Correction du partiel CMP2

EPITA – Promo 2011
Tous documents (notes de cours, polycopiés, livres) autorisés
Calculatrices et ordinateurs interdits.

Mars 2009 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain et Akim Demaille. Le *best of* est tiré des copies des étudiants, fautes de français y compris, le cas échéant.

Une copie synthétique, bien orthographiée, avec un affichage clair des résultats, sera toujours mieux notée qu'une autre demandant une quelconque forme d'effort de la part du correcteur. Une argumentation informelle mais convaincante, sera souvent suffisante.

Cette épreuve est longue ; le but n'est pas de répondre vite et mal à toutes les questions, mais plutôt d'en faire bien une partie significative. Une lecture préalable du sujet est recommandée.

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

1 Bootstrap

1. Combien de nombres entiers différents peut-on représenter avec 64 bits ?

   Correction: 2^{64}.

Best-of: Cette question a suscité de nombreuses réponses différentes.

- 4
- 8
- 16
- 127
- 128
- 2147483648
- 2^6
- 2^8
- 2^{63}
- $2^{63} - 1$
- $2^{64} = 45536$ possibilités
- $2^{64} - 1$
- $2^{64} + 1$
- $2^{64}/8$
- 2^{65}
- $2^{64} \times 2^{64}$
- $$\begin{aligned} 2^{64} &= \left((2^{10})^3 \times 2^2\right)^2 \\ &= (1024^3 \times 4)^2 \\ &= 16 \times 1073741824^2 \\ &= 1152921505606446976 \end{aligned}$$
- 7144842452347387905
- Les entiers sur l'architecture MIPS utilisés étaient sur 32 bits, donc 2, mais cette question dépend totalement du processeur.
- Cela dépend de l'architecture. Typiquement, un entier est codé sur 4 bits, d'où 16.
- 64 bits = 8 octets ; $2^8 = 256$; on peut donc coder 256 entiers.
- On peut coder 2 entiers (de 4 octets).
- On peut représenter $\Big[\overbrace{(2^{64} - 1)}^{\text{entiers positifs}} + \overbrace{(2^{63} - 1)}^{\text{entiers négatifs}} \Big]$ nombres différents avec 64 bits.

2. Combien de nombres *flottants* différents peut-on représenter avec 64 bits ?

Correction: Idem, 2^{64} en théorie. En pratique, c'est souvent « un peu moins » ; voir par exemple les standards IEEE 754, où les valeurs NaN (*Quiet NaN* (QNaN) et *Signalling NaN* (SNaN)) ne sont pas codées de façon unique.

Best-of: Là encore, pléthore de résultats intéressants !
- 2
- 4
- 16
- 127
- 2147483648
- 2^4
- 2^{31}
- 2^{32}
- $2^{32} - 1$
- $2^{32} + 2^{31} - 1$
- 2^{53}
- 2^{54}
- $2^{64} \times 2^{-5} = 2^{59}$
- 2^{63}
- $2^{63} - 1$
- $2^{64} = 45536$ possibilités
- $2^{64} - 1$
- [...] $2^{64-1} - 1$ entiers différents en signés.
- $2^{64}/2$
- $2^{64}/16$
- $2^{64} - 2^{53} + 2$
- Moins de 2^{64}
- $2^{64-(}$nombre de bits de la mantisse$)$
- 1152921505606446976
- Il existe $2^{512}-1$ nombres différents sur 64 bits (1 bits = 8 octets = 2^8-1 nombres).
- Beaucoup ($\gg 2^{64}$)
- Avec 64 bits, on peut représenter énormément de nombres flottants.
- On peut coder 1 flottant (de 8 octets).
- On peut représenter $\left[(2^{32}-1)+(2^{31}-1)\right]$ nombres flottants différents avec 64 bits, car ceci sont sur le double de la taille d'un entier.
- Cela dépend du nombre de chiffres après la virgule.
- Un grand nombre
- Une infinité

3. Pourquoi est-ce que l'écriture

```
print_int (-2147483648 / 42)
```

est interdite par notre compilateur Tiger [1]. ?

