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**	**20153**
Professeur	***Local***	***Téléphone***
Ettore Merlo, responsable	**M-4105**	**5193 - 5758**

Jour	*Date*	*Durée*	*Heures*
Jeudi	**10 décembre 2015**	**2 h 30**	**13:30 à 16:00**

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

Directives particulières

- Joindre l'énoncé à votre cahier d'examen en inscrivant votre nom et votre matricule.

Important

Cet examen contient **6** questions sur un total de **8** pages **(excluant cette page)**

La pondération de cet examen est de **50** **%**

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 – Conversion de types (20 pts)

1.1) Implantez en Java la fonction max de la **figure 1.1** en utilisant la classe de la **figure 1.2**.

1.2) Considérez les déclarations suivantes :

```
int i, j, k;

float x, y, z;

double a, b, c;
```

Écrivez le code à trois adresses correspondant aux expressions suivantes en utilisant les **figures 1.1** et **1.3** :

1.2.1) a + i

1.2.2) k + j

1.2.3) y + c

$$
\begin{aligned}
E \rightarrow E_1 + E_2 \quad \{\ & E.type = max(E_1.type, E_2.type); \\
& a_1 = widen(E_1.addr, E_1.type, E.type); \\
& a_2 = widen(E_2.addr, E_2.type, E.type); \\
& E.addr = \textbf{new}\ Temp\,(); \\
& gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{aligned}
$$

Figure 1.1

```
public class types {

    private HashMap<String, String> hierarchie =
        new HashMap<String, String>();

    public types () {

        hierarchie.put("short", "int");
        hierarchie.put("byte", "short");
        hierarchie.put("long", "float");
        hierarchie.put("int", "long");
        hierarchie.put("float", "double");
        hierarchie.put("char", "int");

        return;
    }

    public String max(String type1, String type2) {

        ...

    }
}
```

Figure 1.2

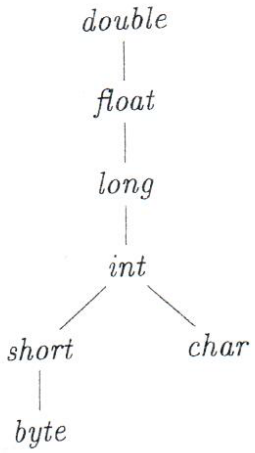


Figure 1.3

a + i => a + (double) i

k + j => k + j

y + c => (double) y + c

Question 2 – Blocs de base (10 points)

En considérant le code à trois adresses de la **figure « 2.1 »**, identifiez les blocs de base et dessinez le graphe de flux de contrôle.

