

RAPPORT COMPILATION

**Utilisation de LEX et YACC pour l'évaluation et génération d'une
forme intermédiaire d'une expression arithmétique**

Mohamed Handaoui 2CS SIL

ECOLE NATIONALE SUPERIEURE D'INFORMATIQUE 2017-2018

Table des matières

Partie 1 :	2
Analyse lexicale :	2
Analyse syntaxique :	3
Exemples :	4
Exemples (erreurs) :	4
Partie 2 :	5
Analyse lexicale :	5
Analyse syntaxique :	6
si(test, expression1, expression2) :	6
moyenne(expression1, expression2, expression3, ...):	7
somme(expression1, expression2, expression3, ...):	7
produit(expression1, expression2, expression3, ...):	7
variance(expression1, expression2, expression3, ...):	7
ecart-type(expression1, expression2, expression3, ...):	8
min(expression1, expression2, expression3, ...):	8
max(expression1, expression2, expression3, ...):	8
Exemples :	8
Exemples (erreurs) :	9
Partie 3 :	10
Analyse lexicale (addition) :	10
Les routines :	10
Les expressions arithmétiques :	12
Les fonctions somme, moyenne, produit :	13
Fonction variance :	14
Fonction écart-type :	16
Fonction min, max :	17
Fonction Si (Test, expr1, expr2) :	19
Exemples :	21

Partie 1 :

Dans cette partie du projet, il est demandé de développer un analyseur syntaxique d'une expression arithmétique en utilisant LEX et YACC. Les opérandes sont des nombres entiers ou flottants. Les erreurs syntaxiques et lexicales devront être signalées par des messages significatifs. La reprise sur erreur est facultative dans cette partie du projet. L'expression en entrée, est lue directement à partir de la ligne de commande ou à partir d'un fichier d'entrée.

Analyse lexicale :

Le code suivant analyse lexicalement une expression arithmétique en utilisant LEX :

```
number [0-9]+(\.[0-9]+)?
spaces [ \t]+
operator [\+ \- \* /\^]
end-of-input [\r\nEOF]
%option yylineno
%%
/* start */
{number} { moveCursor(); yylval.number = atof(yytext); return NUMBER; }

/* escaping spaces */
{spaces}+ { moveCursor(); }

/* handling parenthesis */
"(" { moveCursor(); parenthesis++; return yytext[0]; }

")" { moveCursor(); parenthesis--; return yytext[0]; }

{end-of-input} {
    verifyParenthesisCount();
    moveCursor();
    return EOI;
}

{operator} {
    moveCursor();
    return yytext[0];
}

. {
    moveCursor();
    yyerror("Unexpected character");
}
%%

void moveCursor(){
    cursor += yyleng;
}
```

A chaque fois qu'on lit une entrée et qu'elle soit reconnus, on incrémente la position du curseur avec la taille de la chaîne de caractère (yyleng), ce dernier va aider lors de l'affichage d'une erreur sur la console pour afficher la position de l'erreur. On échappe tous les espaces et tabulation dans une expression.

Si on lit le un nombre, il sera converti vers un nombre réel et on retour un « Token » qui sera utiliser par la suite.

Pour les parenthèses, si on trouve une parenthèse ouvrante on incrémente le nombre des parenthèses ouvertes sinon si on trouve une parenthèse fermante on décrémente le nombre. A la rencontre d'un caractère de fin de ligne (retour à la ligne, retour chariot, fin de fichier) on vérifie le nombre des parenthèses si c'est nul alors soit on n'a pas de parenthèses soit toutes parenthèses ouvertes a été fermées sinon on afficher un message d'erreur.

Les opérateurs des expressions arithmétique sont l'addition, soustraction, multiplication, division, le moins unaire et la puissance.

Analyse syntaxique :

Le code suivant analyse syntaxiquement une expression arithmétique en utilisant YACC :

```
%{
    .....
    #include "error.strings.h"
    .....
}%
%union { double number; }
/* Tokens */
%token <number> NUMBER EOI

/* precedence */
%left '-' '+'
%left '*' '/'
%right '^'
%right unary_minus

/* starting point */
%start Input
%%
Input:
    | Input Line { cursor = 0; }
    ;

Line: EOI
    | Expr EOI { printf("%lf \n", $1); }
    ;

Expr: NUMBER      { $$ = $1; }
    | Expr '-' Expr { $$ = $1 - $3; }
    | Expr '+' Expr { $$ = $1 + $3; }
    | Expr '*' Expr { $$ = $1 * $3; }
```

```

| Expr '/' Expr {
    if($3 == 0) yyerror("division by zero");
    $$ = $1 / $3;
}
| Expr '^' Expr { $$ = pow($1, $3); }
| '-' Expr %prec unary_minus { $$ = -$2; }
| '(' Expr ')' { $$ = $2; }

```

On utilise dans ce programme une grammaire ambiguë, mais il est indispensable de déclarer les règles de précédences ou les priorités pour corriger cette ambiguïté.

