Votre nom :

Correction du partiel CMP2

EPITA – Ing1 promotion 2014
Tous documents (notes de cours, polycopiés, livres) autorisés
Calculatrices, ordinateurs, tablettes et téléphones interdits.

Juin 2012 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain.

Best-of: Le *best of* est tiré des copies des étudiants ; les fautes de français aussi, le cas échéant.

Lisez bien les questions, chaque mot est important. Écrivez court, juste et bien; servez vous d'un brouillon. Une argumentation informelle mais convaincante est souvent suffisante. Gérez votre temps, ne restez pas bloqué sur les questions les plus difficiles. Une lecture préalable du sujet est recommandée.

Cette correction contient 18 pages. Les pages 1–15 contiennent l'épreuve et son corrigé. Ce document comporte en pages 16–18 une enquête (facultative) sur le cours et le projet Tiger (y répondre sur la feuille de QCM qui vous est fournie).

Écrivez votre nom en haut de la première page du sujet et rendez-le avec votre copie et la feuille de QCM.

1 La mémoire dans tous ses états

1. En compilation, qu'appelle-t-on une activation ?

Correction: Une fonction en cours d'exécution.

Best-of:
- L'activation permet un changement d'état.
- C'est lorsqu'un registre est alloué pour une variable.
- Il s'agit d'une partie de code requérant le passage du compilateur.
- Une activation consiste à entrer une valeur en mémoire.
- Une activation est le fait de sauvegarder une variable dans un registre.
- Une requête mémoire d'une variable ou fonction.

2. Qu'est-ce qu'un bloc d'activation ? Que contient-il ?

Correction: Réponse courte : Un bloc d'activation (*activation record*) ou cadre de pile (*stack frame* est un élément de la pile d'appels de fonctions d'un processus ou d'un fil d'exécution (*thread*). Il s'agit d'un espace de mémoire (contigu) contenant les données relatives à une activation.
Éléments de réponse supplémentaires : Parmi les éléments qui composent un bloc d'activation, on trouve habituellement :

- les arguments passés à la fonction ;
- le lien statique vers la fonction "parente" (le cas échéant) ;
- l'adresse de retour, indiquant où l'exécution doit reprendre au terme de la fonction ;
- des variables locales ;
- des variables temporaires (non nommées) ;
- des registres sauvegardés (sous la coupe de l'appelé ; *callee-save registers*) ;

Best-of:

- Il contient du code independant.

3. Citez un mécanisme d'allocation dynamique sur la pile en C et un mécanisme d'allocation dynamique sur le tas en C.

Correction: Le mot "dynamique" était important dans la question, en particulier concernant l'allocation sur la pile : les réponses attendues devaient citer des mécanismes d'allocation capables d'utiliser une taille de bloc connue seulement à l'exécution.
Les tableaux dynamiques (disponibles depuis le C99) et la routine `alloca()` (non POSIX) permettent d'allouer dynamiquement sur la pile. Quant à l'allocation sur le tas, elle se fait habituellement en C avec `malloc()`.
J'ai vu une mention à `stackalloc` concernant l'allocation dynamique sur la pile, ce qui est juste, mais en C# uniquement à ma connaissance.

Best-of:

- Le mot clef `static` permet de faire une allocation sur le tas.
- [Allocation sur le] tas : surcharge de fonctions.
- Déclaration d'une nouvelle fonction ([allocation] sur le tas).
- En C, le linkage des `*.o` est une allocation dynamique sur le tas.
- On a l'allocation par référence et l'allocation par pointeur.
- Allocation dynamique sur le tas : valeur de retour d'une fonction.
- Pour allouer sur la pile, on utilise le mot clé `static` avant l'allocation de la mémoire par `malloc`.

4. Quelle différence faites-vous entre l'allocation dynamique sur la pile (*stack*) et l'allocation dynamique sur le tas (*heap*) ?

   Correction: Réponse courte : Dans le premier cas, cela revient à déplacer le *stack pointer* (pointeur de pile) pour agrandir le cadre de pile de l'activation courante (la fonction en cours d'exécution).
   Le second cas repose sur un allocateur de type `malloc()` ou `new`, qui gèrent la zone mémoire du tas pour y allouer des blocs mémoire.
   Éléments de réponse supplémentaires : L'algorithme d'allocation dynamique ainsi que la structure de données utilisée pour organiser le tas dépend de l'allocateur. Lorsque la taille du tas est insuffisante pour que l'allocateur puisse en extraire un nouvel emplacement mémoire demandé, l'allocateur étend la taille du tas à l'aide des appels systèmes `brk()`, `sbrk()` ou `mmap()`, qui déplacent la limite haute du tas et allouent de nouvelles pages mémoire pour la quantité de mémoire correspondant à cette extension.

   Best-of:
   - Les constantes sont allouées sur le tas.

5. Comment fonctionne la libération de la mémoire dans chaque cas (allocation dynamique sur la pile, allocation dynamique sur sur le tas) ?

   Correction: Dans le cas de l'allocation dynamique sur la pile, la durée de vie du bloc alloué est fonction de l'activation courante : au retour de cette fonction, cet emplacement sera automatiquement libéré, au même titre que l'ensemble des données stockées sur la piles (variables locales).