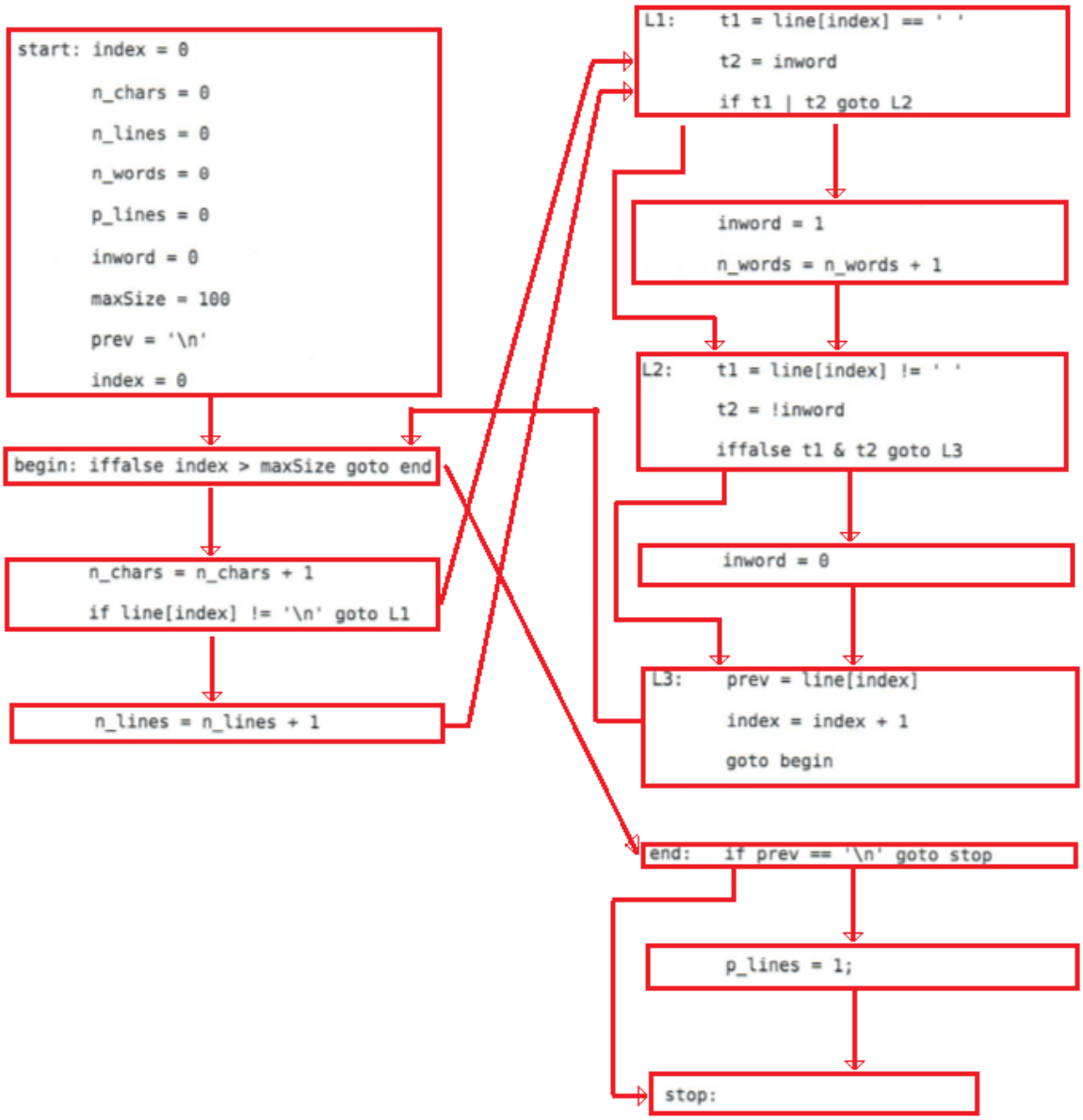
```
start: index = 0
       n_chars = 0
       n_lines = 0
       n_words = 0
       p_lines = 0
       inword = 0
       maxSize = 100
       prev = '\n'
       index = 0
begin: iffalse index > maxSize goto end
       n_chars = n_chars + 1
       if line[index] != '\n' goto L1
       n_lines = n_lines + 1
L1:    t1 = line[index] == ' '
       t2 = inword
       if t1 | t2 goto L2
       inword = 1
       n_words = n_words + 1
L2:    t1 = line[index] != ' '
       t2 = !inword
       iffalse t1 & t2 goto L3
       inword = 0
L3:    prev = line[index]
       index = index + 1
       goto begin
end:   if prev == '\n' goto stop
       p_lines = 1;
stop:
```

Figure 2.1

Identification des leaders :

```
start: index = 0
       n_chars = 0
       n_lines = 0
       n_words = 0
       p_lines = 0
       inword = 0
       maxSize = 100
       prev = '\n'
       index = 0
begin: iffalse index > maxSize goto end
       n_chars = n_chars + 1
       if line[index] != '\n' goto L1
       n_lines = n_lines + 1
L1:    t1 = line[index] == ' '
       t2 = inword
       if t1 | t2 goto L2
       inword = 1
       n_words = n_words + 1
L2:    t1 = line[index] != ' '
       t2 = !inword
       iffalse t1 & t2 goto L3
       inword = 0
L3:    prev = line[index]
       index = index + 1
       goto begin
end:   if prev == '\n' goto stop
       p_lines = 1;
stop:
```

