

1-l'analyse lexical :

L'analyseur lexical travaille généralement en suivant des règles de reconnaissance prédéfinies, qui spécifient les motifs et les structures des différents lexèmes attendus dans le langage source(dans notre cas on travaille avec le langage PCL) . Ces règles peuvent être exprimées à l'aide d'expressions régulières, de grammaires formelles ou d'autres mécanismes de correspondance de motifs.

Voici quelques captures d'écran de l'exécution de notre analyseur lexical

```
C:\Windows\System32\cmd.exe

C:\Users\DELL\Desktop\projet_compil>comp.exe < exemple_while.txt
FFFFFFFF {
VAR {
CONST FLOAT C ;
FLOAT : Tab1 [ 5 ] , Tab2 [ 5 ] ;
INTEGER : Tab3 [ 5 ] ;
CONST F = 3.7 ;
INTEGER L , V ;
INTEGER Z ;
FLOAT I ;
FLOAT W , A , B ;
}
CODE {
W = 1.0 ;
A = 20.0 ;

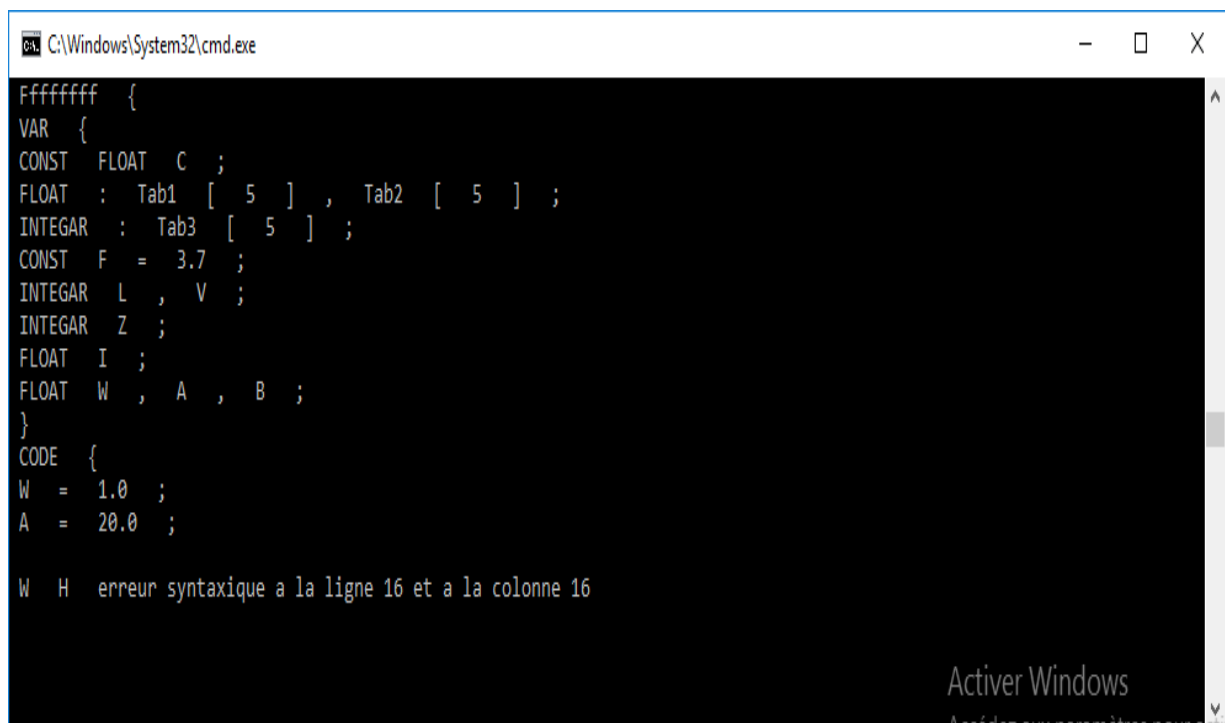
WHILE ( ( ( ( A == 3 ) && ( W == 1.0 ) ) || ( A == 20.0 ) ) ) {
B = B + 2.0 ;
W = W + 1 ;
A = A - 1.0 ;
}
}
}
programme syntaxiquement correct
```

Dans notre exemple, toutes les entités du programme ont été reconnues par l'analyseur lexical et donc elles sont affichées sur écran .

2-l'analyse syntaxique :

L'analyseur syntaxique construit un arbre syntaxique ou une structure de données similaire, représentant la hiérarchie et les relations entre les différentes parties du programme. Cela permet de déterminer la validité syntaxique du programme et de détecter les erreurs de syntaxe telles que les incohérences grammaticales, les expressions mal formées, les parenthèses non appariées,etc.

Voici quelques captures d'écran de l'exécution de notre analyseur syntaxique :



```
C:\Windows\System32\cmd.exe
FFFFFFFF {
VAR {
CONST FLOAT C ;
FLOAT : Tab1 [ 5 ] , Tab2 [ 5 ] ;
INTEGER : Tab3 [ 5 ] ;
CONST F = 3.7 ;
INTEGER L , V ;
INTEGER Z ;
FLOAT I ;
FLOAT W , A , B ;
}
CODE {
W = 1.0 ;
A = 20.0 ;

W H erreur syntaxique a la ligne 16 et a la colonne 16

Activer Windows
Accédez aux paramètres pour...
```

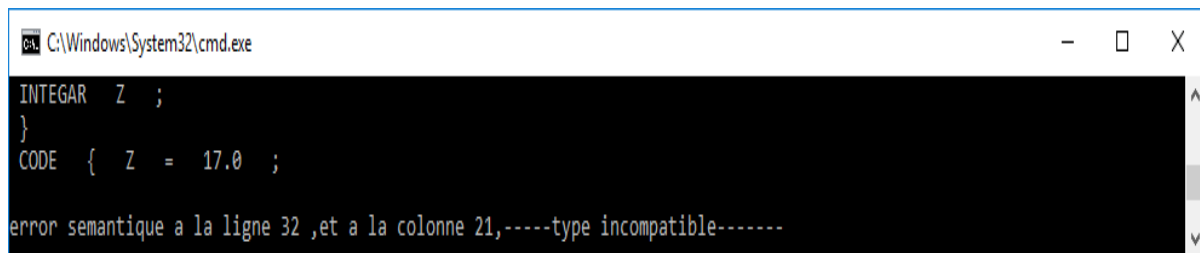
-----> L'erreur syntaxique à la ligne 16 et à la colonne 16 fait référence à une erreur dans la structure ou la formation du code source du programme à ces positions spécifiques. Lorsque l'analyseur syntaxique rencontre une erreur, il peut signaler l'endroit exact où l'erreur a été détectée, en fournissant des informations sur la ligne et la colonne correspondantes.

3-l'analyse sémantique :

L'objectif principal de l'analyse sémantique est de détecter et de traiter les erreurs sémantiques qui ne sont pas détectées lors de l'analyse lexicale ou de l'analyse syntaxique, mais qui sont liées à la logique et à la signification du programme. Ces erreurs peuvent inclure des incohérences de types, des références non définies, des opérations incorrectes, des conflits de portée,etc.

Dans notre projet, nous avons traité les six erreurs sémantiques suivantes :

1-Incompatibilité des types : Nous avons vérifié que les opérations et les affectations entre différents types de données sont cohérentes. Par exemple, si une variable est déclarée comme un entier, nous avons détecté les erreurs lorsque cette variable est utilisée dans une opération arithmétique avec un autre type incompatible, tel qu'une chaîne de caractères.

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\System32\cmd.exe'. The window has a black background with white text. The text inside shows a code snippet: 'INTEGER Z ;' followed by a closing brace '}', and then 'CODE { Z = 17.0 ;'. Below this, an error message is displayed: 'error semantique a la ligne 32 ,et a la colonne 21,-----type incompatible-----'. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

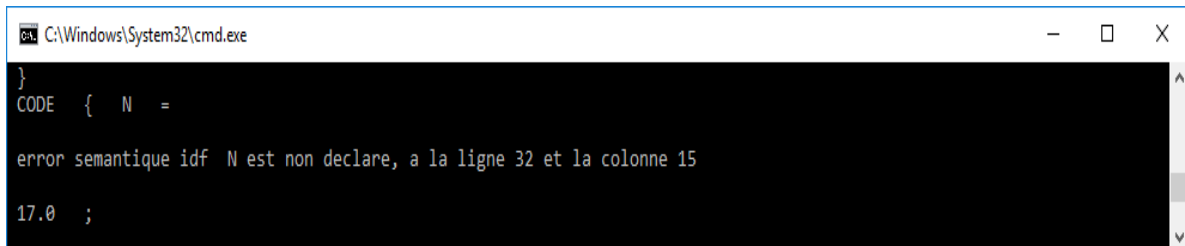
```
C:\Windows\System32\cmd.exe
INTEGER Z ;
}
CODE { Z = 17.0 ;
error semantique a la ligne 32 ,et a la colonne 21,-----type incompatible-----
```

explication :

Dans cet exemple, il y a une incompatibilité de types. La variable "Z" est déclarée comme étant de type "integer" (entier), mais on lui a attribué une valeur de type "float" (nombre à virgule flottante).

2-Utilisation de variables non déclarées : Nous avons vérifié que toutes les

variables utilisées dans le code ont été préalablement déclarées. Si une variable est utilisée sans avoir été déclarée, une erreur sémantique est signalée.



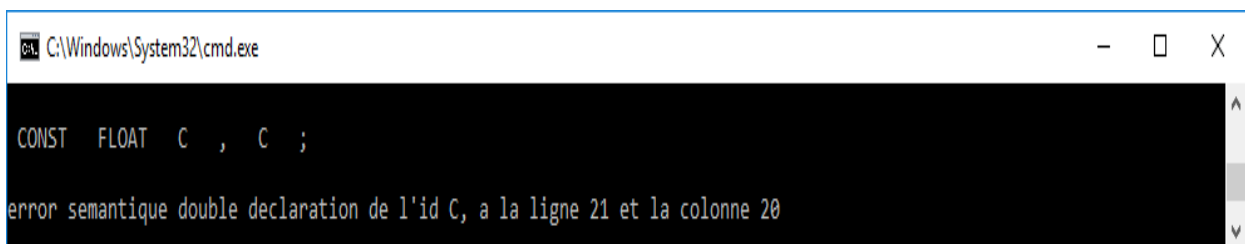
```
C:\Windows\System32\cmd.exe
}
CODE { N =
error semantique idf N est non declare, a la ligne 32 et la colonne 15
17.0 ;
```

explication :

L'erreur sémantique "idf N est non déclaré, à la ligne 32 et à la colonne 15" indique que l'identifiant "N" est utilisé sans avoir été préalablement déclaré dans le code à cet emplacement précis.

