LOG3210 - Élément de langages et compilateur
TP5 : Langage machine

Ettore Merlo – Professeur
Doriane Olewicki – Chargée de laboratoire

Hiver 2020

1 Objectifs

- Générer du code machine à partir du code intermédiaire;
- Calculer les variables vives IN et OUT;
- Calculer les next-use IN et OUT;
- Gérer l'allocation des registres.

2 Travail à faire

Pour ce dernier laboratoire, vous devez convertir un bloc de base de code intermédiaire en code machine. Ensuite, vous utiliserez le code machine généré pour compléter l'implémentation du calcul d'un élément de la suite de Fibonacci en utilisant un interpréteur.

Le langage machine que vous avez à supporter est celui décrit à la section 8.2.1 du livre. Vous n'avez besoin que d'une fraction des commandes, puisque vous ne gérer pas les tableaux, les pointeurs et les flux de contrôle. Vous aurez surtout besoin des commandes "LD", "ST" et "OP". Ici, un résumé des commandes :

- *LD dst, addr* : assignation *dst = addr*. Charge la valeur à l'adresse *addr* dans le registre *dst*.
- *ST x, r*. Charge la valeur du registre *r* à l'adresse *x*.
- *OP addr, src1, scr2* : *OP* est une opération (ex: *ADD*, *SUB*, *MUL*, *DIV*) et *addr*, *scr1* et *scr2* sont des adresses de registres. Assignation de type *addr = src1 OP src2*.

2.1 Langage de ce TP

Le langage que comprend la grammaire fournis avec le TP est différente de celle des TP précédent. Elle comprend un nœud pour enregistrer le nombre de registre disponible, les instructions, qui forme un bloc de base (et ne comporte donc aucun flux de contrôle), et une instruction de statement, qui ne fait pas partie du bloc de base MAIS vous informe des valeurs vives à la sortie du bloc (Life_OUT du dernier statement du bloc). De plus, le langage ne comprend que des entiers et des assignations. Allez voir le format dans les tests.

2.2 Etape à réaliser

1. Traduisez le code intermédiaire en code machine. Pour chaque statement, un ou plusieurs objet(s) MachLine est défini. Chacun des MachLine contient une List "line" dans laquelle vous pouvez mettre chaque partie de votre code machine. Exemple:

a = b + c	=>	[ADD, @a, @b, @c]

   Vous devez aussi définir les REF et DEF de chaque ligne et les PRED ET SUCC. C'est facile pour ces derniers, c'est du code basique ici, donc le succ de N est toujours N+1 (ou rien si on est à la dernière ligne) et le pred est N-1 (ou rien si on est à la première ligne). A noter que dans les expressions "LD @a, a" ont pour REF = [] et DEF = [@a] et les expressions "ST a, @a" pour REF = [@a] et DEF = []. Les variables non-pointeur (ex: 'a') sont considéré comme des constantes.

   A cette étape, j'identifierais encore les registres comme des pointeurs vers les identifiants: "@a" représente donc le registre contenant la valeur de "a". A l'étape suivante, vous transformerez ces "@*something*" en "R*number*".

   Le premier code machine généré à cette étape n'est donc pas optimisé du tout, ni utilisable car les registres ne sont pas encore définis.

   Voici quelques exemples de comment traduire les expressions d'assignation :

   - a = b * c : [MUL, @a, @b, @c]. Attention que si b et c n'existent pas avant la ligne, il faut les charger de la mémoire ([LD, @b, b] et [LD, @c, c]).
   - a = - b : [SUB, @a, #0, @b]. Attention que si b n'existe pas avant la ligne, il faut le charger de la mémoire ([LD, @b, b]).
   - a = b : [ADD, @a, #0, @b]. Attention que si b n'existe pas avant la ligne, il faut le charger de la mémoire ([LD, @b, b]).

   A la fin du bloc de code basique, il faut charger en mémoire toutes les variables ayant été modifiées qui seront encore utilisées après le morceau de code (les variables dans l'instruction return). L'instruction return ne fait PAS partie du code basique. Pour charger en mémoire : [ST a, @a].

2. Adaptez votre implémentation de variables vives au langage machine. L'implémentation ne devrait pas changer de beaucoup. Consultez la grammaire concernant les spécificités de chaque type d'assignation... IN et OUT du TP4 seront redéfinis à Life_IN et Life_OUT. Le Life_OUT de la dernière ligne devra contenir les variables dans l'expression return.

   Rappel de l'algorithme :

```
forall (node in nodeSet) {
   IN[node] = {}
   OUT[node] = {}
}

workList = {}
workList.push(stop_node);

while(!workList.empty()) {
   node = workList.pop();

   for (succNode in successors(node)) {
      OUT[node] = OUT[node] union IN[succNode];
   }
```

```
    OLD_IN = IN[node];
    IN[node] = (OUT[node] - DEF[node]) union REF[node];

    if(IN[node] != OLD_IN) {
        for(predNode in predecessors(node)) {
            workList.push(predNode);
        }
    }
}
```