Pour la division il faut vérifier si on divise par zero, dans ce cas on affiche une erreur.

```

/* gestion des erreurs arithmétiques */
| '(' error { yyerror(EXPRESSION_EXPECTED); }
| '(' error ')' { yyerror(EXPRESSION_EXPECTED); }
| '(' Expr error { yyerror(CLOSING_PARENTHESIS_EXPECTED); }
| Expr '+' error { yyerror(EXPRESSION_EXPECTED); }
| Expr '-' error { yyerror(EXPRESSION_EXPECTED); }
| Expr '*' error { yyerror(EXPRESSION_EXPECTED); }
| Expr '/' error { yyerror(EXPRESSION_EXPECTED); }
| Expr '^' error { yyerror(EXPRESSION_EXPECTED); }
;

```

Pour la gestion des erreurs arithmétique, on utilise le mot réservé par yacc « error » pour savoir exactement quelle erreur est souvenu et a quelle position relative à une certaines constantes

Exemples :

```

handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
2*(5+8--6/8^2)
26.187500

```

On a l'expression : $2*(5+8--6/2^2) = 26.187500$

On remarque la succession des négatives, et selon la grammaire elle est acceptée donc $--6 = 6$

Exemples (erreurs) :

```

handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
5++
Error: Expression expected after operator at position 3

handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
5+(2*8_
Error: Closing parenthesis expected after expression at position 7

handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
5+(
Error: Expression expected after operator at position 4

```

Partie 2 :

Dans cette partie du projet, il s'agit de compléter le travail effectué dans la partie 1 pour évaluer l'expression arithmétique donnée en entrée si cette dernière est syntaxiquement correcte. Des fonctions supplémentaires souvent utilisées dans les évaluateurs d'expressions seront inclus dans l'évaluateur.

Ces fonctions Sont :

- somme(expression1, expression2, expression3, ...)
- produit(expression1, expression2, expression3, ...)
- moyenne(expression1, expression2, expression3, ...)
- variance(expression1, expression2, expression3, ...)
- ecart-type(expression1, expression2, expression3, ...)
- min(expression1, expression2, expression3, ...)
- max(expression1, expression2, expression3, ...)
- si(test, expression1, expression2)

Analyse lexicale :

```
.....
CMP [><=]
.....
%%
.....
"moyenne" { moveCursor(); return AVERAGE; }

"somme" { moveCursor(); return SUM; }

"produit" {moveCursor(); return PRODUCT; }

"variance" { moveCursor(); return VARIANCE; }

"ecart-type" { moveCursor(); return STANDARD_DEVIATION; }

"min" { moveCursor(); return MIN; }

"max" { moveCursor(); return MAX; }

"si" { moveCursor(); return IF; }
.....

",","|";" { moveCursor(); return yytext[0]; }

{CMP} { moveCursor(); return yytext[0]; }
```

Dans ce code on reconnaît les fonctions qui sont acceptées par notre évaluateur, quand on reconnaît une fonction on retourne un Token qui sera utilisé dans YACC comme un non-terminal.

Pour les séparateurs « , » et « ; » et les opérateurs de comparaison « < » « > » et « = » on retourne le même caractère comme token car c'est plus lisible dans le programme YACC.

Analyse syntaxique :

```
.....
%union {
    struct list{
        double value;
        double sqr_value;
        int size;
    } list;
    double number;
    int function;
}

/* Tokens */
%token <number> NUMBER EOI
%token <number> AVERAGE SUM PRODUCT VARIANCE STANDARD_DEVIATION MIN MAX IF
/* precedence */
.....
/* Types definitions */
%type <number> Test Expr Line Function
%type <list> AVERAGE_List PRODUCT_List MIN_List MAX_List VARIANCE_List
.....
%%
.....
Expr: NUMBER { $$ = $1; }
    | Function { $$ = $1; }
    | IF '(' Test ';' Expr ';' Expr ')' { $$ = $3 ? $5 : $7; }
.....
```

Dans la production de l'expression on ajoute deux règles pour le calcul des fonctions, et une autre pour la fonction du test.

Dans la méthode de calcul de la valeur d'une fonction, on calcule au fur et à mesure d'une rencontre avec un nouveau paramètre dans la fonction, donc on garde que la dernière valeur calculée, on garde aussi le nombre de paramètres si c'est nécessaire.

si(test, expression1, expression2) :

```
Test: Expr '>' Expr { $$ = $1 > $3; }
    | Expr '<' Expr { $$ = $1 < $3; }
    | Expr '=' Expr { $$ = $1 == $3; }
    ;
```

On affecte au Terminal la valeur de la comparaison donc soit 0 ou 1, ensuite on compare avec une opération ternaire « ? » si c'est égal à 1 on retourne la valeur de la première expression sinon la deuxième.

moyenne(expression1, expression2, expression3, ...)

```
Function: AVERAGE '(' AVERAGE_List ')' { $$ = $3.value / $3.size; }
.....

AVERAGE_List: AVERAGE_List ',' Expr
               { $$ .value = $1.value + $3; $$ .size = $1.size + 1; }
               | Expr { $$ .value = $1; $$ .size = 1; }
               .....

```

On initialise la première valeur de la liste si on a lu un premier paramètre, la liste contient deux valeurs : le nombre de paramètre qui sera incrémenter à chaque lecture de ce dernier, et la valeur calculée.

A la fin on divise la somme sur le nombre de paramètre.

somme(expression1, expression2, expression3, ...)

```
Function: .....
         | SUM '(' AVERAGE_List ')' { $$ = $3.value; }
         .....

```

La fonctions somme utilise la même liste que celle de la moyenne car on somme seulement les paramètres, donc à la fin on affecte le résultat a la valeur de la fonction.

produit(expression1, expression2, expression3, ...)

```
Function: .....
         | PRODUCT '(' PRODUCT_List ')' { $$ = $3.value; }
         .....

PRODUCT_List: PRODUCT_List ',' Expr { $$ .value = $1.value * $3; }
              | Expr { $$ .value = $1; }
              .....

```

On initialise le résultat avec la valeur du premier paramètre. A la lecture d'un nouveau paramètre on multiplie ce dernier avec le résultat précédant.

variance(expression1, expression2, expression3, ...)

```
Function: .....
         | VARIANCE '(' VARIANCE_List ')'
           { $$ = ($3.sqr_value / $3.size) - pow($3.value / $3.size,2); }
         .....

VARIANCE_List: VARIANCE_List ',' Expr {
               $$ .value = $1.value + $3;
               $$ .sqr_value = $1.sqr_value + pow($3,2);
               $$ .size = $1.size + 1;
               }
               | Expr { $$ .value = $1; $$ .sqr_value = pow($1,2); $$ .size = 1; }
               .....

