Segueni Seif Eddine 171732028289 G2 – Touabti Mohamed 171732028255 G2

**Rapport Projet compilation**

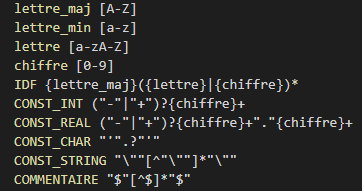
Ce projet on a réalisé un mini-compilateur du langage demande dans l’énoncé en effectuant les différentes phases de la compilation : lexicale, syntaxique, sémantique et génération du code Intermédiaire les quadruplets).

# Les outils utilisés :

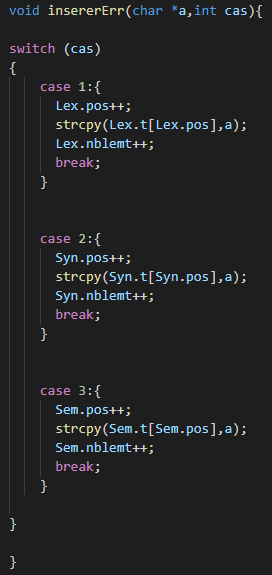
* Flex (Pour la partie lexical).
* Bison (Pour la partie syntaxique et sémantique).
* Langage C.
* VS code (éditeur de texte).

# La partie Lexical :

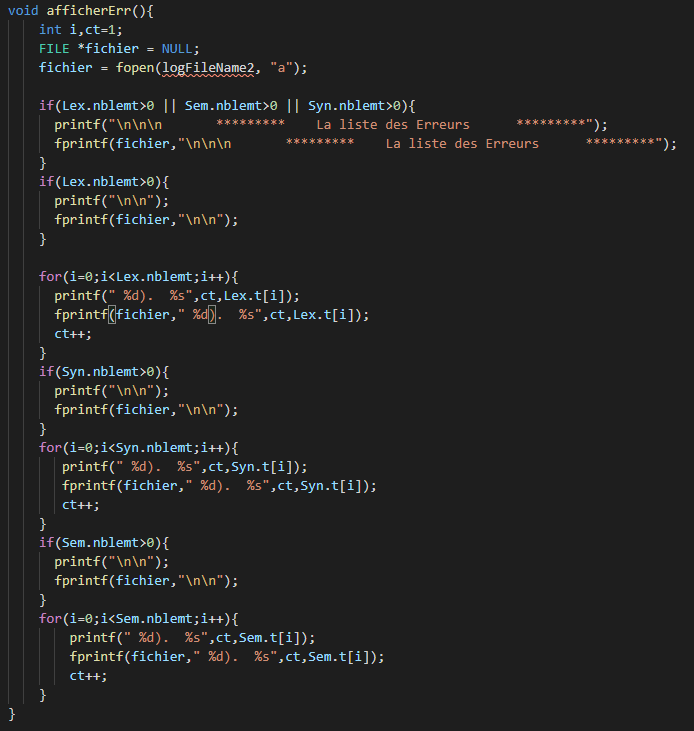
Le but de cette partie est d’associer à chaque mot du programme source la catégorie lexicale à laquelle il appartient. Pour cela on définit la différente entité a l’aide d’expressions régulières suivantes qui sont dans la fichier « lexical.l » la fichier Flex :



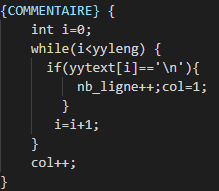
On suite, pour chaque entité reconnue (les mot clés ou bien les séparateurs) on les insère dans la table des symboles et le on incrément la colonne avec la taille de l’entité et envoyer l’entité vers le fichier « syntax.y » pour l’utiliser dans la partie syntaxique et sémantique. Le cas de « Idf » et les constantes on a fait une étape e plus c’est la vérification de la taille avant d’insère dans la table, si la taille est n’est pas valide on n’insère pas et on ajout l’erreur aux la tables des erreurs avec la fonctions « insererErr() » et on suite on affiche tous les erreurs avec la fonction « afficheErr() » et l’insérer dans u fichier log :



Cette fonction consiste à insérer toutes les erreurs avec leur type (lexical, syntaxique, sémantique).