1. Pour information, $-2147483648 = -2^{31}$

Correction: Parce que l'expression $-n$ est désucrée en `0 - n` par le parser, où n est considéré comme un entier positif. Or 2147483648 (=2^{31}) n'est pas un entier 32-bit valide : la plus grand valeur positive acceptable (`MAX_INT`) est 2147483647 (=$2^{31}-1$).
Plusieurs copies affirment que -2^{31} ne peut être codé sur 32 bits, ce qui est faux sur la plupart des machines, où le type « entiers sur 32 bits » représente l'intervalle $[\![-2^{31}, 2^{31}-1]\!]$ (allez donc jeter un oeil dans '`/usr/include/limits.h`').
La vraie justification découle des limitations de notre couple scanner/parser Tiger évoquées précédemment, qui accepte des valeurs dans l'intervalle $[\![-2^{31}-1, 2^{31}-1]\!]$.

```
% echo '-2147483648'  | tc --parse -
standard input:1.2-11: integer out of bounds: 2147483648
```

Cependant, même si l'entier *littéral* -2147483647 est refusé par le scanner, cette valeur est toujours atteignable (représentée) sur 32 bits en Tiger, et le code suivant est parfaitement valide.

```
% echo '(print_int(-2147483647 - 1) ; print("\n"))' \
    | tc -H - >/tmp/foo.hir
% havm /tmp/foo.hir
-2147483648
```

Best-of:
- On ne peut pas garder en mémoire le nombre flottant résultant de l'opération. (*Il n'y a pas de flottants en Tiger !*)
- $-2147483648/42$ est négatif or `print_int` ne print que des entiers positifs.
- Cette écriture est interdite car `print_int` ne gère pas les calculs.
- Ces deux nombres n'ont pas le même type.
- `print_int` prend un entier en paramètre, or `-2147483648 / 42` est une expression.
- L'opérateur '`/`' est prioritaire sur l'opérateur '`-`'.
- Cette notation est interdite par le compilateur parce qu'il doit calculer son argument.
- Tiger ne gère pas les nombres négatifs.
- L'opérateur de division doit certainement causer un buffer-overflow avec ce type de nombres élevés.
- Le nombre -2147483648 n'existe pas.
- NYI [« Not Yet Implemented » ?]

2 Partie centrale

1. Définissez la partie centrale d'un compilateur.

   Correction: *Middle end* : tout ce qui ne dépend ni du langage source, ni du langage cible. En pratique, il est possible qu'il reste des "call-back" ou d'autres formes de paramétrisation en fonction du langage source et/ou cible (par exemple `$v0` dans `tc`), mais le code est essentiellement générique.

   Best-of:
   - La partie centrale d'un compilateur est celle qui se trouve entre la partie primale et la partie terminale.
   - C'est la partie qui va analyser chaque token un à un à l'aide des différents types de parsing que l'on a utilisé dans le cadre de la mise en œuvre de ce compilateur.
   - La partie centrale d'un compilateur correspond à la table de nœuds de l'AST.

2. Ci-après figure la représentation intermédiaire haut-niveau (HIR), en langage Tree, d'un programme Tiger [2].

```
move
 temp rv
 eseq
  move temp t2 const 1
  eseq
  seq
   label l2
   cjump gt temp t1 const 0 name l3 name l1
   label l3
   seq
    move
     temp t2
     binop mul temp t2 temp t0
    move
     temp t1
     binop sub temp t1 const 1
   seq end
   jump name l2
   label l1
  seq end
  temp t2
jump ra
```

   « Décompilez-le » en un programme Tiger ou C équivalent.

2. Les plus attentifs remarqueront que ce code
   - ne comporte que le corps d'une routine (par de prologue ni d'épilogue) ;
   - n'est pas encapsulé dans une `seq` ;
   - comporte une instruction finale `jump ra` (normalement absente d'une sortie HIR de `tc`).

Il s'agit de modifications visant à simplifier l'exercice.

Correction:

```
function pow (x : int,
              n : int) : int =
 let  var res := 1 in
  while n > 0 do
   (
    res := res * x;
    n := n - 1
   );
  res
 end
```

```
int pow (int x, int n)
{
 int res = 1;
 while (n > 0)
  {
   res *= x;
   --n;
  }
 return res;
}
```

3. Que fait ce code ?

Correction: Il s'agit d'un calcul de puissance : $\texttt{t0}^{\texttt{t1}}$ (ou, avec les notations du code Tiger ou C de la réponse précédente : $\texttt{x}^{\texttt{n}}$).

