POLYTECHNIQUE MONTRÉAL

Questionnaire examen final

LOG3210

Sigle du cours

Identification de l'étudiant(e)		
Nom :	**Prénom :**	
Signature :	**Matricule :**	**Groupe :**

Réservé

Sigle et titre du cours	*Groupe*	*Trimestre*
LOG3210 – Éléments de langages et compilateurs	**Tous**	**20171**
Professeur	***Local***	***Téléphone***
Ettore Merlo, responsable ; Ludovic Font, chargé de cours	**M-4105**	**4758-5193**

Jour	*Date*	*Durée*	*Heures*
Jeudi	**4 mai 2017**	**2h30**	**13h30 – 16h00**

Documentation	*Calculatrice*	
☐ Aucune	☒ Aucune	**Les cellulaires, agendas électroniques ou téléavertisseurs sont interdits.**
☒ Toute	☐ Toutes	
☐ Voir directives particulières	☐ Non programmable	

Directives particulières

Important

Cet examen contient **5** questions sur un total de **18** pages **(excluant cette page)**

La pondération de cet examen est de **45** **%**

Vous devez répondre sur : ☒ le questionnaire ☐ le cahier ☐ les deux

Vous devez remettre le questionnaire : ☒ oui ☐ non

L'étudiant doit honorer l'engagement pris lors de la signature du code de conduite.

Question 1 – Code à trois adresses (15 points)

1.1 TRADUISEZ le code suivant en code à trois adresses :

```
void euclDiv ( int i, int j ) {
if ( i < j ) {
      euclDiv ( j , i );
} else {
      int m = 1;
      while ( m * j < i )
            m ++;
      i = i – (m – 1) * j;
}
```

Il y a ici plusieurs solutions, notamment dans la gestion des *goto* redondants

```
euclDiv :     if i < j goto then
              goto else
then:         arg i
              arg j
              call euclDiv, 2
              goto endIf
else :        m = 1
while :       t1 = m * j
              ifFalse t1 < i goto endWhile
              m = m + 1
              goto while
endWhile :    t2 = m-1
              t3 = t2 * j
              I = I – t3
endIf:
```

1.2 Nous avons vu en cours trois façons de représenter ce code : en quadruplets, en triplets directs et en triplets indirects.

1.2.1 Quel est l'avantage de la représentation par triplets par rapport aux quadruplets?

Prend moins d'espace en mémoire, car on a pas à stocker le résultat de chaque opération.

1.2.2 Quel est l'inconvénient de la représentation directe des instructions en triplets?

Lorsque l'on veut utiliser le résultat d'une instruction précédente comme opérande, il faut référer au numéro de l'instruction où on l'a calculé. Si l'on renumérote les instructions pour une raison quelconque (optimisations, etc.), il faut modifier toutes les références aux instructions renumérotées.

1.2.3 Comment la représentation indirecte règle-t-elle ce problème?

En représentation indirecte, les numéros d'instructions sont distincts de leurs adresses concrètes. Lorsque l'on renumérote des instructions, on ne change que leur adresse. Leur numéro, lui, ne change pas, et il n'y a donc pas à recalculer quoi que ce soit.

Question 2 – Variables vives et allocation de registres (35 points)

2.1 COMPLÉTEZ le tableau suivant. Ce tableau contient du code machine dans la première colonne. Les autres colonnes indiquent l'emplacement des variables.

Code	**a**	**b**	**c**	**d**
`0 LD Ra, a`	`a, Ra`	`b`	`c`	`d`
`1 LD Rb, b`	`a, Ra`	`b, Rb`	`c`	`d`
`2 ADD Ra, Ra, Rb`	`Ra`	`b, Rb`	`c`	`d`
`3 LD Rc, c`	`Ra`	`b, Rb`	`c, Rc`	`d`
`4 SUB Rb, Ra, Rc`	`Ra`	`Rb`	`c, Rc`	`d`
`5 LD Rd, d`	`Ra`	`Rb`	`c, Rc`	`d, Rd`
`6 ADD Rd, Rd, Rb`	`Ra`	`Rb`	`c, Rc`	`Rd`
`7 SUB Rc, Rd, Ra`	`Ra`	`Rb`	`Rc`	`Rd`
`(fin du code)`	`Ra`	`Rb`	`Rc`	`Rd`