   L'allocation dynamique sur le tas est par contre indépendante de toute activation (notamment celle qui l'a créé). La libération peut-être manuelle ou automatique. Dans le premier cas, elle s'effectue sous la responsabilité de l'utilisateur, qui doit invoquer le service de récupération mémoire idoine (`free()`, `delete`, etc.). Dans le second cas, c'est le *garbage collector* ("ramasse-miettes"), un composant du *run time* (via une machine virtuelle, une bibliothèque dédiée, etc.) qui s'en chargera, dès lors que le bloc sera identifié comme non référencé (et donc "recyclable").

   Certaines copies affirment que les données allouées sur la pile sont libérées à la fin du bloc (de code) qui les contient ; il s'agit là d'une confusion entre deux notions : la *portée* d'une variable et sa *durée de vie*. L'emplacement mémoire contenant une variable allouée sur la pile est libéré au retour de la fonction qui la contient, bien que la portée de la variable soit limitée au bloc dans lequel elle est déclarée.

   Best-of:
   - (...) l'algo de gestion de la mémoire (first feat, best feat, ...) (*`malloc` fait effectivement des prouesses !*)
   - La libération sur la pile se fait à la fin du programme (sauf libération forcée) alors que l'on doit obligatoirement libérer les objets sur le tas sinon il restent et saturent la mémoire.

6. Quels sont les avantages de chaque système ?

Correction: **Réponse courte :** L'allocation dynamique sur la pile est très peu coûteuse en termes de vitesse et d'utilisation mémoire, ne produit pas de fragmentation et garantit que l'espace alloué sera toujours libéré. Mais contrairement aux allocateur dynamiques, elle offre peu de moyens face aux cas d'erreurs : un échec d'allocation peut se traduire par un `SIGSEGV`. De plus, la pile est un espace mémoire très limité (quelques MiO), qui ne peut être utilisé pour stocker de grosses données. L'allocation dynamique permet d'allouer des blocs mémoire bien plus conséquents. Enfin, la durée de vie d'une variable sur le tas peut excéder celle de la fonction qui l'a créée, contrairement à une variable située sur la pile.
Réponse longue : L'allocation dynamique sur la pile est très peu coûteuse en termes de vitesse et d'utilisation mémoire : elle consiste à déplacer le pointeur de pile, ce qui correspond essentiellement à une instruction machine MOVE. GCC traduit habituellement un appel à `alloca()` par un code mis en ligne (*inlined*) correspondant le plus souvent à cette unique instruction. Par ailleurs, l'allocation dynamique sur la pile ne produit pas de fragmentation mémoire, comme cela peut arriver avec l'allocation sur le tas. Enfin, les données allouées sur la pile sont toujours libérées, même si la fonction ne retourne pas normalement à son appelant, par exemple suite à un appel à `longjmp()` ou à une levée d'exception non rattrapée (en C++).
L'un des défauts majeurs de l'allocation dynamique est le peu de support en cas d'échec d'allocation : le plus souvent, cela se traduira par un signal similaire à celui d'un appel récursif non borné (`SIGSEGV`). Les allocateurs dynamiques sur le tas offrent des mécanismes d'erreur plus clairs et plus utilisables (en cas d'erreur, `malloc()` renvoie `0`, tandis que `new` lève une exception `std::bad_alloc`). De plus la pile est un espace mémoire très limité, qui ne peut être employé pour allouer de gros blocs mémoire. Sous Linux, elle a habituellement une taille de 8 MiO par défaut. Les allocateurs sur le tas peuvent allouer bien plus de mémoire ; leurs limites sont fonction de la quantité de mémoire physique, de la taille du fichier d'échange sur le disque (`swap`) ou encore la quantité de mémoire virtuelle que le processus peut adresser. Enfin, l'allocation sur le tas permet de créer des blocs de mémoire dont la durée de vie ne dépend pas d'une activation, ce qui permet la communication entre fonctions.

7. Rappeler quels sont les stratégies d'allocations utilisées en Tiger et pour quels types de données elles sont utilisées.

Correction: Les variables de types entier (`int`) sont allouées (statiquement) sur la pile. Les chaînes de caractères (`string`) et les types agrégats/utilisateur (tableaux, enregistrements, objets) sont alloués dynamiquement sur le tas.

8. Soit la fonction Tiger `f` suivante :

```
function f (x : int) : int = x + 1
```

`tc` génère pour `f` le code MIPS suivant :

```
1  tc_l0:
2          sw      $fp, -8 ($sp)
3          move    $fp, $sp
4          sub     $sp, $sp, 12
5          sw      $a0, ($fp)
6          sw      $a1, -4 ($fp)
7  l1:
8          lw      $t0, -4 ($fp)
9          add     $v0, $t0, 1
10 l2:
11         move    $sp, $fp
12         lw      $fp, -8 ($fp)
13         jr      $ra
```

(a) Quelle est la taille du cadre de pile de `f` ?

Correction: On voit à la ligne 4 que le stack pointer (`$sp`) est décrémenté de 12 octets (3 mots mémoire de 32 bits) : il s'agit de la taille du cadre de pile de `f`.

