What is context free grammar?

= A Context Free Grammar is a formal set of rules used to describe the structure of sentences in a language (particularly programming language).

A CFG is defined using 4-tuples

$$G = (V, T, P, S)$$

where,

V → A finite set of variables (non-terminals)
T → A finite set of terminals
P → production Rule

Applications

↳ used in automata theory for corresponding to Pushdown Automata (PDA).
↳ Defining syntax of programming languages
↳ Used in parsers and syntax analyzers

Maniruzzaman
13/10/25

Design a context-free grammar (CFG) that can correctly parse following arithmetic equations.

i) a+b*c

ii) a/b+c*d

```
%token ID ADD MUL

%%
expr
     : ID ADD ID MUL ID
     ;
%%
```

one way but precedence missing

```
%token ID ADD MUL
%left ADD
%left MUL
%%
expr
     : expr ADD term
     | term
     ;
term
     : term MUL factor
     | factor
     ;
factor
     : ID
     ;
%%
```

by maintaining precedence (corrected)

Design a CFG for given input

a / b + c * d

<u>Ans:</u>

```
%token ID ADD MUL DIV
%left ADD
%left MUL DIV

%%
expr
      : expr ADD term
      | term
      ;
term
      : term MUL factor
      | term DIV factor
      | factor
      ;
factor
      : ID
      ;
%%
```

Design a CFG that can perform all kinds of operations using the following operators:

+, -, *, /, %, &, I, !, ~, ^

such that the grammar maintain the correct operator precedence & associativity.

```
%token ID NUM
%token ADD SUB MUL DIV MOD AND OR NOT BNOT XOR
%left OR
%left XOR
%left AND
%left ADD SUB
%left MUL DIV MOD
%right NOT BNOT
%%
expr : expr OR expr | expr XOR expr | expr AND expr
     | expr ADD expr | expr SUB expr | expr MUL expr
     | expr DIV expr | expr MOD expr | NOT expr
     | BNOT expr | '(' expr ')' | ID | NUM
     ;
%%
```

For given input

```
int i = 10;
int j = 20;
float a = 20.2;
```

Ans:

```
%token INT FLOAT ID NUM REAL ASSIGN
%token SEMI

%%
program : decl_list
        ;

decl_list : decl_list decl
          | decl
          ;

decl : Type ID ASSIGN value SEMI
     ;

Type : INT | FLOAT
     ;

value : NUM
      | REAL
      ;
%%
```

For given input

```
int a;
int m, n = 20;
double k, p = 10.99;
int b = 20;
float c = 100.10;
```

Ans

```
%token INT FLOAT DOUBLE
%token ID
%token NUM FREAL DREAL
%token ASSIGN SEMI COMMA

%%
program
        : decl_list
        ;

decllist : decl_list decl
         | decl
         ;

decl : type var_list SEMI
     ;

type : INT | FLOAT | DOUBLE

var_list : var_list COMMA var
         | var

var
    : ID
    | ID ASSIGN value
    ;

value : NUM
      | DREAL
      | FREAL
      ;
%%
```

For Given Input

```
int a=0;
int b=20;
if(a==0) a=10;
if(a>b){a=a-b;}
else {b=b+a;}
```

Ans:

```
%token INT IF ELSE ID NUM ASSIGN EQ GT
%token LP RP LB RB SEMI
%%
program
        : declList ifstmt
        ;
declList
        : declList decl
        | decl
        ;
decl
        : INT ID ASSIGN NUM SEMI
        ;
ifstmt
        : IF LP condition RP stmt
        | IF LP condition RP stmt ELSE stmt
        ;
condition
        : ID EQ NUM
        | ID GT ID
        ;
stmt : Assign_stmt | block ;
Assign_stmt : ID ASSIGN
```

```
stmt
    : assign_stmt
    | block
    ;
assign_stmt
    : ID ASSIGN expr SEMI
    ;
block
    : LB stmt_list RB
    ;
stmt_list
    : stmt_list stmt
    | stmt
    ;
expr
    : ID
    | ID ADD ID
    | ID SUB ID
    | NUM
    ;
%%
```

Another way
↳

```
condition
        : ID EQ NUM | ID GT ID
        ;

stmt
      : ID ASSIGN expr SEMI
      | LB ID ASSIGN expr SEMI RB
      ;

expr
     : ID
     | NUM | ID ADD ID | ID SUB ID
     ;
```

For Given Code:

```
int number = 5;
int i = 1;
if (number > 0) {
    printf("The number is positive.\n");
    }
else {
    printf ("The number is not positive.\n");
    }
while (i < number) {
        printf ("count: %d \n", i);
        i++;
    }
```

Ans:

```
%token INT IF ELSE WHILE ID NUM ASSIGN GT
%token LE LP RP LB RB SEMI INC PRINTF STR
%%
program :
        | dec_list if_stmt while_stmt
        ;

dec_list
        : dec_list dec
        | dec
        ;
dec : INT ID ASSIGN NUM SEMI
    ;
```

```
ifstmt : IF LP cond RP block ELSE block
       ;
while_stmt
        : WHILE LP cond RP block
        ;
cond
    : ID GT NUM
    | ID LE ID
    ;
block
    : LB stmt_list RB
    ;
stmt_list
        : printf_stmt
        | inc_stmt
printf_stmt
        : PRINTF LP STR COMMA RP SEMI
        | PRINTF LP STR RP SEMI
        ;
inc_stmt
        : ID INC SEMI
        ;
%%
```