```

Pour calculer la variance on utilise l'expression suivante : $\text{Var}(X) = E[X^2] - E[X]^2$, car on ne garde pas la liste des paramètres.

Donc on doit calculer les deux moyennes simultanément.

ecart-type(expression1, expression2, expression3, ...)

```
Function: .....
| STANDARD_DEVIATION '(' VARIANCE_List ')'
| { $$ = sqrt(($3.sqr_value / $3.size) - pow($3.value / $3.size,2)); }
```

Pour l'écart-type on calcule la variance de la même manière ensuite on calcule la racine carrée de la variance.

min(expression1, expression2, expression3, ...)

```
Function: .....
| MIN '(' MIN_List ')' { $$ = $3.value; }
| .....

MIN_List: MIN_List ',' Expr { if ( $3 < $1.value) $$ .value = $3; }
| Expr { $$ .value = $1; }
| .....
```

On initialise la valeur à la rencontre du premier paramètre, ensuite on compare le nouveau paramètre avec le résultat précédant, si c'est inférieur alors le résultat prend la valeur du nouveau paramètre.

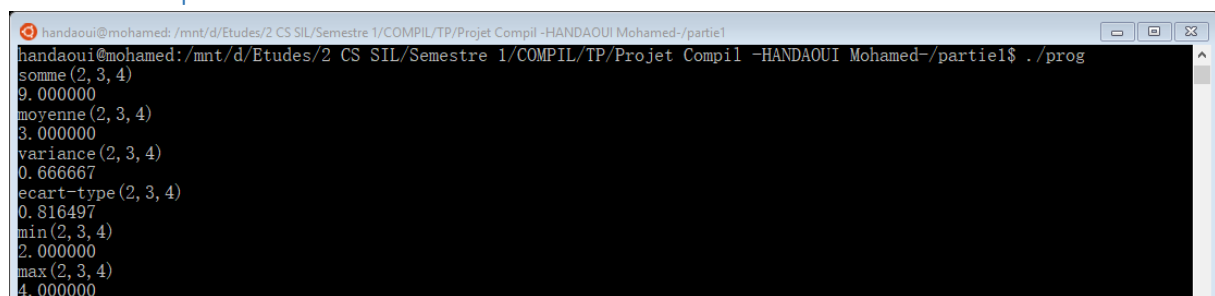
max(expression1, expression2, expression3, ...)

```
Function: .....
| MAX '(' MAX_List ')' { $$ = $3.value; }
| .....

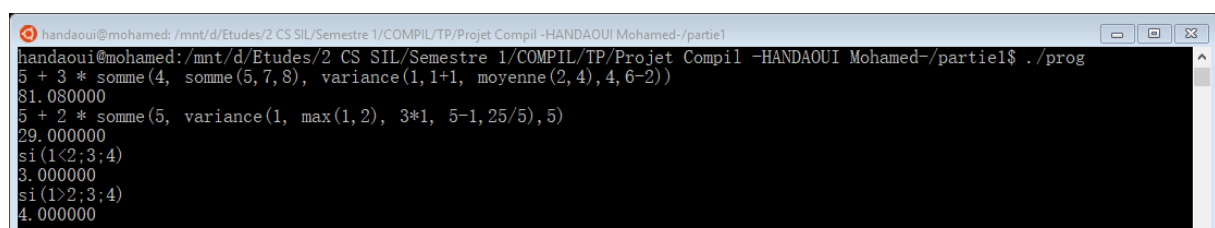
MAX_List: MAX_List ',' Expr { if ( $3 > $1.value) $$ .value = $3; }
| Expr { $$ .value = $1; }
| .....
```

On initialise la valeur à la rencontre du premier paramètre, ensuite on compare le nouveau paramètre avec le résultat précédant, si c'est supérieur alors le résultat prend la valeur du nouveau paramètre.

Exemples :



```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
somme(2,3,4)
9.000000
moyenne(2,3,4)
3.000000
variance(2,3,4)
0.666667
ecart-type(2,3,4)
0.816497
min(2,3,4)
2.000000
max(2,3,4)
4.000000
```



```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
5 + 3 * somme(4, somme(5,7,8), variance(1,1+1, moyenne(2,4),4,6-2))
81.080000
5 + 2 * somme(5, variance(1, max(1,2), 3*1, 5-1,25/5),5)
29.000000
si(1<2;3;4)
3.000000
si(1>2;3;4)
4.000000
```

Exemples (erreurs) :

```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1
handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
5*(2*(moyenne(1,2,3)+5))_
Error: Closing parenthesis expected after expression at position 24

handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
5*(2*(moyenne(1,2,3)+))
Error: Expression expected after operator at position 22

handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
5*(2*(moyenne(1,2,3)+5))
Error: Comma expected after expression at position 19

handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
5*2*(moyenne(1,2,3)+)
Error: Opening parenthesis expected after function name at position 23

handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie1$ ./prog
5*(2*(moyenn(1,2,3)+5))
Error: Unknown function call character at position 13
```

Contenu du fichier « error.strings.h »

```
const char *EXPRESSION_EXPECTED =
    {"Expression expected after operator"};
const char *CLOSING_PARENTHESIS_EXPECTED =
    {"Closing parenthesis expected after expression"};
const char *FUNCTION_OPENING_PARENTHESIS_EXPECTED =
    {"Opening parenthesis expected after function name"};
const char *FUNCTION_PARAMS_EXPECTED =
    {"Expression expected after function opening parenthesis"};
const char *OPERATOR_EXPECTED = {"Operator expected after expression"};
const char *COMMA_EXPECTED = {"Comma expected after expression"};
```

Partie 3 :

Dans cette partie du projet les opérandes ne sont pas connus durant la phase de compilation (exemple : $a*(b+c)-d/e$). Si l'expression en entrée est syntaxiquement correcte, générer une forme intermédiaire sous forme de quadruplets qui une fois exécuté donnera le résultat de l'expression donné au départ.

