République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

ESI ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
Ecole nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

RAPPORT DU PROJET COMPILATION

Thème

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

Réalisé par

- TAKLIT ZINA
- OPTION SIL

Encadré par

- MR AIT AOUDIA

Promotion : 2017/2018

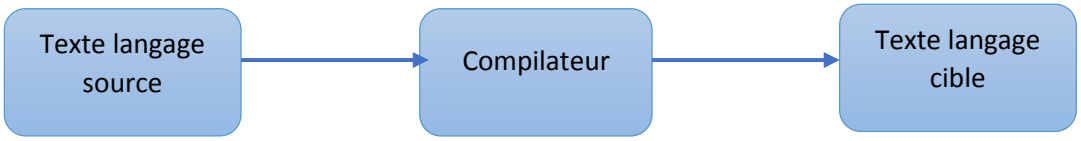
Sommaire

Introduction et problématique

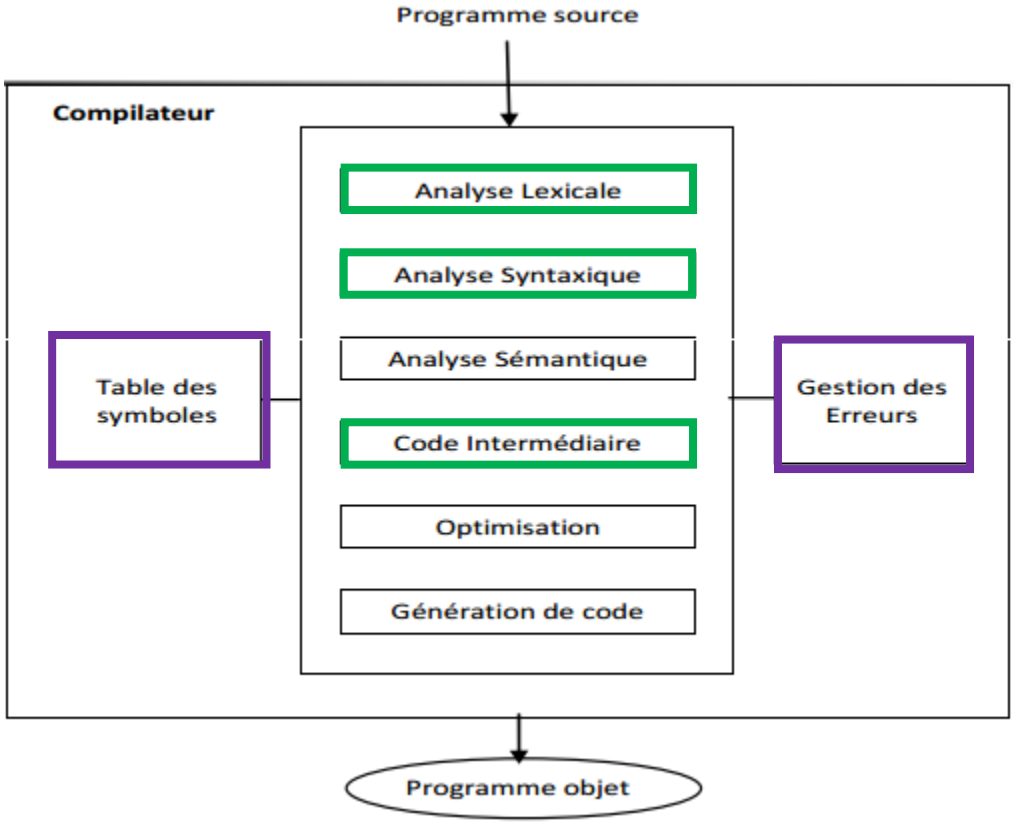
Un compilateur désigne un programme qui permet de transformer un code source écris dans un langage de programmation (langage source) en un autre langage informatique (appelé langage cible) qui peut être exploité par la machine après. Donc :

Compilateur = vérification et traducteur

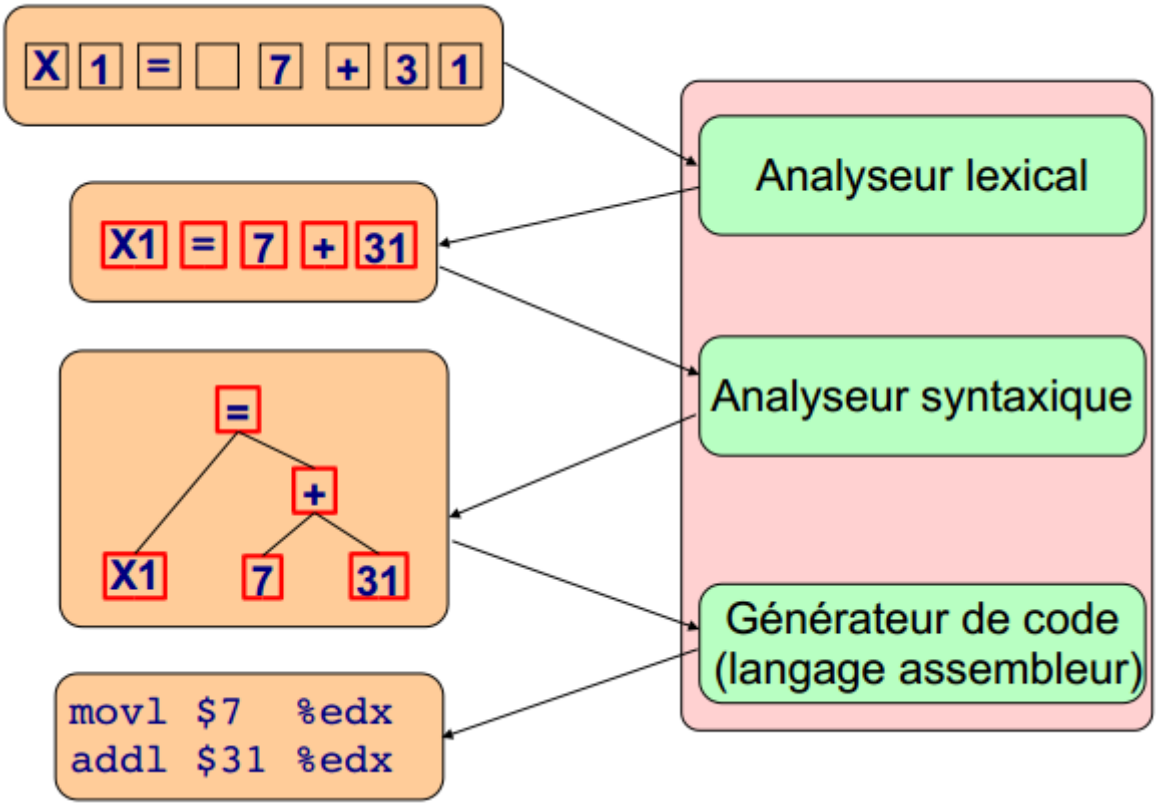
Voici un schéma qui montre le rôle d'un compilateur :



La compilation passe par plusieurs étapes afin d'aboutir un programme objet comme on peut la voir dans cette illustration :



A considérer qu'il y a trois (3) étapes les plus importantes dans la compilation : Analyse lexical, analyse syntaxique, et génération de code. Et pour avoir une idée générale sur ces trois étapes voici ce schéma qui donne un exemple sur chaque étape :



Notre Tp jete la lumière sur ces trois étapes afin de simuler le fonction d'un vraie compilateur :

- Dans la première partie on va développer un analyseur syntaxique d'un expression arithémétique en utilisant LEX et YACC. Les opérandes sont des des nombres entiers ou flottants. Dans cette partie en va aussi signalées les erreur syntaxique et lexical avec un message significatifs au cas d'un erreur.
- Dans la 2ème partie qui consiste a compléter le travail de la première partie pour évaluer l'expression arithmétique donné en entrée si dernière est syntaxiquement correcte. Tout en ajoutant des fonction qui sont souvent utilisées dans les évaluateurs (moyen pondérer).
- La 3ème parties et comme les opérandes ne sont pas connus durant la phase de compilation et donc on va générer une forme intermédiaire sous forme de quadruplets qui une fois exécuté donnera le résultat de l'expression donné au départ.

On utilise Flex comme générateur d'analyse syntaxique et Bison comme générateur d'analyse lexical, et comme la partie une est commune pour les 2 dernière partie donc on va diviser notre travail en 2 grand partie :

1. Partie Evaluation qui contien ces fichier :
   - Ccalc.c : contient la fonction main et les fonctions nécessaire pour les différentes affichage et les fonctions nécessaires pour la gestion erreurs.
   - Ccalc.h : Définition incluse pour tous les modules.
   - Eval.c : contien les fonctions basique et d'autre fonction souvent implémenter pour les évaluateurs.
   - Yacc.y :Grammaire d'entrée pour Bison.
   - Lex.l :Entrée pour Flex.
   - In.txt : contient la fonction à évaluer.
2. Parite code intermédiaire :

Les fichiers ccalc.c, ccalc.h ,lex.l, et yacc.y sont presque les mêmes à ceux du partie évaluation avec des petit modif selon les fonctions implémenté.

- Interm.c : contien les fonctions nécessaires pour générer la forme intermédiaire.
- Evaluation.c : permet d'évaluer et d'éxcuté les forme interrmédiare générées.
- In.t : contient l'exprssion dont on va générer son code intermédiaire.
- Out.txt : le code intermédiare générer pour l'expression du départ.

