UE Compilation Avancée
M1 STL – Master Informatique
Sorbonne université

Compilation Avancée 2018-19 : Travaux Pratiques 1

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 avec une liste doublement chaînée de lignes (classe `Line`).

Lignes

Une ligne (classe `Line`) possède un successeur et un prédécesseur potentiellement `nullptr`. 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, `nullptr` sinon
- `Label* getLabel(Line *)` renvoie le label correspondant à `l` si `l` est un label, `nullptr` sinon
- `Directive* getDirective(Line *)` renvoie la directive correspondant à `l` si `l`est une directive, `nullptr` 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 `nullptr` 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 class t_Format {J, I, R, O, B};`
- Le type d'une instruction par est donné par le type énuméré (fichier `Enum_type.h`): `enum class t_Inst {ALU, MEM, BR, OTHER, BAD};`
- L'opération associée à une instruction est donnée par le type énuméré (fichier `Enum_type.h`) : `enum`

```
class t_Operator.
```

Notez que vous pouvez afficher dans un terminal le contenu de n'importe quel objet (`Program, Function, Basic_block, Line, Instruction`) avec la méthode `display`, implantée dans toutes ces classes. Servez-vous en pour vérifier vos codes.

Exercice 1

Récupérez les suivantes
`/users/Enseignants/heydemann/CoursCA/CodeEtudiant2019.tgz`

Décompressez là en tapant la commande suivante dans un terminal :
`tar –zxvf CodeEtudiant2019.tgz`

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

- `include` qui contient les entêtes (fichier `.h`)
- `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.
  - `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 programmes principaux, il y a normalement des programmes pour tester les différentes fonctions. C'est dans ce répertoire qu'il faut écrire les codes de test. Ce répertoire contient un `README.txt` décrivant chacun des programmes existant.
- `obj` qui contient après compilation les fichiers objets générés
- `lib` qui contient après compilation une bibliothèque rassemblant tous les fichiers objets de parsing et manipulation du code assembleur
- `bin` qui contient après compilation les exécutables (programmes principaux)
- `doc` qui contient la documentation (html et latex)

Le fichier `README.txt` de ce répertoire contient des informations utiles : LISEZ-LE !

1) Ouvrez la documentation disponible dans le répertoire `include/html` en ouvrant le fichier `index.html` dans un navigateur web. Vous avez désormais accès à la documentation des différentes classes du projet et des méthodes à votre disposition.

Pour information, il est possible de générer la documentation avec la commande suivante à la racine du répertoire :

```
doxygen doxyconfig
```

La commande n'existe pas sur les salles de la PPTI, donc elle a été générée et fournie dans l'archive.

2) Positionnez vous dans le répertoire racine de l'archive (`CodeEtudiant`) et compilez le code avec la commande `make`.
La compilation compile les fichiers que vous trouverez dans les répertoires `src/base` et la place dans le répertoire `obj` ; 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 répertoire `bin/c` si le fichier source a une extension `.c`

3) Ouvrez le fichier `test_decoupage_function.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_decoupage_fonction src/example/ex_asm.s`
Que fait ce programme ?
Regarder le code du programme `test_decoupage_fonction.cpp` ainsi que le code de la classe `Program.cpp` et `Function.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 doit créer autant de blocs de base que nécessaire et les ajouter 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.

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 slo*t.**

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 juste devant la première instruction du bloc, une instruction si aucune étiquette ne précède cette première instruction.
2) un pointeur vers la dernière ligne composant le bloc de base, et
3) un pointeur vers la ligne correspondant au branchement terminant le bloc de base, 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::add_BB(Line *, Line *, Line *, int)` qui crée et ajoute un BB à la fonction (voir le commentaire de fonction dans le fichier `Function.cpp` directement)

Il est fortement 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 `t_Inst::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)

Attention il n'y a pas de fonction permettant de tester si un saut est inconditionnel !

Utilisez le programme principal `test_decoupage_bb.cpp` pour tester cette fonction. Vous pouvez bien sur l'adapter si vous le souhaitez. Regardez bien ce qu'il fait, et tester bien votre découpage sur différents exemples (`dep_inst.s` par exemple) et vérifier qu'il est correct. Un mauvais découpage posera problème dans les questions ultérieures et est parfois difficile à trouver… Notamment : les étiquettes doivent faire partie des blocs de base qu'elles désignent, il ne doit pas y avoir de bloc de base sans aucune instruction.

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.

Par déterminer qui est un successeur cible d'un saut, il faut déterminer, étant donnée l'étiquette de la cible saut utilisée dans l'instruction, 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` précédent un bloc de base. Attention à ne pas confondre les deux types/objets bien différents !
On peut récupérer un opérande de type `OPLabel*` avec la méthode :

```
OPLabel* Instruction::get_op_label();
```

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 en paramètre.

La fonction retourne `nullptr` si aucun bloc commençant par ce label dans la fonction n'est trouvé.
Pour que la méthode `Basic_block *Function::find_label_BB(OPLabel* label)` puisse fonctionner il ne faut pas oublier de calculer les étiquettes : il faut au préalable appeler une fois la fonction `comput_label` de la fonction ce qui est vérifié au travers d'un booléen à l'entrée de la fonction (idem pour le calcul des blocs de base !).

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 l'exercice 2, dans lequel ces fonctions sont déjà mentionnées.

On supposera que les codes utilisés pour tester ne contiennent pas de saut indirect sauf pour le retour à la fonction appelante (instruction `jr $31`) . Un bloc avec un saut indirect n'a pas de successeur.

Testez votre fonction avec le programme de test `test_liens_bb.cpp`. Ce code génère un fichier .dot du FCG de la fonction, cela permet de visualiser les liens entres les blocs de base. Testez bien avec différents codes, notamment ceux vus en TD, vous avez ainsi des éléments de comparaison !

Les fonctions demandées ici sont essentielles pour toute la suite des TP et du projet, il faut les finir rapidement et les tester intensément ! Un rendu des premiers TP (1 et 2) sera demandé rapidement afin de vous forcer à avancer.