# Compilation

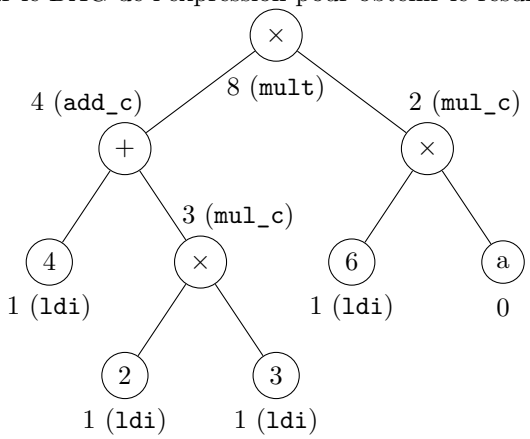
## TD 7 : Génération de code - Correction

G. Iooss & C. Alias

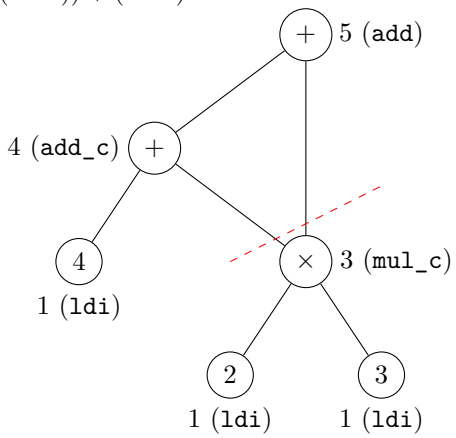
# Partie I - Sélection d'instruction

## Exercice 1. Sélection d'instruction

- On applique algorithme sur le DAG de l'expression pour obtenir le résultat suivant :

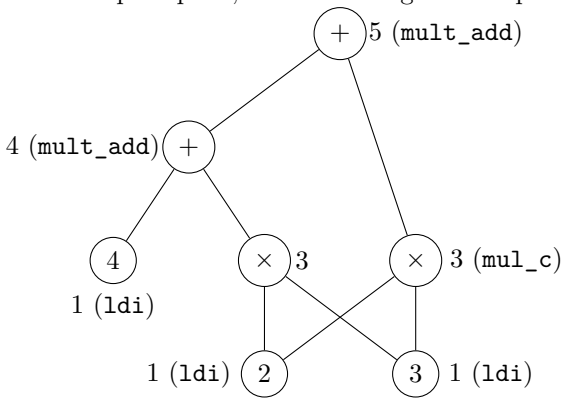



- Le DAG de l'expression $(4 + (2 * 3)) + (2 * 3)$ est :




  En comparaison avec le coût du calcul aprés duplication du nœud "$(2 * 3)$", cette solution est plus économique.

- Maintenant que `mult_add` ne coûte plus que **1**, la donne change et la duplication est plus économique :




  **Remarque :** Dans le calcul des coûts, attention à ne pas compter deux fois le même nœud dans le cas d'un DAG.

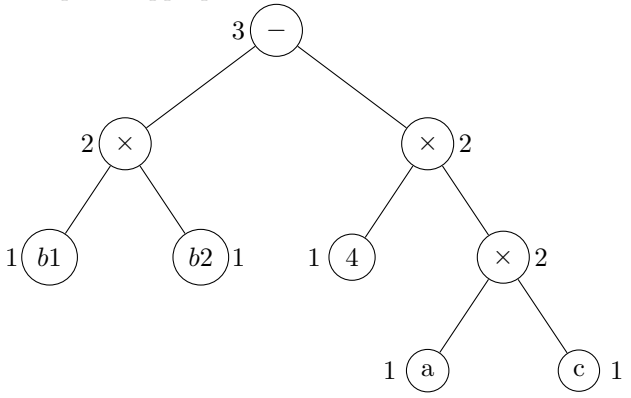
# Partie II - Ordonnancement

## Exercice 2. Algorithme de Sethi-Ulmann et ordonnancement

- Un code assembleur calculant `delta = b1 * b2 - 4*a*c` est :

```
ld r0, [ARP-0]   // b1
ld r1, [ARP-1]   // b2
mul r0, r0, r1
ld r1, [ARP-2]   // a
ld r2, [ARP-3]   // c
mul r1, r1, r2
ldi r2, 4
mul r1, r1, r2
sub r0, r0, r1
st r0, [ARP-4]   // delta
```

- [Sethi-Ulmann] Le DAG a qui on applique Sethi-Ulmann est le suivant :




  L'une des possibilité de code assembleur correspondant a été donné en réponse à la première question

- [Prefetching] Le nouveau code assembleur (sans réutilisation de registre : nécessaire pour faire le calcul des durées de vie aprés) est :

```
ld r0, [ARP-0]   // b1
ld r1, [ARP-1]   // b2
ld r2, [ARP-2]   // a
ld r3, [ARP-3]   // c
mul r4, r0, r1
mul r5, r2, r3
ldi r6, 4
mul r7, r5, r6
sub r8, r4, r7
st r8, [ARP-4]   // delta
```