Lors de l'analyse sémantique, le compilateur vérifie si les identificateurs utilisés dans le code ont été déclarés auparavant. Si un identifiant est utilisé sans déclaration préalable, une erreur sémantique est signalée.

3-double déclaration d'une variable: Cette erreur se produit lorsqu'une variable est déclarée plusieurs fois dans la même portée.



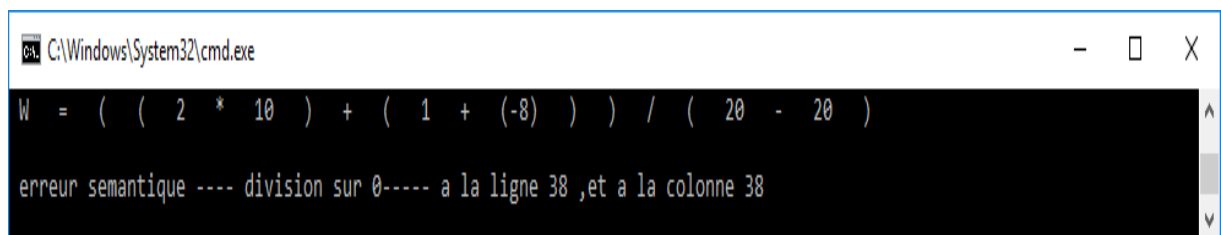
```
C:\Windows\System32\cmd.exe
CONST FLOAT C , C ;
error semantique double declaration de l'id C, a la ligne 21 et la colonne 20
```

explication :

L'erreur sémantique "double déclaration de l'id C, à la ligne 21 et à la colonne 20" indique qu'il y a une redéclaration de l'identifiant "C" à cet emplacement spécifique du code.

une variable ou un identificateur ne peut être déclaré qu'une seule fois dans la même portée. Si vous tentez de redéclarer un identifiant déjà déclaré, une erreur sémantique se produira.

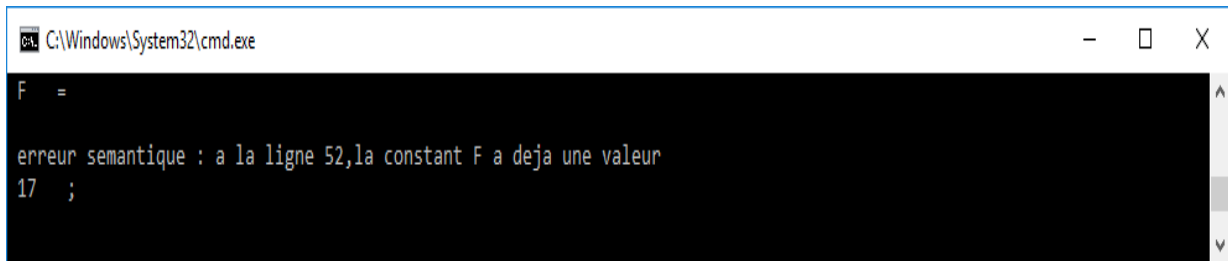
4-division par zéro : L'erreur sémantique "division par 0" se produit lorsqu'une tentative de division est effectuée avec un diviseur égal à zéro. Cette opération est mathématiquement invalide et génère une erreur sémantique car elle ne peut être évaluée correctement.



```
C:\Windows\System32\cmd.exe
W = ( ( 2 * 10 ) + ( 1 + (-8) ) ) / ( 20 - 20 )
erreur semantique ---- division sur 0----- a la ligne 38 ,et a la colonne 38
```

5-affecter une valeur à une constante: Les constantes sont des entités dont la valeur est fixe et ne peut pas être modifiée une fois qu'elle a été initialisée.

Si on va essayer d'affecter une valeur à une constante déjà définie, une erreur sémantique sera générée. Cela est dû au fait que les constantes sont conçues pour rester immuables pendant l'exécution du programme.

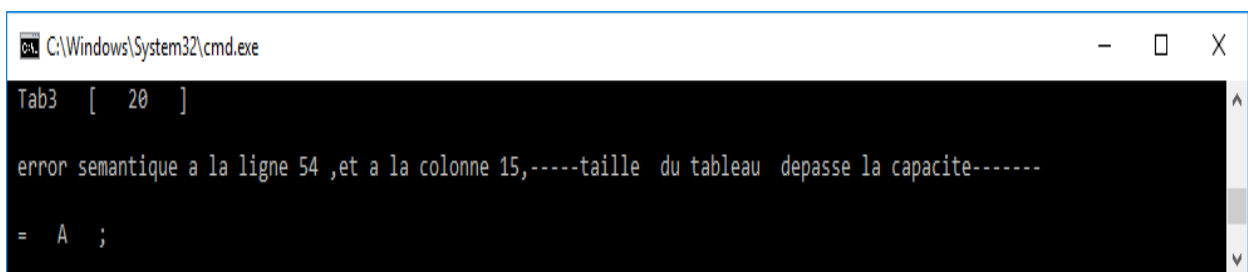


```
C:\Windows\System32\cmd.exe
F =
erreur semantique : a la ligne 52,la constant F a deja une valeur
17 ;
```

explication :

L'erreur sémantique que nous avons décrite à la ligne 52 indique qu'il y a une tentative de redéclaration ou de réaffectation d'une valeur à une constante "F" qui a déjà été déclarée et initialisée auparavant.

6-déplacement de la taille du tableau : Un tableau a une taille fixe déterminée lors de sa déclaration. Si on va essayer d'accéder à un indice en dehors des limites du tableau ou de modifier sa taille après sa déclaration, une erreur sémantique se produira.



```
C:\Windows\System32\cmd.exe
Tab3 [ 20 ]
error semantique a la ligne 54 ,et a la colonne 15,-----taille du tableau depasse la capacite-----
= A ;
```

explication:

L'erreur sémantique "taille du tableau dépasse la capacité" à la

ligne 54 et à la colonne 15 indique qu'il y a un dépassement de capacité lors de l'accès ou de la manipulation de tableau tab3.

4- Gestion de la table de symboles :

La gestion de la table de symboles est une composante essentielle d'un compilateur ou d'un interpréteur. La table de symboles est une structure de données utilisée pour stocker les informations sur les symboles (variables, fonctions, constantes, etc.) rencontrés lors de l'analyse d'un programme.

La table de symboles dans notre projet a été implémentée sous la forme d'une table de hachage. Chaque symbole est associé à des informations telles que son nom, son type, sa portée, sa valeur (le cas échéant), et d'autres attributs pertinents.

La gestion de la table de symboles comprend dans notre projet les opérations suivantes :

Déclaration : Lorsqu'un symbole est rencontré pour la première fois dans le code, il est déclaré et enregistré dans la table de symboles. Les informations associées au symbole, telles que son nom et son type, sont stockées.

Recherche : Lorsqu'un symbole est référencé dans le code, une recherche est effectuée dans la table de symboles pour trouver les informations associées à ce symbole. Cela permet de vérifier si le symbole est déclaré et d'obtenir ses informations pertinentes.

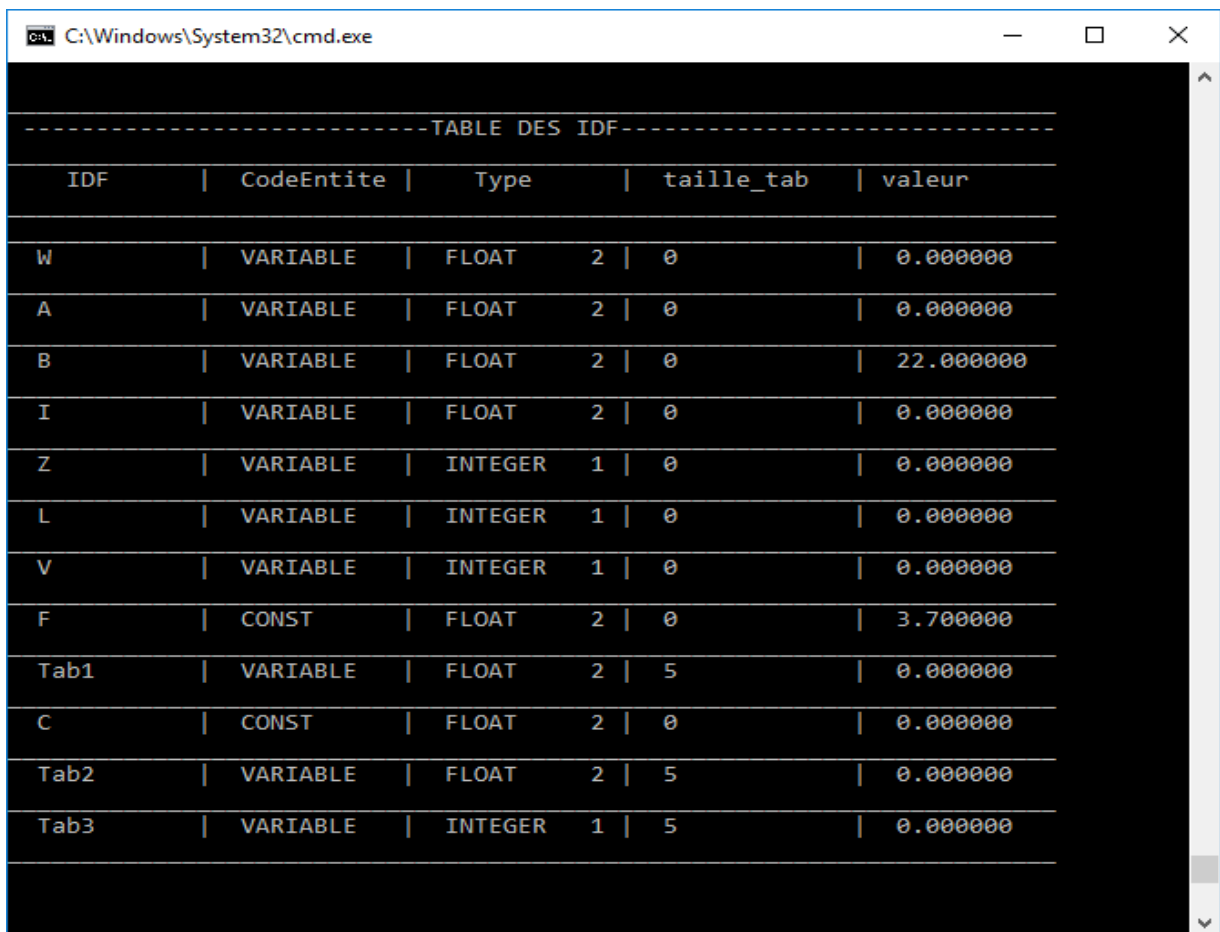
Mise à jour : Si nécessaire, les informations d'un symbole dans la table de symboles peuvent être mises à jour, par exemple lorsqu'une variable est affectée à une nouvelle valeur ou lorsqu'une fonction est redéfinie.