Analyse lexicale (addition) :

Dans cette partie on doit reconnaître les chiffres et les identificateurs, il faut aussi interdire l'utilisation des variables temporaires « temp ».

```
.....
identifieur [a-z0-9]+
.....
%%
/* start */
"exit" moveCursor(); return EXIT;

"temp"[0-9]+ { moveCursor(); yyerror("Variable name is reserved"); }
.....
{identifieur} moveCursor(); return IDENTIFIER;
.....
```

Les routines :

Pour les routines, on utilise les variables suivantes :

```
char stack[100][10];
```

La variable « stack » est une pseudo-pile qui sera utiliser pour empiler les valeurs des expressions et variables temporaires qui seront utilisées par d'autre expressions

```
char quadStack[100][100];
```

La variable « quadStack » est une pile qui contient les quadruplets générés lors de l'analyse syntaxique, on doit l'utiliser pour mettre à jour les adresses des JUMP et ensuite sauvegarder les quadruplets à la fin de l'analyse.

```
int top = 0;
```

La variable « top » représente le numéro du quadruplet courant, nommée « top » pour simplement représente la tête de la pile.

```
int tempNumber = 0;
```

La variable représente le numéro courant de la variable temporaire utilisée, exemple « temp6 » dans tempNumber == 6 ;

```
char temp[10] = "";
char result[100] = "";
```

La variable « temp » est utilisé pour contenir la variable temporaire courante, et « result » contient la valeur du quadruplet courant, ces deux variables sont utilisées seulement pour ne pas allouer du mémoire à chaque utilisation.

```

Input:
| Input Line { cursor = 0; strcpy(line,""); }
| EXIT { closeFiles(); }
;

Line: EOI
| EXIT { closeFiles(); }
| Expr EOI { printQuadruplets(); }
;

```

A la fin de la génération du code intermédiaire et à la fin de la chaîne d'entrée on affiche les quadruplets et on réinitialise des variables comme le curseur.

```

void printQuadruplets()
{
    int i;
    if (fileIsOpen)
    {
        printf("%s\n", line);
    }
    for (i = 1; i <= lineNumber; i++)
    {
        fprintf(yyout, "%03d  %s\n", i, quadStack[i]);
    }
    if (lineNumber == 0 && strstr(stack[top], "temp") == NULL)
    {
        fprintf(yyout, "%03d  temp0 = %s\n", ++lineNumber, stack[top]);
    }
    fprintf(yyout, "%03d  END\n\n", lineNumber + 1);
    lineNumber = 0;
    tempNumber = 0;
}

```

Cette fonction affiche dans la console une sortie de programme n'est spécifiée, sinon le résultat sera sauvegardé dans un fichier en exécutant la commande avec les paramètres suivants :

```
$ ./prog -o output.txt
```

La fonction suivante ajoute le quadruplet courant dans la pile des quadruplets et incrémente le numéro de la ligne.

```
void generateQuadruplet() { sprintf(quadStack[++lineNumber], "%s", result); }
```

La fonction suivante empile la valeur de l'expression analysée, selon la position de la routine contenant cette fonction.

```
void push() { strcpy(stack[++top], yytext); }
```

Les expressions arithmétiques :

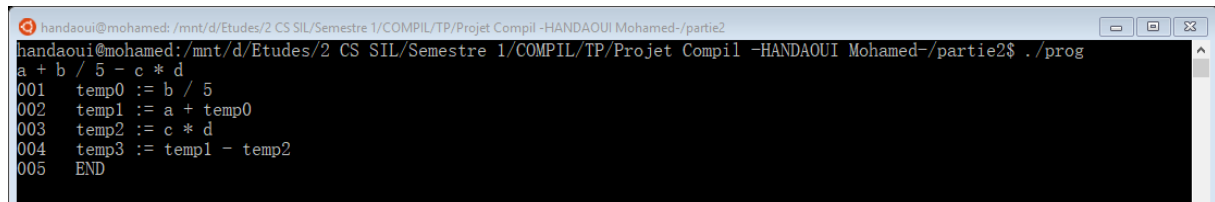
```
Expr: Expr '-' { push(); } Expr { generateArithmeticQuadruplet(); }
| Expr '+' { push(); } Expr { generateArithmeticQuadruplet(); }
| Expr '/' { push(); } Expr { generateArithmeticQuadruplet(); }
| Expr '*' { push(); } Expr { generateArithmeticQuadruplet(); }
| Expr '^' { push(); } Expr { generatePowQuadruplet(); }
| '-' { push(); } Expr %prec unary_minus
                        { generateQuadrupletUnaryMinus(); }
| '(' Expr ')' {}
| IDENTIFIER { push(); }
.....
```

Pour les opérations « +, -, *, / » on la même routine pour générer les quadruplets :

```
void generateArithmeticQuadruplet()
{
    sprintf(temp, "temp%d", tempNumber++);
    sprintf(result, "%s := %s %s %s", temp, stack[top - 2], stack[top - 1],
stack[top]);
    generateQuadruplet();
    top -= 2;
    strcpy(stack[top], temp);
}
```

Top -= 2 ça représente le dépilement de la pile de deux éléments. Et la dernière ligne en empile la dernière variable temporaire utilisée.