Partie Evaluation :

Certains traitement dise action sémantique (routine sémantique) sont effectué au fur et à mesure que se d éroule l'analyse syntaxique. On appelle ça une traduction dériger par la syntaxe qui peut avoir plusieurs forme :

pour les évaluateurs d'expression, tel MS EXCEL, le résultat sera l'évaluation de l'expression mathématique si celle-ci est syntaxiquement correcte.

Analyse lexicale :

L'analyse lexical fait partie de la première phase de la chaine de compilation. Elle consiste à convertir une chaine de caractère en un liste de symboles(Tokens). Ces Tokens seront utilisé pendant l'analyse syntaxique.

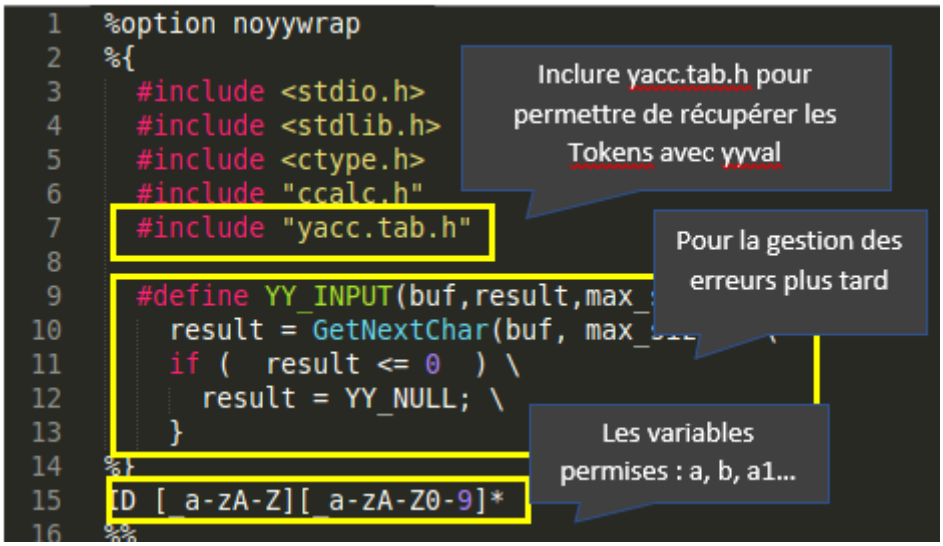
Lex, Flex sont parmi les logiciels qui construit automatiquement un analyseur lexical.

Mise en œuvre :

Pour atteindre le but de cette phase on a créé un fichier « lex.l » qui permet de reconnaitre :

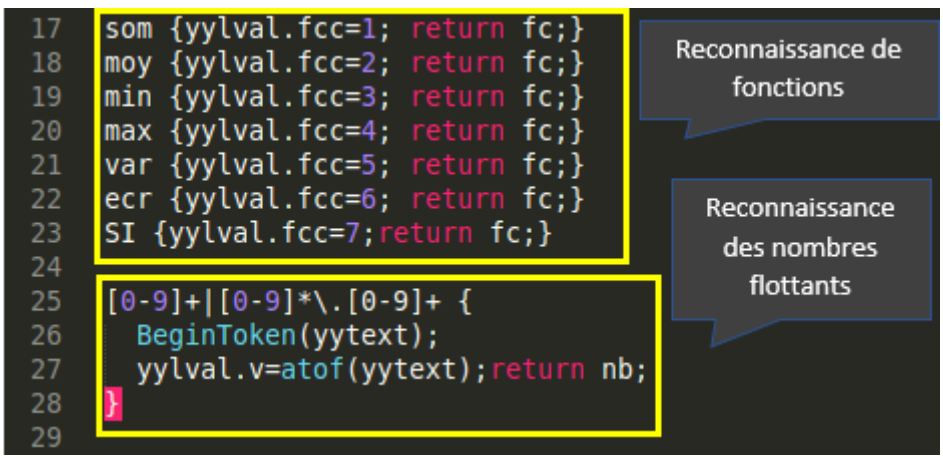
- Identificateur « les variables » (a,b,c…).
- Les nombres flottants (Float) : 2. 5…
- Les noms de différentes fonctions utiliser dans le TP : « som» (somme) , « moy » (moyenne), « var »(variance) , « ecr » (ecart type), « max » , « min » .
- Les espaces « whitespace » ; les caractères séparant entre les unités lexicales. Les espace sont reconnus mais non retournés à l'analyseur syntaxique.
- Tout autres charactères ne sont pas respectés.

Voici une image qui montre comment est structurer ce fichier :




```
%option noyywrap
%{
  #include <stdio.h>
  #include <stdlib.h>
  #include <ctype.h>
  #include "ccalc.h"
  #include "yacc.tab.h"

  #define YY_INPUT(buf,result,max_
    result = GetNextChar(buf, max_
    if (  result <= 0  ) \
      result = YY_NULL; \
    }
%}
ID [_a-zA-Z][_a-zA-Z0-9]*
%%
```




```
som {yylval.fcc=1; return fc;}
moy {yylval.fcc=2; return fc;}
min {yylval.fcc=3; return fc;}
max {yylval.fcc=4; return fc;}
var {yylval.fcc=5; return fc;}
ecr {yylval.fcc=6; return fc;}
SI {yylval.fcc=7;return fc;}

[0-9]+|[0-9]*\.[0-9]+ {
  BeginToken(yytext);
  yylval.v=atof(yytext);return nb;
}
```

Partie Evaluation :

Analyse syntaxique :

L'analyseur syntaxique (parser en englais) est le programme informatique qui consiste à vérifier si la grammaire fixé reconnaisse le programme source donné et puis si oui la question sera comment trouver efficacement un arbre de dérivation pout ce mot (programme) donné.

Parmi les logiciels qui permet de construire automatiquement un analyseur syntaxique on trouve Yacc,Bison... On décrit la grammaire en spécifiant :

- Ses terminaux (ou lexème)
- Ses règles.

Et on associe à chaque règle une action qui sera exécuté lors de la réduction.

Mise en œuvre :

Yacc.y :

Ce fichier contient les définitions des variables et les appels aux fonctions nécessaire dans cette partie. Les déclarations nécessaires dans ce fichier sont les suivantes :

- Bloc littéral optionnelle pour les déclaration C :

```
%{  //déclaration C  %}
```

☝ Il faut inclure la bibliothèque ccalc.h pour les appels de fonctions nécessaire.

```
%{
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "ccalc.h"
#define YYERROR_VERBOSE 1
static Variable *ID;//Global variable
%}
```

- Une directive union décrivant les types des valeurs associées aux symboles terminaux et non terminaux :

Partie Evaluation :

```
%union { //déclaration associé}
```

- La structure « list » est utilisée pour sauvegarder les valeurs d'une liste dans un tableau, affin d'appliquer des traitements plus tard tel le calcul de la moyenne pondérée.

```
%union {
  float v; // The float values
  char *string;//refers to IDs
  int fcc; //The function : som, moy,max, min...

  struct { // Struct of list
  //float value;
    float values[15];
  int cpt;
 } list;
}
```

- Déclaration des terminaux ayant un nom :

```
%token <NOM> nom_terminal
```

- Le terminal « nb » est de type : float. Anoter que « nb » est le token retourné par l'analyse lexical.

```
%token <v> nb
```

- Déclaration de symboles non terminaux, enrichis de leurs types :

```
%type <NOM> nom_non_terminal
```

- Les non terminaux « E » et « fct » sont de type : float.
- Le non terminal « L » est de type : list.
- Le non terminal « nf » (nom de la fonction) est de type : int.

```
%type <v> E fct
%type <list>L
%type <fcc> nf
```

A noter qu'il faut déclarer tous les terminaux et non terminaux avec les directive « token » respectivement « type », sino un message d'erreur sera signalé par exemple :

```
warning Y4003: '$3': attribute is untyped
```

- Priorisation des opérateurs en considérant leurs associativités :

```
%left 'symbole1' 'symbole2' ...
%right 'symbol3' 'symbol4' ...
```

- Les opérateurs binaires « - » et « + » sont associatifs à gauches.
- Les opérateurs binaires « * » et « / » sont associatifs à droites.
- L'opérateur unaire « - » est associatif à droite.
- Pour la priorité des opérateurs les plus prioritaires sont écris le plus bas.

```
%left '-' '+'
%left '*' '/'
%right '^'
%left '>' '<' '='
%right moins_unaire
```

- Déclaration (optionnelle) du symbole de départ :

  %start <NOM>

```
50   %start program
```

- Un ensemble de règles est donné sous la forme :

```
<non terminal> : <membre droit1>
               | <membre droit2>
               .....
               ;
```

  - Le programme est formé d'un ensemble d'instructions séparer « ; ».

```
53   //Rules
54   program
55       : statement ';' program
56       | statement ';'
57       | statement error ';' program
58         {
59         yyerrok;
60         }
61       ;
```

  - La forme de programme a pour but d'exécuté plusieurs instruction en même temps et d'afficher les valeurs de chaque ligne vers la fin.Ex :
    a : 22+3 ;
    b :moy(1,23) ;
    Et donc afficher la valeurs 25 et 2 pour a respectivement b à la foi.

```
statement
: IDENTIFIER {ID = IDGet($1, &@1);}
 ':' E{IDsetValue(ID, $4);}
  | E
;
```

  - Différentes formes de l'expression « E » qui peut être la différence des deux expressions ou leurs additions ou un nombre ou …

```
75     E    :   E '-' E    {
76             $$ = Sub($1, $3);
77           }
```

```
100          |   '-' E     %prec moins_unaire  {$$=-$2;}
101          | '(' E ')'   {$$=$2;}
102          |   nb        {$$=$1;}
103          |   fct       {$$=$1;}
104          | IDENTIFIER  {$$ = IDGetValue($1, &@1);}
105          ;
```

  - L'expression peut être aussi un appel à une fonction (som, moy …) « nf » refera au nom de la fonction ex : 1 pour la somme et 2 pour la moyenne pondérer … som(1,2,3,4) dans ce cas nf=1(sommer les éléments de la liste) et $$=10 (la valeurs retourner par la fonction « Sum ».