Portée : La table de symboles gère la portée des symboles, c'est-à-dire la visibilité des symboles dans différentes parties du code. Lorsqu'une portée est quittée, les symboles correspondants peuvent être supprimés de la table de symboles.

Gestion des erreurs : La table de symboles est également utilisée pour détecter les erreurs de déclaration, telles que les double déclarations de symboles ou les références à des symboles non déclarés.

La gestion efficace de la table de symboles est cruciale pour garantir l'analyse et l'exécution correctes d'un programme. Elle permet de maintenir les informations nécessaires sur les symboles tout au long du processus de compilation ou d'interprétation.

-----> voici la Ts des idfs de notre projet ainsi que la Ts des séparateurs et des mots clés



The screenshot shows a Windows command prompt window with the title bar "C:\Windows\System32\cmd.exe". The window contains a table titled "TABLE DES IDF" with the following data:

IDF	CodeEntite	Type	taille_tab	valeur
W	VARIABLE	Float	2 0	0.000000
A	VARIABLE	Float	2 0	0.000000
B	VARIABLE	Float	2 0	22.000000
I	VARIABLE	Float	2 0	0.000000
Z	VARIABLE	Integer	1 0	0.000000
L	VARIABLE	Integer	1 0	0.000000
V	VARIABLE	Integer	1 0	0.000000
F	CONST	Float	2 0	3.700000
Tab1	VARIABLE	Float	2 5	0.000000
C	CONST	Float	2 0	0.000000
Tab2	VARIABLE	Float	2 5	0.000000
Tab3	VARIABLE	Integer	1 5	0.000000

```

C:\Windows\System32\cmd.exe

/*****Table des symboles mots cl@s*****/

```

NomEntite	CodeEntite
VAR	Mot cle
CONST	Mot cle
FLOAT	Mot cle
INTEGER	Mot cle
=	Mot cle
CODE	Mot cle
FOR	Mot cle

```

/*****Table des symboles s@parateurs*****/

```

NomEntite	CodeEntite
{	Mot cle
;	Mot cle
:	Mot cle
[Mot cle
]	Mot cle
,	Mot cle
}	Mot cle
(Mot cle
)	Mot cle
*	Mot cle
-	Mot cle
+	Mot cle

5- Génération du code intermédiaire :

La génération du code intermédiaire sous forme de quadruples est une étape clé dans le processus de compilation d'un programme. Les quadruples sont une représentation intermédiaire du code qui facilite l'optimisation ultérieure et la génération du code final.

Un quadruple est une structure de données composée de quatre éléments :

Opération : Il s'agit de l'opération à effectuer, telle qu'une opération arithmétique (+, -, *, /), une affectation, un appel de

fonction, etc.

Opérande 1 : C'est le premier opérande de l'opération.

Opérande 2 : C'est le deuxième opérande de l'opération.

Résultat : C'est l'endroit où le résultat de l'opération est stocké.

La génération des quadruples se fait généralement en parcourant l'arbre syntaxique abstrait du programme, créé lors de l'analyse syntaxique. Lors de ce parcours, les nœuds de l'arbre sont traduits en quadruples en utilisant des règles prédéfinies.

----> Dans ce qui suit, nous allons illustrer chaque instruction de notre projet, en fournissant des explications détaillées

1- l'instruction if :

-----> voici l'exemple :

```
C:\Windows\System32\cmd.exe
C:\Users\DELL\Desktop\projet_compil>comp.exe < exemple_if.txt
FFFFFFFF {
VAR {
CONST FLOAT C ;
FLOAT : Tab1 [ 5 ] , Tab2 [ 5 ] ;
INTEGER : Tab3 [ 5 ] ;
CONST F = 3.7 ;
INTEGER L , V ;
INTEGER Z ;
FLOAT I ;
FLOAT W , A , B ;
}
CODE {

IF ( ( F == 3.7 ) ) {
W = ( ( 2 * 10 ) + ( 1 + (-8) ) ) / ( 1 - 3 ) + A ;
I = 5 + 5 ;
Tab1 [ L ] = W / ( 30.0 * 2 - ( 20.0 ) ) ;
}
ELSE {
B = ( ( 2 * 10 ) - (-2) ) + W ;
}
}
}
programme syntaxiquement correct
```

voici leur le code intermédiaire:

```
C:\Windows\System32\cmd.exe
----- les quadruplets -----
0-(BNE ,3 ,F ,3.700000 )
1-(:= ,1 , ,T1 )
2-(BR ,4 , , )
3-(:= ,0 , ,T1 )
4-(BZ ,19 ,T1 , )
5-( * ,2 ,10 ,T2 )
6-( + ,1 , -8 ,T3 )
7-( + ,T2 ,T3 ,T4 )
8-( - ,1 ,3 ,T5 )
9-( / ,T4 ,T5 ,T6 )
10-( + ,T6 ,A ,T7 )
11-(:= ,T7 , ,W )
12-( + ,5 ,5 ,T8 )
13-(:= ,T8 , ,Tab1[L] )
14-( * ,30.000000 ,2 ,T9 )
15-( - ,T9 ,20.000000 ,T10 )
16-( / ,W ,T10 ,T11 )
17-(:= ,T11 , ,Tab1[L] )
18-(BR ,23 , , )
19-( * ,2 ,10 ,T12 )
20-( - ,T12 , -2 ,T13 )
21-( + ,T13 ,W ,T14 )
22-(:= ,T14 , ,B )

-----Partie optimisation-----
```

explication :

0-(BNE, 3, F, 3.700000) : Cette instruction correspond à une branche conditionnelle (BNE) qui vérifie si la valeur stockée dans F est différente de 3.700000. Si la condition est vraie, elle effectue un saut à l'instruction 3.

1- (:=, 1, , T1) : Cette instruction effectue une assignation (:=) de la valeur 1 à la variable temporaire T1.

2- (BR, 4, ,) : Cette instruction effectue un branchement (BR) non conditionnel à l'instruction 4.

3- (:=, 0, , T1) : Cette instruction effectue une assignation de la valeur 0 à la variable temporaire T1.

4- (BZ, 19, T1,) : Cette instruction correspond à une branche conditionnelle (BZ) qui vérifie si la valeur stockée dans T1 est égale à zéro. Si la condition est vraie, elle effectue un saut à l'instruction 19.

5-(*, 2, 10, T2) : Cette instruction effectue une multiplication () entre les valeurs 2 et 10, et stocke le résultat dans la variable temporaire T2.

6-(+, 1, -8, T3) : Cette instruction effectue une addition (+) entre la valeur 1 et -8, et stocke le résultat dans la variable temporaire T3.

7-(+, T2, T3, T4) : Cette instruction effectue une addition entre les valeurs stockées dans T2 et T3, et stocke le résultat dans la variable temporaire T4.

8-(-, 1, 3, T5) : Cette instruction effectue une soustraction (-) entre la valeur 1 et 3, et stocke le résultat dans la variable temporaire T5.

9-(/, T4, T5, T6) : Cette instruction effectue une division (/) entre les valeurs stockées dans T4 et T5, et stocke le résultat dans la variable temporaire T6.

10-(+, T6, A, T7) : Cette instruction effectue une addition entre les valeurs stockées dans T6 et A, et stocke le résultat dans la variable temporaire T7.

11-(:=, T7, , W) : Cette instruction effectue une assignation de la valeur stockée dans T7 à la variable W.

12-(+, 5, 5, T8) : Cette instruction effectue une addition entre les valeurs 5 et 5, et stocke le résultat dans la variable temporaire T8.

13-(:=, T8, , Tab1[L]) : Cette instruction effectue une assignation de la valeur stockée dans T8 à l'élément L du tableau Tab1.

14-(*, 30.000000, 2, T9) : Cette instruction effectue une multiplication entre les valeurs 30.000000 et 2, et stocke le résultat dans la variable temporaire T9.

15-(-, T9, 20.000000, T10) : Cette instruction effectue une soustraction entre la valeur stockée dans T9 et 20.000000, et stocke le résultat dans la variable temporaire T10.

16-(/, W, T10, T11) : Cette instruction effectue une division entre les valeurs stockées dans W et T10, et stocke le résultat dans la variable temporaire T11.

17-(:=, T11, , Tab1[L]) : Cette instruction effectue une assignation de la valeur stockée dans T11 à l'élément L du tableau Tab1.

18-(BR, 23, ,) : Cette instruction effectue un branchement non conditionnel à l'instruction 23.

19-(*, 2, 10, T12) : Cette instruction effectue une multiplication entre les valeurs 2 et 10, et stocke le résultat dans la variable temporaire T12.

20-(-, T12, -2, T13) : Cette instruction effectue une soustraction entre la valeur stockée dans T12 et -2, et stocke le résultat dans la variable temporaire T13.

21-(+, T13, W, T14) : Cette instruction effectue une addition entre les valeurs stockées dans T13 et W, et stocke le résultat dans la variable temporaire T14.

22-(:=, T14, , B) : Cette instruction effectue une assignation de la valeur stockée dans T14 à la variable B.