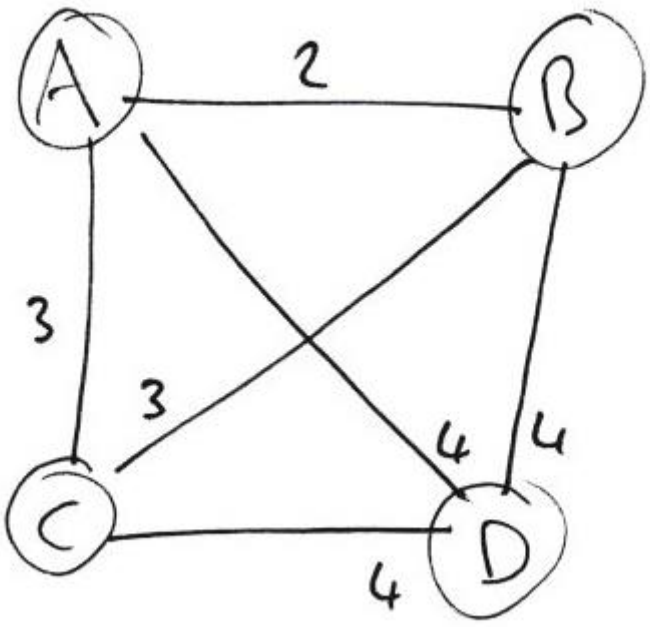
2.2 COMPLÉTEZ le tableau suivant, indiquant, pour le même code, quelles variables sont vivantes et quel est leur prochain usage. À la fin de ce bloc de code, c'est la seule variable vivante et sa prochaine utilisation se trouve à la ligne 12, en dehors du code.

Code	**Vivantes**	**Next-use**
`0 LD Ra, a`	/	/
`1 LD Rb, b`	a	a : 2
`2 ADD Ra, Ra, Rb`	a, b	a : 2, b : 2
`3 LD Rc, c`	a	a : 4
`4 SUB Rb, Ra, Rc`	a, c	a : 4, c : 4
`5 LD Rd, d`	a, b	a : 7, b : 6
`6 ADD Rd, Rd, Rb`	a, b, d	a : 7, b : 6, d : 6
`7 SUB Rc, Rd, Ra`	a, d	a : 7, d : 7
(fin du code)	c	c : 12

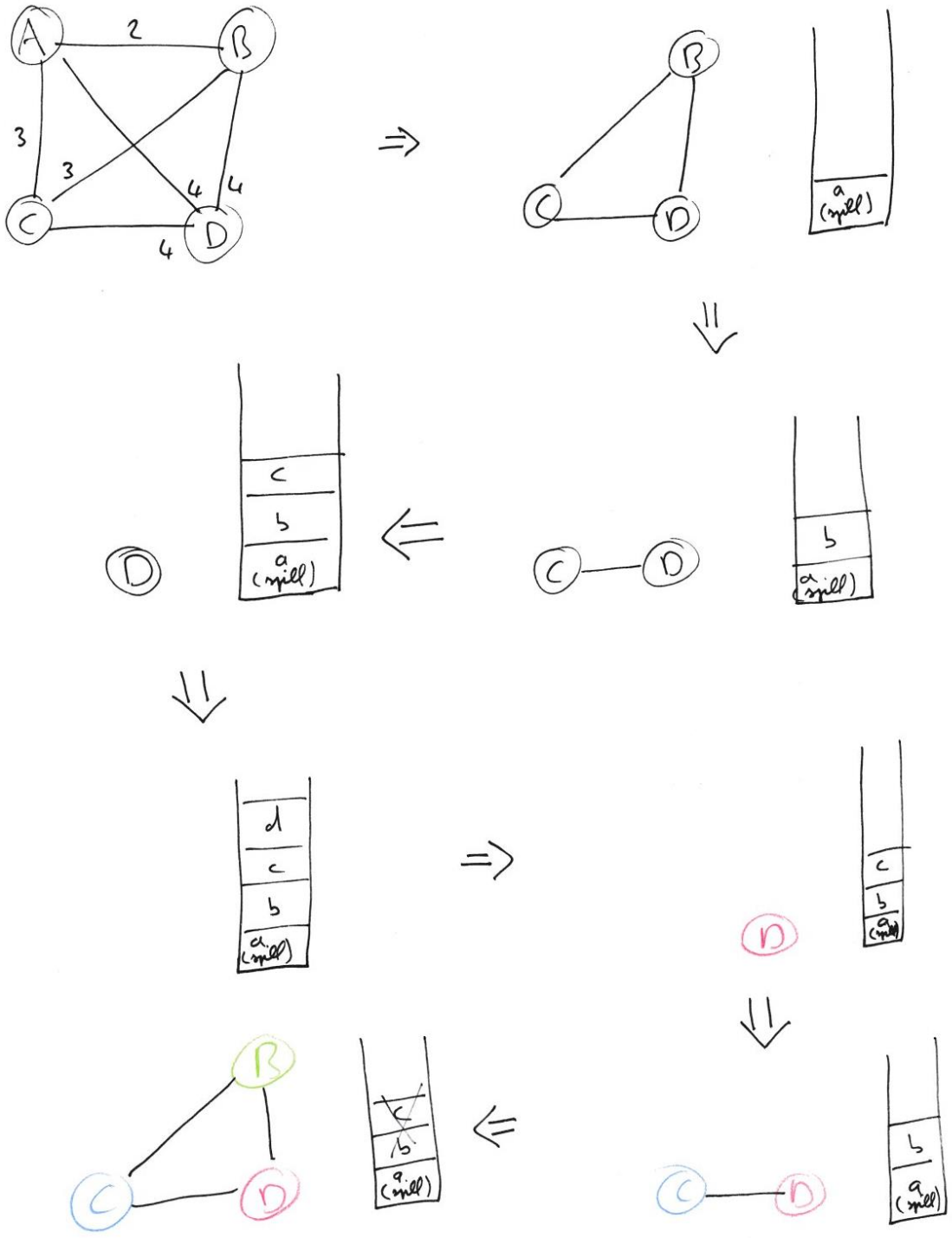
2.3 Considérons un autre fragment de code, avec les variables vives et leur prochain usage déjà calculés.