Et après chaque saut de ligne on incrémente le nombre de ligne est mettre les colonne s a 0. Pour le cas de commentaire :

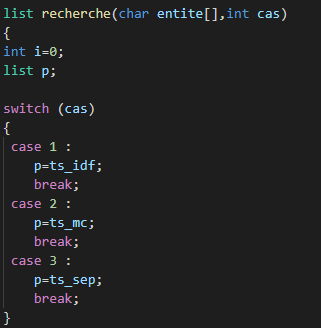


On calcul juste la nombre de ligne est les colonnes.

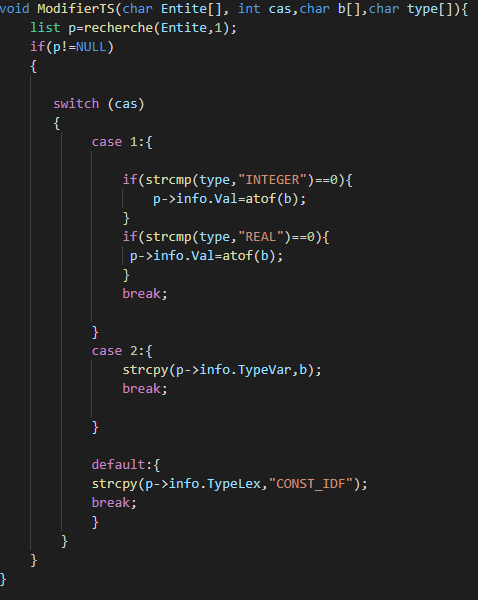
# Table des symboles :

La table des symboles est créée lors la phase de l’analyse l’lexical. Elle regroupe l’ensemble des identificateurs, constantes, les mots clés et les séparateurs définis par le programmeur. Cette table sera mise à jour au fur et à mesure de l’avancement de la compilation. Dans notre projet on a choisi les listes chainées comme un structure pour gérer cette table : 3 liste chainées, «ts\_idf » pour les identificateurs, «ts\_mc » pour les mots clés, «ts\_sep » pour les séparateurs. Cette structure de donnes ainsi les fonctions nécessaires pour la gestion de la table des symboles sont dans le fichier « TS.h ». Les fonctions nécessaires pour la gestion sont :

* Recherche() : pour chercher si l’entité existe dans un des liste avant d’insère.

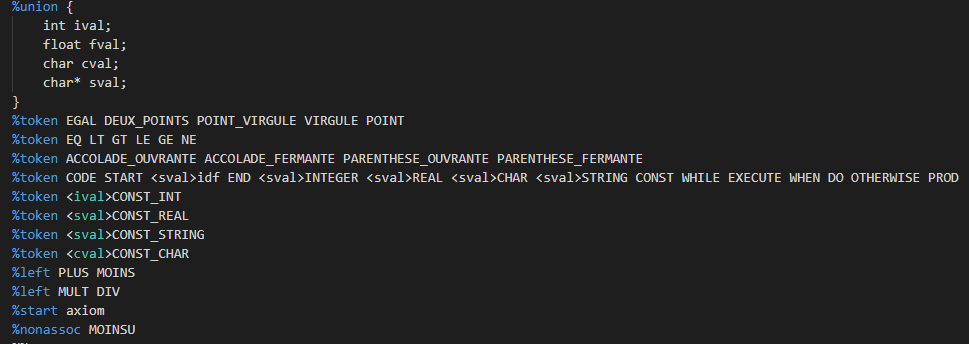


* Inserer() : pour insérer une entité dans sa propre liste chainée si elle n’existe pas déjà dans la liste.
* afficherTs\_IDF(),afficherTs\_MC\_Sep() : pour afficher la liste des identificateurs et les séparateurs et les mot clés respectivement et les insérer dans un fichier log.
* ModifierTS() : pour mettre a jour la table des symboles aux niveau de la partie sémantique, cette fonctions cherche l’entité passe en paramètre et modifier leur type de déclaration ou sa valeur selon la valeur du 2eme paramètre.