2- l'instruction while :

-----> voici l'exemple :


```
C:\Windows\System32\cmd.exe
FFFFFFFF {
VAR {
CONST FLOAT C ;
FLOAT : Tab1 [ 5 ] , Tab2 [ 5 ] ;
INTEGER : Tab3 [ 5 ] ;
CONST F = 3.7 ;
INTEGER L , V ;
INTEGER Z ;
FLOAT I ;
FLOAT W , A , B ;
}
CODE {
W = 1.0 ;
A = 20.0 ;

WHILE ( ( ( ( A == 3 ) && ( W == 1.0 ) ) || ( A == 20.0 ) ) ) {
B = B + 2.0 ;
W = W + 1 ;
A = A - 1.0 ;
}
}
}
programme syntaxiquement correct
```

voici leur le code intermédiaire:

```
C:\Windows\System32\cmd.exe

----- les quadruplets -----
0-(:= ,1.000000 , ,W )
1-(:= ,20.000000 , ,A )
2-(BNE ,5 ,A ,3 )
3-(:= ,1 , ,T1 )
4-(BR ,6 , , )
5-(:= ,0 , ,T1 )
6-(BNE ,9 ,W ,1.000000 )
7-(:= ,1 , ,T2 )
8-(BR ,10 , , )
9-(:= ,0 , ,T2 )
10-(BZ ,14 ,T1 , )
11-(BZ ,14 ,T2 , )
12-(:= ,1 , ,T3 )
13-(BR ,15 , , )
14-(:= ,0 , ,T3 )
15-(BNE ,18 ,A ,20.000000 )
16-(:= ,1 , ,T4 )
17-(BR ,19 , , )
18-(:= ,0 , ,T4 )
19-(BNZ ,23 ,T3 , )
20-(BNZ ,23 ,T4 , )
21-(:= ,0 , ,T5 )
22-(BR ,24 , , )
23-(:= ,1 , ,T5 )
24-(+ ,B ,2.000000 ,T6 )
25-(:= ,T6 , ,B )
26-(+ ,W ,1 ,T7 )
27-(:= ,T7 , ,W )
28-(- ,A ,1.000000 ,T8 )
29-(:= ,T8 , ,A )
30-(BZ ,2 ,T5 , )
```

explication :

0-(:=, 1.000000, , W) : Cette instruction effectue une assignation de la valeur 1.000000 à la variable W.

1-(:=, 20.000000, , A) : Cette instruction effectue une assignation de la valeur 20.000000 à la variable A.

2-(BNE, 5, A, 3) : Cette instruction correspond à une branche

conditionnelle (BNE) qui vérifie si la valeur stockée dans A est différente de 3. Si la condition est vraie, elle effectue un saut à l'instruction 5.

3-(:=, 1, , T1) : Cette instruction effectue une assignation de la valeur 1 à la variable temporaire T1.

4-(BR, 6, ,) : Cette instruction effectue un branchement (BR) non conditionnel à l'instruction 6.

5-(:=, 0, , T1) : Cette instruction effectue une assignation de la valeur 0 à la variable temporaire T1.

6-(BNE, 9, W, 1.000000) : Cette instruction correspond à une branche conditionnelle (BNE) qui vérifie si la valeur stockée dans W est différente de 1.000000. Si la condition est vraie, elle effectue un saut à l'instruction 9.

7-(:=, 1, , T2) : Cette instruction effectue une assignation de la valeur 1 à la variable temporaire T2.

8-(BR, 10, ,) : Cette instruction effectue un branchement (BR) non conditionnel à l'instruction 10.

9-(:=, 0, , T2) : Cette instruction effectue une assignation de la valeur 0 à la variable temporaire T2.

10-(BZ, 14, T1,) : Cette instruction correspond à une branche conditionnelle (BZ) qui vérifie si la valeur stockée dans T1 est égale à zéro. Si la condition est vraie, elle effectue un saut à l'instruction 14.

11-(BZ, 14, T2,) : Cette instruction correspond à une branche conditionnelle (BZ) qui vérifie si la valeur stockée dans T2 est égale à zéro. Si la condition est vraie, elle effectue un saut à l'instruction 14.

12-(:=, 1, , T3) : Cette instruction effectue une assignation de la valeur 1 à la variable temporaire T3.

13-(BR, 15, ,) : Cette instruction effectue un branchement (BR) non conditionnel à l'instruction 15.

14-(:=, 0, , T3) : Cette instruction effectue une assignation de la valeur 0 à la variable temporaire T3.

15-(BNE, 18, A, 20.000000) : Cette instruction correspond à une branche conditionnelle (BNE) qui vérifie si la valeur stockée dans A est différente de 20.000000. Si la condition est vraie, elle effectue un saut à l'instruction 18.

16-(:=, 1, , T4) : Cette instruction effectue une assignation de la valeur 1 à la variable temporaire T4.

17-(BR, 19, ,) : Cette instruction effectue un branchement (BR) non conditionnel à l'instruction 19.

18-(:=, 0, , T4) : Cette instruction effectue une assignation de la valeur 0 à la variable temporaire T4.

19-(BNZ, 23, T3,) : Cette instruction correspond à une branche conditionnelle (BNZ) qui vérifie si la valeur stockée dans T3 est non nulle. Si la condition est vraie, elle effectue un saut à l'instruction 23.

20-(BNZ, 23, T4,) : Cette instruction correspond à une branche conditionnelle (BNZ) qui vérifie si la valeur stockée dans T4 est non nulle. Si la condition est vraie, elle effectue un saut à l'instruction 23.

21-(:=, 0, , T5) : Cette instruction effectue une assignation de la valeur 0 à la variable temporaire T5.

22-(BR, 24, ,) : Cette instruction effectue un branchement (BR) non conditionnel à l'instruction 24.

23-(:=, 1, , T5) : Cette instruction effectue une assignation de la valeur 1 à la variable temporaire T5.

24-(+, B, 2.000000, T6) : Cette instruction effectue une addition (+) entre la valeur stockée dans B et 2.000000, et stocke le résultat dans la variable temporaire T6.

25-(:=, T6, , B) : Cette instruction effectue une assignation de la valeur stockée dans T6 à la variable B.

26-(+, W, 1, T7) : Cette instruction effectue une addition entre les valeurs stockées dans W et 1, et stocke le résultat dans la variable temporaire T7.

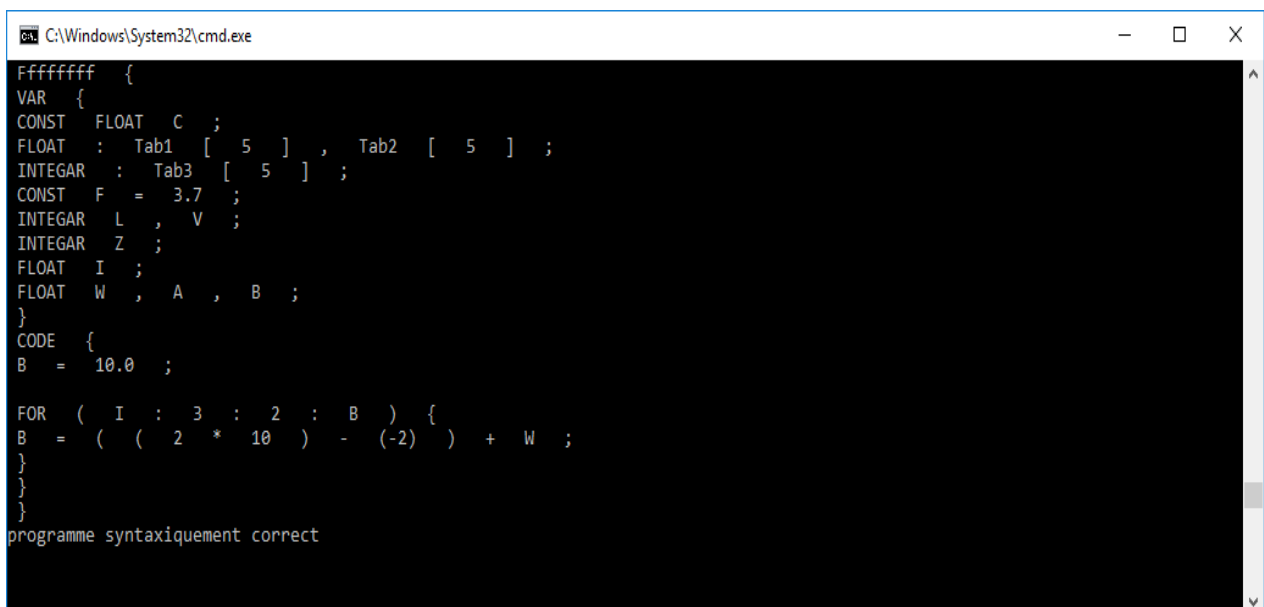
27-(:=, T7, , W) : Cette instruction effectue une assignation de la valeur stockée dans T7 à la variable W.

28-(-, A, 1.000000, T8) : Cette instruction effectue une soustraction (-) entre la valeur stockée dans A et 1.000000, et stocke le résultat dans la variable temporaire T8.

29-(:=, T8, , A) : Cette instruction effectue une assignation de la valeur stockée dans T8 à la variable A.

30-(BZ, 2, T5,) : Cette instruction correspond à une branche conditionnelle (BZ) qui vérifie si la valeur stockée dans T5 est égale à zéro. Si la condition est vraie, elle effectue un saut à l'instruction 2.