Code	**Variables vivantes**	**Prochain usage**
0 : Ld Ra, a	/	/
1 : Ld Rb, b	a	a : 2
2 : Add Rc, Ra, Rb	a, b	a : 2, b : 2
3 : Ld Rd, d	a, b, c	a : 4, b : 5, c : 14
4 : Sub Rd, Rd, Ra	a, b, c, d	a : 4, b : 5, c : 14, d : 5
5 : Add Rd, Rd, Rb	b, c, d	b : 5, c : 14, d : 5
(fin)	c, d	c : 14, d : 18

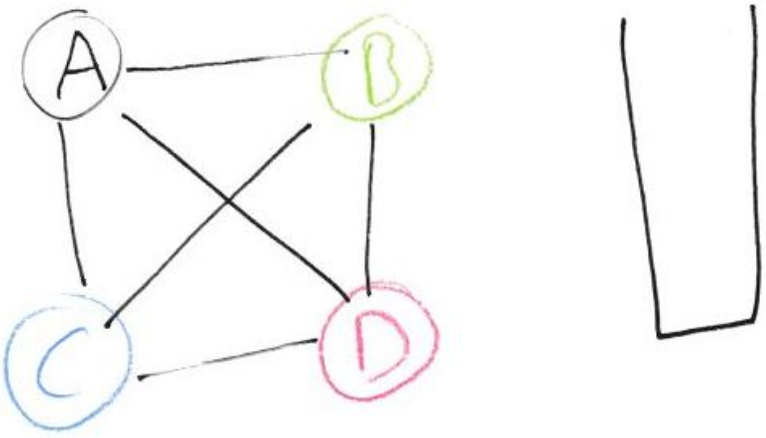
2.3.1 COMPLÉTEZ le graphe d'interférence des registres, en indiquant sur chaque arête l'instruction vous ayant permis de la construire.



2.3.2 APPLIQUEZ l'algorithme. Pour cela, REPRÉSENTEZ l'état de la pile et DESSINEZ le graphe à chaque étape. Traversez les registres dans l'ordre alphabétique : Ra, Rb, Rc, Rd. INDIQUEZ le coloriage final et les déversements (spill), si besoin.



(page blanche pour réponse de la question 2.3.2)

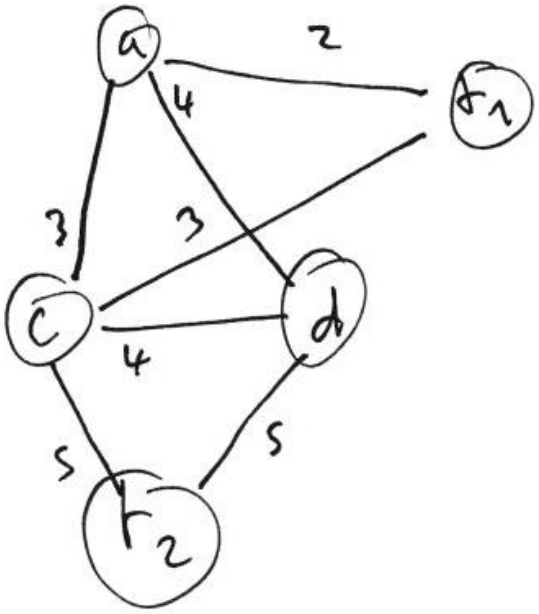


Il faut donc modifier le code. Un exemple possible est :