# La partie syntaxique :

Pour réaliser cette partie on a créé une grammaire syntaxique cohérant qui respect les conditions d’une grammaire valide LALR (sans conflit et récursivité), cette grammaire est dans le fichier « syntax.y ». Cette grammaire utilise les entités retourner depuis le fichier « lexical.l » comme des tokens avec des spécifications de types (%union) et les priorités (%left) et l’associativité(%noassoc) entre les opérateurs :

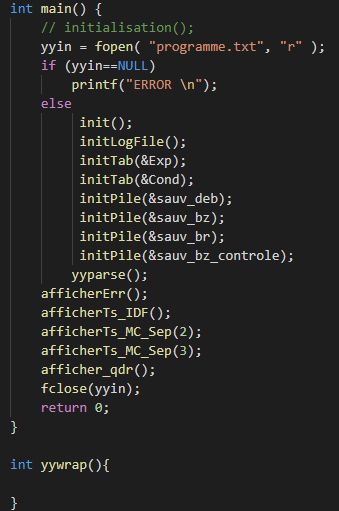
****

L’axiome de cette grammaire est :



Si tout est correct, « YYACCEPT » va être exécuté, sinon pareille comme la partie lexicale, on insère l’erreur avec leur type, dans ce cas Il est de type syntaxique dans la table des erreurs et l’affiche en suite dans le consol et le fichier log.

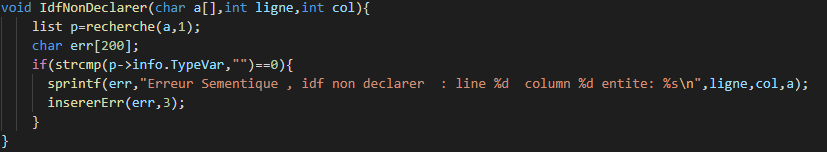
On aussi la fonction main qui sert à ouvrir le fichier qui contient le code a compilé et initialiser les structures de donnes, et afficher les erreurs s’ils existent :



# La partie sémantique :

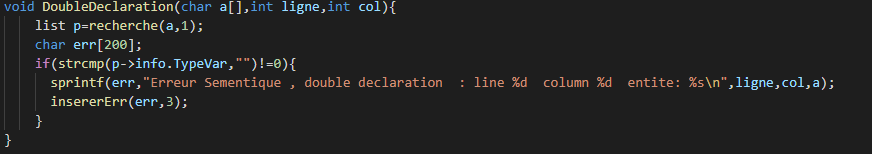
Pour cette partie on implémente des routines sémantiques qui consiste à détecte les diffèrent erreur sémantique :

* **Idf non déclaré :** la fonction IdfNonDeclarer() a comme paramètre l’entité laquelle on doit vérifier. Cette fonction recherche ce dernier dans la liste des idfs, et vérifier si cette entité a un type, sinon ça veut dire l’idf est n’est pas déclarer, par conséquence on générer une erreur sémantique et on l’ajout dans la table des erreurs.



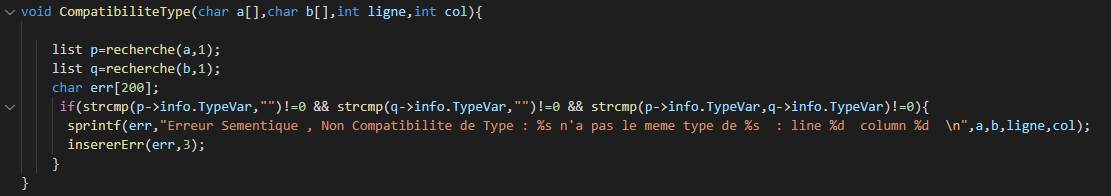
On appelle cette routine dans toutes apparitions de le token IDF dans notre grammaire sauf dans le bloc de déclaration.

* **Idf double déclaration** : la fonction DoubleDeclaration(), de même comme la précédente dans les paramètres et la recherche, si l’entités dans la table des symboles a déjà un type donc cette fonction générer une erreur sémantique et on l’ajout dans la table des erreurs.