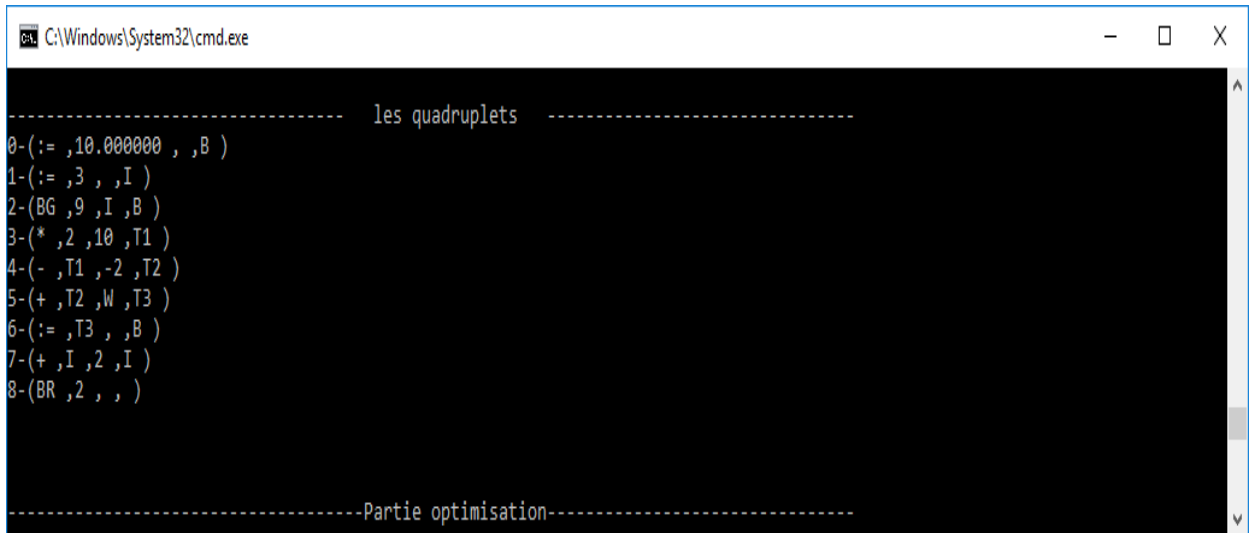
3- l'instruction for :



```
C:\Windows\System32\cmd.exe
FFFFFFFF {
VAR {
CONST  FLOAT  C  ;
FLOAT   :  Tab1  [  5  ] ,  Tab2  [  5  ]  ;
INTEGER :  Tab3  [  5  ]  ;
CONST  F  =  3.7  ;
INTEGER L  ,  V  ;
INTEGER Z  ;
FLOAT   I  ;
FLOAT  W  ,  A  ,  B  ;
}
CODE {
B  =  10.0  ;

FOR ( I  :  3  :  2  :  B ) {
B  =  ( (  2  *  10 ) - (-2) ) + W  ;
}
}
}
programme syntaxiquement correct
```

voici leur le code intermédiaire:



```
----- les quadruplets -----
0-(:= ,10.000000 , ,B )
1-(:= ,3 , ,I )
2-(BG ,9 ,I ,B )
3-( * ,2 ,10 ,T1 )
4-(- ,T1 , -2 ,T2 )
5-(+ ,T2 ,W ,T3 )
6-(:= ,T3 , ,B )
7-(+ ,I ,2 ,I )
8-(BR ,2 , , )

-----Partie optimisation-----
```

explication :

1-(:=, 10.000000, , B) : Cette instruction effectue une assignation de la valeur 10.000000 à la variable B.

2-(:=, 3, , I) : Cette instruction effectue une assignation de la valeur 3 à la variable I.

3-(BG, 9, I, B) : Cette instruction correspond à une branche conditionnelle (BG) qui vérifie si la valeur stockée dans I est supérieure à la valeur stockée dans B. Si la condition est vraie, elle effectue un saut à l'instruction 9.

4-(*, 2, 10, T1) : Cette instruction effectue une multiplication () entre les valeurs 2 et 10, et stocke le résultat dans la variable temporaire T1.

5-(-, T1, -2, T2) : Cette instruction effectue une soustraction (-) entre la valeur stockée dans T1 et -2, et stocke le résultat dans la variable temporaire T2.

6-(+, T2, W, T3) : Cette instruction effectue une addition (+) entre les valeurs stockées dans T2 et W, et stocke le résultat dans la variable temporaire T3.

7-(:=, T3, , B) : Cette instruction effectue une assignation de la valeur stockée dans T3 à la variable B.

8-(+, I, 2, I) : Cette instruction effectue une addition (+) entre la valeur stockée dans I et 2, et stocke le résultat dans la variable I.

9-(BR, 2, ,) : Cette instruction effectue un branchement (BR) non conditionnel à l'instruction 2.

6- optimisation:

L'optimisation du code intermédiaire est une étape importante dans le processus de compilation. Elle vise à améliorer l'efficacité

du code en réduisant sa complexité et en optimisant son exécution. dans notre projet on a utiliser quelques techniques couramment pour optimiser le code intermédiaire :

1-La propagation de copie :La propagation de copie (copy propagation en anglais) est une technique d'optimisation du code qui vise à réduire l'utilisation de variables temporaires en remplaçant les copies inutiles par des références directes aux valeurs d'origine

2- La propagation des expressions : La propagation d'expression est une technique d'optimisation du code qui vise à réduire les calculs redondants en remplaçant les expressions par leurs résultats déjà calculés.

3-L'élimination d'expressions redondantes : L'élimination d'expressions redondantes est une technique d'optimisation du code qui vise à supprimer les expressions dont le résultat est déjà connu ou qui n'ont aucun effet sur le programme.

Lors de la compilation, il peut arriver que certaines expressions soient calculées plusieurs fois sans apporter de nouvelle information ou sans modifier l'état du programme. Ces expressions redondantes peuvent ralentir l'exécution du programme et gaspiller des ressources.

4-La simplification algébrique: La simplification algébrique est une technique d'optimisation du code qui vise à réduire les

expressions mathématiques complexes en les transformant en formes plus simples et équivalentes.

Lors de la compilation, il peut arriver que des expressions mathématiques contiennent des termes redondants, des opérations inutiles ou des formes qui peuvent être simplifiées. La simplification algébrique permet d'optimiser ces expressions en les réécrivant de manière plus concise et efficace.

5-L'élimination de code inutile : L'élimination de code inutile est une technique d'optimisation du code qui consiste à supprimer les parties de code qui ne sont pas utilisées ou qui n'ont aucun impact sur le comportement du programme.

Lors de la compilation, il est courant d'avoir du code qui a été écrit mais qui n'est plus nécessaire. Cela peut être dû à des changements dans la logique du programme, à des conditions qui ne sont jamais satisfaites, à des variables qui ne sont plus utilisées, ou à d'autres parties de code qui ont été modifiées ou supprimées.

on va prendre un exemple dans notre projet :

```
C:\Windows\System32\cmd.exe
C:\Users\DELL\Desktop\projet_compil>comp.exe < exemple_optimisation.txt
FFFFFFFF {
VAR {
FLOAT t6 , t9 , temp ;
FLOAT t10 , t11 , t12 , t13 ;
FLOAT t8 , ttttt ;
FLOAT t14 , t15 , t16 , t17 , t18 ;
FLOAT : A [ 100 ] ;
CONST j1 = 10 ;
}
CODE { t6 = 4 * j1 ;
t8 = j1 - 1 ;
t9 = 2 * t8 ;
temp = A [ t9 ] ;
t10 = j1 + 1 ;
t11 = t10 - 1 ;
t16 = 10.0 * 4.0 ;
t12 = 4.0 + ( 5.0 * 10.0 ) ; %% ceci est un commentaire

t13 = A [ t12 ] ;
t14 = j1 - 1 ;
t15 = 4 * t14 ;
A [ t15 ] = t13 ;
t16 = j1 + 1 ;
t17 = 1 - 1 ;
t18 = 4 * t17 ;
A [ t18 ] = temp ;
}
}
programme syntaxiquement correct
```

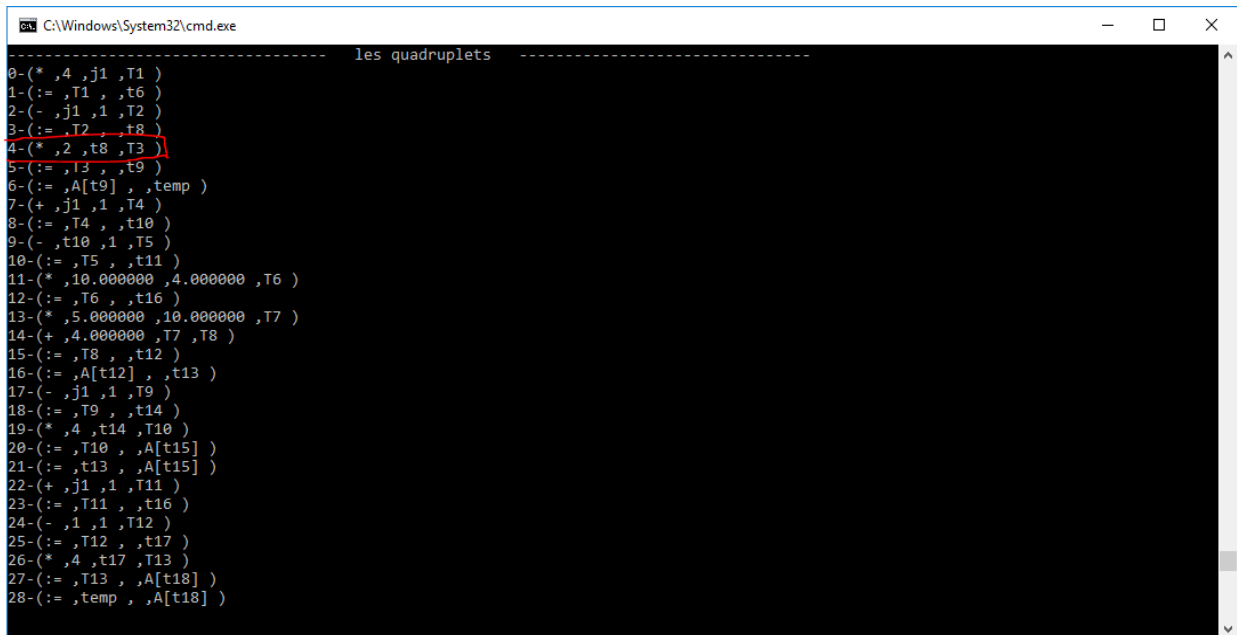
-----> la partie quadruplets

```
C:\Windows\System32\cmd.exe
----- les quadruplets -----
0-( * ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-(:= ,T2 , ,t8 )
4-( * ,2 ,t8 ,T3 )
5-(:= ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-(:= ,T4 , ,t10 )
9-(- ,t10 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-( * ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-( * ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-(:= ,T9 , ,t14 )
19-( * ,4 ,t14 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-(- ,1 ,1 ,T12 )
25-(:= ,T12 , ,t17 )
26-( * ,4 ,t17 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

1- verification 2*X:

avant

optimisation



```
C:\Windows\System32\cmd.exe
----- les quadruplets -----
0-(* ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-(:= ,T2 , ,t8 )
4-(* ,2 ,t8 ,T3 )
5-(:= ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-(:= ,T4 , ,t10 )
9-(- ,t10 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-(* ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-(* ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-(:= ,T9 , ,t14 )
19-(* ,4 ,t14 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-(- ,1 ,1 ,T12 )
25-(:= ,T12 , ,t17 )
26-(* ,4 ,t17 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

après

optimisation

```
C:\Windows\System32\cmd.exe
-----Verification 2*X-----
0-(* ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-(:= ,T2 , ,t8 )
4-(+ ,t8 ,t8 ,T3 )
5-(- ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-(:= ,T4 , ,t10 )
9-(- ,t10 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-(* ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-(* ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-(:= ,T9 , ,t14 )
19-(* ,4 ,t14 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-(- ,1 ,1 ,T12 )
25-(:= ,T12 , ,t17 )
26-(* ,4 ,t17 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

2- Simplification algébrique:

avant

optimisation

