Examen de majeure 2
Compilation

(Durée 2h)

20 mars 2000

On attachera une importance particulière à la clarté, à la précision, mais aussi à la ***concision*** *de la rédaction. Les questions 1 et 4 sont indépendantes, ainsi que certaines sous-questions d'autres parties.*

Mise en pile

Dans le cours, nous avons apporté un soin particulier à l'allocation de registres de façon à éviter au maximum les mises en pile et à éliminer le plus possible d'instructions `move`. Cependant, dans le cas d'une mise en pile, nous avons fait le choix délibéré de la simplicité, parfois au détriment de la qualité (rapidité d'exécution) du code produit. On cherche ici à améliorer le résultat de la mise en pile.

1 De bons exemples

On considère la fonction suivante d'un programme Pseudo-Pascal :

```
function sum (var n : integer) : integer;
begin
  if n > 0 then sum := n + sum (n−1) else sum := 0
end;
```

1.1 Donner un code Spim possible, tel que pourrait le retourner le compilateur du cours (en particulier, on n'effectuera pas de transformation de programme avancée, telle que l'élimination de la récursion).

Réponse:

```
    sum_f=8
sum:
    sub   $sp, $sp, sum_f
    sw    $ra, 4($sp)
    sw    $s0, 0($sp)
    move  $s0, $a0
    ble   $s0, $zero, L2
    sub   $a0, $s0, 1
    jal   sum
    add   $v0, $s0, $v0
L1:
    lw    $s0, 0($sp)
```

```
        lw     $ra, 4($sp)
        add    $sp, $sp, sum_f
        j      $ra
L2:
        move   $v0, $zero
        j      L10
```

1.2 Donner un argument prouvant que la mise en pile est inévitable dans la compilation du programme ci-dessus (on s'interdit toujours des transformations de programme avancées), quelque soit la qualité de l'allocateur de registres et le nombre de registres disponibles.

Réponse: *La fonction* `sum` *est récursive, non terminale. Il faut donc revenir à la fonction après l'appel récursif. Pour cela l'adresse de retour (la valeur du registre* **$ra***) doit être sauvegardée pendant l'appel récursif (qui utilise lui-même* **$ra***). Aucun registre n'est candidat à la sauvegarde de* **$ra***, car il serait lui-même écrasé : c'est le même code qui effectue l'appel récursif donc il utilise potentiellement (*i.e. *c'est une approximation statique) les mêmes registres que l'appel lui-même.*

(On pourrait sauvegarder **$ra** *dans la tas, mais cela revient à implémenter une pile dans le tas.)*

1.3 La mise en pile est-elle satisfaisante dans l'exemple ci-dessus ?

Réponse: *Oui, on ne peut guère faire mieux à la main (sans transformation de programme avancée) : pour chacun des registres* `ra` *et* `s0`*, il n'y a qu'une seule sauvegarde et restauration en pile qui est inévitable.*

1.4 En induire un cas, assez fréquent, de mise en pile bien traité par la solution simple du cours.

Réponse: *Les registres sauvés par l'appelant sont placés dans un temporaire au début du programme et restaurés à la fin. Si le temporaire utilisé pour cette sauvegarde doit être mise en pile, cette mise en pile sera efficace car le temporaire n'a qu'une seule occurrence en écriture et en lecture.*

2 Des insuffisances

Dans toute la suite du problème, on utilise les notations et restrictions suivantes. On suppose qu'il n'existe que trois registres machine **$a0**, **$v0** et **$t0** qui sont aussi des temporaires particuliers, pré-coloriés (on réserve le préfixe $ aux registres machine). Pour simplifier, on traite les registres spéciaux (**$sp**, **$ra** et **$zero**) à part : on ne les considérera pas comme des temporaires et on les exclut de toutes les phases d'analyse (durée de vie, interférence, allocation de registres).

On note $\mathcal{G}(A)$ le graphe d'interférence associé au code assembleur A et $\varphi(A)$ le code produit par la mise en pile d'un seul temporaire t dans le code A en utilisant la procédure du cours (le temporaire t est toujours le même et laissé implicite).

2.1 On considère comme exemple le code C suivant :

```
1:    add  t,   $a0, 1
2:    add  t,   t, $a0
3:    add  $a0, t, t
4:    jal  foo
5:    add  $v0, t, $v0
```

où la fonction `foo` lit **$a0**, écrase les trois registres machines et retourne son résultat dans `v0`. On suppose que le registre **$v0** est vivant à la fin du code C. Donner le code $\varphi(C)$.