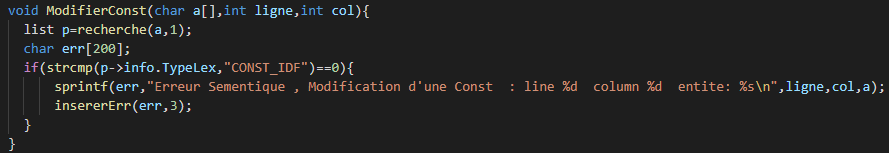
On appelle cette routine dans toutes apparitions de le token IDF dans notre grammaire dans le bloc de déclaration.

* **Non compatibilité de type** : la fonction CompatibiliteType(), prend en paramètre 2 entités ensuite recherche le type de chaque entité dans la liste, s’ils ont des types différents cette fonctions générer une erreur sémantique et on l’ajout dans la table des erreurs.



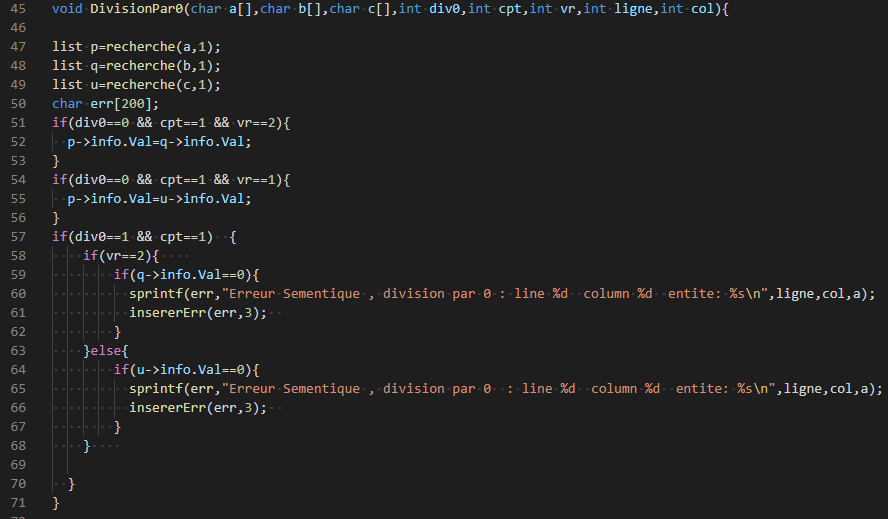
On appelle cette routine dans les affectations et la déclaration d’une constante.

* **Modification d’une constante** : la fonction MoifierConst(), prend en paramètre une entité et ensuite récupère le type de cette entité s’il est de type Const dons une erreur va être générer et insérer dans la table des erreurs.



On appelle cette routine dans tous les blocs d’apparitions dans la grammaire et vérifier le membre gauche d l’affectation.

* **Division Par 0 :**



* div0 : ce paramètre définie s’il y a une division ou non
* cpt : si div0==0 alors cpt == nombre des opérandes dans la dernière expression

si div0==1 alors cpt == nombre des opérandes après la division.

* vr  : 1 si le dernier opérande de l’expression est une constante

2 si le dernier opérande de l’expression est un IDF.

Donc :

Si div0==0 alors il y’a pas de division.

Si cpt==1 alors il y’a un seul opérande dans la dernière expression

Donc on peut modifier la valeur de l’idf ( Le champ Val dans la TS) à gauche de l’affectation

vers la valeur d’une constante si vr==1 ou un idf si vr==2.

**Exemples :** X1:=1999.222; X1 :=(Var1) ; X1 :=Var1 ;

Si div0==1 alors il y’a une division .

Si cpt==1 alors il y’a un seul opérande après la division

Donc ce cas on doit vérifier si la valeur de l’opérande (constante si vr==1 , idf si vr==2)

Est égale a 0.