```
C:\Windows\System32\cmd.exe
----- les quadruplets -----
0-(* ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-(:= ,T2 , ,t8 )
4-(* ,2 ,t8 ,T3 )
5-(:= ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-(:= ,T4 , ,t10 )
9-(- ,t10 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-(* ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-(* ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-(:= ,T9 , ,t14 )
19-(* ,4 ,t14 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-(- ,1 ,1 ,T12 )
25-(:= ,T12 , ,t17 )
26-(* ,4 ,t17 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

optimisation

après

```
C:\Windows\System32\cmd.exe

-----Simplification alg|@brique-----
0-(* ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-(:= ,T2 , ,t8 )
4-(+ ,t8 ,t8 ,T3 )
5-(:= ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-(:= ,T4 , ,t10 )
9-(- ,t10 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-(* ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-(* ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-(:= ,T9 , ,t14 )
19-(* ,4 ,t14 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-( , , )
25-(:= ,t12 , ,t17 )
26-(* ,4 ,t17 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

3- le code mort:

avant

optimisation

```
C:\Windows\System32\cmd.exe
C:\Users\DELL\Desktop\projet_compil>comp.exe < exemple_optimisation.txt
Ffffffff {
VAR {
  FLOAT t6 , t9 , temp ;
  FLOAT t10 , t11 , t12 , t13 ;
  FLOAT t8 , ttttt ;
  FLOAT t14 , t15 , t16 , t17 , t18 ;
  FLOAT : A [ 100 ] ;
  CONST j1 = 10 ;
}
CODE { t6 = 4 * j1 ;
t8 = j1 - 1 ;
t9 = 2 * t8 ;
temp = A [ t9 ] ;
t10 = j1 + 1 ;
t11 = t10 - 1 ;
t16 = 10.0 * 4.0 ;
t12 = 4.0 + ( 5.0 * 10.0 ) ; %% ceci est un commentaire

t13 = A [ t12 ] ;
t14 = j1 - 1 ;
t15 = 4 * t14 ;
A [ t15 ] = t13 ;
t16 = j1 + 1 ;
t17 = 1 - 1 ;
t18 = 4 * t17 ;
A [ t18 ] = temp ;
}
}
programme syntaxiquement correct
```

optimisation

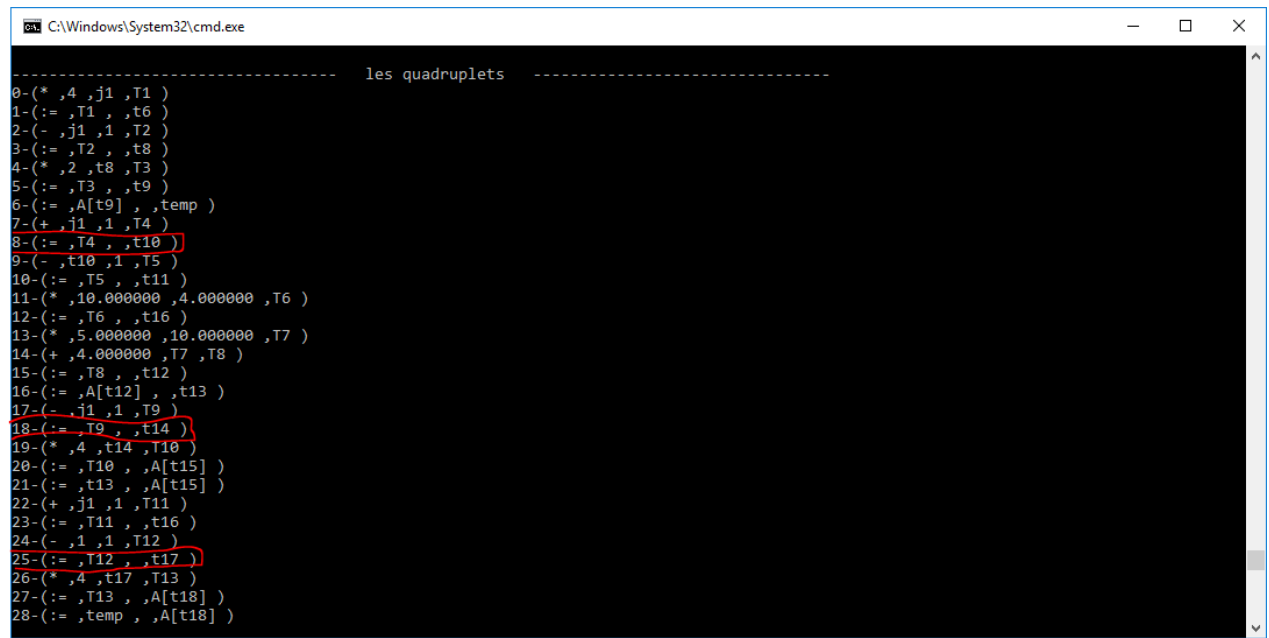
après

```
C:\Windows\System32\cmd.exe
----- Les IDSs non utilis|es -----
!!! Attention !!! : IDF ( ttttt ) declare et non utiliser
0-( * , 4 , j1 , T1 )
1-( := , T1 , , t6 )
2-( - , j1 , 1 , T2 )
3-( := , T2 , , t8 )
4-( + , t8 , t8 , T3 )
5-( := , T3 , , t9 )
6-( := , A[t9] , , temp )
7-( + , j1 , 1 , T4 )
8-( := , T4 , , t10 )
9-( - , t10 , 1 , T5 )
10-( := , T5 , , t11 )
11-( * , 10.000000 , 4.000000 , T6 )
12-( := , T6 , , t16 )
13-( * , 5.000000 , 10.000000 , T7 )
14-( + , 4.000000 , T7 , T8 )
15-( := , T8 , , t12 )
16-( := , A[t12] , , t13 )
17-( - , j1 , 1 , T9 )
18-( := , T9 , , t14 )
19-( * , 4 , t14 , T10 )
20-( := , T10 , , A[t15] )
21-( := , t13 , , A[t15] )
22-( + , j1 , 1 , T11 )
23-( := , T11 , , t16 )
24-( , , , )
25-( := , T12 , , t17 )
26-( * , 4 , t17 , T13 )
27-( := , T13 , , A[t18] )
28-( := , temp , , A[t18] )
```


4- propagation de copie-:

avant

optimisation



```
C:\Windows\System32\cmd.exe

----- les quadruplets -----
0-(* ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-(:= ,T2 , ,t8 )
4-(* ,2 ,t8 ,T3 )
5-(:= ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-(:= ,T4 , ,t10 )
9-(- ,t10 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-(* ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-(* ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-(:= ,T9 , ,t14 )
19-(* ,4 ,t14 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-(- ,1 ,1 ,T12 )
25-(:= ,T12 , ,t17 )
26-(* ,4 ,t17 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

après

optimisation

```
C:\Windows\System32\cmd.exe

-----propagation de copie-----
0-(* ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-( , , , )
4-(+ ,T2 ,t8 ,T3 )
5-(:= ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-( , , , )
9-(- ,T4 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-(* ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-(* ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-( , , , )
19-(* ,4 ,T9 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-( , , , )
25-( , , , )
26-(* ,4 ,T12 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

5- propagation arithemtique-:

optimisation

avant

```
C:\Windows\System32\cmd.exe

----- les quadruplets -----
0-(* ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-(:= ,T2 , ,t8 )
4-(* ,2 ,t8 ,T3 )
5-(:= ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-(:= ,T4 , ,t10 )
9-(- ,t10 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-(* ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-(* ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-(:= ,T9 , ,t14 )
19-(* ,4 ,t14 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-(- ,1 ,1 ,T12 )
25-(:= ,T12 , ,t17 )
26-(* ,4 ,t17 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

optimisation

après

```
C:\Windows\System32\cmd.exe

----- les quadruplets -----
0-(* ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-(:= ,T2 , ,t8 )
4-(* ,2 ,t8 ,T3 )
5-(:= ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-(:= ,T4 , ,t10 )
9-(- ,t10 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-(* ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-(* ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-(:= ,T9 , ,t14 )
19-(* ,4 ,t14 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-(- ,1 ,1 ,T12 )
25-(:= ,T12 , ,t17 )
26-(* ,4 ,t17 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

6- elimination des tempons repetees:

optimisation

avant

```
C:\Windows\System32\cmd.exe

----- les quadruplets -----
0-(* ,4 ,j1 ,T1 )
1-(:= ,T1 , ,t6 )
2-(- ,j1 ,1 ,T2 )
3-(:= ,T2 , ,t8 )
4-(* ,2 ,t8 ,T3 )
5-(:= ,T3 , ,t9 )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-(:= ,T4 , ,t10 )
9-(- ,t10 ,1 ,T5 )
10-(:= ,T5 , ,t11 )
11-(* ,10.000000 ,4.000000 ,T6 )
12-(:= ,T6 , ,t16 )
13-(* ,5.000000 ,10.000000 ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-(:= ,T8 , ,t12 )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-(:= ,T9 , ,t14 )
19-(* ,4 ,t14 ,T10 )
20-(:= ,T10 , ,A[t15] )
21-(:= ,t13 , ,A[t15] )
22-(+ ,j1 ,1 ,T11 )
23-(:= ,T11 , ,t16 )
24-(- ,1 ,1 ,T12 )
25-(:= ,T12 , ,t17 )
26-(* ,4 ,t17 ,T13 )
27-(:= ,T13 , ,A[t18] )
28-(:= ,temp , ,A[t18] )
```

après

optimisation

```
C:\Windows\System32\cmd.exe
-----elimination des tempons repetees-----
0-(* ,4 ,j1 ,T1 )
1-( , , , )
2-(- ,j1 ,1 ,T2 )
3-( , , , )
4-(+ ,T2 ,t8 ,T3 )
5-( , , , )
6-(:= ,A[t9] , ,temp )
7-(+ ,j1 ,1 ,T4 )
8-( , , , )
9-(- ,T4 ,1 ,T5 )
10-( , , , )
11-(:= ,40.00 , ,T6 )
12-( , , , )
13-(:= ,50.00 , ,T7 )
14-(+ ,4.000000 ,T7 ,T8 )
15-( , , , )
16-(:= ,A[t12] , ,t13 )
17-(- ,j1 ,1 ,T9 )
18-( , , , )
19-(* ,4 ,T9 ,T10 )
20-( , , , )
21-( , , , )
22-(+ ,j1 ,1 ,T11 )
23-( , , , )
24-( , , , )
25-( , , , )
26-(* ,4 ,T12 ,T13 )
27-( , , , )
28-( , , , )