Best-of:
- Fibbonachie (un truc du genre).
- Calcul de factoriel.
- Ce code calcule un PGCD.
- Ce programme donne le $n^{\text{ème}}$ nombre premier.
- Ce code effectue des calculs (multiplication et soustraction) sur des variables.
- Ce code multiplie `t0`, `t1` fois. (*C'est pas faux !*)
- Cette fonction renvoie la $n^{\text{ième}}$ multiplication d'un nombre donné par lui-même. (*C'est pas faux non plus !*)

4. Nous allons *canoniser* ce programme TREE pour le transformer en représentation intermédiaire bas-niveau (LIR). La première étape consiste en la *linéarisation* du programme. Pour chacune des règles ci-dessous, écrivez la partie droite de la règle de réécriture (sur votre copie).

   (a) **move** (**temp** t, **eseq** (s, e)) =>

   Correction:

   move (**temp** t, **eseq** (s, e)) => **seq** (s, **move** (**temp** t, e))

   (b) **move** (**mem** (**eseq** (s, e1)), e2) =>

   Correction:

   move (**mem** (**eseq** (s, e1)), e2)
   => **seq** (s, **move** (**mem** (e1), e2))

   (c) **move** (**mem** (e1), **eseq** (s, e2)) =>

   Correction: Si `e1` et `s` commutent :

   move (**mem** (e1), **eseq** (s, e2))
   => **seq** (s, **move** (**mem** (e1), e2))

   Si `e1` et `s` ne commutent pas :

   move (**mem** (e1), **eseq** (s, e2))
   => **seq** (**move** (**temp** t, e1), s, **move** (**mem** (**temp** t), e2))

   où `t` est une temporaire fraîche.

5. Aidez-vous des règles que vous avez écrites à la question précédente ainsi que de celles de l'annexe A pour réécrire le programme en une version linéarisée (sans `eseq` ni `seq`).

Correction:

```
move temp t2 const 1
label l2
cjump gt temp t1 const 0 name l3 name l1
label l3
move
 temp t2
 binop mul temp t2 temp t0
move
 temp t1
 binop sub temp t1 const 1
jump name l2
label l1
move temp rv temp t2
jump ra
```

6. Découpez le code de la question précédente en blocs basiques (commençant par une étiquette et terminant par un saut conditionnel ou inconditionnel).

Correction:

```
label l4
move temp t2 const 1
jump name l2
```

```
label l2
cjump gt temp t1 const 0 name l3 name l1
```

```
label l3
move
 temp t2
 binop mul temp t2 temp t0
move
 temp t1
 binop sub temp t1 const 1
jump name l2
```

```
label l1
move temp rv temp t2
jump ra
```

Notez qu'il a fallu ajouter une étiquette `label l4` au début du premier bloc pour que celui-ci obéisse à la définition de bloc basique.

Best-of: J'ai pas de ciseaux...

7. Réassemblez les blocs basiques de la question précédente en une trace valide (de tel sorte que les `cjumps` soient normalisés en saut à une branche).

Correction: La couture des blocs précédents en une trace est immédiate : le premier bloc se termine avec un `jump` vers l'étiquette du second bloc, on peut donc les coudre ensemble et supprimer l'instruction `jump name l2` inutile. Le deuxième bloc se termine avec un `cjump` dont la branche « faux » est connectée à l'étiquette `l1`, qui commence le quatrième bloc. Comme ce bloc est libre, il suffit de le placer juste après le deuxième. Enfin, on peut poser le troisième bloc à la fin, puisque le bloc précédemment posé se termine par un `jump` (ce qui n'impose aucune contrainte sur le bloc qui le suit).

```
label l4
move temp t2 const 1
label l2
cjump gt temp t1 const 0 name l3 name l1
label l1
move temp rv temp t2
jump ra
label l3
move
  temp t2
  binop mul temp t2 temp t0
move
  temp t1
  binop sub temp t1 const 1
jump name l2
```

Best-of: . . .et encore moins de colle.