Partie Evaluation :

```
107         fct: nf '(' L ')'
108         {
109          switch($1)
110          {
111            case 1:
112                 $$=Sum($3.values,$3.cpt);
113            break;
114            case 2:
115                 $$=Moy($3.values,$3.cpt);
116            break;
```

- La liste « L » peut être un ensemble d'éléments « E » et/ou d'autres liste « L ». On sauvegarde les éléments de chaque liste « L » dans un tableau afin de l'utiliser plus tard.

```
138          L    :   L ',' E {
139               $$=$1; $$.values[$$.cpt++]=$3;
140           }
141          | E {
142          $$.values[0]=$1;  $$.cpt=1;}
143          ;
144   %%
```

Voici un exemple de l'appelle à une fonction :

```
75    E    :   E '-' E    {
76           $$ = Sub($1, $3);
77          }
```

Dans cette règle l'appelle au fonction « Sub » permet d'affecté le résultat de la différence des deux expressions à droite ($1 et $2) à l'expression qui se trouve à gauche ($$).

Voici d'autre fichier qui permet de mener cette évaluation :

Ccalc.h :

Ce fichier contient :

- Déclaration nécessaire et la structures de variable qui permet de sauvegarder la valeur de chaque instruction afin de les afficher plus tard.

```
13  #include "yacc.tab.h"
14
15  // Functions of lex & yacc
16  extern int yylex(void);
17  extern int yyparse(void);
18  extern void yyerror(char*);
19
20  typedef struct _Variable {
21    char    *name;
22    float  value;
23  } Variable;
```

- L'ensemble des fonctions implémenter pour ce Tp.
- Les fonctions qui permet de lire les instructions de programme depuis un fichiers.

- Les fonctions qui permet de gérer les erreurs.
- Les fonctions qui permet de gérer manipuler les différent IDS ainsi qu'afficher leurs valeurs vers la fin.

```
25  // Functions of TP
26  extern float SI(float values[15]);
27  extern float Ecrt(float [15],int);
28  extern float Var(float [15],int);
29  extern float Moy(float [15],int);
30  extern float Sum(float [15],int);
31  extern float Max(float [15],int);
32  extern float Min(float [15],int);
33  extern float Pow(float, float);
34  extern float Add(float, float);
35  extern float Sub(float, float);
36  extern float Mult(float, float);
37  extern float Div(float, float, YYLTYPE*);
38
39  //Need it for the final display and to print significant errors
40  extern void DisplayLine(void);
41  extern int GetNextChar(char *b, int maxBuffer);
42  extern void BeginToken(char*);
43  extern void PrintError(char *s, ...);
44
45  //To manipulate the deferent IDs
46  extern Variable *IDGet(char*, YYLTYPE*);
47  extern void IDsetValue(Variable*, float);
48  extern float IDGetValue(char*, YYLTYPE*);
49  extern void DisplayAllIDs();
50
51  #endif /*CCALC_H_*/
```

eval.c :

- Une bonne pratique et de séparer toutes les fonctions dans un fichier appart.
  Voici un exemple d'une fonction basique « Add » qui permet de faire la somme de deux nombres.

```
81  float Add(float a, float b) {
82    return a + b;
83  }
```

Une autre fonction qui permet de calculer la moyens pondérée d'une liste donné sous forme d'un tableau :

```
39  float Moy( float values[15],int cpt)
40  {
41      float moy=0;
42              moy = (Sum(values,cpt)/cpt);
43      return moy;
44  }
```

Cette fonction ci-dessous permet de calculer la variance des éléments de la liste :

Partie Evaluation :

```c
float Var( float values[15],int cpt) {
  //vriance (n1,n2,n3,...,n)=(n1-moy(n1...n)carré+n2-moy(n2....n)carré+....))/n-1
  float variance=0;
  float moy=Moy(values,cpt);
    for (int j=0;j<cpt;j++)
        variance+=pow((values[j]-moy),2);
  return (variance/(cpt));
}
```

ccalc.c :

On se concentre ici sur la fonction main qui fait appel à la fonction getNexLine() qui permet de lire les donné de fichier ligne par ligne et puis exécuté la fonction yyparse() qui permet de faire l'analyse syntaxique et qui fait un appelle implicite au fonction yylex() qui fait l'analyse lexical. Enfin l'appelle au fonction DisplayAllIDs() qui permet d'afficher le contenue de l'évaluation de chaque ligne :

```c
int main() {
  //int i;
  char *infile=NULL;
  printf("======================== Evaluation ======================== \n\n");
  infile = "in.txt";
  file = fopen(infile, "r");
  if (  file == NULL  ) {
    printf("cannot open input\n");
    return 12;
    }
  buffer = malloc(MaxBuff);
  if (  buffer == NULL  ) {
    printf("cannot allocate %d bytes of memory\n", MaxBuff);
    fclose(file);
    return 12;
  }
  DisplayLine();//Display the read line
  if (  getNextLine() == 0  )
    yyparse();
  free(buffer);
  fclose(file);
  printf("final content of variables\n");
  DisplayAllIDs();
  return 0;
}
```

Exemple d'évaluation :

A noter que pour que générer l'exécutable il faut exécuter ces instructions :

- flex lex.l
- bison -d yacc.y
- gcc -std=c99 -o tp ccalc.c yacc.tab.c lex.yy.c -lm

Voici un exemple d'évaluation :

La gestion des erreurs :

```
1  a:2+5*3-(2+6);
2  b:som(2,3,5,6,8);
3  c:moy(1,2,3);
4  d: 5 + 3 * som(4, som(5,7,8), var(1,1+1, moy(2,4),4,6-2));
5  e:max(3,4,max(8,3));
6  f:min(10,5,max(2,3));
7  g:ecr(3,5,6,8,9);
8  h:3^3;
```

Qui donne comme résultats :

```
======================== Evaluation ========================

    1 |a:2+5*3-(2+6);
    2 |b:som(2,3,5,6,8);
    3 |c:moy(1,2,3);
    4 |d: 5 + 3 * som(4, som(5,7,8), var(1,1+1, moy(2,4),4,6-2));
    5 |e:max(3,4,max(8,3));
    6 |f:min(10,5,max(2,3));
    7 |g:ecr(3,5,6,8,9);
    8 |h:3^3;final content of variables
  ID------------------ Value----------
  'a                  ' 9
  'b                  ' 24
  'c                  ' 2
  'd                  ' 81.08
  'e                  ' 8
  'f                  ' 3
  'g                  ' 2.13542
  'h                  ' 27
```

La gestion des erreurs :

La fonction qui permet d'afficher des erreurs significatives est utiliser par les 2 parties. On la prend plus en détails de cette section afin de montrer l'utilité d'afficher des erreurs significatives qui peuvent après aider pour débugger :

Processus de gestion des erreurs :

- Pour afficher des messages d'erreurs significatifs on utilise la macro : `YYERROR_VERBOSE` et donc le premier message d'erreurs 'syntax error' va devenir :

  Error 'syntax error, unexpected IDENTIFIER, expecting SEMICOLON'

- Flex possède aussi la macro `YY_INPUT` qui lit les données pour l'interprétation de « token ». Donc on ajoute un appel dans YY_INPUT à la fonction GetNextChar() qui lit les donné de fichier et sauvegarde l'information sur la position du caractère suivant à lire en utilisant un buffer qui va contenir une ligne du fichier. Voici comment est utilisé la macro YY_INPUT dans notre TP :

La gestion des erreurs :

```
#define YY_INPUT(buf,result,max_size)  {\
  result = GetNextChar(buf, max_size); \
  if (  result <= 0  ) \
    result = YY_NULL; \
  }
```

- La fonction BeginToken() est appelé par chaque règle pour rappeler le début et la fin de chaque jeton. Voici un exemple d'utilisation de cette fonction :

```
[0-9]+|[0-9]*\.[0-9]+ {
  BeginToken(yytext);
  yylval.v=atof(yytext);return nb;
}
```

- La fonction PrintError () est appelé à chaque fois qu'il y a une erreur pour afficher un message d'erreur sophistiqué.
- Lorsque on a plusieurs erreurs qui sont commis en même temps dans la même ligne on est besoins de la position de l'expression pour y faire on mit la position exacte du « token » dans la variable global « yyloc » de type YYLTYPE. Ensemble avec la macro YYLOC_DEFAULT, Bison calcule la position de l'expression. La version par défaut de YYLTYP est :

```
typedef struct YYLTYPE
{
int first_line;
int first_column;
int last_line;
int last_column;
} YYLTYPE ;
```

Mais elle peut être redéfinie pour ajouter d'autre informations.
Les différent champs de la structure YYLTYPE sont mis à jours à chaque appelle de la fonction BeginToken() :

```
//Mark the beginig of new token
extern void BeginToken(char *t) {
  /* remember last read token ----------*/
  StartToken = NextStartToken;
  TokenSize = strlen(t);
  NextStartToken = cptBuffer; // + 1;
  /* location for bison ----------------*/
  yylloc.first_line = NumLine;
  yylloc.first_column = StartToken;
  yylloc.last_line = NumLine;
  yylloc.last_column = StartToken + TokenSize - 1;
}
```

Sauvegarder le dernier token lit.