Activer Windows
Accédez aux paramètres pour activer Wi
```

-----> Jusqu'à maintenant, nous avons expliqué chaque fonction individuellement pour vous montrer leur utilité respective. Dans la suite, nous allons appliquer ces fonctions à plusieurs reprises lors d'un balayage complet, puis afficher les résultats

balyage1

```
C:\Windows\System32\cmd.exe

----> (balyage num 1 ) <----

0-( * ,4 ,j1 ,T1 )
1-(- ,j1 ,1 ,T2 )
2-( + ,T2 ,t8 ,T3 )
3-(:= ,A[t9] , ,temp )
4-( + ,j1 ,1 ,T4 )
5-(- ,T4 ,1 ,T5 )
6-(:= ,40.00 , ,T6 )
7-( , , , )
8-( + ,4.000000 ,50.00 ,T8 )
9-(:= ,A[t12] , ,t13 )
10-(- ,j1 ,1 ,T9 )
11-( * ,4 ,T9 ,T10 )
12-( , , , )
13-( + ,j1 ,1 ,T11 )
14-( , , , )
15-( * ,4 ,T12 ,T13 )
16-( , , , )
```

balyage2

```
C:\Windows\System32\cmd.exe

----> (balyage num 2 ) <----

0-( * ,4 ,j1 ,T1 )
1-(- ,j1 ,1 ,T2 )
2-( + ,T2 ,t8 ,T3 )
3-(:= ,A[t9] , ,temp )
4-( + ,j1 ,1 ,T4 )
5-(- ,T4 ,1 ,T5 )
6-(:= ,40.00 , ,T6 )
7-(:= ,54.00 , ,T8 )
8-(:= ,A[t12] , ,t13 )
9-(- ,j1 ,1 ,T9 )
10-( * ,4 ,T9 ,T10 )
11-( + ,j1 ,1 ,T11 )
12-( * ,4 ,T12 ,T13 )
```

balyage3

```
C:\Windows\System32\cmd.exe

----> (balyage num 3 ) <----

0-(* ,4 ,j1 ,T1 )
1-(- ,j1 ,1 ,T2 )
2-(+ ,T2 ,t8 ,T3 )
3-(:= ,A[t9] , ,temp )
4-(+ ,j1 ,1 ,T4 )
5-(- ,T4 ,1 ,T5 )
6-(:= ,40.00 , ,T6 )
7-(:= ,54.00 , ,T8 )
8-(:= ,A[t12] , ,t13 )
9-(- ,j1 ,1 ,T9 )
10-(* ,4 ,T9 ,T10 )
11-(+ ,j1 ,1 ,T11 )
12-(* ,4 ,T12 ,T13 )
```

final

optimisation

```
C:\Windows\System32\cmd.exe

***** bravo! bravo! bravo! objectif atteint vous etes les meilleurs *****

-----
0-(* ,4 ,j1 ,T1 )
1-(- ,j1 ,1 ,T2 )
2-(+ ,T2 ,t8 ,T3 )
3-(:= ,A[t9] , ,temp )
4-(+ ,j1 ,1 ,T4 )
5-(- ,T4 ,1 ,T5 )
6-(:= ,40.00 , ,T6 )
7-(:= ,54.00 , ,T8 )
8-(:= ,A[t12] , ,t13 )
9-(- ,j1 ,1 ,T9 )
10-(* ,4 ,T9 ,T10 )
11-(+ ,j1 ,1 ,T11 )
12-(* ,4 ,T12 ,T13 )
-----
```

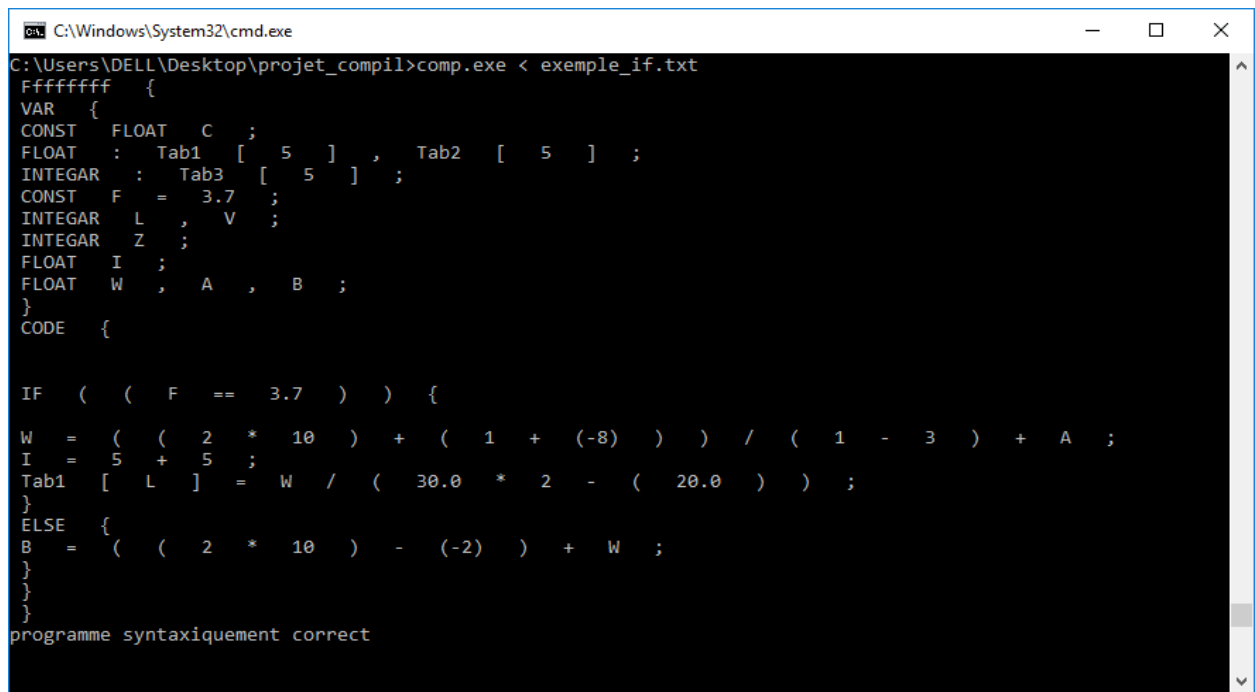
7- code machine:

Le langage assembleur utilise une représentation mnémonique pour chaque instruction du processeur. Chaque instruction

assembleur correspond généralement à une instruction machine unique exécutée par le processeur. Le langage assembleur permet de manipuler les registres du processeur, d'accéder à la mémoire, de réaliser des opérations arithmétiques et logiques, de gérer les branchements conditionnels et les boucles, et bien plus encore.

Dans notre projet, nous allons détailler chaque instruction individuellement.

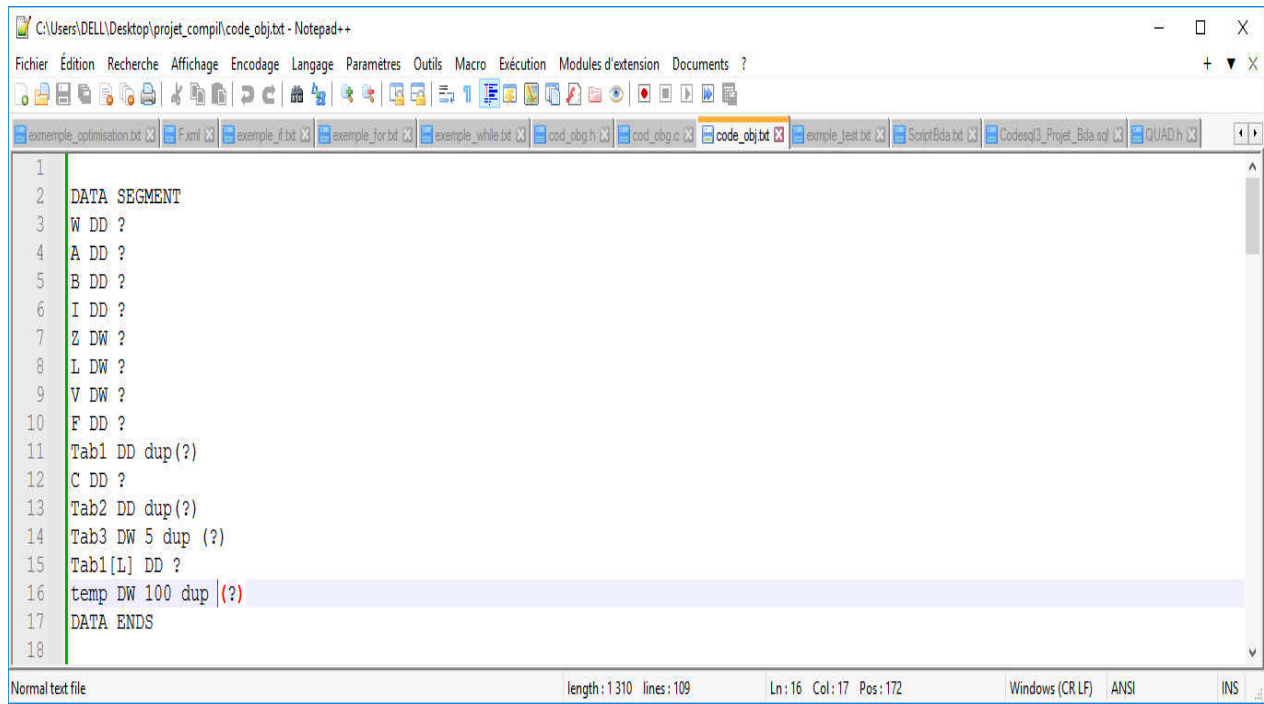
1- l'instruction if : _____



```
C:\Windows\System32\cmd.exe
C:\Users\DELL\Desktop\projet_compil>comp.exe < exemple_if.txt
Ffffffff {
VAR {
CONST FLOAT C ;
FLOAT : Tab1 [ 5 ] , Tab2 [ 5 ] ;
INTEGER : Tab3 [ 5 ] ;
CONST F = 3.7 ;
INTEGER L , V ;
INTEGER Z ;
FLOAT I ;
FLOAT W , A , B ;
}
CODE {