3 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`, `r3`. Le premier est sous la responsabilité de l'appelant, et les deux autres sous celle de l'appelé.

1. Définissez la partie terminale d'un compilateur.

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

   Best-of: La partie terminale d'un compilateur se trouve après la partie centrale.

2. Écrivez un programme Tiger ou C correspondant au programme suivant, avant allocation des registres.

```
enter:  u <− r3
        t <− r2
        n <− r1
        a <− 0
        b <− 1
loop:   if n <= 0 jump exit
        c <− a + b
        a <− b
        b <− c
        n <− n − 1
        jump loop
exit:   r1 <− b
        r2 <− t
        r3 <− u
        return
```

   Correction:

```
function fib (n : int) : int =
 let var a := 0
     var b := 1
 in
  while n > 0 do
   let var c := a + b
   in
     a := b;
     b := c;
     n := n − 1
   end;
  b
 end
```

```
int fib (int n)
{
 int a = 0;
 int b = 1;
 while (n > 0)
  {
   int c = a + b;
   a = b;
   b = c;
   −−n;
  }
 return b;
}
```

   Attention, beaucoup se sont trompés sur le sens de l'inégalité dans le test de la boucle `while` ; il fallait bien faire attention aux étiquettes correspondant aux branches « vrai » et « faux ».

3. Que fait ce programme ?

   Correction: Il calcule le $n^{\text{ième}}$ terme de la suite de Fibonacci.

Best-of:
- Ce programme fait une boucle.
- Il modifie certains registres.
- Ce programme génère les n premiers termes de la suite de Bernouilli.
- Ce programme swap les valeurs des registres.
- Ce programme calcule 2^n.
- Ce programme retourne $(n-1)$.
- Ce programme renvoit $n \times b$.
- Ce programme retourne la valeur passée après des calculs et allocations inutiles.
- [Place] $\sum_{i=1}^{n} i$ dans `c` et retourne du `void`.
- Si $n = 0, 1$, alors retourne 1 ; sinon, retourne $n + 2^{(n-2)}$.
- Il fait un truc bizarre, mais après une série [de] calculs retourne `b`.
- Ce programme calcule la suite de fibonacci du nombre `r1`, très utile pour modéliser la reproduction des lapins en milieu naturel.

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

   Correction: Il est sûr que `r1` sert pour le premier argument, et aussi pour retourner une valeur. Puisque `r2` et `r3` sont sous la responsabilité de l'appelé, il ne peuvent clairement pas servir ni aux arguments, ni aux retours.

   Best-of:
   - Les registres `r2` et `r3` sont internes à la fonction et contiennent des informations qui la concerne comme son nom par exemple.

5. Étant données les conventions d'appel, le sens du programme, expliquer quelles sont les variables vives lors du `return`.

   Correction: Puisqu'elles sont sous la responsabilité de l'appelé, `r2` et `r3` sont vives. Puisqu'elle sert au retour de valeur, `r1` l'est aussi.

6. Expliquez pourquoi les lignes

```
u <- r3
t <- r2
```

   et

```
r2 <- t
r3 <- u
```

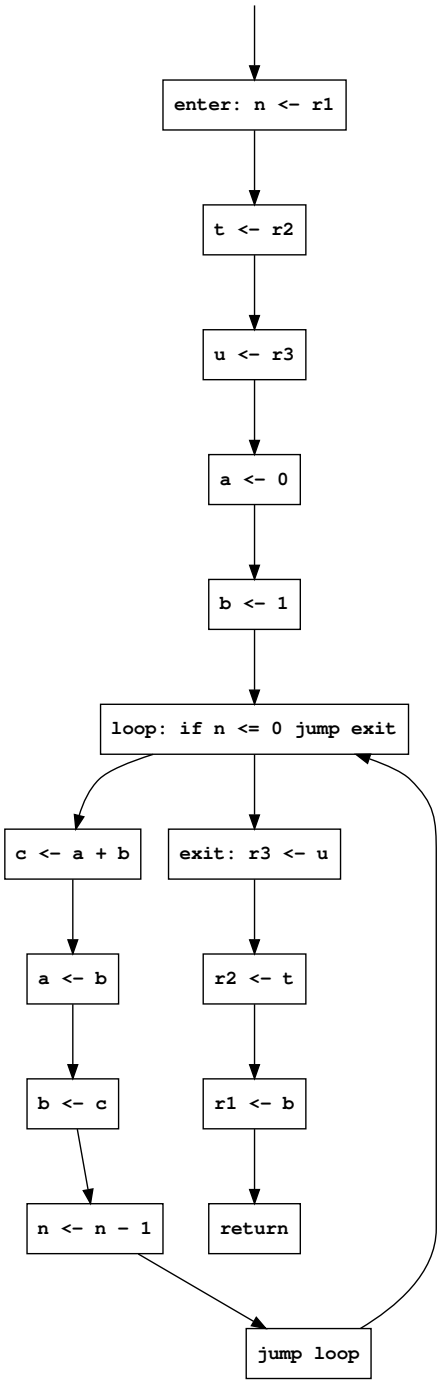
   ont été générées.