Best-of: De nombreuses propositions, parfois sans unité :

- 1 octet
- 8 bits
- 12 bits
- 32 bits
- 8 octets
- 16 octets
- 20 octets
- 2
- 5
- 6
- 13
- 20
- $2 \times$ `sizeof(int)`

(b) Détaillez le contenu de ce cadre de pile.

Correction: Dans les lignes qui suivent, $\texttt{\$fp}_\texttt{f}$ désigne le frame pointer de `f`. Le cadre de pile de `f` contient trois variables de la taille d'un mot mémoire (qui est fixé à 32 bit dans le code généré par `tc`).

- À l'adresse $\texttt{\$fp}_\texttt{f}$: le *static link* de la fonction "parente" de `f`, initialisé avec la valeur du registre d'argument `$a0` (cf. ligne 5).
- À l'adresse $\texttt{\$fp}_\texttt{f}$`-4` : l'argument `x` de `f`, initialisé avec la valeur du registre d'argument `$a1` (cf. ligne 6).
- À l'adresse $\texttt{\$fp}_\texttt{f}$`-8` : la sauvegarde du frame pointer de l'activation précédente (la fonction ayant appelé `f`). Celui-ci est stocké à l'adresse `[$sp-8]` par l'instruction de la ligne 2, où `$sp` est le stack pointer de l'activation précédente, égal au frame pointer de `f` (cf. ligne 2).

Best-of:

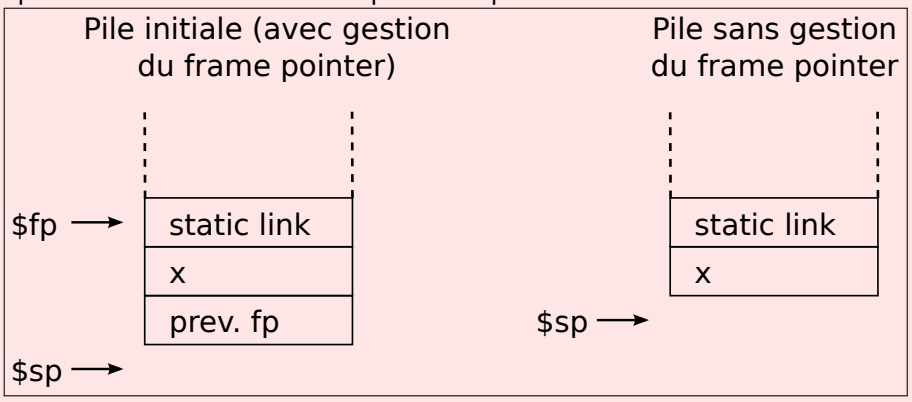
- Un entier.

(c) `tc` pourrait tout à fait générer le code de `f` sans utiliser le frame pointer (`$fp`). Réécrivez ce code sans vous servir de `$fp`.

Correction: Réponse courte :

```
tc_l0:
        sub     $sp, $sp, 8
        sw      $a0, 8 ($sp)
        sw      $a1, 4 ($sp)
l1:
        lw      $t0, 4 ($sp)
        add     $v0, $t0, 1
l2:
        add     $sp, $sp, 8
        jr      $ra
```

Éléments de réponse supplémentaires : Si l'on s'abstient d'utiliser `$fp`, la modification la plus essentielle est de manipuler les données présentes sur la pile en utilisant `$sp`. Étant donnée que la taille du cadre de pile d'une fonction Tiger est fixe, cela est simple. Il faut également remarquer que la sauvegarde et la restauration du frame pointer de la fonction appelante (resp. lignes 2 et 12 du code originel) deviennent superflus et diminuent d'un mot mémoire le cadre de pile, qui a alors une taille de 8 octets.
Le diagramme ci-dessous montre la structure du cadre de pile de `f` avant et après la modification introduite par cette question.



(d) La suppression du frame pointer du code généré, illustrée dans la question précédente, est en fait une optimisation proposée par certains compilateurs, qui permet de simplifier les prologues et les épilogues des routines. Il existe cependant des situations dans lesquelles cette optimisation n'est pas envisageable. Citez-en une.

Correction: Le frame pointer devient essentiel dès lors que la taille du cadre de pile n'est plus statiquement connue, car alors le stack pointer ne peut plus servir de point de référence. On peut citer les deux situations suivantes :

- l'allocation dynamique sur la pile, qui déplace le stack pointer dynamiquement pour agrandir le bloc d'activation ;
- les fonctions ayant une liste d'arguments variable (comme `printf`), dont la taille du cadre de pile n'est pas connue statiquement.

Correction: (suite)
Sur certaines machines, le frame pointer est également nécessaire au bon fonctionnement du debugger.

J'ai aussi accepté comme réponse valide le support des variables non locales, car le fonctionnement des variables locales en Tiger vu en cours s'appuie sur le static link, dépendant lui-même de la sauvegarde du frame pointer. Il faut cependant bien comprendre que l'on pourrait tout aussi bien implémenter le static link en utilisant le stack pointer *et* l'information de la taille du cadre de pile de chaque fonction (à condition que celle-ci soit connue statiquement, ce qui nous ramène au problème précédent), puisque ces deux données suffisent à retrouver le frame pointer.

