Compilation Avancée : Travaux Pratiques 1 et 2

L'objectif de ce premier TP est en premier lieu la prise en main du code fourni que vous utiliserez pendant les TPs associés à ce cours et pour le projet à rendre.

C'est un code écrit en C++ (plutôt C orienté ++…). Vous trouverez sur internet facilement de la documentation C++, mais globalement c'est entre le C et le Java.

Ce code n'est pas forcément écrit de manière optimale toutefois il a l'avantage d'être commenté et documenté, dans le code se trouvent des directives de documentation permettant à *Doxygen* de générer automatiquement une documentation des classes et des méthodes disponibles, indiquant notamment les relations d'héritage.

Le code fourni permet de représenter et de manipuler du code assembleur MIPS. Notamment, il permet de parser un code assembleur MIPS produit par le compilateur *GNU gcc*.

Un fichier assembleur MIPS, une fois parsé, est représenté par un objet de type `Program`. Un `Program` est représenté une liste doublement chaînée de lignes (classe `Line`).

Lignes

Une ligne (classe `Line`) possède un successeur et un prédécesseur potentiellement `NULL`. Une ligne de type `Line` n'existe pas en soi, la classe `Line` est une classe abstraite (interface en java). Sont dérivées de la classe `Line`, les classes correspondant aux types de lignes que l'on trouve dans un fichier assembleur MIPS à savoir :

- La classe `Directive` qui correspond aux lignes directives (voir un exemple de code)
- La classe `Label` qui correspond aux lignes contenant une étiquette (de la forme `etiq:`)
- La classe `Instruction` qui correspond aux instructions du jeu d'instructions MIPS

Etant donné un pointeur vers une ligne `Line* l`, on peut savoir si la ligne pointée est une instruction, un label ou une directive en appelant des méthodes de la classe Line :

- `l->isInst()` rend `true` si `l` est une instruction
- `l->isLabel()` rend `true` si c'est un label, et
- `l->isDirective()` renvoie `true` si `l` est une directive.

De plus, des fonctions permettent de récupérer les objets pointés avec le bon type :

- `Instruction* getInst(Line * l)` renvoie l'instruction correspondant à `l` si `l` est une instruction, `NULL` sinon
- `Label* getLabel(Line *)` renvoie le label correspondant à `l` si `l` est un label, `NULL` sinon
- `Directive* getDirective(Line *)` renvoie la directive correspondant à `l` si `l`est une directive, `NULL` sinon

Instructions

Une instruction comporte entre autre :

- Un type (`t_Inst`)
- Un code opération (`t_Operateur`)
- Un format de codage (`t_Format`)
- Un nombre d'opérandes
- 3 opérandes (classe virtuelle `Operand` ; mais certains sont à `NULL` si l'instruction n'a pas 3 opérandes)
- …

Les types de la forme `t_X` sont souvent des types énumérés dont la définition se trouve dans le fichier `include/Enum_type.h`

Notamment

- Le format de codage des instructions est donné par le type énuméré (fichier Enum_type.h) : `enum t_Format {J, I, R, O, B};`
- Le type d'une instruction par est donné par le type : `enum t_Inst {ALU, MEM, BR, OTHER, BAD};`
- L'opération associée à une instruction est donnée par le type énuméré `enum t_Operator.`

Notez que vous pouvez afficher dans un terminal le contenu de n'importe quel objet avec la méthode `display`, implantée dans toutes les classes. Servez-vous en pour vérifier vos codes.

Exercice 1

Récupérez les suivantes
`/users/Enseignants/heydemann/CoursCA/CodeCAEtudiant.tgz`
`/users/Enseignants/heydemann/CoursCA/boost.tgz`

Décompressez là en tapant la commande suivante dans un terminal :
`tar -zxvf CodeCAEtudiant.tgz`

IMPORTANT :

Copier l'archive `boost.tgz` dans `CodeEtudiant/include/`
Décompressez l'archive `boost.tgz`

Vous voilà avec un répertoire racin nommé `CodeEtudiant` contenant plusieurs sous-répertoires :

- `include` qui contient les entêtes mais aussi dans les sous-répertoires `html` et `latex` la documentation automatiquement générée par *Doxygen*.
- `src` qui contient les répertoires :
  - `base` qui contient les fichiers .cpp correspondant aux classes mentionnées ci-dessus, notamment les fichiers dans lesquels vous aurez à coder. C'est ici que vous aurez à ajouter des fichiers plus tard.
  - `parsing` qui contient les fichiers *lex* et *yacc* utilisés pour parser le code assembleur MIPS
  - `examples` qui contient des fichiers assembleur MIPS produits par gcc, notamment ceux que l'on a utilisés en TD (`ex_asm.s` et `test_asm32.s`)
  - `mains` qui contient des codes avec des programmes principaux utilisant la bibliothèque, c'est là qu'il faudra écrire des codes pour tester votre travail.
- `bin` qui contient après compilation les exécutables
- `lib` qui contient après compilation la bibliothèque de parsing, manipulation et représentation d'un code assembleur (à ne pas toucher)