   Correction: Il faut préserver `r2` et `r3`. Sans ces lignes, cela signifierait simplement que `r2` `r3`, étant vives sur tous les arcs, seraient en conflit avec toutes les temporaires, donc inutilisables par l'allocation des registres. En clair, on compilerait avec un registre (`r1`) au lieu de trois. Ici, on se donne la possible de se servir de `r2` et `r3` quitte à devoir sauver leurs valeur initiales sur la pile, puis la restaurer à la fin : `r2` et `r3` restent disponible pour le corps de la boucle.

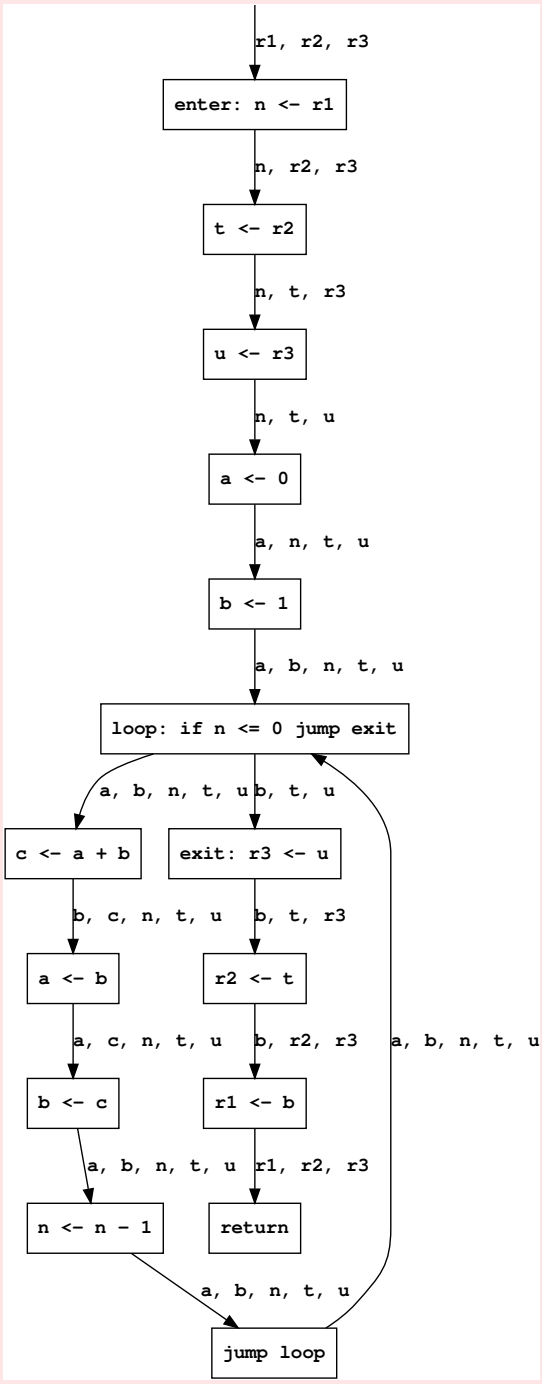
7. Étiquetez les ~~arêtes~~ arcs du graphe de contrôle de flux de ce programme ci-après avec les temporaires vives (**répondez directement sur le sujet**).