Le support de la récursion, mentionnée dans plusieurs copies, ne nécessite pas intrinsèquement de frame pointer. Rappelons au passage qu'un langage (ou même tout simplement un programme) sans fonctions récursives ne nécessite pas de pile en théorie : comme chaque fonction a au plus une activation à tout instant, on peut borner à la compilation la quantité de mémoire utilisée par l'ensemble des fonctions et l'allouer statiquement, sans avoir recours à une pile.

Best-of:
- Pour la routine `main`, ce n'est pas possible.
- Cette optimisation n'est pas envisageable si on a alloué une variable sur le tas.
- L'inlining ne permet pas s'effectuer sur des fonctions recursives (*sic*).

2 Partie terminale

Dans cet exercice, on considère une machine hypothétique, dont le jeu d'instructions est suffisamment clair pour ne pas avoir à être défini et qui possède trois registres `r1`, `r2` et `r3`. Les deux premiers sont sous la responsabilité de l'appelant (*caller-save*) et le dernier sous celle de l'appelé (*callee-save*). Par soucis de simplification, on considérera comme complètement implicite la gestion d'éléments tels que l'adresse de retour d'une routine, le frame pointer ou le stack pointer.

1. Définissez la partie terminale d'un compilateur.

   Correction: *Back end* : tout ce qui est dédié à la cible.

2. Considérons le programme suivant, constitué de deux fonctions `f` et `g`, pour lequel l'allocation des registres n'a pas encore été effectuée.

```
1  f:      t   ←  r3
2          n   ←  r1
3          r1  ←  n
4          r2  ←  1
5          call g
6          v   ←  r1
7          r1  ←  v
8          r3  ←  t
9          return
```

```
10 g:      t   ←  r3
11         a   ←  r1
12         b   ←  r2
13         if a <= 1 jump l1
14         d   ←  b * a
15         c   ←  a - 1
16         r1  ←  c
17         r2  ←  d
18         call g
19         v   ←  r1
20         jump exit
21 l1:     v   ←  b
22 exit:   r1  ←  v
23         r3  ←  t
24         return
```

Écrivez un programme Tiger ou C équivalent à ce programme.

Correction: Tout d'abord, mes excuses concernant la coquille présente dans le code de `f`, à la ligne 2 de l'énoncé originel, qui comportait l'instruction '`a ← r1`', au lieu de l'instruction '`n ← r1`' (rectifiée dans la présente version du sujet). J'en ai bien entendu tenu compte dans lors de la correction.

Voici une version "naïve" du code précédent en Tiger :

```
function f (n : int) : int =
  g (n, 1)

function g (a : int, b : int) : int =
  if a <= 1 then
    b
  else
    let
      var d := b * a
      var c := a - 1
    in
      g (c, d)
    end
```

Voici une version plus élégante de ce programme[a], où `f` et `g` ont été renommés respectivement en `fact` et `fact_` :

```
function fact (n : int) : int =
  let
    function fact_ (n : int, acc : int) : int =
      if n <= 1 then acc else fact_ (n - 1, acc * n)
  in
    fact_ (n, 1)
  end
```

Les versions sans récursion terminale ou itératives, étant non fidèles au programme initial, n'ont pas été acceptées.

[a] Il s'agit en fait du code originel dont on a déduit la version en langage d'assemblage de l'énoncé.

3. À quoi peut servir ce programme ?

   Correction: Il permet de calculer $n!$ (factorielle n), de façon récursive.

   Best-of:
   - Ce programme peut servir à trier des données.
   - Il permet de calculer la puissance d'un nombre.

4. D'après ce programme, détailler les conventions d'appel de cette machine ; en particulier, pour chacun des registres, dire lesquels servent au passage d'argument(s) et au retour de résultat.

   Correction: Il est sûr que `r1` sert pour le premier argument et aussi pour retourner une valeur ; `r2` est uniquement utilisé pour passer un second argument.

5. Étant données les conventions d'appel, le sens du programme, expliquer quelles sont les variables vives lors du `return` de la fonction `g` (ligne 24).

   Correction: Puisqu'elle est sous la responsabilité de l'appelé, `r3` est vive. Puisqu'elle sert au retour de valeur, `r1` l'est aussi.