Graphe de flux de contrôle :



Question 3 – Optimisations locales (20 points)

Considérez le bloc de base suivant :

1: c = a + b

2: a = b - c

3: b = a + d

4: c = a - b

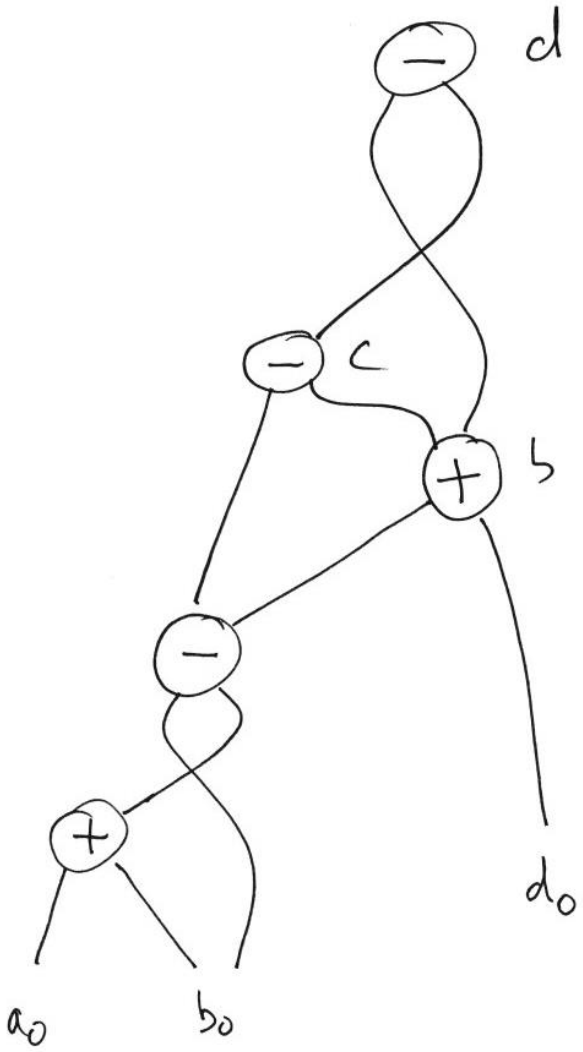
5: d = b - c

3.1) DESSINEZ le graphe orienté sans cycle (DAG) correspondant au bloc de base et aux sous-expressions.

Commençons par les définitions les plus récentes :

	a	b	c	d
1 : c = a + b	a0	b0	c0	d0
2 : a = b – c	a0	b0	1	d0
3 : b = a + d	2	b0	1	d0
4 : c = a – b	2	3	1	d0
5 : d = b - c	2	3	4	d0
	2	3	4	5

Graphe dirigé acyclique :



3.2) CALCULEZ la fonction « PROCHAINE_UTILISATION » (« NEXT-USE ») pour chaque variable utilisée dans chaque instruction en supposant qu'aucune variable n'est utilisée après la fin du bloc. REMPLISSEZ le **tableau 3.1**.

a	b	c	d	Prochaine utilisation ("NEXT-USE")
1	1	/	3	

1: c = a + b

a	b	c	d	Prochaine utilisation ("NEXT-USE")
/	2	2	3	

2: a = b – c

a	b	c	d	Prochaine utilisation ("NEXT-USE")
3	/	/	3	

3: b = a + d

a	b	c	d	Prochaine utilisation ("NEXT-USE")
4	4	/	/	

4: c = a – b

a	b	c	d	Prochaine utilisation ("NEXT-USE")
37	5	5	/	

5: d = b - c

a	b	c	d	Prochaine utilisation ("NEXT-USE")
37	16	/	12	

Tableau 3.1

3.3) En utilisant les réponses des points précédents, est-ce qu'il y a des sous-expressions en commun qui pourraient être éliminées? Lesquelles ?