Réponse:

```
1:      add  p1,  $a0, 1
        sw   p1,  0($sp)
2:      lw   q1,  0($sp)
        add  p2,  q1, $a0
        sw   p2,  0($sp)
3:      lw   q2,  0($sp)
        add  $a0, q2, q2
4:      jal  foo
5:      lw   q3,  0($sp)
        add  $v0, q3, $v0
```

2.2 Donner au moins une raison pour laquelle le code $\varphi(C)$ n'est pas bon.

Réponse: *L'instruction **lw** **q1**, 0(**$sp**) rappelle le contenu du sommet de pile dans le temporaire auxiliaire **q1**. C'est absurde, car ce contenu est aussi celui du temporaire **p1** : on pourrait certainement remplacer **lw** **p1**, 0(**$sp**) par l'instruction **move** **q1**, **p1** qui évite une lecture en pile inutile (de plus, cette instruction sera ensuite éliminée par l'allocation de registres). En général, certaines occurences de t n'ont pas besoin d'être traitées en pile (qu'il s'agisse d'une lecture ou d'une écriture).*

2.3 Donner un code de mise en pile meilleure que $\varphi(C)$, obtenu manuellement, en utilisant au mieux les registres disponibles (on donnera le code final, après allocation des registres).

Réponse:

```
1:      add  $t0,  $a0, 1
2:      add  $t0,  $t0, $a0
        sw   $t0,  0($sp)
3:      add  $a0,  $t0, $t0
4:      jal  foo
5:      lw   $t0,  0($sp)
        add  $v0,  $t0, $v0
```

*(On pourrait inclure les instructions d'ajustement de pile **sub** **$sp**, **$sp**, 4 au début, et **add** **$sp**, **$sp** 4 à la fin, ici comme à la question précédente, bien qu'elles ne font pas partie du code de mise en pile, mais de la phase finale postérieure à l'allocation de registres.)*

3 Approche mécanique

On note $\varphi^\dagger(A)$ le programme A transformé par la procédure suivante :

1. On donne un numéro $i \in I$ à chaque instruction qui lit le temporaire t et un numéro $j \in J$ à chaque instruction qui écrit le temporaire t (une instruction peut avoir à la fois un numéro $i \in I$ et un numéro $j \in J$ si t y est utilisé à la fois en lecture et en écriture.)
2. On se donne des nouveaux temporaires $q_i^{i \in I}$ et $p_j^{j \in J}$.
3. On remplace dans chaque instruction $i \in I$ toutes les occurrences de t en lecture par q_i.
4. On remplace chaque instruction $j \in J$ par la séquence d'instructions composée, dans l'ordre :
   (a) de l'instruction j dans laquelle toutes les occurrences en écriture de t sont remplacées par p_j,
   (b) des instructions $(\textbf{move}\ q_i,\ p_j)^{i \in I}$ (on rappelle que **move** u, v copie v dans u).

3.1 Donner, très soigneusement, le code $\varphi^\dagger(C)$.

Réponse:

```
                                        # Temporaires vivants en sortie

1:      add   p1,  $a0, 1               # $a0, p1
        move  q1,  p1                   # $a0, p1, q1
        move  q2,  p1                   # $a0, p1, q1
        move  q3,  p1                   # $a0, q1
2:      add   p2,  q1, $a0              # p2
        move  q1,  p2                   # p2
        move  q2,  p2                   # p2, q2
        move  q3,  p2                   # q2, q3
3:      add   $a0, q2, q2               # $a0, q3
4:      jal   foo                       # $v0, q3
5:      add   $v0, q3, $v0              # $v0
```

3.2 Montrer, dans le cas général, que la transformation de A en $\varphi^\dagger(A)$ est correcte, c'est-à-dire que les deux codes assembleurs sont équivalents (pour une machine qui préserverait la valeur des temporaires à chaque appel de fonction).