IF ( ( F == 3.7 ) ) {
W = ( ( 2 * 10 ) + ( 1 + (-8) ) ) / ( 1 - 3 ) + A ;
I = 5 + 5 ;
Tab1 [ L ] = W / ( 30.0 * 2 - ( 20.0 ) ) ;
}
ELSE {
B = ( ( 2 * 10 ) - (-2) ) + W ;
}
}
}
programme syntaxiquement correct
```

le code machine correspondent :



```
1  
2 DATA SEGMENT  
3 W DD ?  
4 A DD ?  
5 B DD ?  
6 I DD ?  
7 Z DW ?  
8 L DW ?  
9 V DW ?  
10 F DD ?  
11 Tab1 DD dup(?)  
12 C DD ?  
13 Tab2 DD dup(?)  
14 Tab3 DW 5 dup (?)  
15 Tab1[L] DD ?  
16 temp DW 100 dup (?)  
17 DATA ENDS  
18
```

"W DD ?" : déclare une variable nommée "W" de type double word (32 bits).

"A DD ?" : déclare une variable nommée "A" de type double word.

"B DD ?" : déclare une variable nommée "B" de type double word.

"I DD ?" : déclare une variable nommée "I" de type double word.

"Z DW ?" : déclare une variable nommée "Z" de type word (16 bits).

"L DW ?" : déclare une variable nommée "L" de type word.

"V DW ?" : déclare une variable nommée "V" de type word.

"F DD ?" : déclare une variable nommée "F" de type double word.

"Tab1 DD dup(?)" : déclare un tableau nommé "Tab1" de type double word avec une taille non spécifiée (indiquée par "?").

"C DD ?" : déclare une variable nommée "C" de type double word.

"Tab2 DD dup(?)" : déclare un tableau nommé "Tab2" de type double word avec une taille non spécifiée.

"Tab3 DW 5 dup (?)" : déclare un tableau nommé "Tab3" de type word avec une taille de 5 éléments, chaque élément initialisé avec une valeur non spécifiée.

"Tab1[L] DD ?" : déclare un élément spécifique "Tab1[L]" du tableau "Tab1" de type double word avec une valeur non spécifiée.

"temp DW 100 dup (?)" : déclare un tableau nommé "temp" de type word avec une taille de 100 éléments, chaque élément initialisé avec une valeur non spécifiée.

Ces déclarations permettent de réserver de l'espace en mémoire pour stocker les données nécessaires à l'exécution du programme.

C:\Users\DELL\Desktop\projet_compil\code_obj.txt - Notepad++

Fichier Edition Recherche Affichage Encodage Langage Paramètres Outils Macro Exécution Modules d'extension Documents ?

exemple_optimisation.txt F.xml exemple_f.txt exemple_for.txt exemple_while.txt cod_obj.h cod_obj.c code_obj.txt exemple_test.txt ScriptBda.txt Codesql3_Projet_Sda.sql QUAD.h

```
19 CODE SEGMENT
20 ASSUME CS:CODE, DS:DATA
21 MAIN :
22 MOV AX,DATA
23 MOV DS,AX
24
25 MOV AX, F
26 CMP AX,3.700000
27 JNE etiquette 3
28 MOV AX,
29 CMP AX,
30 JMP etiquette 4
31 MOV SI,NOT
32 ADD SI,SI
33 MOV AX,t[SI]
34 MOV T1,AX
35 etiquette3:MOV SI,1
36 ADD SI,SI
37 MOV AX, t[SI]
38 CMP AX,
39 JZ etiquette 19
40 etiquette4:MOV AX, 1
41 ADD AX, -8
42 MOV SI,3
43 ADD SI,SI
44 MOV t[SI],AX
45 MOV SI,3
46 ADD SI,SI
47 MOV AX, t[SI]
48 ADD AX, 20
```

Normal text file length : 1 310 lines : 109 Ln : 16 Col : 17 Pos : 172 Windows (CR LF) ANSI INS

Activer Windows
Accédez aux paramètres pour activer Windows.

C:\Users\DELL\Desktop\projet_compil\code_obj.txt - Notepad++

Fichier Edition Recherche Affichage Encodage Langage Paramètres Outils Macro Exécution Modules d'extension Documents ?

exemple_optimisation.txt Fxnl exemple_f.txt exemple_for.txt exemple_while.txt cod_obj.h cod_obj.c code_obj.txt exemple_test.txt ScriptBda.txt Codesql3_Projet_Bda.sql QUAD.h

```
48 ADD AX, 20
49 MOV SI, 4
50 ADD SI, SI
51 MOV t[SI], AX
52 MOV SI, 4
53 ADD SI, SI
54 MOV AX, t[SI]
55 DIV AX, 2
56 MOV SI, 6
57 ADD SI, SI
58 MOV t[SI], AX
59 MOV SI, 6
60 ADD SI, SI
61 MOV AX, t[SI]
62 ADD AX, A
63 MOV SI, 7
64 ADD SI, SI
65 MOV t[SI], AX
66 MOV SI, 0
67 ADD SI, SI
68 MOV AX, t[SI]
69 MOV T0, AX
70 MOV AX, 30.000000
71 MUL AX, 2
72 MOV SI, 9
73 ADD SI, SI
74 MOV t[SI], AX
75 MOV SI, 9
76 ADD SI, SI
77 MOV AX, t[SI]
```

Activer Windows
Accédez aux paramètres pour activer Windows.

Normal text file length : 1 310 lines : 109 Ln : 16 Col : 17 Pos : 172 Windows (CR LF) ANSI INS

```
77 MOV AX, t[SI]
78 SUB AX, 20.000000
79 MOV SI, 1
80 ADD SI, SI
81 MOV t[SI], AX
82 MOV SI, 7
83 ADD SI, SI
84 MOV AX, t[SI]
85 DIV AX, t[1]
86 MOV SI, 1
87 ADD SI, SI
88 MOV t[SI], AX
89 MOV AX,
90 CMP AX,
91 JMP etiquette 23
92 MOV AX, 20
93 SUB AX, -2
94 MOV SI, 1
95 ADD SI, SI
96 MOV t[SI], AX
97 MOV SI, 1
98 ADD SI, SI
99 MOV AX, t[SI]
100 ADD AX, W
101 MOV SI, 1
102 ADD SI, SI
103 MOV t[SI], AX
104 FIN :
105 MOV AH, 4CH
106 INT 21h
```

interprétation :

CODE SEGMENT

ASSUME CS:CODE, DS:DATA

MAIN:

MOV AX, DATA

MOV DS, AX

Cette partie du code est située dans la section CODE SEGMENT.
Elle commence par l'instruction ASSUME, qui indique au
programme comment les segments de code et de données sont

associés aux registres segment et de données respectivement.

Ensuite, nous avons l'étiquette MAIN, qui marque le point d'entrée du programme. À partir de là, les instructions suivantes sont exécutées séquentiellement.

La première instruction est MOV AX, DATA. Elle déplace la valeur du segment de données dans le registre AX. Ici, le segment de données est associé au segment DS (registre de données). Ainsi, cette instruction initialise le registre DS avec la valeur du segment de données.

La deuxième instruction est MOV DS, AX. Elle déplace la valeur du registre AX dans le registre DS, finalisant ainsi l'initialisation du registre DS avec le segment de données.

Ces instructions sont généralement présentes au début d'un programme pour établir la correspondance entre les segments de code et de données et les registres appropriés, permettant ainsi un accès correct à la mémoire et aux données du programme.

MOV AX, F

CMP AX, 3.700000

JNE etiquette3

MOV AX, ?

CMP AX, ?

JMP etiquette4

Ces instructions chargent la valeur de F dans AX, comparent AX avec 3.700000 et effectuent un saut à etiquette3 si la

comparaison est fausse (JNE signifie "jump if not equal"). Sinon, elles exécutent les instructions suivantes.

etiquette3:

MOV SI, 1

ADD SI, SI

MOV AX, t[SI]

CMP AX, ?

JZ etiquette19

Ces instructions définissent une étiquette etiquette3. Elles déplacent 1 dans le registre SI, effectuent une opération de décalage à gauche sur SI (ADD SI, SI), chargent la valeur de t[SI] dans AX, comparent AX avec une valeur manquante (?) et effectuent un saut à etiquette19 si la comparaison est vraie (JZ signifie "jump if zero").

etiquette4:

MOV AX, 1

ADD AX, -8

MOV SI, 3

ADD SI, SI

MOV t[SI], AX

MOV SI, 3

ADD SI, SI

MOV AX, t[SI]

ADD AX, 20

MOV SI, 4

ADD SI, SI

MOV t[SI], AX

MOV SI, 4

ADD SI, SI

MOV AX, t[SI]

DIV AX, 2

MOV SI, 6

ADD SI, SI

MOV t[SI], AX

MOV SI, 6

ADD SI, SI

MOV AX, t[SI]

ADD AX, A

MOV SI, 7

ADD SI, SI

MOV t[SI], AX

Ces instructions définissent une étiquette `etiquette4`. Elles chargent la valeur 1 dans AX, ajoutent -8 à AX, effectuent des opérations sur t en utilisant les indices 3, 4 et 6, et stockent le résultat dans t à différents emplacements.

MOV AX, ?

CMP AX, ?

JMP `etiquette23`

Ces instructions chargent une valeur manquante (?) dans AX, effectuent une comparaison de AX avec une autre valeur manquante (?) et effectuent un saut à `etiquette23`.

`etiquette19`:

MOV AX, 20

SUB AX, -2

MOV SI, 1

ADD SI, SI

MOV t[SI], AX

MOV SI, 1

ADD SI, SI

MOV AX, t[SI]

ADD AX, W

MOV SI, 1

ADD SI, SI

MOV t[SI], AX

Ces instructions définissent une étiquette etiquette19. Elles chargent 20 dans AX, effectuent une soustraction -2 à AX, effectuent des opérations sur t1 en utilisant l'indice 1, W et AX, et stockent le résultat dans Tab1 à l'emplacement spécifié.

FIN:

MOV AH, 4CH

INT 21h

Ces instructions définissent une étiquette FIN. Elles chargent 4CH dans AH (une fonction pour terminer le programme) et effectuent une interruption 21h pour terminer le programme.