Exemple : $a + b / 5 - c * d$

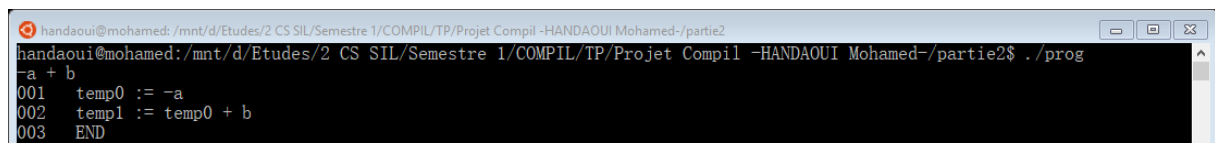


```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
a + b / 5 - c * d
001 temp0 := b / 5
002 temp1 := a + temp0
003 temp2 := c * d
004 temp3 := temp1 - temp2
005 END
```

Pour le moins unaire on a un seul opérande donc :

```
void generateQuadrupletUnaryMinus()
{
    sprintf(temp, "temp%d", tempNumber++);
    sprintf(result, "%s := -%s", temp, stack[top]);
    generateQuadruplet();
    top--;
    strcpy(stack[top], temp);
}
```

Exemple : $-a + b$



```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
-a + b
001 temp0 := -a
002 temp1 := temp0 + b
003 END
```

Pour la puissance on suit l'algorithme suivant :

- 1- entier résultat = 1 ;
- 2- si (b == 0) alors aller à 6
- 3- résultat = résultat * a
- 4- b = b -1
- 5- si (b != 0) aller à 3
- 6- FIN

Exemple : a^b

```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
a b
001 temp0 := b
002 temp1 := 1
003 CMP b
004 JZ 8
005 temp1 := temp1 * a
006 temp0 := temp0 - 1
007 JNZ 5
008 END
```

Les fonctions somme, moyenne, produit :

Les routines de la somme sont la même que celle de la moyenne sauf la division sur la taille des paramètre, or la routine du produit est différente de celle de la somme seulement dans l'opération de production.

Dans la suite on va détaillera les routines de la moyenne :

```
SUM_List: SUM_List { push(); } ',' Expr { generateSumQuadruplet(); $$++; }
          | Expr { $$ = 1;}
```

On commence par générer les quadruplets de la somme, et à chaque nouveau paramètre on incrémente le nombre.

```
void generateSumQuadruplet()
{
    sprintf(temp, "temp%d", tempNumber++);
    sprintf(result, "%s := %s + %s", temp, stack[top - 2], stack[top]);
    generateQuadruplet();
    top -= 2;
    strcpy(stack[top], temp);
}
```

Cette routine est suffisante pour générer les quadruplets de la somme, exemple : somme(a,b,5,g)

```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
somme(a, b, 5, g)
001 temp0 := a + b
002 temp1 := temp0 + 5
003 temp2 := temp1 + g
004 END
```

Ensuite pour la moyenne on divise sur le nombre de parametre :

```
Function: SUM '(' SUM_List ')'
          | AVERAGE '(' SUM_List ')' { generateAverageQuadruplet($3); }
```

```

void generateAverageQuadruplet(int size)
{
    sprintf(temp, "temp%d", tempNumber++);
    sprintf(result, "%s := %s / %d", temp, stack[top], size);
    generateQuadruplet();
    strcpy(stack[top], temp);
}

```

Exemple : moyenne(a,b,5,g)

```

handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
moyenne(a,b,5,g)
001 temp0 := a + b
002 temp1 := temp0 + 5
003 temp2 := temp1 + g
004 temp3 := temp2 / 4
005 END

```

Exemple de la fonction produit : produit(a,b,5,g)

```

handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
produit(a,b,5,g)
001 temp0 := a * b
002 temp1 := temp0 * 5
003 temp2 := temp1 * g
004 END

```

Fonction variance :

Les routines de la variance sont les même que l'écart type a l'exception de la dernière routine ou la fonction de l'écart type calcule la racine carrée de la variance.

```

Function: .....
| VARIANCE '(' VARIANCE_List ')' { generateVarianceQuadruplet($3); }
| STANDARD_DEVIATION '(' VARIANCE_List ')'
| { generateVarianceQuadruplet($3); generateDeviationQuadruplet($3); }

```

```

VARIANCE_List: VARIANCE_List { push(); } ',' Expr
                { generatePreVarianceQuadruplet($$++); }
| Expr { generateInitVarianceQuadruplet(); $$ = 1; }

```

Dans une première étape on doit initialiser notre algorithme lors du premier paramètre de la fonction

```

void generateInitVarianceQuadruplet()
{
    sprintf(temp, "temp%d", tempNumber++);
    sprintf(result, "%s := %s * %s", temp, stack[top], stack[top]);
    generateQuadruplet();
    sprintf(temp, "temp%d", tempNumber++);
    sprintf(result, "%s := %s", temp, stack[top]);
    generateQuadruplet();
    top--;
}

```

Si on a `variance(a,b)`, la fonction prétendante va générer le code suivant :

```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2
handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
variance(a,b)
001  temp0 := a * a
002  temp1 := a
```

De cette manière on connaîtra par la suite les quelles des variables temporaires utiliser, donc minimiser le nombre de ces derniers.

```
void generatePreVarianceQuadruplet(int size)
{
    tempNumber++;
    sprintf(temp, "temp%d", tempNumber - size);
    sprintf(result, "%s := %s * %s", temp, stack[top], stack[top]);
    generateQuadruplet();
    char prevTemp[10] = "";
    sprintf(prevTemp, "temp%d", tempNumber - (size + 1));
    sprintf(result, "%s := %s + %s", prevTemp, prevTemp, stack[top]);
    generateQuadruplet();
    sprintf(prevTemp, "temp%d", tempNumber - (size + 2));
    sprintf(result, "%s := %s + %s", prevTemp, prevTemp, temp);
    generateQuadruplet();
    top--;
}
```

Ensuite on génère les quadruplets à chaque lecture d'un nouveau paramètre en réutilisant des variables temporaires utilisées dans l'initialisation, exemple :

```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2
handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
variance(a,b)
001  temp0 := a * a
002  temp1 := a
003  temp2 := b * b
004  temp1 := temp1 + b
005  temp0 := temp0 + temp2
```