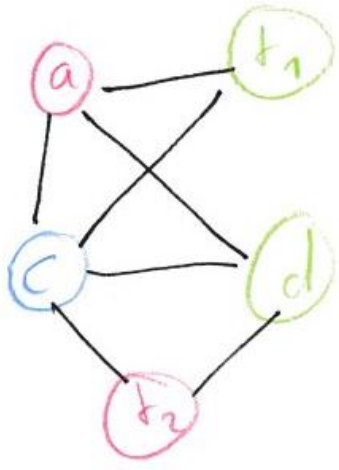
Code	**Variables vivantes**
0 : Ld Ra, a	/
1 : Ld Rt1, b	a
2 : Add Rc, Ra, Rt1	a, t1
3 : Ld Rd, d	c, a
4 : Sub Rd, Rd, Ra	c, d, a
5 : Ld Rt2, b	c, d
6 : Add Rd, Rd, Rt2	c, d, t2
(fin)	c, d

(page blanche pour réponse de la question 2.3.2)

Ce nouveau code donne le graphe suivant :



Que l'on peut colorier comme suit :



2.3.3 RÉÉCRIVEZ le code machine avec les vrais registres R1, R2, R3, répartis selon votre coloriage.

Rouge = R1, Vert = R2, Bleu = R3

```
0 : Ld R1, a

1 : Ld R2, b

2 : Add R3, R1, R2

3 : Ld R2, d

4 : Sub R2, R2, R1

5 : Ld R1, b

6 : Add R2, R2, R1
```

Question 3 – Gestion de la mémoire (20 points)

Considérons un gestionnaire de mémoire sans garbage collector, comme en C/C++. Ce gestionnaire utilise la stratégie Best-fit et utilise des *bins* de blocs vides. Un *bin* est une liste de blocs vide de taille donnée. Les *bins* sont les suivants :

I : contient les blocs vides dont la taille est comprise entre 1 et 15 octets : [1 , 16[
II : contient les blocs vides dont la taille est comprise entre 16 et 255 octets : [16 , 256[
III : contient les blocs vides dont la taille est comprise entre 256 et 2047 octets : [256-2048[
IV : contient les blocs vides dont la taille est supérieure à 2048 octets : [2048, infini[

Lorsqu'un bloc est libéré, le gestionnaire regarde si les blocs voisins sont vides. Si oui, il les retire de leurs *bins* respectifs, les fusionne avec le bloc libéré, puis ajoute le bloc résultant au *bin* approprié.

Lorsqu'un bloc est alloué, le gestionnaire l'enlève de son *bin* et, en cas de fragmentation, ajoute le nouveau bloc libre au *bin* approprié.

Sans entrer dans les détails de la constitution d'un bloc en mémoire, nous les représenterons de la façon suivante :

Nom	Taille	Occupé	*Bin*

Ainsi, le bloc A :

A	100	Vide	II

occupe 100 octets en mémoire, est actuellement vide, et appartient au *bin* II.

On ne se préoccupe pas de comment les *bins* gèrent leur liste de blocs.

Lorsque plusieurs blocs vides sont disponibles pour allouer, on choisit systématiquement le premier dans l'ordre alphabétique.

Lors de la création d'un bloc, que ce soit par allocation, fragmentation ou fusion, le nom du bloc est la première lettre non utilisée dans l'ordre alphabétique.

