TP06 - CONSTRUCTION ET RÉSOLUTION DU GRAPHE D'ANALYSE

1. Objectif

L'objectif de ce TP est double. Il s'agit d'une part de créer le graphe d'analyse correspondant à un programme en pré-assembleur et, d'autre part, de résoudre les équations induites par ce graphe pour calculer les ensembles de registres virtuels qui sont vivants sur chaque arc du graphe d'analyse.
La construction du graphe et la résolution des équations est fondé sur les trois packages `fg`, `util.graph` et `util.intset`, qui ont été ajoutés au dépot git.
Le détail de ces trois packages est décrit dans les transparents disponibles sur le site du cours.

2. Construction du graphe d'analyse

La construction du graphe d'analyse est une méthode de la classe `Fg`.

```
public class Fg implements NasmVisitor <Void> {
    public Nasm nasm;
    public Graph graph;
    Map< NasmInst, Node> inst2Node;
    Map< Node, NasmInst> node2Inst;
    Map< String, NasmInst> label2Inst;
}
```

Elle consiste à parcourir le pré-code `nasm`, qui est une instance de la classe `nasm`. C'est pendant ce parcours que sont créés les sommets et les arcs du graphe d'analyse.

2.1. **Création des sommets du graphe.** La première étape consiste à créer les sommet du graphe. Pour chaque instruction `nasm` un sommet est créé et associé à l'instruction grâce aux deux tables de hash `inst2Node` et `node2Inst`.
De plus, pour chaque instruction étiquetée, l'association entre l'étiquette et l'instruction est indiquée par une entrée dans la table de hash `label2inst`.

2.2. **Création des arcs du graphe.** La seconde étape consiste à créer les arcs du graphe. Pour cela, on parcours les instructions `nasm` et pour chaque instruction i, on établit des arcs (i, j) où j sont les sommets correspondant aux instructions qui peuvent suivre directement l'instruction i.
Illustrons cela sur le cas suivant :

```
1   mov r1, 4
2   add r1, 5
```

l'instruction correspondant à $i = 1$, est une instruction `mov`. Il s'agit d'une instruction simple, dans le sens où elle ne peut être suivie que par l'instruction suivante : l'instruction `add`. Dans ce cas, on crée l'arc $(1, 2)$.
Certaines instructions peuvent avoir plus d'un successeur, auquel cas, plus d'un arc devront être créés. C'est le cas en particulier des sauts conditionnels :

```
1         cmp r1, r2
2         je  l1
3         mov r3, 0
4         jmp l2
5   l1 : mov r3, 1
```

Ici, l'instruction 2 (`je l1`) peut être suivie par l'instruction 3 qui la suit directement ou par l'instruction 5 qui correspond à l'étiquette `l1`. Dans ce cas, deux arcs seront créés : $(2, 3)$ et $(2, 5)$.
La construction du graphe d'analyse peut être réalisée de deux manières, soit à l'aide du visiteur `NasmVisitor` qui définit une méthode `public T visit(X inst);` pour chaque type d'instruction `X`, soit sous la forme d'une séquence d'instructions `if` où la nature de l'instruction traitée est identifiée à l'aide de l'opérateur `instanceof`.

3. Résolution des équations du graphe d'analyse

La résolution des équations du graphe d'analyse permet de calculer, pour chaque instruction du pré-assembleur `nasm`, les ensembles `in` et `out`, qui indiquent les registres qui sont vivants à l'entrée de l'instruction et à sa sortie.
Elle repose sur le package `FgSolution` qui associe à chaque instruction quatre ensembles `IntSet` par l'intermédiaire de quatre tables de hash :

```
public class FgSolution{
  int iterNum = 0;
  public Nasm nasm;
  Fg fg;
  public Map< NasmInst, IntSet> use;
  public Map< NasmInst, IntSet> def;
  public Map< NasmInst, IntSet> in;
  public Map< NasmInst, IntSet> out;
}
```

La résolution des équations se décompose en deux parties : l'initialisation des ensembles `use` et `def` et le calcul des ensembles `in` et `out`.

3.1. Initialisation des ensembles `use` et `def`.

Elle consiste simplement à parcourir la séquence des instructions du pré-assembleur et à calculer, pour chacune, les ensembles `use` et `def`. Le premier indique les registres qui sont utilisés par l'instruction et le second, les registres dont la valeur est modifiée par l'instruction (on dit dans ce cas qu'elle les définit).
L'instruction `mov r1, r2`, par exemple, utilise le registre `r2` et modifie le registre `r1`.
On peut savoir si une instruction particulière utilise ou définit ses opérandes grâce aux variables booléennes `destUse`, `destDef`, `srcUse` et `srcDef` définies au niveau de la classe abstraite `NasmInst` :

```
public abstract class NasmInst{
    public NasmOperand label = null;
    public NasmOperand dest = null;
    public NasmOperand src = null;
    public NasmOperand address = null;
    public boolean destUse = false;
    public boolean destDef = false;
    public boolean srcUse = false;
    public boolean srcDef = false;
    String comment;
    }
```

Voici, par exemple, la classe `NasmMov` qui définit son opérande `dest` et utilise son opérande `src`.

```
public class NasmMov extends NasmInst {
  public NasmMov(NasmOperand label, NasmOperand dest, NasmOperand src, String comment){
      destDef = true;
      srcUse = true;
      this.label = label;
      this.dest = dest;
      this.src = src;
      this.comment = comment;
  }
}
```

Deux remarques importantes :

— On ne s'intéresse ici qu'aux registres *généraux*, c'est à dire les registres `r0`, `r1`, ... qui correspondent aux temporaires du code trois adresses. Cela exclut les registres spéciaux,

en particulier les registres `ebp` et `esp`. Pour savoir si un registre est un registre général, on peut utiliser la méthode `isGeneralRegister()`, définie dans la classe `NasmOperand`.

— Dans certains cas, des adresses `nasm` correspondent à des calculs à partir de valeurs contenues dans des registres, comme dans les exemples ci-dessous :

```
1    mov r1, [ebp + r2]
2    mov r2, [r3 + r4]
```

Dans l'instruction `1`, le registre `r1` est modifié et le registre `r2` est utilisé. Dans l'instruction `2`, le registre `r2` est modifié et les deux registres `r3` et `r4` sont utilisés.

Comme pour la création du graphe d'analyse, l'initialisation des ensembles `use` et `def` peut être réalisée à l'aide du visiteur `NasmVisitor` ou bien à l'aide de `if` imbriqués.

3.2. **Calcul des ensembles `in` et `out`.** Le cœur de la résolution des équations est l'algorithme itératif vu en cours, qui est repris ci-dessous. Cet algorithme suppose que ensembles `def` et `use` de chaque instruction ont été initialisés et calculent de manière itérative les ensembles `in` et `out` de chaque instruction, juqu'à convergence.
Les opérations ensemblistes sont définies dans la classe `IntSet`.

Algorithm 1 Calcul itératif de $in(s)$ et $out(s)$

for all s **do**
  $in(s) = \{\}$
  $out(s) = \{\}$
end for
repeat
  for all s **do**
    $in'(s) = in(s)$
    $out'(s) = out(s)$
    $in(s) = use(s) \cup (out(s) - def(s))$
    $out(n) = \cup_{s \in succ(s)} in(s)$
  end for
until $in'(s) = in(s)$ et $out'(s) = out(s)$, $\forall s$