Mettre à jours la position exacte de l'expression.

- Pour les règles pour utiliser la position « location » il faut étendre la règle et ceci pour éviter de copier toute la structure dans la règle, un pointeur est généré. Pour l'emplacement du token $3 est référencé via @3 est le pointeur générer est &@3.

```
|   E '/' E    {
 $$ = Div($1, $3, &@3);
}
```

- Pour les fonction les champs de YYLTYPE sont mis à jour directement pour chaque erreur détecter, comme on peut l'avoir dans la fonction qui traite la division entre deux nombres, les champs de YYLTYP sont mis à jour directement au cas où le second membre égale à « 0 ».

```
//divide two numbers, check for zero
extern float Div(float a, float b, YYLTYPE *bloc) {
  if (  b == 0  ) {
 PrintError("division by zero! Line %d:c%d to %d:c%d",
                      bloc->first_line, bloc->first_column,
                      bloc->last_line, bloc->last_column);
    return FLT_MAX;
  }
  return a / b;
}
```

Exemple de messages d'erreurs significatifs :

```
a:2/0;
b:2+3();
c:som(2,3,);
d:min(max);
f:var(2,3;
g!!!!mp;
```

```
======================== Evaluation ====================

     1 |a:2/0;

...... !.....^   token6:6
Error: division by zero! Line 1:c4 to 1:c4
     2 |b:2+3();

...... !.....^..   token6:6
Error: syntax error, unexpected '(', expecting ';'
     3 |c:som(2,3,);

...... !.........^..   token10:10
Error: syntax error, unexpected ')'
     4 |d:min(max);

...... !.....^.....   token6:6
Error: syntax error, unexpected ')', expecting '('
     5 |f:var(2,3;

...... !.........^   token10:10
Error: syntax error, unexpected ';', expecting ')' or ','
     6 |g!!!!mp;
...... !.^.....   token2:2
Error: Reference to unknown variable '(null)'
```

Partie code Intermédiare :

La forme intermédiaire est une abstraction du programme source qui peut être un arbre abstrait (AST for Abstract Syntax Tree) ou un code à trois adresse (quadruplets, triples). La forme intermédiaire est indépendante de la machine et donc elle n'est pas liée à ses jeux d'instruction et est facile de transformer le code intermédiaire au code machine. Une optimisation du code peut s'effectué à ce niveau-là avant de générer le code objet final. Dans

Partie code Intermédiare :

notre Tp on a opté pour le 2ème choix, code à trois (3) adresse qui est le plus utilisé et plus proche du jeu d'instruction de la machine, et on va utiliser les quadruplets pour représenter la forme intermédiaire.

La structure du Quadruplet est de la forme : (op, source1, source2, destination). La source1, source2 et destination sont des variables, constantes ou variables temporaire générés par le compilateur. Op représente un opérateur tel un opérateur arithmétique ou un branchement. Le résultat de l'opération effectué entre la source1 et source2 et assigné aux destinations comme ceci : destination = source1 op source2.

Mise en œuvre :

Structure et fonction essentiel pour générer QuadrUplet :

- Structure « Variable » qui contient le nom de la variable et sa valeur (la valeur sera utilisé pour la partie évaluation).

```
typedef struct _Variable {
  char    *name;
  float  value;
} Variable;
```

- Tableau IDS qui contient l'ensemble des variables déclaré par l'utilisateur, avec un compteur qui permet de parcourir cette table :

```
static Variable IDs[100];
static int cptIDs;
```

- Tableau Tmps qui contient les variables temporaires générer pendant la traduction, avec un compteur qui permet de parcourir cette table :

```
static Variable Tmps[200];
static int cptTmps;
```

- Fichier « out » là où on va écrire les quadruplets générer.
- Une variable « NumLine » qui est incrémenter à chaque fois qu'une écriture sur le fichier est effectué et ça sert pour générer les « Jump » après.

```
static int NumLine;
```

- La fonction « WriteQuad » qui permet d'écrire directement les quadruplets générer dans le fichier « out ».

```
// To generate Quad  And put it in the file Out
extern void WriteQuad(char *opCode,char *src1,char *src2,char *dest)
{
  char *cmd=malloc(50);
  sprintf(cmd,"%s %s %s %s\n",opCode,src1,src2,dest);
  fputs(cmd,file);
  free(cmd);
  NumLine++;
}
```

Le fichier « lex » est le meme que celui utilisé dans la partie évalutation. Pour le fichier « yacc » qui contient l'ensmble de déclaration et de règle pour l'enalyse lexical la différence avec celui de la partie écaluation est :

- Dans cette partie on évlaue pas, tout ce qu'on fait est la gestion de code intermédiare correspondant aux entrées donc ce qui va etres manipuler est les variables et les tmps et donc le type des expression change ainssi que le type des élément de la liste :

```
%type <string> E fct
```

```
%type <list> L
```

- Pour chaque opération dans cette partie on générer un code intermédiaire au lieu d'évaluer, dans cette exemple l'appel au fonction « Mul » permet de générer le code intermédiaire correspond à la fonction de multiplication et returnet le tmps générer pour ça :

```
|   E '*' E    {
 $$ = Mul($1, $3);
}
```

Générer les Quad correspondants aux opérations :

Affectation :

```
: IDENTIFIER  ':' E
  {
     IDGet($1, &@1);
    AffQuad($1,$3);
  }
```

L'affectation est opération de base donc la génération du Quad est simple :

- Vérifier si la variable existe dans ce cas on génaire le Quad.
- sino on créer d'abord la variablepuis générer le Quad :
  voici le contenue de différente champs du Quad :

Op	Source1	Source2	Destination
=	$3	#	$1

Et voici la fonction qui permet de générer ce Quad :

```
//Generate The Quad and tmp for Affectation
extern void AffQuad(char *dest,char *src)
{
        if(idIndex(dest,0)==-1){
         IDs[cptIDs].name=dest;
         cptIDs++;
        }
        WriteQuad("=",src,"#",dest);

}
```

L'Addition :

```
E
      :   E '+' E   {
       $$= Add($1,$3);
      }
```

L'addition est aussi une fonction de base, la génération du Quad est aussi simple :

- Créer une variable tmp_i.
- Le résultat de l'addition sera dans ce tmp_i.
- Retourner le tmp générer : $$=tmp_i$.
  Voici le contenue des différents champs du Quad pour cette opération :

Partie code Intermédiare :

Op	Source1	Source2	Destination
+	$1	$3	tmpi

Et voici la fonction qui permet de générer ce Quad :

```
//Add 2 numbers
extern
char* Add(char * a, char * b) {
  Tmps[cptTmps].name=malloc(12);
  sprintf(Tmps[cptTmps].name,"tmp%d",cptTmps+1);
  WriteQuad("+",a,b,Tmps[cptTmps].name);
  cptTmps++;
  return Tmps[cptTmps-1].name;
}
```

A noter que les autres opérations de base tel : la soustraction, multiplication, division, le moins unaire se génère de la même façon. Pour le moins unaire c'est juste qu'on pas la source2 donc à sa place on mit « # ».

Somme d'une liste d'éléments :

```
fct: nf '(' L ')'
 {
  switch($1)
  {
    case 1:
          $$=Sum($3.values,$3.cpt);
    break;
```

- On génère un tmp initialiser à 0.

Op	Source1	Source2	Destination
=	0	#	tmpi

- On parcoure le tableau est on ajoute chaque élément du tableau « values » tempi générer.

Op	Source1	Source2	Destination
+	tmpi	Values[i]	tmpi

- Retourner le tmp générer : $$=tmpi.

Voici la fonction qui permet de générer les Quads de cette opération :

```
extern char* Sum( char* values[50],int cpt)
{
  int i=0;
  Tmps[cptTmps].name=malloc(12);
  sprintf(Tmps[cptTmps].name,"tmp%d",cptTmps+1);
  WriteQuad("=",values[i],"#",Tmps[cptTmps].name);
  for (i=1;i<cpt;i++)
  {
    WriteQuad("+",Tmps[cptTmps].name,values[i],Tmps[cptTmps].name);
  }
  cptTmps++;
  return Tmps[cptTmps-1].name;
}
```

Partie code Intermédiare :

La moyenne pondérée :

```
case 2:
        $$=Moy($3.values,$3.cpt);
break;
```

La moyenne est basée sur la somme des éléments d'une liste :

- Générer les Quad correspondants a la somme des éléments d'une liste.
- Générer le Quad correspondants à la division du tmp_i générer de l'opération de la somme avec le cpt (nombre des éléments de la table).