Et à la fin on divise sur le nombre des paramètres et on soustrait les deux moyennes,

$$\text{Var}(X) = E[x^2] - E[x]^2$$

```
void generateVarianceQuadruplet(int size)
{
    char prevTemp[10] = "";
    sprintf(prevTemp, "temp%d", tempNumber - size - 1);
    sprintf(result, "%s := %s / %d", prevTemp, prevTemp, size);
    generateQuadruplet();
    char prevTemp2[10] = "";
    sprintf(prevTemp2, "temp%d", tempNumber - size);
    sprintf(result, "%s := %s / %d", prevTemp2, prevTemp2, size);
    generateQuadruplet();
    sprintf(result, "%s := %s * %s", prevTemp2, prevTemp2, prevTemp2);
    generateQuadruplet();
    sprintf(temp, "temp%d", tempNumber++);
}
```

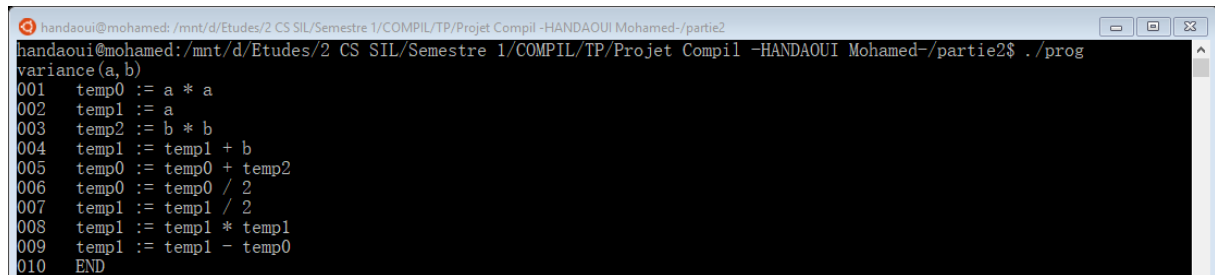


```

    sprintf(result, "%s := %s - %s", prevTemp2, prevTemp2, prevTemp);
    generateQuadruplet();
    tempNumber -= size + 1;
    sprintf(temp, "temp%d", tempNumber);
    top -= size - 2;
    strcpy(stack[top], temp);
}

```

Donc pour l'exemple précédant le résultat final est le suivant :



```

handaoui@mohamed:/mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
variance(a,b)
001 temp0 := a * a
002 temp1 := a
003 temp2 := b * b
004 temp1 := temp1 + b
005 temp0 := temp0 + temp2
006 temp0 := temp0 / 2
007 temp1 := temp1 / 2
008 temp1 := temp1 * temp1
009 temp1 := temp1 - temp0
010 END

```

Fonction écart-type :

Pour l'écart type on applique les mêmes routines, à la fin on doit générer les quadruplets de racine carrée de la variance.

L'algorithme de la racine carrée :

```

void main()
{
    int n;
    float temp, sqrt;
    printf("Enter number:");
    scanf("%d", &n);
    sqrt = n / 2;
    temp = 0;
    while (sqrt != temp)
    {
        temp = sqrt;
        sqrt = (n / temp + temp) / 2;
    }
    printf("the square root of %d is %f\n", n, sqrt);
}

```

Si on traduit cette algorithme on aura le code suivant :

```

void generateDeviationQuadruplet()
{
    char temp1[10] = "";
    sprintf(temp1, "temp%d", tempNumber++); // n
    char temp2[10] = "";
    sprintf(temp, "temp%d", tempNumber++); //sqrt
    sprintf(temp2, "temp%d", tempNumber++); // temp
    sprintf(result, "%s := %s / 2", temp, temp1);
    generateQuadruplet();
}

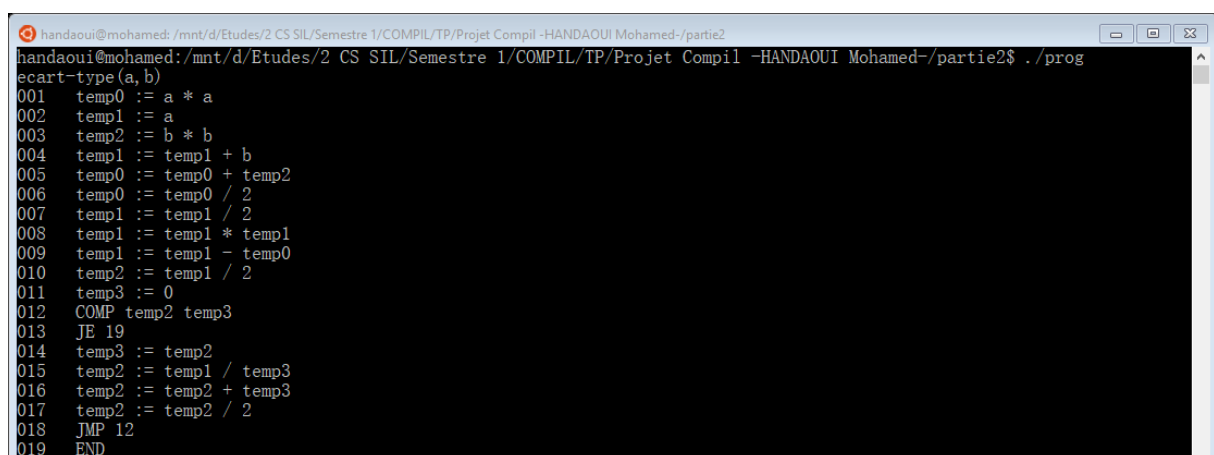
```

```

    sprintf(result, "%s := 0", temp2);
    generateQuadruplet();
    sprintf(result, "COMP %s %s", temp, temp2); // while
    generateQuadruplet();
    sprintf(result, "JE %d", lineNumber + 7);
    generateQuadruplet();
    sprintf(result, "%s := %s", temp2, temp);
    generateQuadruplet();
    sprintf(result, "%s := %s / %s", temp, temp1, temp2);
    generateQuadruplet();
    sprintf(result, "%s := %s + %s", temp, temp, temp2);
    generateQuadruplet();
    sprintf(result, "%s := %s / 2", temp, temp);
    generateQuadruplet();
    sprintf(result, "JMP %d", lineNumber - 5);
    generateQuadruplet();
    strcpy(stack[top], temp);
}

