TP07 - ALLOCATION DE REGISTRES

1. Objectif

L'objectif de ce TP est :

- de construire le graphe d'interférence à partir de la solution au graphe d'analyse
- d'implementer l'algorithme de coloration de graphe vu en cours
- de colorer le graphe d'interférence
- d'utiliser le résultat de la coloration de graphe pour effectuer l'allocation de registres du pré-assmbleur et, finalement, produire l'assembleur final.

2. Graphe d'interférence

```
public class Ig {
    public Graph      graph;//le graphe d'interférence
    public FgSolution fgs;  //les ensembles in et out
    public int        R;    //le nombre de registres fictifs
    public Nasm       nasm; //le pré-nasm
    public Node       int2Node[];

    public       Ig(FgSolution fgs);
    public void  construction();
    public int[] getPrecoloredTemporaries();
    public void  affiche(String baseFileName);
    public void allocateRegisters();
    }
```

Le graphe d'interférence est représenté par un objet de la classe `Ig`. Cette dernière a pour variables d'instances

- le graphe proprement dit : `graph`,
- la solution `fgs` du graphe d'analyse, qui comporte, pour chaque sommet du graphe d'analyse, les deux ensembles `in` et `out`,
- le nombre de registres fictifs `R`, qui correspond au nombre de sommets du graphe d'interférence,
- le code pré-nasm : `nasm`,
- le tableau `int2Node`, qui permet d'accéder rapidement à un sommet du graphe d'interférence à partir de son identifiant (un entier).

Ses méthodes sont :

- `construction()`, qui crée le graphe d'interférence à partir du graphe d'analyse. C'est l'implémentation directe de l'algorithme 1, vu en cours.
- `getPrecoloredTemporaries()`, qui renvoie un tableau d'entiers `couleur` de dimension `R`. Si le registre `i` du code pré-assembleur est précoloré avec la couleur `c`, alors `couleur[i]` vaut `c`. Cette méthode parcourt les instruction du code pré-assembleur et, pour chaque opérande de chaque instruction, si cette opérande est un registre pré-coloré, l'entrée qui lui correspond dans le tableau `couleur` est mise à jour.

Algorithm 1 Construction du graphe d'interférence $G = (S, A)$ à partir du graphe d'analyse $G_A = (S_A, A_A)$

$A \leftarrow \emptyset$
$S \leftarrow \{1, \dots R\}$
for all $i \in S_A$ **do**
  for all $(r, r') \in in(s) \times in(s),\ r \neq r'$ **do**
    $A \leftarrow A \cup (r, r')$
  end for
  for all $(r, r') \in out(s) \times out(s),\ r \neq r'$ **do**
    $A \leftarrow A \cup (r, r')$
  end for
end for

3. Coloration de graphe

```
public class ColorGraph {
    public  Graph          G;       // le graphe a colorer
    public  int            R;       // nombre de sommets
    public  int            K;       // nombre de couleurs
    private Stack<Integer> pile;
    public  IntSet         enleves; // sommets enlevés
    public  IntSet         deborde; // sommets qui débordent
    public  int[]          couleur; // tableau des couleurs
    public  Node[]         int2Node;
    static  int            NOCOLOR = -1;

    public        ColorGraph(Graph G, int K, int[] phi);
    public IntSet couleursVoisins(int s);
    public int    choisisCouleur(IntSet colorSet);
    public int    nbVoisins(int s);
    public int    simplification();
    public void   selection();
    public void   debordement();
    }
```

La coloration de graphe est matérialisée par la classe `ColorGraph` du package `graph`. Les variables d'instance de la classe `ColorGraph` correspondent aux notations des transparents du cours : `G`, `R`, `K` correspondent successivement au graphe d'interférence, au nombre de sommets de ce dernier et aux nombre de couleurs.
L'ensemble `enleves` permet de représenter les sommets qui sont enlevés du graphe, lors de l'étape de simplification. Le tableau `couleur` contiendra à la fin du traitement les couleurs des différents sommets (ou registres fictifs du pré-assembleur). Enfin, le tableau `int2Node` permet d'accéder de manière efficace à un sommet du graphe à partir de son identifiant (un entier).
Les trois méthodes `simplification`, `debordement` et `selection` correspondent aux trois algorithmes 2,3 et 4, qui constituent les trois étapes de l'algorithme vu en cours.
On définit de plus les trois méthodes auxiliaires suivantes ;

- `nbVoisins(int s)`, qui renvoie le nombre de voisins du sommet d'indice `s`.
- `couleursVoisins(int s)`, qui renvoie un ensemble contenant les couleurs des voisins du sommet d'indice `s`.
- `choisisCouleur(IntSet ColorSet)`, qui choisit une couleur dans l'ensemble de couleurs `ColorSet`.

Algorithm 2 Simplification du graphe $G = (S, A)$, ϕ est la fonction de pré-coloration

```
1: pile ← ∅
2: N ← R− nombre de sommets pré-colorés
3: modif ← vrai
4: while taille(pile) ≠ N et modif = vrai do
5:    modif ← faux
6:    for all s ∈ S do
7:       if nb_voisins(s) < K et φ(s) = 0 then
8:          empile(s)
9:          S ← S − {s}
10:         modif ← vrai
11:      end if
12:   end for
13: end while
```

Algorithm 3 Débordement

```
1: deborde ← ∅
2: while taille(pile) ≠ R do
3:    s ← choisis_sommet
4:    empile(s)
5:    S ← S − {s}
6:    deborde = deborde ∪ {s}
7:    Simplifie
8: end while
```

Algorithm 4 Sélection d'une couleur pour les sommets du graphe $G = (S, A)$ avec la pile produite par l'algorithme de Simplification

```
1: ∀s ∈ S couleur[s] ← φ(s)
2: while taille(pile) ≠ 0 do
3:    s ← depile
4:    C ← couleurs_voisins(s)
5:    if |C| ≠ K then
6:       couleur[s] ← choisis_couleur(C − C)
7:    end if
8: end while
```

4. Allocation de registres

Une fois les sommets du graphe d'interférences colorés, il ne reste plus qu'à associer à chaque registre fictif du pré-assembleur un registre réel. Cela est réalisé par la méthode `allocateRegisters` de la classe `Ig`. L'allocation est réalisée lors d'un parcours des instructions du code pré-assembleur. Ce parcours ressemble à celui réalisé par la méthode `getPrecoloredTemporaries`. Il consiste à parcourir toutes les opérandes de toutes les instructions du code pré-assembleur. Pour toute opérande visitée, si cette dernière correspond à un registre qui ne possède pas une couleur, on récupère sa couleur dans le tableau `couleur` et on l'affecte à l'opérande par l'intermédiaire de la méthode `colorRegister(int color)` de la classe `NasmRegister` du package `nasm`.

5. Exécution du code `nasm`

Une fois le code `nasm` produit, il ne reste plus qu'à créer un exécutable. Cela se fait en deux étapes : la génération d'un fichier objet à l'aide du programme d'assemblage `nasm` et l'édition de liens à l'aide du linker `ld`, comme dans l'exemple ci-dessous :

```
java Compiler prog.l
nasm -f elf -dwarf -g prog.nasm
ld -m elf_i386 -o prog prog.o
```