Op	Source1	Source2	Destination
/	tmp_i	cpt	tmp_{ii+1}

- Retourner le tmp générer : $$=tmp_{i+1}$.

Voici la fonction qui permet de générer les Quads de cette opération :

```
char* Moy( char* values[50],int cpt)
{
  char *tmp;
  tmp=malloc(12);
  sprintf(tmp,"%d",cpt);
  //It's a same as Som here we just divide by cpt
  Sum(values,cpt);
  WriteQuad("/",Tmps[cptTmps-1].name,tmp,Tmps[cptTmps-1].name);
  cptTmps++;
  return Tmps[cptTmps-2].name;
}
```

Puissance :

```
| E '^' E
 {
     //We have first to ensure that b is a degit
     printf("isNumeric($3) %d\n",isNumeric($3) );
     if (isNumeric($3))
       $$=Pow($1,$3);//Generate the Quad of Pow
     else
       printf("The given base is not a number\n");
 }
```

Le principe ici est de multiplier la source par elle-même tanque la base est différente de zéro, et pour chaque itération on soustrait 1 de la base. Le processus se déroule comme ceci :

Partie code Intermédiare :

1. Générer un tmpi initier avec la valeur de la source(a).
2. Générer un autre tmpi+1 qui va contenir la base.
3. Si la temp correspond à la base égale à 0 allez à fin (7).
4. Sino Multiplier le tmpi par la source(a).
5. Et soustrait 1 de la base (tmpi+1).
6. Allez à 2.
7. Fin

1	= 1 dest
2	Load b
3	Jz 7
4	* dest a dest
5	-b 1 b
6	Jmp 2
7	

Voici la fonction qui permet de générer les Quads de cette opération :

```
char* Pow(char * a, char * b) {
Tmps[cptTmps].name=malloc(12);
sprintf(Tmps[cptTmps].name,"tmp%d",cptTmps+1);//The tmp for the dest

Tmps[cptTmps+1].name=malloc(12);
sprintf(Tmps[cptTmps+1].name,"tmp%d",cptTmps+2);

//We affecte the value  of the b to the generate tmp
WriteQuad("=",b,"#",Tmps[cptTmps+1].name);

WriteQuad("=","1","#",Tmps[cptTmps].name);
WriteQuad("Load",Tmps[cptTmps+1].name,"#","#");
char *whereJmp=malloc(12);
sprintf(whereJmp,"%d",NumLine+5); // we are in the 2nd line and we need to jump the 7th line
WriteQuad("JZ","#","#",whereJmp);
//Maybe we need first to load the dest
WriteQuad("*",Tmps[cptTmps].name,a,Tmps[cptTmps].name);
WriteQuad("-",Tmps[cptTmps+1].name,"1",Tmps[cptTmps+1].name);
sprintf(whereJmp,"%d",NumLine-3); // we are in the 5 line and we need to jum to the 2nd line
WriteQuad("JMP","#","#",whereJmp);
free(whereJmp);
cptTmps=cptTmps+2;
return Tmps[cptTmps-2].name;
}
```

Racine carré[1] *:*

Le principe utilisé pour faire cette opération et illustrait dans cette fonction :

- Définir une précision de calcule soit « n ».
- Soit « m » le résultat de l'opération de la racine carré sur le nombre donné soit « num ».
- Tanque m*m <num on ajout n à m et on itère.

```
void sqrt()
{
float m,n;   float num;
 n=0.0001;
 printf("ENTER A NUMBER : ");   scanf("%f",&num);
 for(m=0;m<num;m=m+n){
   if((m*m)>num){
   m=m-n;
   break;
   }
  }
}
```

[1] Cette fonction permet de calculer les racine carré contenue dans l'intervalle [1, +infinie]

Partie code Intermédiare :

Pour le code intermédiaire on a qu'à traduire ce processus :

1. Générer un tmpi qui va contenir le résultat initier à 0.
2. Générer un tmpi+1 qui va contenir le la précision (tmpi+1=0.0001).
3. Soustraire la source (le nombre dont on veut calculer la racine carré) au tmpi et maitre le résultat dans tmpi+2. Ça permet de comparer le résultat obtenu à la source.
4. SI le résultat est supérieur à 0 on termine le processus (aller à fin).
5. Sino générer le carré de tmpi et le maitre dans tmpi+2.
6. Comparer à nouveau ce résultat avec la source.
7. Si (résultat*résultat> src) allez à 10.
8. Sino ajouter la précision au résultat (tmpi) .
9. Et refaire à nouveau le processus (allez à 3).
10. Soustraie la précision du résultat et terminer le processus.
11. Fin.

1	= 0 tmp_i
2	=0.0001 tmp_{i+1}
3	-tmpi src tmp_{i+2}
4	JG 12
5	*tmp_i tmp_i tmp_{i+2}
6	-tmp_{i+2} src tmp_{i+2}
7	JG 10
8	+ tmp_i tmp_{i+1} tmp_i
9	JMP 3
10	-tmp_i tmp_{i+1} tmp_i
12	Fin

La variance :

```
case 5:
        $$=Variance($3.values,$3.cpt);
break;
```

Voici la formule utilisé pour calculer la variance :

variance(n1,n2,….,n)=(carré(n1-moy(n1..n))+carré(n2-moy(n1..n))+…)/n-1.

- Calculer la moyenne des éléments de la liste.
- Générer un tmpi initialiser à 0 qui va contenir le résultat du calcul de la variance.

Op	Source1	Source2	Destination
=	0	#	tmpi

- Parcourir la liste et pour chaque élément faire :
  - Soustraire la moyenne des élément (tmpi) de cet élément de la liste et mettre le résultat dans tmpi+1.

Op	Source1	Source2	Destination
-	Values[i]	tmp_{moy}	$tmpi_{+1}$

  - Calculer le carré de ce résultat et le mettre dans $tmpi_{+1}$.

Op	Source1	Source2	Destination
*	tmp_{i+1}	tmp_{i+1}	tmp_{i+1}

  - Ajouter ce résultat au tmpi.

Op	Source1	Source2	Destination
*	tmp_i	tmp_{i+1}	tmp_i

- Diviser le résultat par le nombre d'élément de la liste (comme le cpt commence par 0 donc on divise par cpt au lieu de cpt-1).

Op	Source1	Source2	Destination
/	tmp_i	cpt	tmp_i

- Retourner le tmpi ($$=tmp_i).

Partie code Intermédiare :

Voici la fonction qui permet de générer les Quads de cette opération :

```
char* Variance( char* values[15],int cpt) {
   Moy(values,cpt);
  Tmps[cptTmps].name=malloc(12);//tmp3
  sprintf(Tmps[cptTmps].name,"tmp%d",cptTmps+1);
  WriteQuad("=","0","#",Tmps[cptTmps].name);
  Tmps[cptTmps+1].name=malloc(12);//tmp4
  sprintf(Tmps[cptTmps+1].name,"tmp%d",cptTmps+2);
    for (int j=0;j<cpt;j++)
    {
      WriteQuad("-",values[j],Tmps[cptTmps-2].name,Tmps[cptTmps+1].name);
      WriteQuad("*",values[j],Tmps[cptTmps+1].name,Tmps[cptTmps+1].name);
      WriteQuad("+",Tmps[cptTmps+1].name,Tmps[cptTmps].name,Tmps[cptTmps].name);
    }
    char * tmp=malloc(12);
    sprintf(tmp,"%d",cpt);
    WriteQuad("/",Tmps[cptTmps].name,tmp,Tmps[cptTmps].name);
    free(tmp);
    cptTmps=cptTmps+2;
  return Tmps[cptTmps-2].name;
}
```

L'écart type :

```
        case 6:
            $$=ecrt($3.values,$3.cpt);
        break;
```

- Calculer la variance des éléments de la liste.
- Puis calculer racine carrée des résultats de la variance et retourné le résultat.

Voici la fonction qui permet de générer les Quads de cette opération :

```
extern char* ecrt( char* values[15],int cpt){
  //return (sqrt(Var( values,cpt)));
  Variance(values,cpt);
  Sqrt(Tmps[cptTmps-2].name);
  return Tmps[cptTmps-3].name;
}
```

Max :

```
    case 4:
        $$=Max($3.values,$3.cpt);
    break;
```

L'évaluation :

- Générer un tmp_i qui va contenir la valeur du max initier avec le premier élément du tableau « values ».
- Générer un tmp_{i+1} qui va contenir la valeur de l'élément i de la table « values ».
- Générer un tmp_{i+2} qui va contenir le résultat de soustraction de tmp_{i+1} au tmp_i.
- Pour chaque élément de la liste faire :
    1. Affecter l'élément i da la list (values[i]) au $tmpi_{+1}$.
    2. Soustraire la valeur de tmp_{i+1} du contenue de tmp_i (Pour comparer values[i] avec le max).
    3. SI la valeur du max est supérieure à la valeur l'élément i de la table Allez à 1.
    4. Sino affecter la valeur l'élément i de la table au max.