Réponse: *On considère comme une macro-instruction l'image $\varphi^\dagger(k)$ par réécriture d'une instruction k de A. On exécute en parallèle les codes A et $\varphi^\dagger(A)$. On donne au départ des valeurs indéfinies à t et à tous les ${q_i}^{i\in I}$. On montre facilement, par récurrence, qu'à chaque pas $s \in I\!N$ (un pas correspond à l'exécution d'une instruction ou d'une macro-instruction)*

- *si le programme A est à l'instruction k, alors le programme $\varphi^\dagger(A)$ est à l'instruction $\varphi^\dagger(k)$;*
- *les valeurs des ${q_i}^{i\in I}$ sont toutes identiques dans le programme $\varphi^\dagger(A)$ et égales à la valeur de t dans le programme A ;*
- *pour tous les autres temporaires de A leur valeur dans $\varphi^\dagger(A)$ est égale à leur valeur dans A.*

  (Formellement, comme c'est évidemment vrai au début du programme, il suffit de vérifier que si la propriété est vérifiée pour s, elle l'est aussi pour $s+1$. Soit $k \in A$ la position du programme dans A à l'instant s. Si l'instruction k effectue un saut dans A à l'instruction l, alors compte tenu de la correspondance entre les valeurs des registres, l'instruction $\varphi^\dagger(k)$ effectue un saut à l'instruction l de $\varphi^\dagger(A)$, donc le programme est à l'instruction $l \in A$ et $\varphi^\dagger(l) \in \varphi^\dagger(A)$ à l'instant $s+1$.

  - *Si l'instruction l lit une valeur v dans t (la valeur de t à l'instant s), la macro-instruction $\varphi^\dagger(l)$ lit à la place la valeur de q_i, donc v, pour un certain i de I.*
  - *Si l'instruction l écrit v dans t , alors l'instruction $\varphi^\dagger(l)$ écrit v dans p_j puis écrit la valeur de p_j donc v dans tous les ${q_i}^{i\in I}$.)*

Ainsi, à une trace d'exécution de A correspond une trace d'exécution de $\varphi^\dagger(A)$ ayant un comportement identique vis à vis de l'extérieur. Comme l'exécution est déterministe, la trace observée pour $\varphi^\dagger(A)$ est la seule possible.

3.3 Donner, soigneusement, pour chaque instruction de $\varphi^\dagger(C)$ la liste des temporaires vivants en sortie (on ne demande pas le détail du calcul, mais seulement le résultat).

Réponse: *Voir réponse à la question 3.1.*

3.4 En déduire immédiatement (donc sans faire d'allocation de registres) que l'on peut supprimer certaines instructions **move**.

Réponse: *On peut supprimer les instructions **move** dont le registre de destination n'est pas vivant en sortie.*

3.5 Donner le code résultant que l'on notera C'.

Réponse:

```
1:   add   p1,  $a0, 1      # $a0, p1
     move  q1,  p1          # $a0, p1, q1
2:   add   p2,  q1, $a0     # p2
     move  q2,  p2          # p2, q2
     move  q3,  p2          # q2, q3
3:   add   $a0, q2, q2      # $a0, q3
4:   jal   foo              # $v0, q3
5:   add   $v0, q3, $v0     # $v0
```

4 Interférences parasites

On s'intéresse maintenant au bloc de code B suivant :

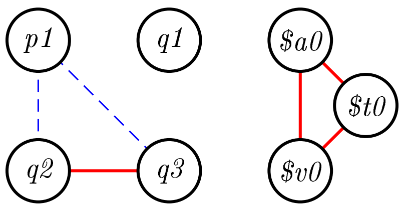
```
add  p2, q1, $a0
move q2, p2
move q3, p2
```

On suppose que les registres q2 et q3 sont les seuls vivants en sortie du bloc. Donner pour chaque instruction les registres vivants en sortie. Donner le graphe d'interférence (sans justification).

Réponse:

```
                       # vivants     # Interférences engendrées
add  p2, q1, $a0       # p2          #
move q2, p2            # p2 q2       #
move q3, p2            # q2 q3       # q2−q3
```

- *Le graphe d'interférence comporte les trois registres et quatre autres nœuds **p2**, **q1**, **q2**. **q3**. Il comporte un seul arc* (q2, q3) *d'interférence en plus du sous-graphe plein des interférences entre registres (que l'on pourrait omettre).*
- *Les arcs d'attirance qui ne sont pas demandés.*



4.1 Exhiber un arc superflu. En quoi est-ce gênant ?

Réponse: *Les temporaires **q2** et **q3** interfèrent. Comme ces temporaires ont toujours la même valeur (lorsque celle-ci est utile), on pourrait en fait leur associer le même registre machine, ce qui est ici interdit par leur interférence, au risque de ne pas pouvoir les colorier du tout.*

4.2 On regroupe les deux instructions **move** `q2, p2` et **move** `q3, p2` ensemble en une macro-instruction **mmv** `q2 q3, p2` (lire "multi-move") qui lit `p1` et écrit `q2` et `q3`.