**Exemples** :

X1:=1999.222/0;

X1 :=0 ; X2 :=5000\*123/X1 ;

# Les Quadruplets :

**1).  *Les quadruplets d’une expression arithmétique exp avec des parenthèses :***

**L’algorithme :**

* **Etape :** La recherche dans exp (tableau de chaine de caractère) de la dernière parenthèse ouvrante qui se trouve à la position i.

Après on recherche sa parenthèse fermente qui se trouve à la position j Et il est clair que l’expression exp1 qui se trouve être la position i+1 et j-1 ne contient aucune parenthèse.

* **Etape :** L’évaluation de l’expression (exp1)
  + La décomposition de exp1 en deux tableaux de chaine de caractère :
    - Opr : Ce Tableau contient tous les opérandes du exp1.
    - Opt : Ce Tableau contient tous les opérateurs du exp1.
  + Pour chaque opérateur opt \* ou / a la position i du Opt, en prend deux opérandes du Opr : op1 a la position i, op2 a la position i+1.

Et on ajoute le quadruplet :

InsererQuadr(opt,opr1,opr2,‘’’’);

Apres en remplace op1 de la position i dans Opr par ‘’’’ le résultats du (op1 opt op2)

* + Pour chaque opérateur opt + ou - a la position i du Opt, en prend deux opérandes du Opr : op1 a la position i, op2 a la position i+1.

Et on ajoute le quadruplet :

InsererQuadr(opt,opr1,opr2,‘’’’);

Apres en remplace op1 de la position i dans Opr par ‘’’’ le résultats du (op1 opt op2), k ϵ N\*

* **Etape :** Apres l’étape 2 le résultat de exp1 ‘’’’ est le seul élément

Restants dans le tableau Opr.

Donc On remplace dans exp : exp1 par le résultat de exp1 ‘’’’.

* **Etape :** Refaire les trois étapes précédentes.

A la fin on sauvegarde le résultats finale ‘’’’ de exp dans une variable sauveExp

**Exemple d’une affectation :**

Var1 := ( X1 + (10.55 \* 20 \* (Const1\*8.55) + 15 \* (8/12\*18) ));

1). exp= ( X1 + (10.55 \* 20 \* (Const1\*8.55) + 15 \* (8/12\*18) ));

exp1 = 8/12\*18

Opr= [ 8 , 12 , 18]

Opt= [ /, \* ]

24 - ( /  ,  8  ,  12  ,  T1 )

Opr= [ T1 , 18]

Opt= [ \* ]

  25 - ( \*  ,  T1  ,  18  ,  T2 )

Opr= [ T2 ]

Opt= []

2). exp= ( X1 + (10.55 \* 20 \* (Const1\*8.55) + 15 \* T2 ) );

exp1 = Const1\*8.55

Opr= [ Const1 ,8.55]

Opt= [ \* ]

  26 - ( \*  ,  Const1  ,  8.55  ,  T3 )

Opr= [T3]

Opt= []

3). exp= ( X1 + (10.55 \* 20 \* T3 + 15 \* T2 ) );

exp1 = 10.55 \* 20 \* T3 + 15 \* T2

Opr= [ 10.55 ,20,T3,15,T2]

Opt= [ \*, \* ,+ , \* ]

  27 - ( \*  ,  10.55  ,  20  ,  T4 )

Opr= [T4,T3,15,T2]

Opt= [ \* ,+ , \* ]

  28 - ( \*  ,  T4  ,  T3  ,  T5 )

Opr= [T5,15,T2]

Opt= [ + , \* ]

  29 - ( \*  ,  15  ,  T2  ,  T6 )

Opr= [T5,T6]

Opt= [ + ]

30 - ( +  ,  T5  ,  T6  ,  T7 )

Opr= [T7]

Opt= []

4). exp= ( X1 +T7 );

exp1 = X1+ T7

Opr= [X1,T7]

Opt= [ + ]

31 - ( +  ,  X1  ,  T7  ,  T8 )

Opr= [T8]

Opt= [ ]

  32 - ( :=  ,  T8  ,    ,  Var1 )

***2). Les quadruplets d’une condition (deux operandes et un opérateur de comparaison):***

**Condition :** opr1 Opt opr2

Opt ϵ {GT,GE,EQ,NE,LE,LT}

**Quadruplet :**

InsererQuadr(opt,op1,op2,"TC");