Correction: Il y avait deux erreurs dans le sujet, qui faisaient que le graphe de contrôle de flux ne correspondait pas exactement au code de la question 2 : en entrée, les instructions `n <- r1` et `u <- r3` étaient permutées ; de même, les instructions `r3 <- u` et `r1 <- b` étaient échangée en sortie.
Las suite de ce corrigé considère que le graphe de contrôle de flux est la bonne version, car c'est la décision qui a le moins d'impact sur la correction [a].
J'en bien entendu tenu compte de cette erreur lors de la correction des copies et adapté ma notation en conséquence.

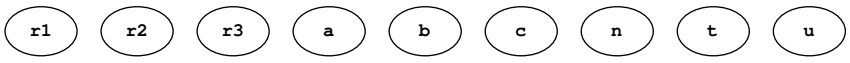
a. Même si c'est vraiment sur le code de la question 2 que je voulais vous faire travailler, car il était plus simple.


enter: n <- r1
t <- r2
u <- r3
a <- 0
b <- 1
loop: if n <= 0 jump exit
c <- a + b
exit: r3 <- u
a <- b
r2 <- t
b <- c
r1 <- b
n <- n - 1
return
jump loop

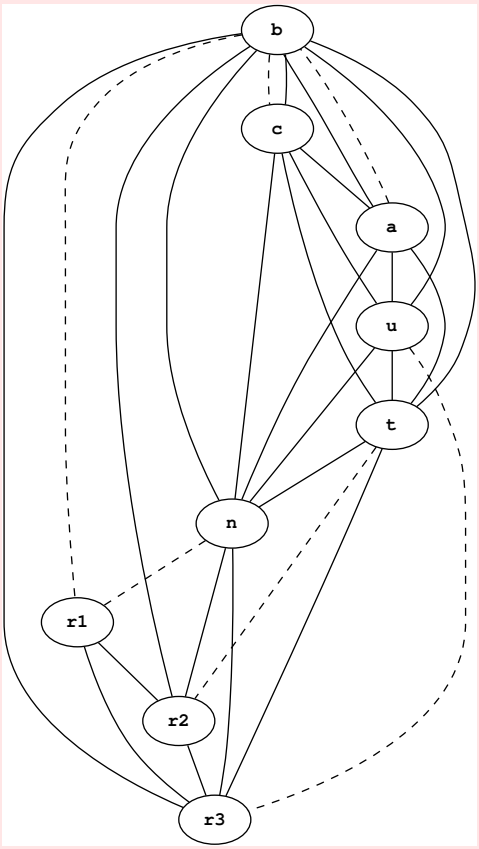
Correction:



8. Complétez le graphe d'interférence en en traçant les arêtes, y compris les fusions (*coalesce*) possibles avec des arêtes en pointillés (**répondez directement sur le sujet**).



Correction:



9. Expliquez pourquoi il est impossible de colorier ce graphe tel quel.

Correction: Les temporaires (a, b, c, n, t, u) sont toutes de degré significatif (>3), donc on ne peut colorier ce graphe par simplification.
Certains avancent le fait que le graphe soit biparti (ce qui n'est pas le cas, d'ailleurs) pour justifier le fait qu'on ne peut le colorier. Or il s'agit d'un argument en faveur de l'assertion inverse : un graphe biparti peut être colorié avec seulement deux couleurs !

Best-of:
– Car un registre va être relié à lui-même.
– Car je n'ai rien d'autre qu'un stylo bleu !
– Il est impossible de colorier ce graphe tel quel car ce graphe n'est pas connexe.
– Mmh, j'arrive à colorier le graphe tel quel, bizarre.
– J'ai oublié mes crayons de couleurs, impossible de colorier.

10. Nous allons donc devoir utiliser la mémoire pour diminuer la pression sur les registres. Combien de temporaires va-t-il au moins falloir verser (*spill*) sur la pile pour que le problème de l'allocation soit soluble ? Justifiez votre réponse.

Correction: Le sous-graphe du graphe d'interférence restreint aux temporaires (a, b, c, n, t, u) constitue un *graphe complet* (on dit aussi que {a, b, c, n, t, u} est une 6-clique). Or nous ne disposons que de trois registres. Il va donc falloir verser trois temporaires sur la pile.

Best-of:
– 42
– Il va falloir faire deux temporisation.