Non : aucun nœud n'est annoté par deux variables.

3.4) En utilisant les réponses des points précédents, est-ce qu'il y a du code mort (inutile) qui pourrait être éliminé? Lequel?

Non : tout le code est utile. La seule variable morte à la fin du code est c, mais on en a besoin pour la dernière instruction d = b – c.

Question 4 – Environnements d'exécution (15 points)

Considérez un environnement d'exécution par enregistrement d'allocation des fonctions actives sur une pile.

4.1) Présentez brièvement l'approche par liens d'accès et par « display registers ».

Les liens d'accès servent à indiquer l'emplacement contenant les données accessibles par la procédure appelée, mais qui ne sont pas directement dans son bloc d'activation, par exemple des variables statiques.

(l'approche par display registers n'est plus dans le cours)

4.2) COMPAREZ les deux approches du point **4.1)** par rapport à la manipulation et au maintien des informations lors d'appel et de retour des procédures en adressant EXPLICITEMENT les avantages, désavantages, limitations, efficacité et performance. (Suggestion : examinez les appels directement ou indirectement récursifs)

(l'approche par display registers n'est plus dans le cours)

Question 5 – Allocation de registres par coloriage de graphe (20 points)

Considérez le code machine avec registres *symboliques* suivant :

```
1:   LD   Rb, b
2:   LD   Rd, d
3:   ADD  Ra, Rb, Rd
4:   LD   Rc, c
5:   SUB  Rd, Ra, Rc
6:   ADD  Rc, Rb, Rd
7:   ADD  Rb, Ra, Rc
8:   ADD  Ra, Rb, Rc
```

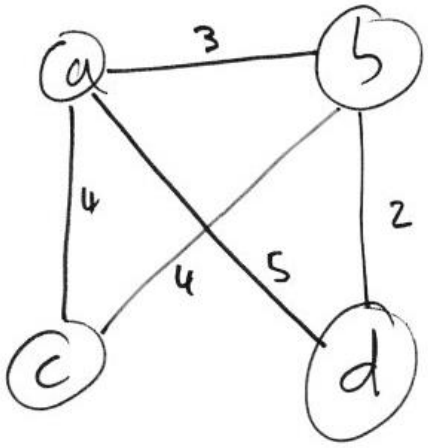
5.1) CALCULEZ l'ensemble des registres vifs (« LIVE ») avant et après chaque instruction pour le code machine ci-haut et REMPLISSEZ le **tableau 5.1**.

IMPORTANT : Considérez qu'aucun registre ne soit **VIF** après l'instruction 8.