1) Ouvrez la documentation disponible dans le répertoire `include/html` en ouvrant n'importe quel fichier html dans un navigateur ou en ouvrant le fichier `refman.pdf` du répertoire `include/latex`

2) Positionnez vous dans le répertoire racine de l'archive (`CodeEtudiant`) et compilez le code avec la commande `make`.
La compilation compile en une bibliothèque les fichiers que vous trouverez dans les répertoires `src/base` et la place dans le répertoire `lib` ; la compilation produit automatiquement pour tous les codes contenus dans `src/mains` un exécutable pour chacun des fichiers et place cet exécutable dans le répertoire `bin/cpp` si le fichier source a une extension `.cpp` ou dans le répetoire `bin/c` si le fichier source a une extension `.c`

3) Ouvrez le fichier `test_karine.cpp` se trouvant dans le répertoire `src/mains`

4) Exécutez l'exécutable issu de la compilation de ce fichier en le lançant à la racine de l'archive avec la commande suivante (vous pouvez tester avec d'autres fichiers assembleur) :
`./bin/cpp/test_karine src/example/ex_asm.s`
Que fait ce programme ?
Regarder le code du programme `test_karine.cpp` ainsi que le code de la classe `Program.cpp` correspondant aux fonctions utilisées dans le programme.

Exercice 2

Vous devez dans cette question coder la méthode `comput_basic_bloc` de la classe `Function.`
Cette méthode doit délimiter les blocs de base de la fonction et construire liste des blocs de base de la fonction.
Elle crée autant de blocs de base que nécessaire et les ajoute à la liste `_myBB`. Il y a dans le fichier `Function.cpp` des indications sur comment faire. Il s'agit ici d'appliquer l'algorithme donné en cours, à savoir

déterminer les entêtes des blocs de bases et d'utiliser l'algorithme esquissé en TD.

IMPORTANT : n'oubliez pas qu'en MIPS l'instruction qui suit un branchement fait partie du même bloc de base que le branchement, c'est l'instruction du ***delayed slot*.**

Pour information, un bloc de base doit contenir :

1) un pointeur vers la première ligne du programme qui correspond à la première ligne du bloc de base : cela doit être une étiquette si une étiquette se trouve avant l'entête qui est la première instruction du bloc, une instruction si aucune étiquette ne précède l'entête.
2) un pointeur vers la dernière ligne composant le bloc de base, et
3) un pointeur vers la ligne correspondant au branchement le terminant s'il y en a un
4) un identifiant ou index du bloc : les blocs de base sont numérotés dans l'ordre du programme en commençant à 0.

Vous utiliserez la méthode `void Function::addBB(Line *, Line *, Line *, int)` qui crée et ajoute un BB à la fonction (voir la fonction directement)

Il est conseillé d'utiliser les méthodes disponibles dans la classe `Instruction` pour déterminer le type d'une instruction et notamment celle permettant de déterminer si l'instruction sur laquelle est appelée un saut.

- `bool Instruction::is_branch()` : teste si l'instruction est une opération de type BR (type associé à toutes les instructions de saut et branchement)

D'autres méthodes permettent de déterminer la nature du saut plus précisément (utile plus tard dans le TME) :

- `bool Instruction::is_call()` : teste si l'instruction est un appel de fonction
- `bool Instruction::is_cond_branch()` : teste si l'instruction est un saut conditionnel
- `bool Instruction::is_indirect_branch()` : test si l'instruction est un saut indirect (adresse du saut dans un registre)

Utilisez le programme principal `test_bb_cfg.cpp` (qui teste l'ensemble des fonctions demandées dans les questions 2 à 5) pour tester cette fonction, vous pouvez aussi l'utiliser pour modifier le programme principal `test_karine.cpp`.

Exercice 3

Dans cette question, vous devez implanter la méthode `void Function::compute_succ_pred_BB()`. Il s'agit de déterminer les blocs successeurs et prédécesseurs d'un bloc de base. Pour cela, vous aurez besoin de déterminer, étant donnée une étiquette cible d'un saut, quel bloc commence par cette étiquette dans une fonction donnée.