6. Expliquez pourquoi les lignes

```
t  ←  r3
```

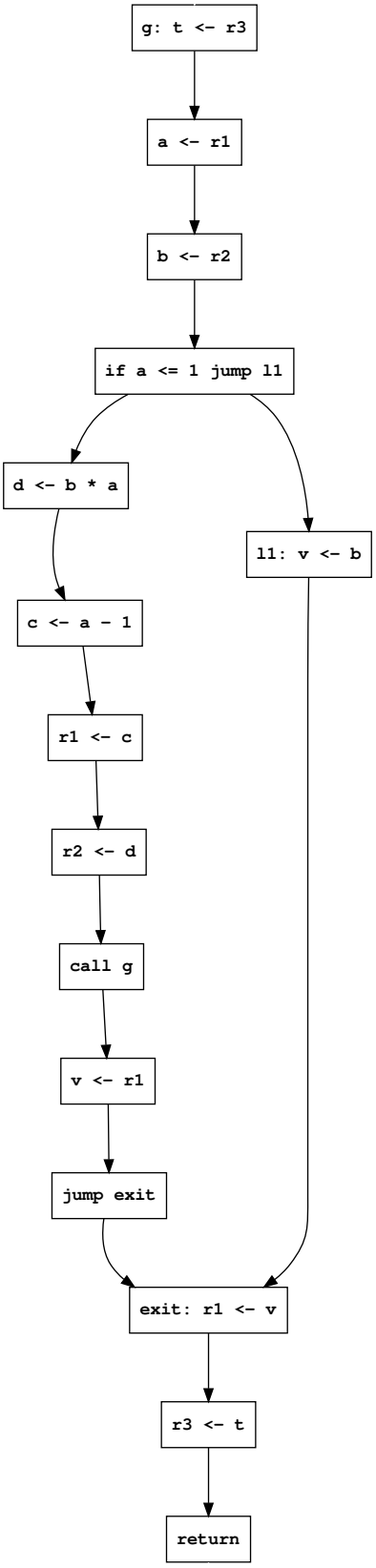
   et

```
r3  ←  t
```

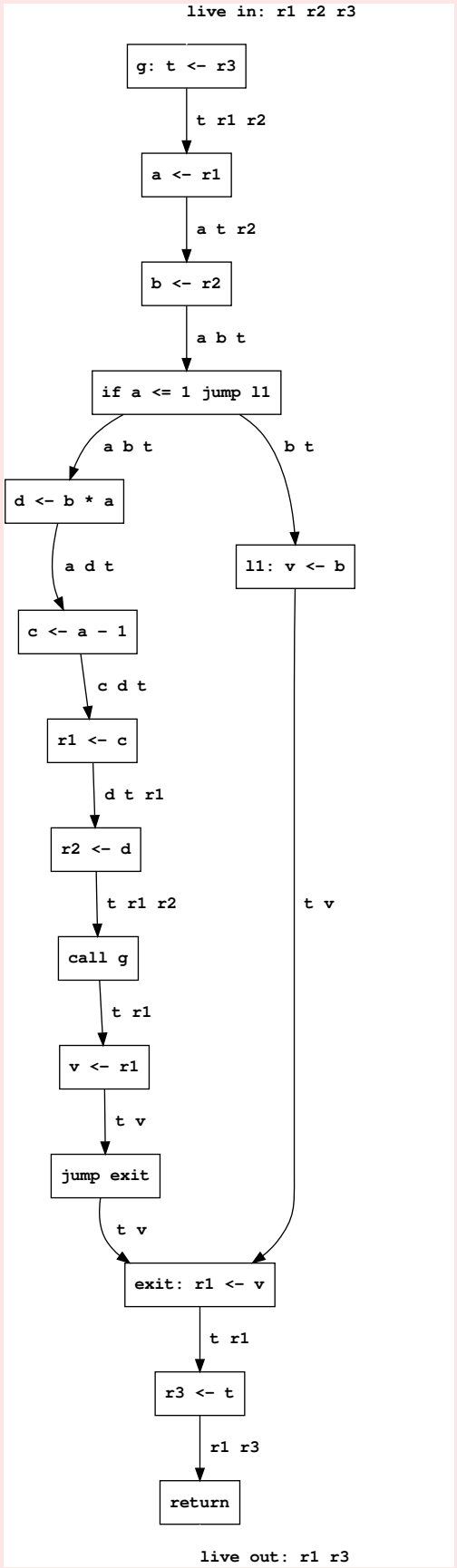
   ont été générées.

   Correction: Il faut préserver `r3`. Sans ces lignes, cela signifierait simplement que `r3`, étant vive sur tous les arcs, serait en conflit avec toutes les temporaires, donc inutilisable par l'allocateur des registres. En clair, on compilerait avec deux registres (`r1` et `r2`) au lieu de trois. Ici, on se donne la possible de se servir de `r3`, quitte à devoir sauver sa valeur initiale sur la pile, puis la restaurer à la fin : `r3` reste disponible pour le corps de la fonction.