– Pour le calcul des durées de vie, on rappelle qu'une variable commence à vivre après l'instruction la définissant et termine sa vie à la fin de la dernière instruction l'utilisant. Du coup, on trouve qu'il y a 4 registres devant être présent simultanément après le quatrième load. Si on a que 3 registres, il nous faudrait donc spiller l'une des variables que l'on vient de récupérer, ce qui est bien entendu stupide... :)

- En pré-fetchant tant bien que mal sans augmenter le MAX_LIVE, un code possible est :

```
ld r0, [ARP-0]   // b1
ld r1, [ARP-1]   // b2
ld r2, [ARP-2]   // a
mul r0, r0, r1
ld r1, [ARP-3]   // c
mul r1, r2, r1
ldi r2, 4
mul r1, r1, r2
sub r0, r0, r1
st r0, [ARP-4]   // delta
```

### Exercice 3. Optimisation globale

- Dû à la longueur du code assembleur, je skippe les questions d'écriture de ces derniers et de comptage de cycles d'exécution. Si vous voulez vraiment connaître la réponse à cette partie, envoyez-moi un email.
- En terme de cycle, le corps de la boucle va donner cela :

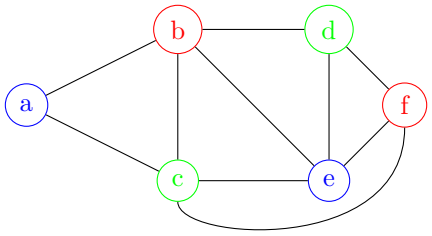
```
ld r0, [...]   // load A[i] (que l'on suppose non bloquant)
nop            // Foutu temps de chargement...
nop
ldi r1, 2      // Juste avant que le load termine, récupérer le 2
mul r0, r1, r0 // On a enfin A[i], donc on peut faire la multiplication
st r0, [...]   // store V[i]
ldi r4, 1      // Pour incrémenter $i$
add r3, r3, r4 // i++
jump for_loop
```

- Comme on a vu dans la question précédente, il y a des cycles non-utilisés du fait de l'attente de la fin du `load`. Du coup, on peut en profiter pour envoyer d'autres `load` en attendant que le premier `load` termine. Vu que l'on a 2 trous, on peut envoyer au total 3 `load`s avant de charger la constante 2, puis de faire la multiplication du premier `load`.
  En fait, cela revient à dérouler la boucle `for` en interne et avoir 3 itérations faites à chaque fois (d'où le 6=2*3 qui n'était pas du tout innocent :) ). Du coup, on peut paralléliser les accès mémoire pour optimiser le programme encore plus.
  Une autre solution fréquemment proposée est de dérouler complètement la boucle `for` (ce qui évite en effet les tests et produit le code le plus optimal possible), cependant cette solution ne tient pas si la borne supérieure de la boucle `for` est à $3n$ au lieu de 6.

## Partie III - Allocation de registres

### Exercice 4. Échauffement

**Algo de Chaitin :** On peut éliminer directement $a$ du graphe (degré <3). Puis, vu que plus aucun nœud est trivialement coloriable, on retire temporairement $e$. Le graphe devient trivialement coloriable et on arrive même à colorier $e$ après-coup. Donc, le graphe est 3-coloriable et une possibilité de coloration est la suivante :

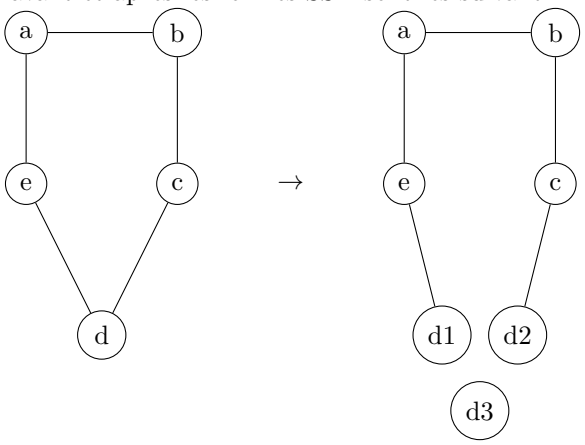



**Critère de Georges :** Après suppression du nœud $a$, les nœuds $c$ et $d$ ont le même ensemble de voisins non-trivialement coloriable. D'après le critère de Georges, on peut forcer à ce qu'ils aient la même couleur et donc les contracter en un unique nœud.

## Exercice 5. Optimalité du coloriage par algorithme glouton pour la forme SSA

**Préliminaire : Impact de la forme SSA sur le coloriage**

- Le graphe d'interférence avant et aprés les formes SSA sont les suivant :




Remarque : En pratique, $d1$ et $d2$ interfèrent. Cependant, comme elles ne peuvent pas être définies simultanément, on peut retirer le lien dans le graphe d'interférence (vu que ces deux variables n'iront jamais s'intersecter).
Dans tous les cas, le graphe devient 2-coloriable après passage sous forme SSA.