Les étiquettes présentes dans les instructions sont des opérandes, donc sont de types `OPLabel`. Elles désignent une ligne de type `Label`. Attention à ne pas confondre les deux types/objets bien différents !

La fonction `Basic_block *Function::find_label_BB(OPLabel* label)` rend le bloc de base de la fonction sur laquelle cette méthode est appelée qui commence par le label donné. La fonction retourne NULL si aucun bloc commençant par ce label dans la fonction n'est trouvé. Pour que la fonction puisse fonctionner il ne faut pas oublier de calculer les étiquettes : il faut au préalable appeler une fois la fonction `compute_label` de la fonction.

La méthode `Basic_Block :: set_link_succ_pred(Basic_Block *succ)` ajoute au bloc sur laquelle elle est appelée (`this`) le bloc `succ` et le bloc `this` est ajouté comme prédécesseur au bloc `succ`.

Il y a dans le fichier `Function.cpp` des indications sur comment faire juste avant la fonction dont le corps est vide et à remplir. Pensez à utiliser les méthodes de la classe `Instruction` pour déterminer si une instruction est un saut, et si une instruction est un saut conditionnel, un saut inconditionnel, un appel de fonction ou un saut indirect. Voir ci-dessus.

On supposera que les codes utilisés pour tester ne contiennent pas de saut indirect sauf pour le retour à la fonction appelante. Un bloc avec un saut indirect n'a pas de successeur.

Exercice 4

Vous devez implanter la fonction `comput_CFG` de la classe `Program`. Il s'agit ici de construire la liste des CFG présents dans un programme. Un CFG (voir la définition de la classe dans `include/Cfg.h`) contient un pointeur vers le premier bloc de base de la fonction et le nombre de blocs de base que contient la fonction. La construction d'un CFG prend en paramètre un pointeur vers le premier bloc de base du CFG et le nombre de blocs de base du CFG.
Par exemple :

```
Cfg * cfg = new Cfg(bb0, n);
```

crée un CFG dont le BB d'entrée est `bb0` et contenant `n` BBs.

Les successeurs et prédécesseurs des blocs de base permettant de se déplacer dans le CFG, il n'est pas nécessaire de coder explicitement les arêtes entre les blocs.

Il y a dans le fichier `Program.cpp` des indications.

Testez l'ensemble des fonctions écrites jusqu'à présent avec le programme principal contenu dans le fichier `test_bb_cfg.cpp`.
Pensez à regarder la tête de vos CFG en utilisant la possibilité de créer un fichier .dot ! Des exemples d'utilisation de cette production de fichiers sont donnés dans les exemples de programmes principaux.
Vous devez retrouver les CFG déterminés à la main en TD.

Exercice 5

Cet exercice a pour but d'implanter la méthode `Function::compute_dom()` de la classe `Function`. Il s'agit ici de déterminer les blocs de base dominants des les blocs d'une fonction.

Chaque bloc de base possède un tableau de NB_MAX_BB booléens et nommé `Domin`.
La ième case de ce tableau indique si le ième bloc de la fonction domine le bloc auquel il est associé. Ce tableau est accessible sans accesseurs. Lors de la création d'un bloc de base tous les éléments de ce tableau sont initialisés à vrai.
Le calcul des blocs dominant devra suivre l'algorithme suivant et utiliser une liste de blocs à traiter.

- La liste est initialisée avec les blocs sans prédécesseurs (un seul normalement). Pour ces blocs, le seul dominant est lui-même.
- Ensuite tant que la liste est non vide, on recalcule les dominants du premier bloc B de la liste (et on l'enlève). Si un changement a lieu dans un des dominants du bloc B, il faut ajouter à la liste de blocs à traiter tous les successeurs du bloc B.
-

Rappel : on peut récupérer le numéro d'un bloc avec la méthode `Basic_block::get_index()` de la classe `Basic_block`. On peut récupérer le bloc d'un numéro voulu avec `Basic_block::get_BB(int index)` de la classe `Basic_block`.

Testez l'ensemble des fonctions écrites jusqu'à présent avec le programme principal contenu dans le fichier `test_bb_cfg.cpp`.
Pensez à regarder la tête de vos CFG et vos calculs de dominant pour détecter les potentielles erreurs. Vous devriez retrouver les résultats trouvés en TD sur les CFG exemples utilisés et disponibles dans les exemples pour les tests.

Afin d'éviter que vous ne passiez pas ces étapes de programmation qui ne constitue qu'une petite partie du projet final, il vous sera demander de tester votre code en séance sous forme de démo qui constituera une étape et note intermédiaire (au plus tard lors de la séance avance les vacances de printemps).