignment\n");
        else if(flag==7) printf("\nValid Position(?) Operation\n");
        flag=0;
        printf("%s\n",s);
        strcpy(s,"\n--> 3 ADDRESS CODE <--\n");
        return 0;
      }
```

```
expr : SDT VAR str
     | CDT VAR char
     | IDT VAR int

str : ASSGN rhsstring {strcat(s,"varname = T");}
     | {strcat(s,"No 3 Address Code");flag=1;}

char : ASSGN rhschar {strcat(s,"varname = T");}
     | {strcat(s,"No 3 Address Code");flag=2;}

int : ASSGN rhsint {strcat(s,"pos = T");}

rhsstring : VAR SOP VAR    {strcat(s,"T = varname operand varname\n");flag=3;}
     | DOUBLE VALSTR DOUBLE rhsstring1 {strcat(s,"T = \"value\"\n");}

rhsstring1 : SOP DOUBLE VALSTR DOUBLE    {flag=4;}
     |       {flag=5;}

rhschar : SINGLE VALCHAR SINGLE     {strcat(s,"T = 'charvalue'\n");flag=6;}
rhsint : VAR IOP VAR    {strcat(s,"T = varname1 ? varname2\n");flag=7;}

%%


int main()
{
   while(1)
   {
      printf("\nEnter : ");
      yyparse();
   }
}


int yyerror(char *str)
{
   printf(" %c %c\n",24,24);
   fprintf(stderr,"\n--> SYNTAX ERROR <--
    \nCheck your last Token generated to Detect Error\n");
   exit(0);
}
```

3.3 Execution environment setup

http://gnuwin32.sourceforge.net/packages/flex.htm
http://gnuwin32.sourceforge.net/packages/bison.htm
when installing on windows you store this in c:/gnuwin32 folder and not in c:/program files(X86)/gnuwin32

https://sourceforge.net/projects/orwelldevcpp/

set environment varialble and then run program

Open a prompt, cd to the directory where your ".l" and ".y" are, and compile them with:

flex lexical.l
bison -dy syntax.y
gcc lex.yy.c y.tab.c -o compiler.exe
compiler.exe

3.4 Output screenshots of yacc based implementation

Valid Input: -

```
Enter : string s;

--> VALID TOKENS <--
< Keyword , string>
 < ID , s >

--> CORRECT SYNTAX <--
Valid String Declaration

--> 3 ADDRESS CODE <--
No 3 Address Code
```

```
Enter : string str = "Mahesh";

--> VALID TOKENS <--
< Keyword , string>
 < ID , str >
 < Assop , = >
 < Double Quote , " >
< StringValue , Mahesh >
< Double Quote , " >

--> CORRECT SYNTAX <--
Valid String Assignment

--> 3 ADDRESS CODE <--
T = "value"
varname = T
```

```
Enter : string s = a + b;

--> VALID TOKENS <--
< Keyword , string>
 < ID , s >
 < Assop , = >
 < ID , a >
 < Relop , + >
 < ID , b >

--> CORRECT SYNTAX <--
Valid String Operation with Variable

--> 3 ADDRESS CODE <--
T = varname operand varname
varname = T
```

```
Enter : string str = "Mahesh" <-> "Vegada";

--> VALID TOKENS <--
< Keyword , string>
 < ID , str >
 < Assop , = >
 < Double Quote , " >
< StringValue , Mahesh >
< Double Quote , " >
 < Relop , <-> >
 < Double Quote , " >
< StringValue , Vegada >
< Double Quote , " >

--> CORRECT SYNTAX <--
Valid String Operation with Value

--> 3 Address Code <--
T1 = "value1"
T2 = "value2"
T3 = T1 operand T2
varname = T3
```

```
Enter : char c;

--> VALID TOKENS <--
< Keyword , char>
 < ID , c >

--> CORRECT SYNTAX <--
Valid Char Declaration

--> 3 ADDRESS CODE <--
No 3 Address Code
```

```
Enter : char c = 'A';

--> VALID TOKENS <--
< Keyword , char>
 < ID , c >
 < Assop , = >
  < Single Quote , ' >
 < CharValue , A >
 < Single Quote , ' >

--> CORRECT SYNTAX <--
Valid Char Assignment

--> 3 ADDRESS CODE <--
T = 'charvalue'
varname = T
```

```
Enter : int pos = abc ? pqr;

--> VALID TOKENS <--
< Keyword , int>
 < ID , pos >
 < Assop , = >
 < ID , abc >
 < Relop , ? >
 < ID , pqr >

--> CORRECT SYNTAX <--
Valid Position(?) Operation

--> 3 ADDRESS CODE <--
T = varname1 ? varname2
pos = T
```

Invalid Input: -

Variable names should start with “small case character” & cannot contain integer. The maximum size of variable name can be 3.

```
Enter : string str1;

--> VALID TOKENS <--
< Keyword , string>
 < StringValue , str1 >
        ↑       ↑

--> SYNTAX ERROR <--
Check your last Token generated to Detect Error
```

Char value should be of length 1.

```
 Enter : char c = 'Mahesh';

 --> VALID TOKENS <--
 < Keyword , char>
  < ID , c >
  < Assop , = >
   < Single Quote , ' >
 < StringValue , Mahesh >
     ↑           ↑

 --> SYNTAX ERROR <--
 Check your last Token generated to Detect Error
```

Return type of position operator should be ‘int’ datatype.

```
Enter : string str = abc ? pqr;

--> VALID TOKENS <--
< Keyword , string>
 < ID , str >
 < Assop , = >
 < ID , abc >
 < Relop , ? >
     ↑     ↑

--> SYNTAX ERROR <--
Check your last Token generated to Detect Error
```

Assignment operator is missing.

```
Enter : string str "Mahesh";

--> VALID TOKENS <--
< Keyword , string>
 < ID , str >
 < Double Quote , " >
     ↑           ↑

--> SYNTAX ERROR <--
Check your last Token generated to Detect Error
```

Declaration of variable without any Datatype.

```
 Enter : str;
 < ID , str >
   ↑     ↑

 --> SYNTAX ERROR <--
 Check your last Token generated to Detect Error
```

4. CONCLUSION

As an IT student we have learned and worded with many programming languages, but we have hardly thought the process that is running behind the picture. There is very complex logic that is implemented to make this language that we are using right now. But the subject of Language Translator makes us understand this complex mechanism.

By implementing this project, we have learnt the lexical, syntax, semantic, intermediate code generator, code optimizer and target code generation phases of compiler design.