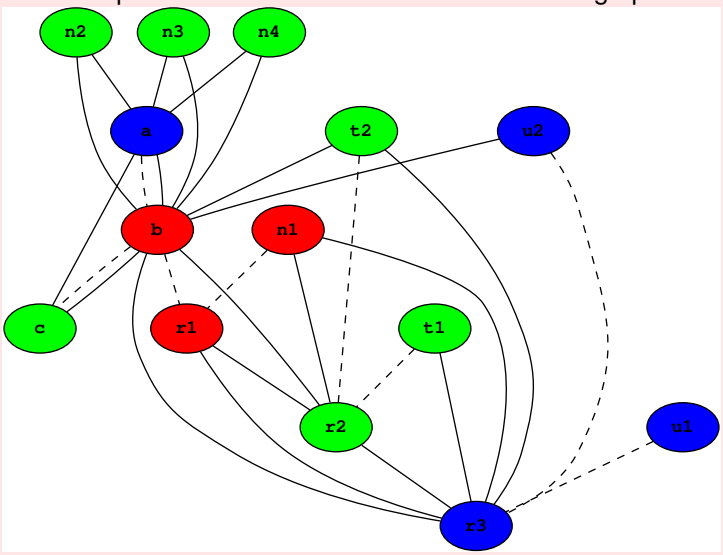
11. Réécrivez le programme en versant une ou plusieurs variables sur la pile de sorte à ce que le nouveau graphe associé puisse être colorié.

Correction: Pour commencer `t` et `u` sont de bons candidats, qui n'apparaissent pas dans la boucle. On remplace donc ces temporaires `t` et `u` par des accès mémoire `M[mt]` et `M[mu]`[a]). Mais ce n'est pas suffisant : il faut verser une troisième variable sur la pile. Cela devient un peu plus critique, puisque `a`, `b`, `c`, et `n` sont toutes les quatre utilisées dans la boucle. Choisissons `n`, qui interfère avec presque toutes les variables, et remplaçons-la par `M[mn]`. Pour finir, réécrivons les lectures/écritures sur `t`, `u` et `n` pour charger/décharger depuis la pile. Par exemple :

```
enter:  n1 <- r1
        M[mn] <- n1
        t1 <- r2
        M[mt] <- t1
        u1 <- r3
        M[mu] <- u1
        a <- 0
        b <- 1
        n2 <- M[mn]
loop:   if n2 <= 0
          jump exit
        c <- a + b
        a <- b
        b <- c
        n3 <- M[mn]
        n4 <- n3 - 1
        M[mn] <- n4
        jump loop
exit:   u2 <- M[mu]
        r3 <- u2
        t2 <- M[mt]
        r2 <- t2
        r1 <- b
        return
```

a. En toute exactitude, bien entendu `M[`m`]` serait plutôt quelque chose comme `[fp + 4]`, 4 étant pris au titre d'exemple : en fait n'importe quel endroit de la pile. Mais dans le cas présent, pour simplifier, cachons l'existence de ce registre supplémentaire.

Correction: Nous pouvons maintenant colorier le nouveau graphe :



Le programme réécrit (fig. 1(a)) peut donc passer l'allocation de registres (fig. 1(b)). Enfin, les instructions `move` inutiles peuvent êtres supprimées (fig. 1(c)).

figs/
```
enter: n1 <- r1
       M[mn] <- n1
       t1 <- r2
       M[mt] <- t1
       u1 <- r3
       M[mu] <- u1
       a <- 0
       b <- 1
       n2 <- M[mn]
loop:  if n2 <= 0
         jump exit
       c <- a + b
       a <- b
       b <- c
       n3 <- M[mn]
       n4 <- n3 - 1
       M[mn] <- n4
       jump loop
exit:  u2 <- M[mu]
       r3 <- u2
       t2 <- M[mt]
       r2 <- t2
       r1 <- b
       return
```
(a) Réécriture