```

Exemple : ecart-type(a,b)



```

handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
ecart-type(a,b)
001  temp0 := a * a
002  temp1 := a
003  temp2 := b * b
004  temp1 := temp1 + b
005  temp0 := temp0 + temp2
006  temp0 := temp0 / 2
007  temp1 := temp1 / 2
008  temp1 := temp1 * temp1
009  temp1 := temp1 - temp0
010  temp2 := temp1 / 2
011  temp3 := 0
012  COMP temp2 temp3
013  JE 19
014  temp3 := temp2
015  temp2 := temp1 / temp3
016  temp2 := temp2 + temp3
017  temp2 := temp2 / 2
018  JMP 12
019  END

```

Fonction min, max :

La routine d'initialisation des fonctions min et max sont identique,

```

MIN_List: MIN_List { push(); } ',' Expr { generatePreMinQuadruplet(); }
          | Expr { generateInitMinMaxQuadruplet();}

```

```

MAX_List: MAX_List { push(); } ',' Expr { generatePreMaxQuadruplet(); }
          | Expr { generateInitMinMaxQuadruplet();}

```

Cette routine vérifie si le premier paramètre est une variable temporaire, si c'est le cas on doit par l'initialiser, sinon on créer une nouvelle variable temporaire avec comme valeur le premier paramètre de la fonction qui sera utiliser par la routine spécifique pour la fonction min ou max.

```

void generateInitMinMaxQuadruplet()
{
    if (strstr(stack[top], "temp") == NULL)
    {
        sprintf(temp, "temp%d", tempNumber++);
        sprintf(result, "%s := %s", temp, stack[top]);
        generateQuadruplet();
        // top--;
        strcpy(stack[top], temp);
    }
}

```

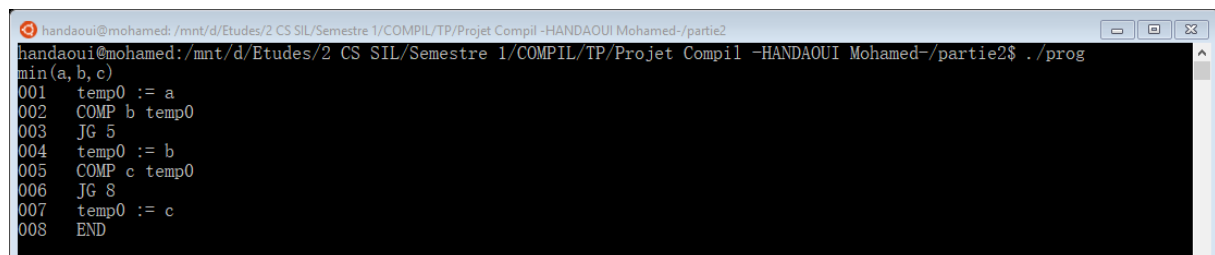
La seule différence entre les fonctions min et max dans la deuxième routine c'est lors de la comparaison on fait un jump si supérieur pour le min (car on réaffecte seulement si on a un nouveau min) et inférieur pour le max.

```

void generatePreMinQuadruplet()
{
    sprintf(result, "COMP %s %s", stack[top], stack[top - 2]);
    generateQuadruplet();
    sprintf(result, "JG %d", lineNumber + 3);
    generateQuadruplet();
    sprintf(result, "%s := %s", stack[top - 2], stack[top]);
    generateQuadruplet();
    top -= 2;
}

```

Exemple : min(a,b,c)

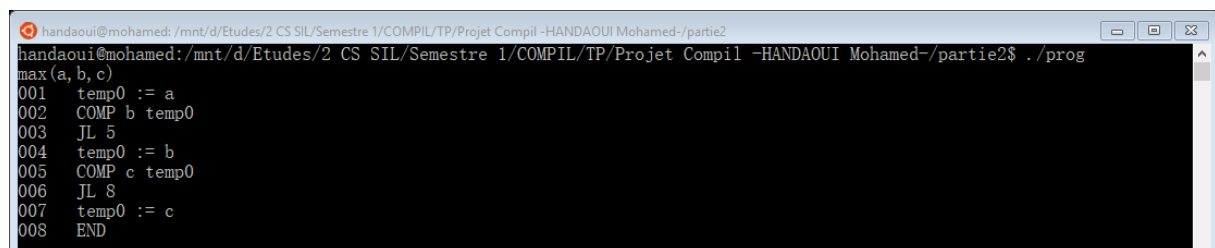


```

handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
min(a, b, c)
001 temp0 := a
002 COMP b temp0
003 JG 5
004 temp0 := b
005 COMP c temp0
006 JG 8
007 temp0 := c
008 END

```

Exemple : max(a,b,c)



```

handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
max(a, b, c)
001 temp0 := a
002 COMP b temp0
003 JL 5
004 temp0 := b
005 COMP c temp0
006 JL 8
007 temp0 := c
008 END

```

Fonction Si (Test, expr1, expr2) :

Cette fonction suit l'algorithme suivant :