***3). Les quadruplets de l’instruction de contrôle WHEN :***

**Grammaire après traduction :**

CONTROLE: CONTROLE\_B OTHERWISE BLOCK

 {

itoa(qc,tmp,10);

Maj\_Quad(depiler(&sauv\_br),1,tmp);

}

;

CONTROLE\_B: CONTROLE\_A DO BLOCK

 {

    empiler(&sauv\_br,qc);

    InsererQuadr("BR","","","");

    itoa(qc,tmp,10);

    Maj\_Quad(depiler(&sauv\_bz\_controle),1,tmp);

 }

;

CONTROLE\_A: WHEN {cpt2=1;} CONDITION

{

empiler(&sauv\_bz\_controle,qc);

InsererQuadr("BZ","","",resCond);

}

;

Dans le cas où il y’a deux instructions WHEN imbriques on doit utiliser deux pile sauve\_br\_controle et sauve\_bz\_controle au lieu de deux entiers.

Et ça pour sauvegarder pour chaque instruction WHEN, ses propres valeurs de

sauve\_br \_controle respectivement(sauve\_bz\_controle) dans le sommet de la pile sauve\_br respectivement(sauve\_bz\_controle).

Alors pour enregistrer les valeurs de sauve\_bz on utilise la fonction :

empiler(&sauv\_bz\_controle,qc);

Et pour récupérer les valeurs on utilise la fonction :

depiler(&sauv\_bz\_controle,qc);

**Exemple d’une instruction WHEN** :

**Instruction:**

WHEN 555 EQ (5 + 3)

DO

Var1:= 1999;

OTHERWISE

WHEN "sss" GT "ss"

DO

Var1:= Var1 + Var2;

OTHERWISE

Var1:=0;

**Les quadruplets:**

 64 - ( /  ,  Var1  ,  8  ,  T1 )

 65 - ( \*  ,  T1  ,  5  ,  T2 )

 66 - ( +  ,  5  ,  3  ,  T3 )

 67 - ( EQ  ,  T2  ,  T3  ,  TC )

 68 - ( BZ  ,  71  ,    ,  TC )

 69 - ( :=  ,  1999  ,    ,  Var1 )

 70 - ( BR  ,  77  ,    ,   )

 71 - ( GT  ,  "sss"  ,  "ss"  ,  TC )

 72 - ( BZ  ,  76  ,    ,  TC )

 73 - ( +  ,  Var1  ,  Var2  ,  T1 )

 74 - ( :=  ,  T1  ,    ,  Var1 )

 75 - ( BR  ,  77  ,    ,   )

 76 - ( :=  ,  0  ,    ,  Var1 )

**4). Les quadruplets de l’instruction BOUCLE :**

**Grammaire après traduction :**

BOUCLE: BOUCLE\_B EXECUTE ACCOLADE\_OUVRANTE INSTRUCTIONS

ACCOLADE\_FERMANTE POINT\_VIRGULE

 {

  itoa(depiler(&sauv\_deb),tmp,10);

  InsererQuadr("BR",tmp,"","");

  itoa(qc,tmp,10);

  Maj\_Quad(depiler(&sauv\_bz),1,tmp);

 }

;

BOUCLE\_B: BOUCLE\_A CONDITION

 {

  empiler(&sauv\_bz,qc);

  InsererQuadr("BZ","","",resCond);

 }

;

BOUCLE\_A: WHILE {empiler(&sauv\_deb,qc);cpt2=1;}

;

**Exemple d’une instruction BOUCLE :**

**Instruction:**

WHILE (5\*3 + 5/5) LT Var1 EXECUTE

{

Var2 := 22/(((Var1)));

Var2 := 20/(55);

WHILE "abn" EQ "a" EXECUTE

{

Var2 := 2022;

WHILE 'e' NE 'a' EXECUTE

{

Var2 := 2021 + Var2;

};

};

Var2 := 1999;

};