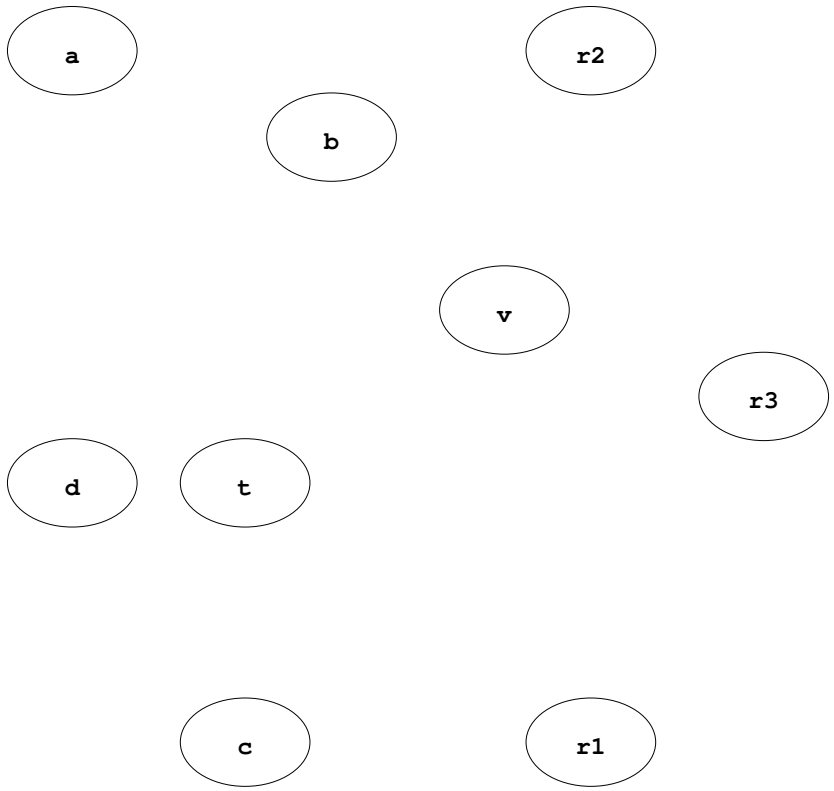
7. Étiquetez les arcs du graphe de contrôle de flux de la fonction `g` ci-après avec les temporaires vives. N'oubliez pas d'indiquer également les temporaires vives en entrée et en sortie du programme. **Répondez directement sur le sujet**.


g: t <- r3
a <- r1
b <- r2
if a <= 1 jump l1
d <- b * a
l1: v <- b
c <- a - 1
r1 <- c
r2 <- d
call g
v <- r1
jump exit
exit: r1 <- v
r3 <- t
return

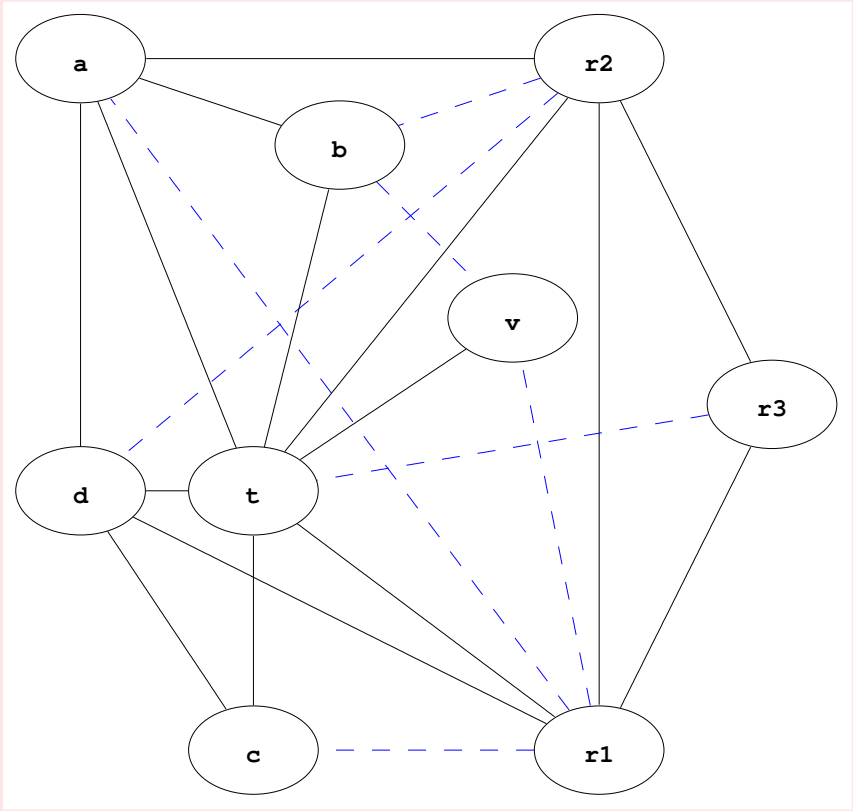
Correction:


live in: r1 r2 r3
g: t <- r3
t r1 r2
a <- r1
a t r2
b <- r2
a b t
if a <= 1 jump l1
a b t
b t
d <- b * a
l1: v <- b
a d t
c <- a - 1
c d t
r1 <- c
d t r1
r2 <- d
t r1 r2
t v
call g
t r1
v <- r1
t v
jump exit
t v
exit: r1 <- v
t r1
r3 <- t
r1 r3
return
live out: r1 r3

8. Complétez le graphe d'interférence ci-dessous en traçant une arête pour chaque interférence entre deux variables. Faites-y aussi figurer les fusions possibles (*coalesces*) avec des arêtes en pointillés (et de préférence en utilisant un stylo d'une autre couleur). **Répondez directement sur le sujet.**