1	= values[i] tmp_{i+1}
2	-tmp_i tmp_{i+1} tmp_{i+2}
3	JG 1
4	= tmp_i tmp_{i+1}

Voici la fonction qui permet de générer les Quads de cette opération :

```
extern char *Max(char* values[50],int cpt)
{
  int i=0;
  Tmps[cptTmps].name=malloc(12);
  sprintf(Tmps[cptTmps].name,"tmp%d",cptTmps+1);
  WriteQuad("=",values[i],"#",Tmps[cptTmps].name); //tmp1 contien la valeur du max
  Tmps[cptTmps+1].name=malloc(12);
  sprintf(Tmps[cptTmps+1].name,"tmp%d",cptTmps+2);
  Tmps[cptTmps+2].name=malloc(12);
  sprintf(Tmps[cptTmps+2].name,"tmp%d",cptTmps+3);
  for (i=1;i<cpt;i++)
  {
    WriteQuad("=",values[i],"#",Tmps[cptTmps+1].name);// = tab[i] tmp2
    WriteQuad("-",Tmps[cptTmps].name,Tmps[cptTmps+1].name,Tmps[cptTmps+2].name);//Sub tmp1 tmp2 tmp3

    WriteQuad("Load",Tmps[cptTmps+2].name,"#","#");
    char *tmp=malloc(12);
    sprintf(tmp,"%d",NumLine+3);//il faut jumper a =[i]  tmp2
    WriteQuad("JG","#","#",tmp);
    WriteQuad("=",Tmps[cptTmps+1].name,"#",Tmps[cptTmps].name);// = tmp2 tmp1
    free(tmp);
  }
  cptTmps=cptTmps+3;
  return Tmps[cptTmps-3].name;
}
```

Min :

C'est le même principe que le « Max » sauf qu'ici on fait le jump si la valeur du max est inférieure à la valeur de l'élément i de la table values.

L'évaluation :

Dans cette partie optionnelle on va interpréter le code intermédiaire ce qui permet de l'exécuter afin d'avoir le résultat de l'expression donné au départ :

Le principe ici est de lire le fichier « out » qui contient le code intermédiaire et de l'interpréter ligne par ligne. On divise chaque linge on quatre champs ce qui permet de regénérer le Quadruplet et puis analyser chaque champ du Quadruplet et effectuer le traitement nécessaire.

Exemple de fonction d'interprétation :

Affectation :

```
else if (strcmp(array[0],"=")==0)
{
    EvaluateAff(array[1],array[3]);
    i++;
}
```

Si le champ d'opération est le symbole « = » et donc l'affectation à ce moment-là on effectuer au champs destination la valeur de la source si la destination est un tmp_i donc on affecte la valeur de la source au champ valeur du ième élément de la table du tmp. Sino on cherche la variable et on affecte la valeur de la source au champ valeur de cette variable. Voici la fonction qui permet de faire ça :

```
extern void EvaluateAff(char *src,char* dest)
{
    float num1=getValueVar(src);
    int index=getIndexVar(dest);
    if (strstr(dest,"tmp") != NULL) {
      Tmps[index].value=num1;
    }
    else
    {
      IDs[index].value=num1;
    }
}
```

Addition :

```
if (strcmp(array[0],"+")==0)
{
  EvaluateAdd(array[1],array[2],array[3]);
  i++;
}
```

Dans le cas de l'addition on récupère la valeur du deux champs source et on effectuer l'addition puis on affecte le résultat à la destination. Voici la fonction qui permet de faire ça :

```
void EvaluateAdd(char *a, char *b,char * dest)
{
    float num1=getValueVar(a);
    float num2= getValueVar(b);
    int index=getIndexVar(dest);
     Tmps[index].value=num1+num2;
}
```

Les autres opérations arithmétiques tel la soustraction, multiplication suit exactement le même principe. Pour l'opération du moins unaire on considère pas le 2ème champ de la source et on retourne directement -source1.

JZ :

L'évaluation :

```
else if (strcmp(array[0],"JZ")==0)
{
   indexLine= EvaluateJZ(LoadValue,array[3]);
   if( indexLine==-1)
      i=i+1; //cause i begin with 0
  else
    i=indexLine-1;/*we jump to the given line and because
    it start from 0 so we sub 1*/
}
```

Dans ce cas on évalue la valeur de la source si celle-ci égale à 0 en jump ver la valeur de la destination sino on lit la ligne suivante du fichier. Voici la fonction qui permet cette évaluation :

```
extern int EvaluateJZ(float a,char *dest)
{
  int indexLine=0;
  if (a==0)
      indexLine=atoi(dest);
  else
  indexLine=-1;
  return indexLine;
}
```

JMP :

C'est le même principe que JZ sauf que dans ce cas on fait aucune évaluation de la source mais on jump directement à la destination.

JG :

C'est le même principe que JZ sauf que dans celle-ci on fait le Jump au cas où la valeur est supérieure à 0.

JL :

C'est l'inverse du JG.

Load :

On a ajouté cette directive dans notre Tp surtout pour nous aider dans l'opération de puissance la ou on doit comparer la base avec « 0 » et donc utilisé le JZ.

```
else if (strcmp(array[0],"Load")==0)
{
   LoadValue=EvaluateLoad(array[1]);
   i++;
}
```

Le traitement de « load » est très simple, on a qu'à récupérer la valeur de la source et l'affecter à une variable globale LoadValue afin de l'utiliser après pour les JZ. Voici la fonction qui permet cette évaluation :

```
extern float EvaluateLoad(char *src)
{
  return (getValueVar(src));
}
```

L’évaluation :

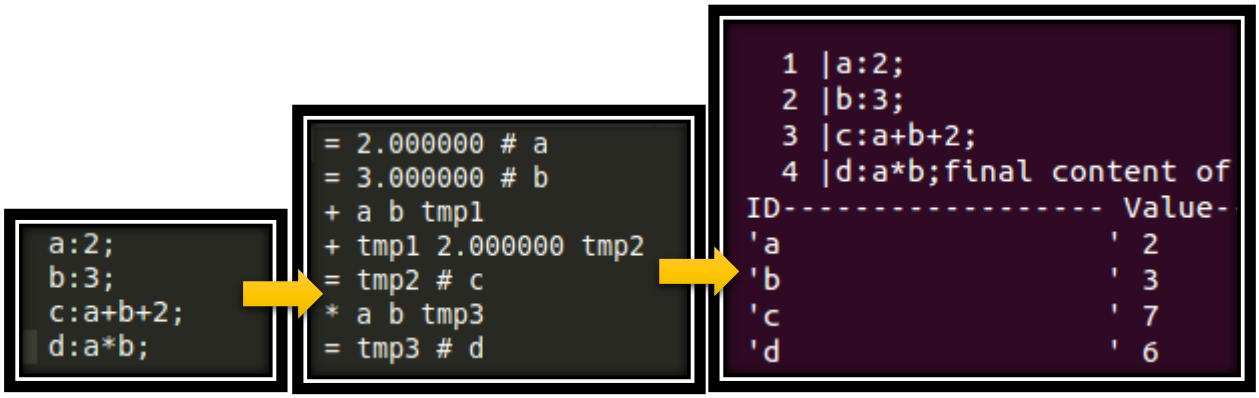
Le résultat de cette interprétation est l’exécution du code intermédiaire et donc retourner le résultat de l’expression donné au départ.

Exemple de code intermédiaire et d’évaluation :

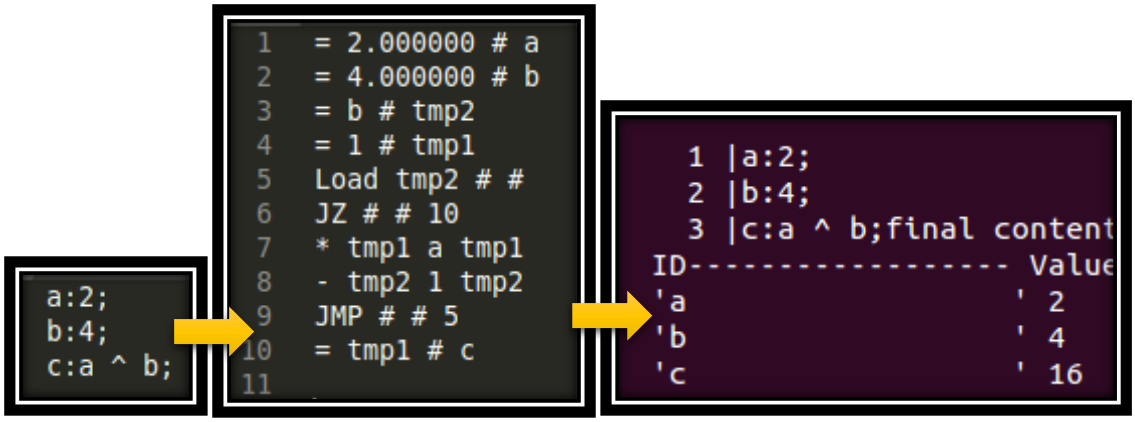
Ici on simule un vraie compléter donc il faut initier toutes les variables à l’avance. On ajoute quelque variable qui va contenir le résultat de chaque expression et ceci dans le but d’afficher le résultat final :

Le premier cadre contient l’expression du départ, le 2ème cadre contient le code intermédiaire et enfin le cadre le plus à droite contient le résultat de l’exécution :