Si (test) alors return expr1 sinon expr2

La routine de test contient qu'une seule fonction, or la fonction Si on doit sauvegarder l'adresse du dernier quadruplet pour mettre à jour le jump générer dans la routine du test,

```
Expr: .....  
    | IF '(' Test { $3 = lineNumber; } ';' Expr  
        { generateTempQuadruplet(); $6 = lineNumber; } ';' Expr ')'   
        { generateIfQuadruplet($3, $6); }  
    .....  
    ;
```

```
Test: Expr '<' { push(); } Expr { generateTestQuadruplet(); }  
    | Expr '>' { push(); } Expr { generateTestQuadruplet(); }  
    | Expr '=' { push(); } Expr { generateTestQuadruplet(); }  
    ;
```

On fait un switch sur le type de test et on génère un quadruplet contenant le jump associé au test, l'adresse de ce dernier doit être mise à jour.

```
void generateTestQuadruplet()  
{  
    sprintf(result, "CMP %s %s", stack[top - 2], stack[top]);  
    generateQuadruplet();  
    switch (stack[top - 1][0])  
    {  
        case '<': sprintf(result, "JGE"); break;  
        case '>': sprintf(result, "JLE"); break;  
        case '=': sprintf(result, "JNE"); break;  
    }  
    generateQuadruplet();  
    top -= 2;  
    strcpy(stack[top], temp);  
}
```

Si le test est vrai on exécute la première expression, à la fin de cette dernière on doit faire un Jump non conditionne vers la fin de la 2eme expression or on ne la connaît pas, donc on sauvegarde l'adresse du dernier quadruplet de la première expression pour la mettre à jour par la suite.

```
void generateTempQuadruplet()  
{  
    lineNumber++;  
    sprintf(quadStack[lineNumber], "%s", stack[top]);  
    sprintf(result, "JMP");  
    generateQuadruplet();  
}
```

Dans la dernière routine, on peut mettre à jour les deux adresses de jump sauvegarder, on rajoute un test supplémentaire pour vérifier si la deuxième expression n'est qu'un simple identificateur si c'est le cas on le met dans une variable temporaire car les deux expressions doivent finir leur exécution avec la même variable temporaire pour reprendre l'exécution d'une nouvelle expression.

```
void generateIfQuadruplet(int testAddress, int exprAddress)
{
    setJumpAddress(testAddress, exprAddress + 1);
    if (strstr(stack[top], "temp") == NULL)
    {
        setJumpAddress(exprAddress, lineNumber + 2);
        sprintf(temp, "temp%d", tempNumber++);
        sprintf(result, "%s := %s", temp, stack[top]);
        generateQuadruplet();
        strcpy(stack[top], temp);
    }
    else
    {
        setJumpAddress(exprAddress, lineNumber + 1);
    }

    char quadruplet[100] = "";
    sprintf(quadruplet, "%s = %s", stack[top], quadStack[exprAddress - 1]);
    strcpy(quadStack[exprAddress - 1], quadruplet);
    sprintf(result, "JMP");
    top -= 2;
    strcpy(stack[top], temp);
}
```

Exemple :

```

handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
si(a=b;5-a;b*2)
001  CMP a b
002  JNE 6
003  temp0 := 5 - a
004  temp1 = temp0
005  JMP 7
006  temp1 := b * 2
007  END

handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog
si(a<b;c;5+p)
001  CMP a b
002  JGE 5
003  temp0 = c
004  JMP 6
005  temp0 := 5 + p
006  END

```

Exemples :

1) $a + b * \text{somme}(c, \text{somme}(d,e,f), \text{variance}(a,b, c,d,e))$

```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2$ ./prog -f input
a + b * somme(c, somme(d,e,f), variance(a,b, c,d,e))
001 temp0 := d + e
002 temp1 := temp0 + f
003 temp2 := c + temp1
004 temp3 := a * a
005 temp4 := a
006 temp5 := b * b
007 temp4 := temp4 + b
008 temp3 := temp3 + temp5
009 temp5 := c * c
010 temp4 := temp4 + c
011 temp3 := temp3 + temp5
012 temp5 := d * d
013 temp4 := temp4 + d
014 temp3 := temp3 + temp5
015 temp5 := e * e
016 temp4 := temp4 + e
017 temp3 := temp3 + temp5
018 temp3 := temp3 / 5
019 temp4 := temp4 / 5
020 temp4 := temp4 * temp4
021 temp4 := temp4 - temp3
022 temp4 := temp2 + temp4
023 temp5 := b * temp4
024 temp6 := a + temp5
```

2) $5 * 2 + \text{variance}(a, \text{variance}(b, c, \text{max}(d,e)), f) - 4 * g$

```
handaoui@mohamed: /mnt/d/Etudes/2 CS SIL/Semestre 1/COMPIL/TP/Projet Compil -HANDAOUI Mohamed-/partie2
5 * 2 + variance(a, variance(b, c, max(d,e)), f) - 4 * g
001 temp0 := 5 * 2
002 temp1 := a * a
003 temp2 := a
004 temp3 := b * b
005 temp4 := b
006 temp5 := c * c
007 temp4 := temp4 + c
008 temp3 := temp3 + temp5
009 temp6 := d
010 COMP e temp6
011 JL 13
012 temp6 := e
013 temp6 := temp6 * temp6
014 temp5 := temp5 + temp6
015 temp4 := temp4 + temp6
016 temp4 := temp4 / 3
017 temp5 := temp5 / 3
018 temp5 := temp5 * temp5
019 temp5 := temp5 - temp4
020 temp5 := temp5 * temp5
021 temp4 := temp4 + temp5
022 temp3 := temp3 + temp5
023 temp5 := f * f
024 temp4 := temp4 + f
025 temp3 := temp3 + temp5
026 temp3 := temp3 / 3
027 temp4 := temp4 / 3
028 temp4 := temp4 * temp4
029 temp4 := temp4 - temp3
030 temp4 := temp0 + temp4
031 temp5 := 4 * g
032 temp6 := temp4 - temp5
033 END
```