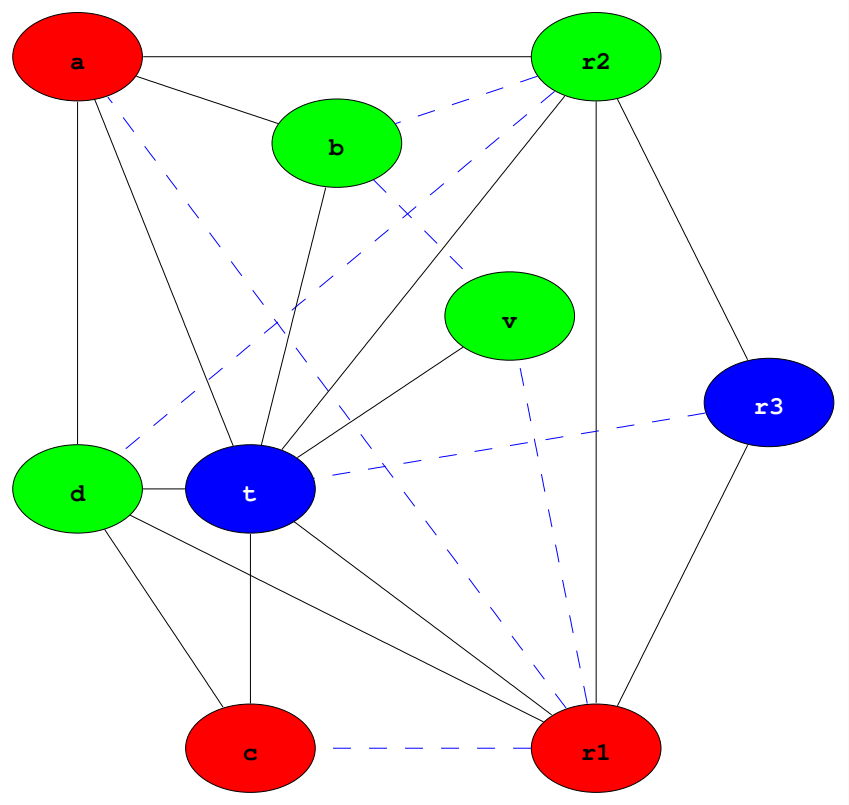
Correction:



9. Le graphe précédent est-il colorable ?

   - Si oui, précisez quelles registres vous affectez à chaque variable (de préférence en coloriant directement le graphe précédent en utilisant des stylos de couleurs **différentes** ou à défaut en indiquant ces affectations sur votre copie).
   - Si non, indiquez et justifiez le meilleur choix de variable à verser sur la pile, puis réécrivez le code de la fonction `g` en intégrant le versement de cette variable sur la pile.

Correction: Ce graphe est colorable tel quel.



Dans beaucoup trop de copies, la coloration proposée attribue la même couleur à deux registres matériels (c'est-à-dire, deux variables nommées '`r`*x*') : ce n'est juste pas possible *physiquement* ! De même, les solutions exhibant plus de trois couleurs (ou des registres inexistants, ce qui revient au même) sont aberrantes. Certaines copies avancent que le graphe n'est pas colorable puisque *l'un* de ses nœuds est de degré significatif (c'est-à-dire, a 3 voisins ou plus), mais cet argument n'est pas suffisant pour justifier cette conclusion.

Best-of:

- Ce graphe n'est pas coloriable. Il faut mettre `r3` dans la pile plutôt que dans `t`.
- Oui [ce graphe est colorable] car il est planaire.

10. **Bonus.** Le code de la fonction `g`, visible à la question 2, est un cas de récursion terminale (*tail-recursion*) : `g` s'appelle récursivement (cf. ligne 18), mais cet appel est la dernière instruction à être évaluée (avant le retour de la fonction). Dans cette situation, il est possible d'optimiser le code en transformant cette récursion en une *itération*. Réécrivez le code de `g` en lui appliquant cette optimisation.

Correction: L'idée consiste à transforme l'appel récursif (`call g`) en un saut au début de la fonction (`jump g`), ce qui donne le code du Listing 2.

Listing 1: Code initial de `g`

```
10 g:     t  ← r3
11        a  ← r1
12        b  ← r2
13        if a <= 1 jump l1
14        d  ← b * a
15        c  ← a - 1
16        r1 ← c
17        r2 ← d
18        call g
19        v <- r1
20        jump exit
21 l1:    v  ← b
22 exit:  r1 ← v
23        r3 ← t
24        return
```

Listing 2: Code de `g` sans *tail call*

```
10 g:     t  ← r3
11        a  ← r1
12        b  ← r2
13        if a <= 1 jump l1
14        d  ← b * a
15        c  ← a - 1
16        r1 ← c
17        r2 ← d
18        jump g


21 l1:    v  ← b
22        r1 ← v
23        r3 ← t
24        return
```

3 À propos de ce cours

Pour terminer cette épreuve, nous vous invitons à répondre à un petit questionnaire. Les renseignements ci-dessous ne seront bien entendu pas utilisés pour noter votre copie. Ils ne sont pas anonymes, car nous souhaitons pouvoir confronter réponses et notes. En échange, quelques points seront attribués pour avoir répondu. Merci d'avance.

Sauf indication contraire, vous pouvez cocher plusieurs réponses par question. Répondez sur la feuille de QCM. N'y passez pas plus de dix minutes.