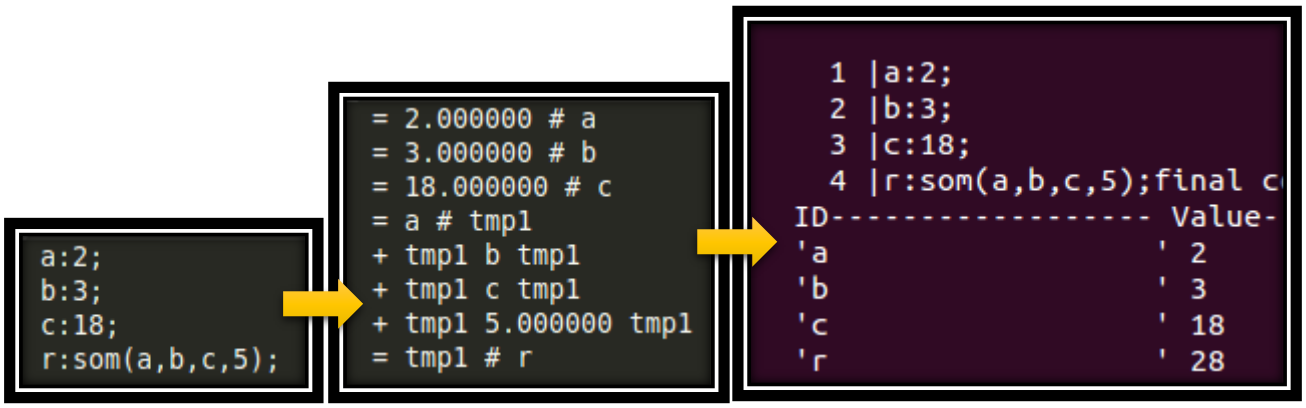
Fonctions de base :



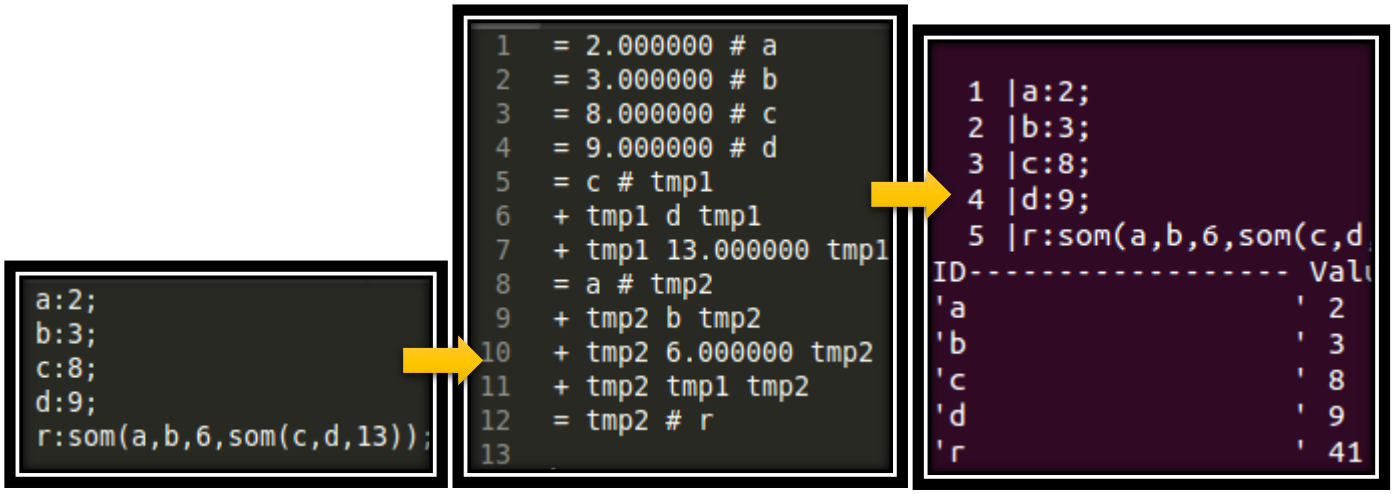
Puissance :



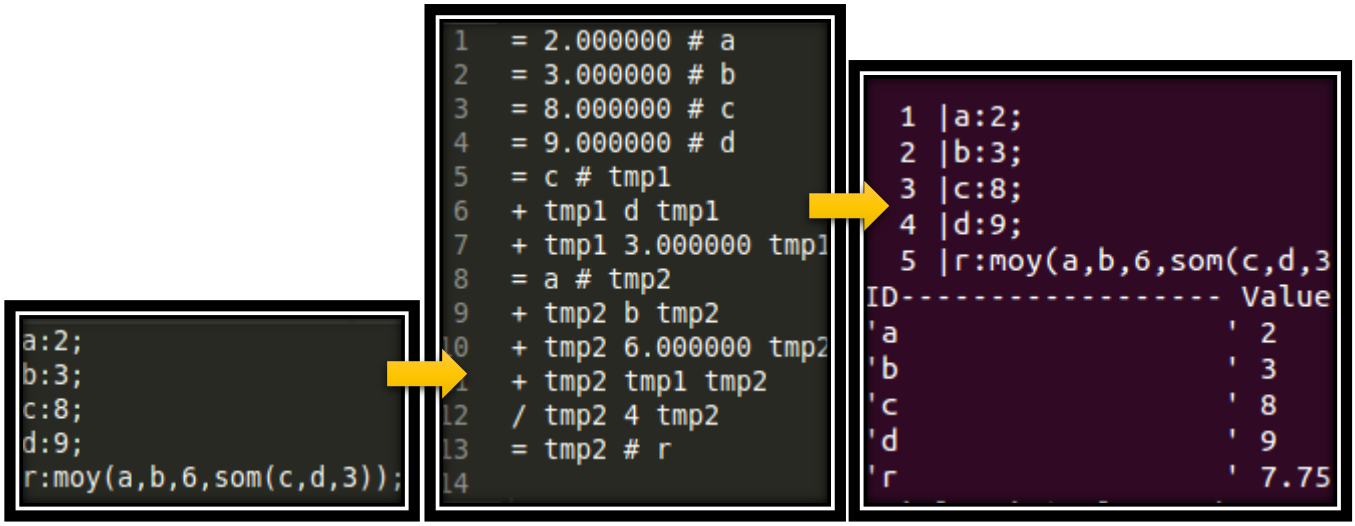
Somme :



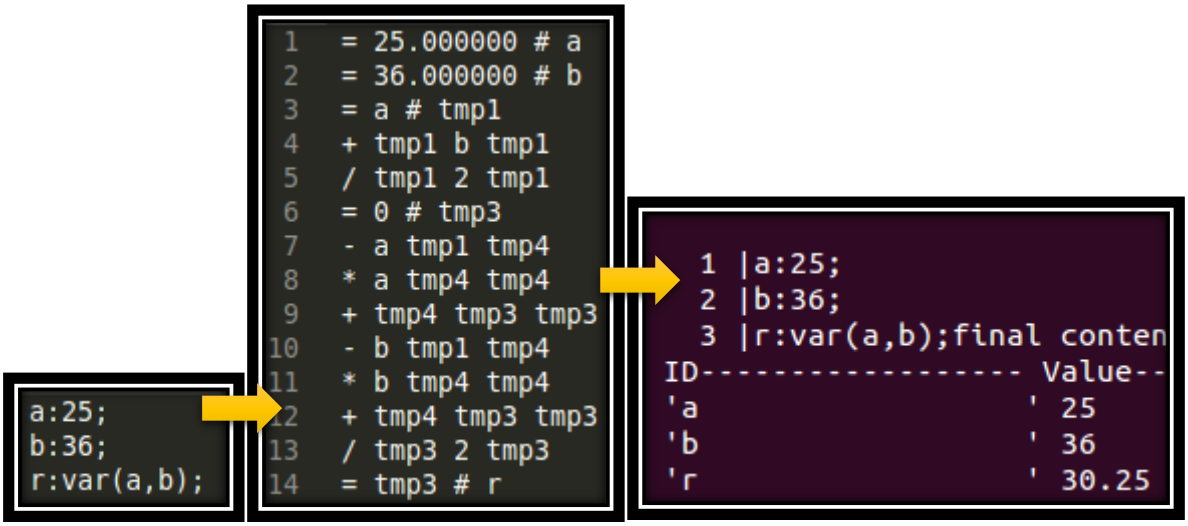
Somme imbriqué :



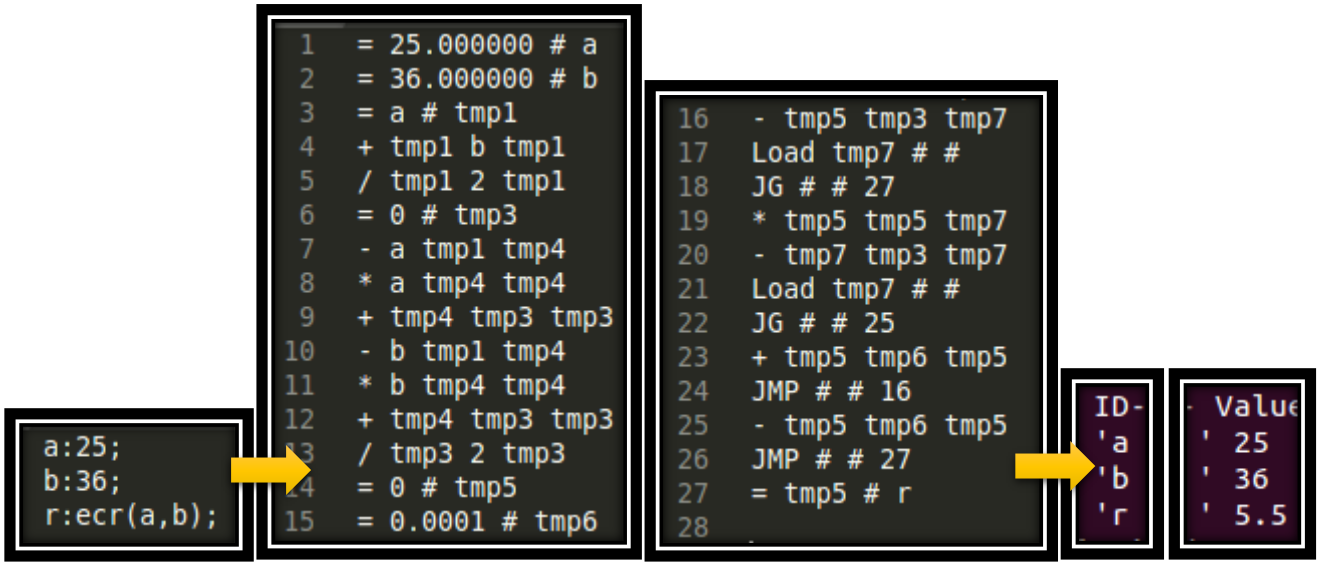
Moyenne pondérée :