Que devient le graphe d'interférence si le multi-move est traité comme une instruction de calcul ordinaire (nœud Oper dans le cours) ?

Réponse: *Les arcs d'interférence sont inchangés (on ignore les arcs d'attirance) : les 2 temporaires* ***q2*** *et* ***q3*** *qui survivent à l'instruction sont écrits par celle-ci donc ils interfèrent entre eux.*

4.3 On traite le multi-move comme une forme généralisée de **move** : l'instruction **mmv** $d_1 \ldots d_n,\ s$ copie le temporaire s dans les temporaires $(d_k)^{k\in 1..n}$.

Décrire les interférences minimales crées par une instruction **mmv** $d_1 \ldots d_n,\ s$ en fonction de l'ensemble *dest* des temporaires $(d_k)^{k\in 1..n}$, de s et de l'ensemble *out* des temporaires vivants en sortie de cette instruction.

Réponse: *Chaque temporaire d_k interfère avec les temporaires vivants autres que s et ceux de dest. Au total, le sous-graphe induit par une telle instruction est* $dest \times (out \setminus (\{s\} \cup dest))$.

4.4 Donner le graphe d'interférence $\mathcal{G}(B)$ en considérant les deux moves successifs comme un multi-move.

Réponse: *Les seuls registres vivants en sortie de $\mathcal{G}(B)$ sont dans la destination du multi-move, donc n'interfèrent pas. Ainsi $\mathcal{G}(B)$ est le graphe vide (*i.e.* sans arcs, il comporte bien sûr des nœuds).*

5 Exemple à suivre

Dans la suite, on note $\varphi^\ddagger(A)$ le code assembleur obtenu à partir de $\varphi^\dagger(A)$ en remplaçant les séquences de **move** introduites par des instructions **mmv**, puis en simplifiant ces instructions après l'analyse de durée de vie (en retirant certaines destinations, donc certains move) comme indiqué à la question 3.4.

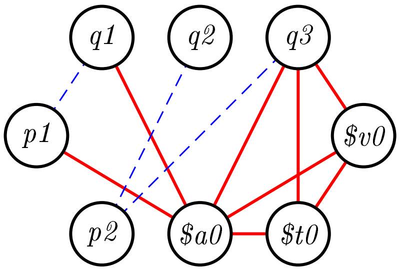
5.1 Retrouver les temporaires vivants à la sortie de chaque instruction du code $\varphi^\ddagger(C)$ (en réutilisant simplement les réponses aux questions 3.3 et 3.5). En déduire $\mathcal{G}(\varphi^\ddagger(C))$.

Réponse:

```
                                # vivants     # Interférences engendrées

1:   add   p1,  $a0, 1          # $a0, p1     # $a0-p1
     mmv   q1,  p1              # $a0, q1     # $a0-q1
2:   add   p2,  $a0, q1         # p2          #
     mmv   q2 q3, p2            # q2, q3      #
3:   add   $a0, q2, q2          # $a0, q3     # $a0-q3
4:   jal   foo                  # $v0, q3     # $a0-q3, $v0-q3, $t0-q3
5:   add   $v0, q3, $v0         # $v0
```

- *On pourrait omettre les arcs entre registres qui forment un sous-graphe plein.*
- *On a mis les arcs d'attirance qui ne sont pas demandés mais qui aident au choix des registres demandé dans la question suivante.*



5.2 Donner un résultat possible pour l'allocateur de registres, en tenant compte des contraintes d'interférence, et si possible des préférences (arcs "move") : on indiquera dans une table pour chaque temporaire le registre-machine associé ou bien le temporaire lui-même s'il doit être mis en pile.

Réponse:

Temporaire	*$a0*	*$v0*	*$t0*	*p1*	*p2*	*q1*	*q2*	*q3*
Registre	*$a0*	*$v0*	*$t0*	*$t0*	*$t0*	*$t0*	*$t0*	*q3*

5.3 Donner le code résultant après expansion des instructions **mmv** et simplification des instructions **move** devenues inutiles par la fusion de leur source et de leur destination.

Réponse:

```
1:    add   $t0, $a0, 1
2:    add   $t0, $t0, $a0
      move  q3,  $t0
3:    add   $a0, $t0, $t0
4:    jal   foo
5:    add   $v0, q3, $v0
```

6 Retour au cas général

On reprend le cas général d'un code A, coloriable à l'exception d'un seul temporaire t qu'il faut placer en pile.

