Grammar For the Language lgrp23

```
strtP->GlobalDecl
        |FuncDefn
        |MainFunc
        |EndP %end of program
        |Err_empty

GlobalDecl->( Glb VarDecl | Glb PointerDecl |Glb ArrayDecl | Glb TypeDecl )*
VarDecl->Type ID [=Initvar] (',' ID [=Initvar])* ';'
Initvar-> Number|H|Boolean
Number is [0-9]+
H (hex number) is '0x'[a-fA-F0-9]+
Boolean is {0,1}
PointerDecl->Type '*' ID [=Initialize] (',' '*' ID [=Initialize])* ';'
Initialize->'NULL'
        | ['&'] ID
        | H
        | ['*'] ID
//No dynamic allocation currently supported
//No more than 2d array supported
ArrayDecl-> Type ID '[' Number ']' [ '[' Number ']' ] [=ArrayInit] ';'
ArrayInit-> '{' ['{'] Initvar ( ',' Initvar)* ['}'] ( ','  ['{'] Initvar ( ',' Initvar)* ['}'] )* '}'
TypeDecl-> 'typedef' Type ID ';'
Glb->'GLOBAL'
EndP->'EOF'
Err_empty->'EMPTY_FILE'

Types
Type->Prefix TypeId
Prefix->'const'
TypeId->'int'|'bool'

//No command-line argument supported
Functions
MainFunc->int main '(' ')' BlockStmt
FuncDefn->Type ['*'] ID '(' Type ['*'] ['*'] ID (',' Type ['*'] ['*'] ID)* ')' BlockStmt
BlockStmt-> '{'LocalDecl Stmt* '}'
LocalDecl->(VarDecl | PointerDecl | Typedecl)*
Stmt->BlockStmt
        | ';'
        |Exp
        |For
        |DoWhile
        |IfStmt
        |ReturnStmt

For-> 'for' '(' Exp ';' Exp ';' Exp ')' Stmt
IfStmt-> Mif | Uif
Mif->'if' '(' Exp ')' Mif [ 'else' Mif ]
Uif-> 'if' '(' Exp ')' Exp ';'
        | 'if' '(' Exp ')' Mif 'else' Uif
```

DoWhile->'do' Stmt 'while' '(' Exp ')' ';'
ReturnStmt-> 'return' [Exp] ';'

Exp-> ID
| NAT /****Not a token*****/
| '(' Exp ')'
|Exp Assign Exp
| Unaryop Exp
| Exp Binaryop Exp
| Exp '(' Args ')'
| 'true'
| 'false'

Args-> [Exp (',' Exp)*]
Assign-> '='
Unaryop-> '-' | '!' | '~' | '&' | '*'
Binaryop-> '+'| '-'| '*' | '/' |'::'|'='|','

Precedence
associativity and precedence listed from highest to lowest
left ()
left []
left ::
right !
right unary -
right &
right *
left */
left +-
right =
left ,

Operators
() Parenthesis alter the evaluation order
[] Array subscrit
:: Scope resolution
! Logical negation(NOT)
~ Bitwise (1's complement)
- Unary Minus
& Address
* Indirection
* Integer multiplication
/ Integer division
+ Integer addition
- Integer subtraction
= simple assignment
, evaluate

Identifiers(ID)
The valid identifier names are described by the following regular expression:
[a-zA-Z_]([a-zA-Z0-9_])*
Examples

Valid Identifiers:
a,C,b2_,d3
Invalid Identifiers:
1, 2b, 3C, 5d6
//Recursive function is future enhancement

```
/*********Tokenizer**********/
%{
#include<stdio.h>
%}

%%
"if"|"else"|"for"|"do"|"while"|"const"|"GLOBAL"|"int"|"bool"|"typedef"|"main"|"EMPTY_FILE"|"tr
ue"|"false"|"return"|"print"|"scan" printf("%s is a reserved keyword\n",yytext);
"switch"|"case"|"continue"|"break"|"void" printf("%s is reserved keywords for future
usage\n",yytext);
[a-zA-Z_][a-zA-Z0-9_]* printf("%s is an identifier\n",yytext);
"+"|"-"|"*"|"/" printf("%s is a binaryop\n",yytext);
"!" printf("%s is a logical negation\n",yytext);
"%d" printf("%s is the format specifier for both integer and boolean\n",yytext);
[0-9]+  printf("%s is an int(integer) data\n",yytext);
"0x"[a-fA-F0-9]+ printf("%s is a hex number\n",yytext);
"&" printf("%s is a reference operator\n",yytext);
%"*" printf("%s is a dereference operator\n",yytext);
"," printf("%s is a comma-separator\n",yytext);
"("  printf("%s is the left parenthesis\n",yytext);
")"  printf("%s is the right parenthesis\n",yytext);
"{"  printf("%s is the left curly brace(start of BlockStmt)\n",yytext);
"}"  printf("%s is the right curly brace(end of BlockStmt)\n",yytext);
"\"" printf("%s is the double quote\n",yytext);
"'" printf("%s is the single quote\n",yytext);
";"  printf("%s is the semicolon(end of expression)\n",yytext);
"\n\t" printf("%s is a whitespace\n",yytext);
" " printf("%s is a whitespace\n",yytext);
"\\"  printf("%s is a escape character\n",yytext);
"::" printf("%s is a scope-resolution operator",yytext);
"~"  printf("%s is a bitwise (1's) complement operator",yytext);
. printf("%s is a NAT(not a token)\n",yytext);
%%
main(){
yylex();
}
```