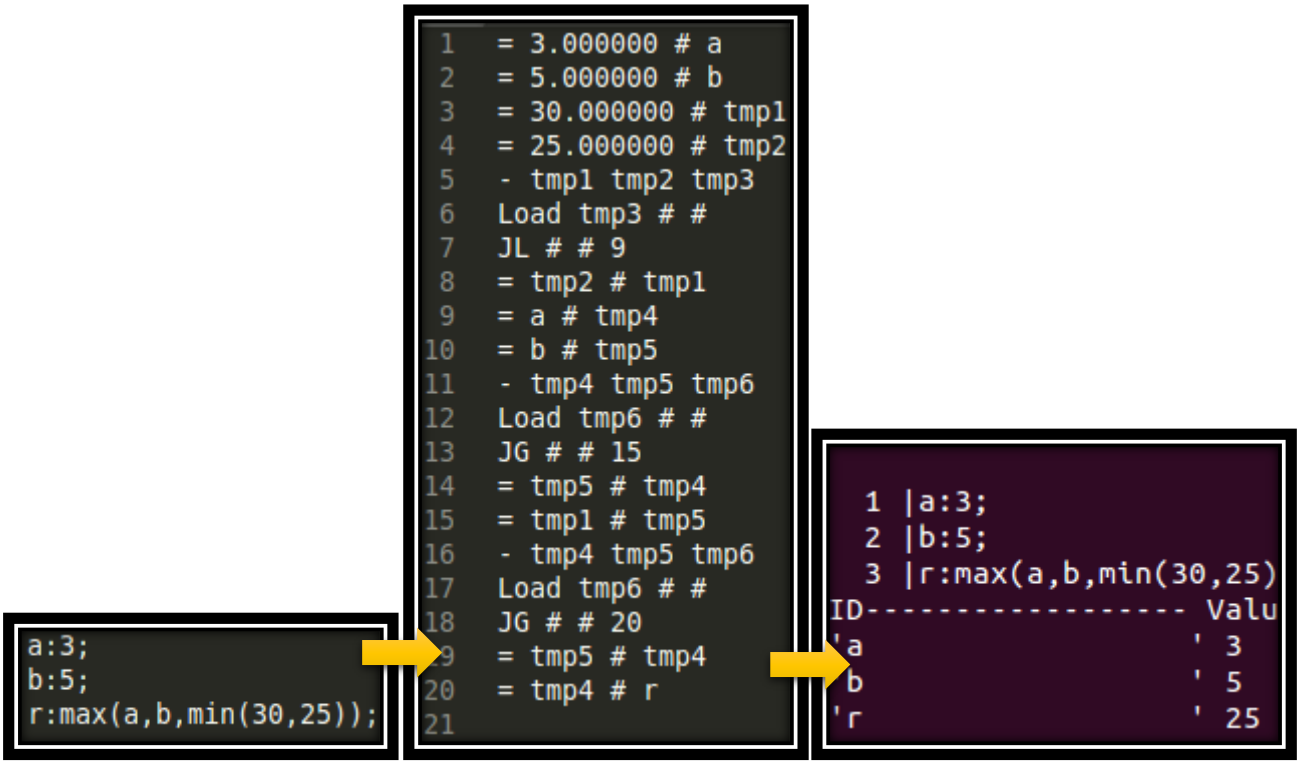
Variance :



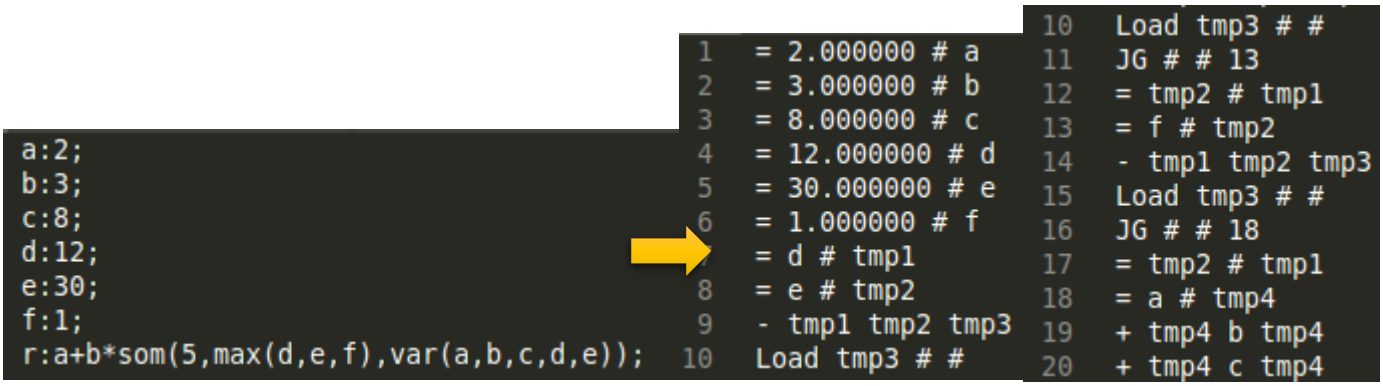
Ecart type :



Max Min :



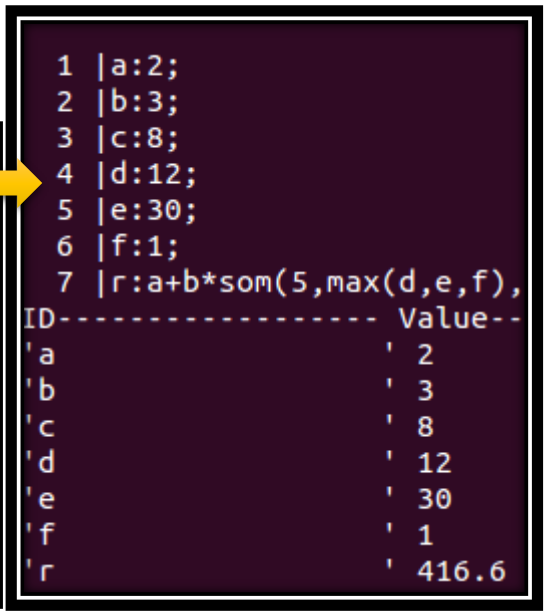
Autre exemple :



L’évaluation :

```
21   + tmp4 d tmp4
22   + tmp4 e tmp4
23   / tmp4 5 tmp4
24   = 0 # tmp6
25   - a tmp4 tmp7
26   * a tmp7 tmp7
27   + tmp7 tmp6 tmp6
28   - b tmp4 tmp7
29   * b tmp7 tmp7
30   + tmp7 tmp6 tmp6
31   - c tmp4 tmp7
32   * c tmp7 tmp7
33   + tmp7 tmp6 tmp6
```

```
34   - d tmp4 tmp7
35   * d tmp7 tmp7
36   + tmp7 tmp6 tmp6
37   - e tmp4 tmp7
38   * e tmp7 tmp7
39   + tmp7 tmp6 tmp6
40   / tmp6 5 tmp6
41   = 5.000000 # tmp8
42   + tmp8 tmp1 tmp8
43   + tmp8 tmp6 tmp8
44   * b tmp8 tmp9
45   + a tmp9 tmp10
46   = tmp10 # r
```



Conclusion

Ce Tp nous a vraiment permet de comprendre le fonctionnement d'un compilateur tout en générant le code qui permet l'évaluation et la génération d'une forme intermédiaire d'une expression arithmétique, et aussi de comprendre comment serai interpréter le code intermédiaire en faisant la partie optionnelle. Ce tp a également ouvert notre esprit à d'autre outils tel Flex et Bison.

Bibliographie

- Christian Hagen.Better error handling using Flex and Bison, developerWorks.
- Site web www.labri.fr, article : « Qu'est-ce qu'un compilateur ? Trois étapes importante », lien : http://www.labri.fr/perso/zeitoun/enseignement/archive/topics/Compilation/01-02/Transparents-2x2-Compilation.pdf.
- Ait-Aoudia,S,(2017), chapitre 0_Introduction_aux_Compilateurs, page 6.
- Ait-Aoudia,S,(2017), chapitre 5_Traduction-Dirigée-Syntaxe.