3.1 La mémoire est initialement occupée comme suit :

Nom	**Taille**	**Occupé**	***Bin***
A	300	Vide	III
B	50	Occupé	-
C	100	Vide	II
D	20	Occupé	-
E	30	Occupé	-
F	20	Vide	II
G	100	Occupé	-

REPRÉSENTEZ, dans les tableaux suivants, l'état de la mémoire après les opérations suivantes :

1 – Libération du bloc D

2 – Allocation d'un bloc de taille 250

3 – Libération du bloc E

4 – Allocation d'un bloc de taille 10

Tableau 1 – Libération du bloc D

Nom	**Taille**	**Occupé**	***Bin***
A	300	Vide	III
B	50	Occupé	-
C	120	Vide	II
E	30	Occupé	-
F	20	Vide	II
G	100	Occupé	-

Tableau 2 – Allocation d'un bloc de taille 250

Nom	Taille	Occupé	*Bin*
A	250	Occupé	-
D	50	Vide	II
B	50	Occupé	-
C	120	Vide	II
E	30	Occupé	-
F	20	Vide	II
G	100	Occupé	-

Tableau 3 – Libération du bloc E

Nom	Taille	Occupé	*Bin*
A	250	Occupé	-
D	50	Vide	II
B	50	Occupé	-
C	170	Vide	II
G	100	Occupé	-

Tableau 4 – Allocation d'un bloc de taille 10

Nom	**Taille**	**Occupé**	***Bin***
A	250	Occupé	-
D	10	Occupé	-
E	40	Vide	II
B	50	Occupé	-
C	170	Vide	II
G	100	Occupé	-

Question 4 – Enregistrements d'activation (15 points)

Considérons l'implémentation (très naïve) du calcul de la suite de Fibonacci :

```
int Fibonacci (int n) {
      if (n == 0)
            return 0;
      if (n == 1)
            return 1;
      int fib1 = Fibonacci(n – 1);
      int fib2 = Fibonacci(n – 2);
      return fib1 + fib2;
}
```

Un appel à Fibonacci(3) génèrerait les activations suivantes :

```
enter Fibonacci(3)
      enter Fibonacci(2)
            enter Fibonacci(1)
            leave Fibonacci(1)
            enter Fibonacci(0)
            leave Fibonacci(0)
      leave Fibonacci(2)
      enter Fibonacci(1)
      leave Fibonacci(1)
leave Fibonacci(3)
```

4.1 EXPLIQUEZ, dans **vos propres mots,** en quoi consiste un enregistrement d'activation, quels champs il contient, et à quoi ils servent.

Un enregistrement d'activation est créé lors de l'appel à une fonction. Il sert à stocker toutes les données nécessaires à la bonne exécution de la fonction.

Il peut contenir :

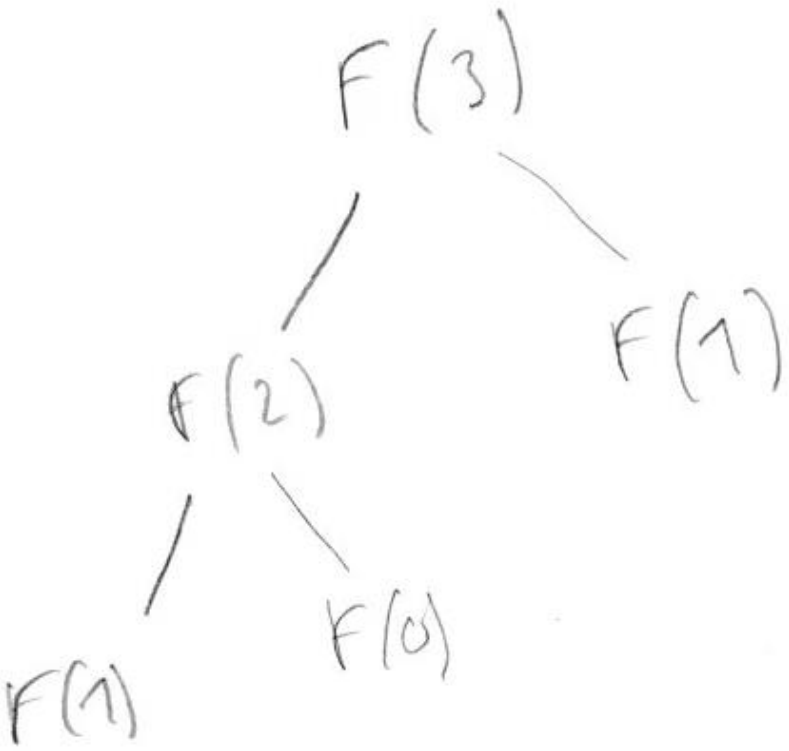
- Arguments de la fonction, permettant à l'appelant de passer des données à l'appelé
- Espace de retour (si besoin), permettant à l'appelé de renvoyer des données à l'appelant
- Liens de contrôle, référant à l'enregistrement d'activation de la fonction appelante
- Liens d'accès (dans certains langages), indiquant un emplacement contenant des données accessibles par l'appelé, mais qui ne se trouvent pas dans son environnement
- Sauvegarde de l'état de la machine, contenant les informations sur l'état de la machine juste avant l'appel de la fonction, permettant de les restaurer à la sortie (par exemple, l'adresse de l'instruction juste après le retour de la fonction, et l'état des registres)
- Données locales, espace réservé aux variables utilisées exclusivement dans la fonction appelée

- Valeurs temporaires qu'il faut parfois créer lors de la traduction en code machine.