**Partie 1 - Graphe d'interférence sous forme SSA** .

**1) MAX_CLIQUE = COLOR :**

- (Théorème de Gavril) Considérons $G$ graphe d'intersection d'une famille de sous-arbre d'un arbre $\mathcal{A}$. Supposons (par l'absurde) que l'on a dans $G$ un cycle $(a_1, \ldots, a_k)$ de taille $k \geq 4$ ayant aucune corde. Ainsi, deux sommets consécutifs ne sont pas connectés.
  Vu que l'on a un arc de $a_1$ vers $a_2$, il existe un point de l'arbre $n_1 \in a_1 \cap a_2$. De même, on arrive à construire les points de l'arbre $n_i \in a_i \cap a_{i+1}$ jusqu'à $n_k \in a_k \cap a_1$.
  Plaçons nous maintenant dans $\mathcal{A}$. Vu que $n_1$ et $n_2$ sont tous les deux dans $a_2$, il existe un chemin $p_1$ dans $\mathcal{A}$ entre ces deux points. De même, on construit $p_2, \ldots, p_k$.
  Considérons maintenant deux sous-arbres $a_i$ et $a_j$ qui ne sont pas consécutifs dans le cycle considéré. Du coup, $a_i \cap a_j = \emptyset$ et, par inclusion des $p_k$ respectifs, on a $p_i \cap p_j = \emptyset$. Du coup, en mettant les $p_i$ bouts à bouts, on a un cycle dans $\mathcal{A}$, ce qui est impossible vu qu'il s'agit d'un arbre!!!
- Sous SSA, le domaine de vivacité d'une variable correspond à un sous-arbre de l'arbre de dominance. Du coup, le graphe d'interférence est bien un graphe d'intersection d'une famille de sous-arbre (les domaines de vivacité des variables du programme) d'un sous-arbre (l'arbre de dominance).

**2) MAX_LIVE = MAX_CLIQUE :**

- Plaçons-nous sous forme SSA. Soient $u_1, \ldots, u_n$ formant une clique dans le graphe d'interférence. Comme on est en forme SSA, tous les $u_i$ n'ont été définis qu'une seule fois (à un point nommé $def(u_i)$). Du coup, vu que l'on a une clique, pour tout $i$ et $j$ $def(u_i)$ est comparable à $def(u_j)$. D'où, tous les $u_i$ sont comparables entre eux.
  Prenons $def(u_k)$ le point de définition où est définie la dernière variable. Vu que $u_i$ interfère avec $u_k$, alors $u_i$ est vivant en $def(u_k)$. D'où, tous les $u_i$ sont vivants simultanément en $def(u_k)$.
- De plus, par définition du graphe d'interférence, si $u_1, \ldots, u_n$ sont vivantes en même temps, elle sont toutes connectées et forment bien une clique. D'où, on a MAX_CLIQUE = MAX_LIVE. On finit par en conclure que MAX_LIVE = COLOR.

**Partie 2 - Optimalité de la méthode gloutone** .

- Soit $G$ le graphe d'interférence qu'on essaye de k-colorier et $s \in G$ un nœud simplicial dans G. Les voisins de $s$ forment donc une clique de G. Du coup, $\{s\} \cup Voisin(s)$ forme également une clique de $G$.
  Or, $|s \cup Voisin(s)| \leq MAX_CLIQUE = MAX_LIVE = k$. Du coup, $|Voisin(s)| < k$ et $s$ est trivialement coloriable. Il ne reste plus qu'à itérer ce processus sur $G - \{s\}$ qui est aussi cordal.

**Exo bonus - Les graphes cordaux (triangulé) sont parfait** .

Un *séparateur* est un ensemble de sommets qui coupe un graphe en plusieurs parties connexes. Du coup, un *séparateur minimal* est un séparateur avec un nombre minimal de sommets.

La preuve (plus compliquée que prévu initialement) fonctionne de la manière suivante :

- Prouver que G triangulé => tout séparateur minimal de G est une clique.
- Prouver que G triangulé => G est une clique ou contient 2 sommets simpliciaux non adjacents.
- En enlevant les simpliciaux un par un, vu qu'ils sont trivialement colorable (cf question de la partie 2), on peut les enlever un par un et colorer tout G.

Le premier point se prouve par l'absurde : si on a $u_1$ et $u_2$ dans le séparateur qui n'est pas connecté, on peut construire un cycle sans corde de taille $\geq 4$ passant par $u_1$, une première composante connexe, $u_2$, une seconde composante connexe, avant de revenir à $u_1$. Contradiction avec la triangularité du graphe !

Le second point est trivial si G est de taille 2 ou est une clique. Du coup, par récurrence sur n, si G est pas une clique et est de taille n, on prend un séparateur minimal K (qui est donc une clique) et on considère deux composantes connexes A et B dans $G - K$. On applique les hypothèses de récurrence sur $A \cup K$ et $B \cup K$. On trouve donc 2 sommets simpliciaux non connectés dans $A \cup K$ et 2 autres sommets simpliciaux dans $B \cup K$. Parmis les deux premiers, il y en a au moins un dans $A$, et il y en a également un autre dans $B$. Du coup, ces deux sommets sont toujours simpliciaux dans le graphe $G$.