Cette épreuve

Q.1 Sans compter le temps mis pour remplir ce questionnaire, combien de temps ce partiel vous a-t-il demandé (si vous avez terminé dans les temps), ou combien vous aurait-il demandé (si vous aviez eu un peu plus de temps pour terminer) ?

a. Moins de 30 minutes.
b. Entre 30 et 60 minutes.
c. Entre 60 et 90 minutes.
d. Entre 90 et 120 minutes.
e. Plus de 120 minutes.

Q.2 Ce partiel vous a paru

a. Trop difficile.
b. Assez difficile.
c. D'une difficulté normale.
d. Assez facile.
e. Trop facile.

Le cours

Q.3 Quelle a été votre implication dans les cours CMP2?

a. Rien.
b. Bachotage récent.
c. Relu les notes entre chaque cours.
d. Fait les annales.
e. Lu d'autres sources.

Q.4 Ce cours

a. Est incompréhensible et j'ai rapidement abandonné.
b. Est difficile à suivre mais j'essaie.
c. Est facile à suivre une fois qu'on a compris le truc.
d. Est trop élémentaire.

Q.5 Ce cours

a. Ne m'a donné aucune satisfaction.
b. N'a aucun intérêt dans ma formation.
c. Est une agréable curiosité.
d. Est nécessaire mais pas intéressant.
e. Je le recommande.

Q.6 La charge générale du cours en sus de la présence en amphi (relecture de notes, compréhension, recherches supplémentaires, etc.) est

a. Telle que je n'ai pas pu suivre du tout.
b. Lourde (plusieurs heures de travail par semaine).
c. Supportable (environ une heure par semaine).
d. Légère (quelques minutes par semaine).

Les formateurs

Q.7 L'enseignant

a. N'est pas pédagogue.
b. Parle à des étudiants qui sont au dessus de mon niveau.
c. Me parle.
d. Se répète vraiment trop.
e. Se contente de trop simple et devrait pousser le niveau vers le haut.

Q.8 Les assistants

a. Ne sont pas pédagogues.
b. Parlent à des étudiants qui sont au dessus de mon niveau.
c. M'ont aidé à avancer dans le projet.
d. Ont résolu certains de mes gros problèmes, mais ne m'ont pas expliqué comment ils avaient fait.
e. Pourraient viser plus haut et enseigner des notions supplémentaires.

Le projet Tiger

Q.9 Vous avez contribué au développement du compilateur de votre groupe (une seule réponse attendue) :

a. Presque jamais.
b. Moins que les autres.
c. Équitablement avec vos pairs.
d. Plus que les autres.
e. Pratiquement seul.

Q.10 La charge générale du projet Tiger est

a. Telle que je n'ai pas pu suivre du tout.
b. Lourde (plusieurs jours de travail par semaine).
c. Supportable (plusieurs heures par semaine).
d. Légère (une ou deux heures par semaine).
e. J'ai été dispensé du projet.

Q.11 Y a-t-il de la triche dans le projet Tiger ? (Une seule réponse attendue.)

a. Pas à votre connaissance.
b. Vous connaissez un ou deux groupes concernés.
c. Quelques groupes.
d. Dans la plupart des groupes.
e. Dans tous les groupes.

Questions 12-18 Le projet Tiger vous a-t-il bien formé aux sujets suivants ? Répondre selon la grille qui suit. (Une seule réponse attendue par question.)

a Pas du tout	**b** Trop peu	**c** Correctement	**d** Bien	**e** Très bien

Q.12 C++.

Q.13 modélisation orientée objet et *design patterns*.

Q.14 anglais technique.

Q.15 compréhension du fonctionnement des ordinateurs.

Q.16 compréhension du fonctionnement des langages de programmation.

Q.17 travail collaboratif.

Q.18 outils de développement (contrôle de version, systèmes de construction, débogueurs, générateurs de code, etc.)

Questions 19-34 Comment furent les étapes du projet ? Répondre selon la grille suivante. (Une seule réponse attendue par question ; ne pas répondre pour les étapes que vous n'avez pas faites.)

a Trop facile. **b** Facile. **c** Nickel. **d** Difficile. **e** Trop difficile.

Q.19 Rush `.tig` : mini-projet en Tiger (Logomatig).

Q.20 `TC-0`, Scanner & Parser.

Q.21 `TC-1`, Scanner & Parser, Tâches, Autotools.

Q.22 `TC-2`, Construction de l'AST et pretty-printer.

Q.23 `TC-3`, Liaison des noms et renommage.

Q.24 `TC-4`, Typage et calcul des échappements.

Q.25 `TC-5`, Traduction vers représentation intermédiaire.

Q.26 `TC-6`, Simplification de la représentation intermédiaire.

Q.27 `TC-7`, Sélection des instructions.

Q.28 `TC-8`, Analyse du flot de contrôle.

Q.29 `TC-9`, Allocation de registres.

Q.30 `TC-D (option)`, Désucrage (boucles `for`, comparaisons de chaînes de caractères).

Q.31 `TC-I (option)`, Mise en ligne du corps des fonctions.

Q.32 `TC-B (option)`, Vérification dynamique des bornes de tableaux.

Q.33 `TC-A (option)`, Surcharge des fonctions.

Q.34 `TC-O (option)`, Désucrage des constructions objets.