4.2 Pourquoi est-il possible de stocker les enchaînements d'enregistrements d'activation sur une pile, qui est une structure de données assez contraignante?

On peut représenter la suite des activations de fonctions par un arbre, car une fonction appelée n'a qu'un seul appelant. La séquence des appels correspond à un parcours préfixe, la séquence des retours à un parcours postfixe. On est donc certains que l'on n'aura jamais à supprimer un enregistrement d'activation au milieu de la pile.

4.3 DESSINEZ l'arbre d'activations de l'appel à Fibonacci(3). Vous pouvez nommer les nœuds F pour Fibonacci.



Question 5 – Ramasse-miettes (15 points)

5.1 EXPLIQUEZ en quoi consistent les localités spatiale et temporelle dans le cadre de la gestion de la mémoire et en quoi c'est important.

La localité spatiale est présente lorsqu'une adresse X a de fortes chances d'être accédée peu après que l'on ait accédé à une adresse proche de X.

La localité temporelle est présente lorsqu'une adresse X a de fortes chances d'être accédée plusieurs fois dans un court laps de temps.

Ces deux notions sont importantes car cela permet d'optimiser la gestion de la mémoire : si l'on sait que la localité spatiale d'un programme donné est très forte, il devient intéressant d'allouer systématiquement des variables proches en mémoire lorsqu'elles sont proches dans le code.

5.2 EXPLIQUEZ, en quelques mots, comment fonctionnent les algorithmes par comptage de références et par collecte générationnelle.

Comptage de références : chaque variable est associée à un compteur, indiquant le nombre d'endroits où cette variable est utilisée. À chaque opération ajoutant ou supprimant une référence, on modifie le compteur en conséquence. Lorsqu'il atteint 0, on libère la mémoire.

Collecte générationnelle : on considère que les variables qui sont en mémoire depuis très longtemps ont de fortes chances d'y rester encore longtemps. Par conséquent, on priorise, lors de la collecte, les variables « jeunes », i.e. allouées récemment. Chaque fois qu'une variable « survit » à une collecte, elle devient plus ancienne.

5.3 COMPAREZ ces deux algorithmes des points de vue : temps d'exécution total, bonne gestion de la mémoire (et de la fragmentation), interruption de l'exécution du programme, et exploitation de la localité spatiale et temporelle des programmes.

Temps total d'exécution :

L'algorithme par décompte des références prend beaucoup de temps car il demande de faire une opération arithmétique, certes courtes, à chaque fois que l'on modifie une référence.

L'algorithme générationnel priorise les variables jeunes : le nettoyage de la mémoire est donc plus court qu'un ramasse-miette classique. De plus, on peut ajuster le degré de priorisation, ce qui fait que l'on a un bon contrôle du temps d'exécution.

Espace mémoire :

L'algorithme par décompte de références libère immédiatement la mémoire. Cependant, aucune défragmentation n'est effectuée a priori, et les références cycliques ne sont pas gérées, ce qui peut entraîner des fuites mémoires conséquentes.

L'algorithme générationnel a un inconvénient majeur : lorsqu'une variable commence à être assez ancienne et n'est plus utilisée, il est possible qu'elle ne soit pas collectée avant très longtemps. Par conséquent, la mémoire risque de se remplir plus rapidement qu'avec un algorithme par décompte de références.

Interruption de l'exécution :

L'algorithme par décompte de références n'interrompt jamais le programme, la libération est faite immédiatement, dès que l'objet est rendu inaccessible.

L'algorithme générationnel interrompt l'exécution, mais de manière courte (c'est l'intérêt) : en ciblant la collecte, on réduit autant que l'on souhaite le temps d'interruption.

Localité spatiale et temporelle :

L'algorithme par décompte de références ne tire absolument pas parti de la localité. Cependant, il n'en a pas besoin car les objets sont libérés aussitôt que possible.

L'algorithme générationnel exploite fortement la localité temporelle, car son postulat de départ est qu'une variable va servir dans un laps de temps court, puis ne plus jamais être utilisée.