figs/
```
enter: r1 <- r1
       M[mn] <- r1
       r2 <- r2
       M[mt] <- r2
       r3 <- r3
       M[mu] <- r3
       r3 <- 0
       r1 <- 1
       r2 <- M[mn]
loop:  if r2 <= 0
         jump exit
       r2 <- r3 + r1
       r3 <- r1
       r1 <- r2
       r2 <- M[mn]
       r2 <- r2 - 1
       M[mn] <- r2
       jump loop
exit:  r3 <- M[mu]
       r3 <- r3
       r2 <- M[mt]
       r2 <- r2
       r1 <- r1
       return
```
(b) Allocation

figs/
```
enter:
       M[mn] <- r1

       M[mt] <- r2

       M[mu] <- r3
       r3 <- 0
       r1 <- 1
       r2 <- M[mn]
loop:  if r2 <= 0
         jump exit
       r2 <- r3 + r1
       r3 <- r1
       r1 <- r2
       r2 <- M[mn]
       r2 <- r2 - 1
       M[mn] <- r2
       jump loop
exit:  r3 <- M[mu]

       r2 <- M[mt]



       return
```
(c) Retrait des `move`s inutiles.

FIGURE 1 – Programme réécrit et alloué après *spilling*

4 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 qui vous est remise. N'y passez pas plus de dix minutes.

Le cours

1. Quelle a été votre implication dans les cours CMP1, CMP2 et TYLA ?
   A Rien.
   B Bachotage récent.
   C Relu les notes entre chaque cours.
   D Fait les annales.
   E Lu d'autres sources.
2. 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.
3. 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.
4. 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 par semaine).
   C Supportable (environ une heure de travail par semaine).
   D Légère (quelques minutes par semaine).

Les formateurs

5. 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.
6. 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

7. 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.
8. 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 de travail par semaine).
   D Légère (une ou deux heures par semaine).
   E J'ai été dispensé du projet.
9. 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 10-16 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

10. Formation au C++.
11. Formation à la modélisation orientée objet et aux *design patterns*.
12. Formation à l'anglais technique.
13. Formation à la compréhension du fonctionnement des ordinateurs.
14. Formation à la compréhension du fonctionnement des langages de programmation.
15. Formation au travail collaboratif.
16. Formation aux outils de développement (contrôle de version, systèmes de construction, débogueurs, générateurs de code, etc.)

Questions 17-29 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.

17. Rush `.tig` : mini-projet en Tiger (Bistromatig).
18. `TC-0`, Scanner & Parser.
19. `TC-1`, Scanner & Parser, Tâches, Autotools.
20. `TC-2`, Construction de l'AST et pretty-printer.
21. `TC-3`, Liaison des noms et renommage.
22. `TC-E`, Calcul des échappements.
23. `TC-4`, Typage.
24. Désucrage des constructions objets (transformation Tiger → Panther).
25. `TC-5`, Traduction vers représentation intermédiaire.
26. Option `TC-A`, Surcharge des fonctions.
27. Option `TC-D`, Suppression du sucre syntaxique (boucles `for`, comparaisons de chaînes de caractères).
28. Option `TC-B`, Vérification dynamique des bornes de tableaux.
29. Option `TC-I`, Mise en ligne du corps des fonctions.

A Quelques règles de réécriture pour la canonisation

```
eseq (s1, eseq (s2, e))      =>  eseq (seq (s1, s2), e)

binop (op, eseq (s, e1), e2)  =>  eseq (s, binop (op, e1, e2))

jump (eseq (s, e1))          =>  seq (s, jump (e1))

cjump (op, eseq (s, e1), e2, l1, l2)
 =>  seq (s, cjump (op, e1, e2, l1, l2))

seq (ss1, seq (ss2), ss3)    => seq (ss1, ss2, ss3)
```

Si `e1` et `s` commutent :

```
binop (op, e1, eseq (s, e2))
 =>  eseq (s, binop (op, e1, e2))

cjump (op, e1, eseq (s, e2), l1, l2)
 =>  seq (s, cjump (op, e1, e2, l1, l2))
```

Si `e1` et `s` ne commutent pas :

```
binop (op, e1, eseq (s, e2))
 =>  eseq (seq (move (temp t, e1), s),
        binop (op, temp t, e2))

cjump (op, e1, eseq (s, e2), l1, l2)
 =>  seq (seq (move (temp t, e1), s),
       cjump (op, temp t, e2, l1, l2))
```

(où `t` est une temporaire fraîche).