**Les quadruplets:**

 39 - ( \*  ,  5  ,  3  ,  T1 )

 40 - ( /  ,  5  ,  5  ,  T2 )

 41 - ( +  ,  T1  ,  T2  ,  T3 )

 42 - ( LT  ,  T3  ,  Var1  ,  TC )

 43 - ( BZ  ,  59  ,    ,  TC )

 44 - ( /  ,  22  ,  Var1  ,  T1 )

 45 - ( :=  ,  T1  ,    ,  Var2 )

 46 - ( /  ,  20  ,  55  ,  T1 )

 47 - ( :=  ,  T1  ,    ,  Var2 )

 48 - ( EQ  ,  "abn"  ,  "a"  ,  TC )

 49 - ( BZ  ,  57  ,    ,  TC )

 50 - ( :=  ,  2022  ,    ,  Var2 )

 51 - ( NE  ,  e  ,  a  ,  TC )

 52 - ( BZ  ,  56  ,    ,  TC )

 53 - ( +  ,  2021  ,  Var2  ,  T1 )

 54 - ( :=  ,  T1  ,    ,  Var2 )

 55 - ( BR  ,  51  ,    ,   )

 56 - ( BR  ,  48  ,    ,   )

 57 - ( :=  ,  1999  ,    ,  Var2 )

 58 - ( BR  ,  39  ,    ,   )

***5). Les quadruplets de l’instruction PROD :***

Prod(<exp1> ,<exp2>,.., <expn> ) 🡺 Prod((R1) (<exp1>(R2) ,<exp2> (R3) ,...,<expn> (Rn+1) )

**Mèthode :**

(":=",1,,idf)

{quadruples du <exp1> return temp1}

("BMZ", exp2, , temp1) ===> On ne passe directement vers exp2 si temp1<=0 ( BMZ )

( "\*" , temp1 , idf , idf) ===> On multiplier temp1 avec idf et le résultats est stocker dans idf

exp2 : {quadruples du <exp2> return temp2}

("BMZ", exp3 , , temp2)

("\*", temp2 , idf , idf)

exp3:{quadruples du <exp3> return temp3}

("BMZ",exp4,,temp3)

("\*",temp3,idf,idf)

.....

.....

expn{quadruples du <expn> return tempn}

("BMZ",fin,,tempn)

("\*",tempn,idf,idf)

fin:

Le résultats finale du Prod est stocker dans idf , et si tous les expressions sont <=0

Alors aucune multiplication est effectuée et le résultat final est 1.

**Les routines sémantiques :**

R1:{

InsererQuadr( ":=" , 1 , "idf");

}

R2…....Rn+1:{

InsererQuadr("BMZ" , qc+2 , , temp) ; ==> si qc est l’indice du quadruplets BMZ alors

qc+2 est l’indice du prochaine expression.

InsererQuadr( "\*" , temp , 1 , idf) ;

}

**Exemple d’une instruction PROD :**

Var3 := PROD(20 + 10 \* 20 ,5 - 8 + 3 - 8 + 6 ,15\*45 - 59);

 4 - ( :=  ,  1  ,    ,  Var3 )

 5 - ( \*  ,  10  ,  20  ,  T1 )

 6 - ( +  ,  20  ,  T1  ,  T2 )

 7 - ( BMZ  ,  9  ,    ,  T2 )

 8 - ( \*  ,  T2  ,  Var3  ,  Var3 )

 9 - ( -  ,  5  ,  8  ,  T3 )

 10 - ( +  ,  T3  ,  3  ,  T4 )

 11 - ( -  ,  T4  ,  8  ,  T5 )

 12 - ( +  ,  T5  ,  6  ,  T6 )

 13 - ( BMZ  ,  15  ,    ,  T6 )

 14 - ( \*  ,  T6  ,  Var3  ,  Var3 )

 15 - ( \*  ,  15  ,  45  ,  T7 )

 16 - ( -  ,  T7  ,  59  ,  T8 )

 17 - ( BMZ  ,  19  ,    ,  T8 )

 18 - ( \*  ,  T8  ,  Var3  ,  Var3 )