6.1 Montrer que le graph $\mathcal{G}(\varphi^{\ddagger}(A))$ est coloriable à l'exception peut-être de certains nœuds parmi ${q_i}^{i\in I}$ et ${p_j}^{j\in J}$.

Réponse: *Il suffit de vérifier qu'une fois retirés tous les nœuds ${q_i}^{i\in I}$ and ${p_j}^{j\in J}$ de $\mathcal{G}(\varphi^{\ddagger}(A))$, on obtient un sous-graphe de $\mathcal{G}(A)$, donc coloriable. En l'occurence, il est exactement égal au graphe $\mathcal{G}(A)$ privé du nœud t.*

*En effet, pour les autres noeuds (*i.e. *temporaires), les variables lus et écrites dans chaque instruction de A et de $\varphi^{\ddagger}(A)$ sont identiques. Les temporaires restants vivants en sortie de chaque instruction sont donc identiques (les instructions ajoutées dans $\varphi^{\ddagger}(A)$ ne concerne pas ces temporaires). Par conséquent les contraintes d'interférence générées pour les temporaires restant sont aussi identiques.*

6.2 Peut-on être plus précis si l'on sait que $\mathcal{G}(\varphi(A))$ est coloriable (on ne fera pas de preuve) ?

Réponse: *Les nœuds* ${t_j}^{j\in J}$ *sont alors aussi coloriables dans* $\mathcal{G}(\varphi^\ddagger(A))$ *(on inclut les nœuds* ${t_j}^{j\in J}$ *et on remplace* A *par* $\varphi^\dagger(A)$ *dans le raisonnement précédent).*

6.3 On définit $\varphi^\#(A)$ à partir de $\varphi^\ddagger(A)$ en remplaçant chaque instruction i par l'instruction **move** $r_i,\ q_i$ suivie de l'instruction i dans laquelle q_i est remplacé par r_i où r_i est un nouveau temporaire. Quel est l'intérêt de colorier $\mathcal{G}(\varphi^\#(A))$ plutôt que $\mathcal{G}(\varphi^\ddagger(A))$? En déduire, dans certains cas, une forme définitive, $\varphi^\infty(A)$, du code.

Réponse: *Dans ce cas, les nœuds* ${r_i}^{i\in I}$ *sont aussi coloriables dans* $\mathcal{G}(\varphi^\#(A))$*. Par conséquent, il existe un coloriage pour lequel l'ensemble* S *des nœuds non coloriables est un sous ensemble de* Q *égal à* ${q_i}^{i\in I}$*. Comme les temporaires* S *n'apparaissent que dans des instructions de la forme* **mmv** $Q_j,\ p_j$ *(où* $Q_j \subset Q$*) ou bien* **move** $r_i,\ q_i$*, il suffit de remplacer les premières par* ***sw*** p_j*, 0(****\$sp****) suivi de* **mmv** $Q_j \setminus S,\ p_j$ *et les secondes par* ***lw*** r_i*, 0(****\$sp****) pour obtenir le code* $\varphi^\infty(A)$*. Le graphe d'interférence résultant est un sous-graphe du précédent, sans les nœuds mis en pile, donc il est coloriable, avec les mêmes couleurs. Aussi on n'a pas besoin de refaire un nouveau coloriage.*

6.4 *(facultative)* Quand le graphe n'est pas coloriable, il existe, en général plusieurs coloriages partiels possibles. Comment peut-on guider l'algorithme de coloriage du cours pour qu'il aboutisse, autant que possible, vers une forme permettant d'obtenir le code définitif $\varphi^\infty(A)$ à partir du coloriage de $\mathcal{G}(\varphi^\#(A))$?

Réponse: *Lors de l'allocation de registres, lorsqu'il n'existe plus de nœuds simplifiables (*i.e. *on n'est alors pas sûr de pouvoir colorier le graphe), on choisit de façon optimiste un nœud complexe que l'on simplifie. On peut guider ce choix en prenant par priorité :*

- *les registres* ${q_i}^{i\in I}$ *(que l'on s'attend à mettre en pile)*
- *les registres anciens (autres que* ${q_i}^{i\in I}$*,* ${r_i}^{i\in I}$ *et* ${p_j}^{j\in J}$*) que l'on espère ne pas avoir à mettre en pile.*
- *les registres éphémères* ${p_j}^{j\in J}$ *et* ${r_i}^{i\in I}$*, que l'on ne veut surtout pas mettre en pile.*