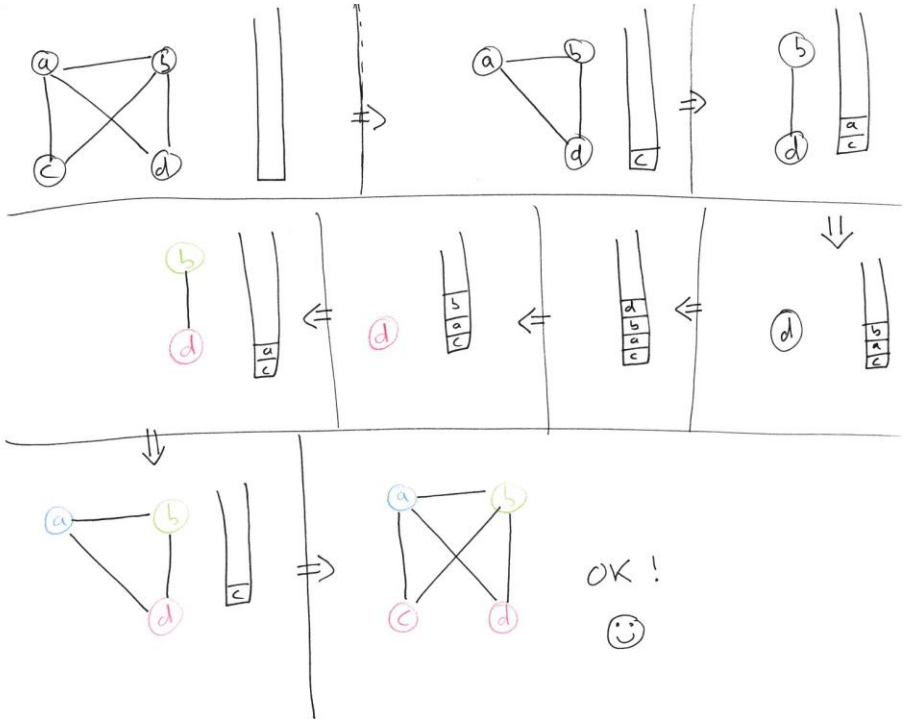
LIVE:	/
1:	LD Rb, b
LIVE:	Rb
2:	LD Rd, d
LIVE:	Rb, Rd
3:	ADD Ra, Rb, Rd
LIVE:	Ra, Rb
4:	LD Rc, c
LIVE:	Ra, Rb, Rc
5:	SUB Rd, Ra, Rc
LIVE:	Ra, Rb, Rd
6:	ADD Rc, Rb, Rd
LIVE:	Ra, Rc
7:	ADD Rb, Ra, Rc
LIVE:	Rb, Rc
8:	ADD Ra, Rb, Rc
LIVE:	Ø

Tableau 5.1

5.2) DESSINEZ le graphe d'interférence de registres pour le code machine ci-haut.



5.3) En utilisant l'algorithme de coloriage de graphe vu en cours et le graphe d'interférence des registres du point précédent, DESSINEZ l'état de la pile et l'état du graphe à CHAQUE ÉTAPE de l'algorithme en utilisant 3 couleurs et en TRAVERSANT les registres dans l'ORDRE ALPHABÉTIQUE Ra, Rb, Rc, Rd. INDIQUEZ le coloriage final et les déversements (« spill ») requis au besoin.



Question 6 – Ramasse-miette (15 points)

EXPLIQUEZ les différences entre les algorithmes de type **incrémentiel et partiel** par rapport aux algorithmes de type « décompte des références » (« reference counting ») en discutant, entre autres, les avantages et désavantages de chacun par rapport à l'autre en terme des aspects suivants : temps total d'exécution, espace mémoire requise, interruption de l'exécution du programme, et 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 incrémentiel s'exécute en même temps que le programme. Le temps d'exécution total est plus faible que le précédent, et exploite le multithreading.

L'algorithme partiel est lui aussi assez rapide car il fait en sorte de ne pas perdre trop de temps sur des objets ayant peu de chances d'être libérés.

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 incrémentiel libère la mémoire assez rapidement car, comme l'exécution est simultanée, il est possible d'avoir en permanence un nettoyage en cours.

L'algorithme partiel est rapide, mais la priorité donnée aux objets à vérifier fait que certains objets qui devraient être libérés peuvent rester inutilement en mémoire très longtemps.

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 incrémentiel n'interrompt pas non plus l'exécution car il s'exécute en parallèle.

L'algorithme partiel interrompt l'exécution, même si la collecte ciblée tente de limiter au maximum la durée de l'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 incrémentiel n'exploite pas non plus la localité temporelle. Il exploite d'une certaine manière la localité spatiale, car deux variables déclarées à la suite ont de bonnes chances d'être traitées lors de la même passe de nettoyage.

L'algorithme partiel exploite la localité temporelle pour identifier quelles variables sont souvent utilisées et lesquelles ne le sont pas.