3. Implémentez l'algorithme "Next Use". L'algorithme est décrit dans le bloc suivant. IN et OUT de next-use seront redéfinis par Next_IN et Next_OUT.

```
forall (node in nodeSet) {
    IN[node] = {}
    OUT[node] = {}
}

workList = {}
workList.push(stop_node);

while(!workList.empty()) {
    node = workList.pop();

    for (succNode in successors(node)) {
        OUT[node] = OUT[node] union IN[succNode];
    }

    OLD_IN = IN[node];
    for ((v,n) in OUT[node]) {
        if (v not in DEF[node]) {
            IN[node] = IN[node] union {(v,n)}
        }
    }

    for (v in REF[node]) {
        IN[node] = IN[node] union {(v, current_line_number)}
    }

    if(IN[node] != OLD_IN) {
        for(predNode in predecessors(node)) {
            workList.push(predNode);
        }
    }
}
```

4. Générez le graphe interférence du bloc de code. Le graphe d'interférence est un graphe non-dirigé dont les nœuds et arrêtes sont définis comme suit:

   - Noeuds: pour chaque pointeur d'identifiants ("@*something*"), on crée un noeud.
   - Arrêtes: Next_OUT, ajouter des arrêtes entre chaque pointeur présent dans le set. Exemple:

```
MUL @a @a @c (Next_OUT : [a:1, b:2, c:1, d:3])
=> 4 noeuds (@a, @b, @c, @d)
=> 6 arretes (@a–@b, @a–@c, @a–@d, @b–@c, @b–@d, @c–@d)
```

5. Assignation de registre en utilisation la coloration de graphe. Une fois votre graphe interférence généré, vous voulez "colorer" les noeuds (c'est-à-dire donner une numéro de registre à chaque pointeur, ex: "@a" devient "R0"). Dans un premier temps implémentez une coloration sans limite de registre (la limite est tout de même à 256 donc garder vos morceaux de code petit). Pour une coloration "k" (dans ce cas k=256):

   (a) Définissez un Stack pour empiler les noeuds.

   (b) Sélectionnez le noeud ayant le nombre de voisin le plus proche et inférieur à k. S'il y a plusieurs nœuds avec le même nombre de voisin, prendre le premier dans l'ordre alphabétique.

   (c) Enlevez le noeud du graphe ainsi que ses arrêtes et "push" le noeud sur la stack.

   (d) Recommencez à partir de (5b) jusqu'à ce que le graphe soit vide.

   (e) Pour chaque nœud de la Stack, "Pop" un nœud et le replacer dans le graphe d'interférence avec ses arrêtes connectés avec les nœuds popped avant celui-ci. Lui donner un numéro de registre différent de tous ses voisins déjà assignés. (Commencez la numérotation à 0, si un voisin à le registre 0, passez à 1... jusqu'à trouver un registre qui fonctionne.)

6. Maintenant, gérons en plus la limitation de registre. Le nombre de registres limite est donné au début des fichiers tests et est déjà stocké dans la variable "REG" du visiteurs.

   Un problème arrive à l'étape (5b) s'il n'y a plus de nœuds ayant moins de k voisins lors de l'empilage de la Stack. Alors, il faut arrêter l'empilage et "spill" le nœud, ensuite, recommencer le Stack.

   Pour le "spill", il faut alors changer le code machine. Parmi les nœuds restant dans le graphe d'interférence, vous sélectionnez le nœud avec le plus de voisin (s'il y a une égalité, prendre le premier dans l'ordre alphabétique). On appellera ce nœud "@a", représentant le pointeur vers la variable "a", pour la suite de l'explication. En utilisant Next_OUT, vous allez pouvoir adapter le code machine.

   Si Next_OUT de @a ne contient qu'un utilisation/adresse (le numéro de ligne d'utilisation de la variable "a"), alors il suffit d'ajouter une ligne de code machine "ST a @a" à la suite de la ligne d'utilisation si "@a" a été modifié ET est dans les valeurs de return. Ensuite, il faut recalculer les ensembles live variable et next-use, puis, recommencer à zéro la coloration (depuis l'étape 4).

   Si Next_OUT de @a contient plusieurs utilisations/adresses, la modification du code est à peu plus complexe. A nouveau, il faut ajouter une ligne de code machine "ST a @a" à la suite de la première ligne d'utilisation, cette fois si "@a" a été modifié. Ensuite, toutes les autres lignes d'utilisation doivent être modifiée pour considéré un nouveau pointeur. Tous les "@a" suivant la première ligne d'utilisation sont remplacés par "@a!" avec le symbole "!" représentant un nouveau pointeur. La deuxième utilisation de "@a" devient donc la première utilisation de "@a!" et il faut donc ajouter l'expression "LD @a! a" avant cette ligne. Pour voir un exemple, consultez les slides du cours 12.

7. Réduction de code. Si des expressions dans votre code machine final sont triviales, vous pouvez les enlever. Exemple: la ligne "ADD R0, #0, R0" peut-être supprimée. Si vous faites des réductions complémentaires, veuillez le préciser dans le rapport.

8. Affichage de l'output. Vous devez écrire dans "m_writer" vos résultats sous le format suivant:

```
*code machine Line0*
// ===== LINE 0 =====
// Life_IN = [...]
```

```
// Life_OUT = [...]
// Next_IN = [...]
// Next_OUT = [...]
*code machine Line1*
// ===== LINE 1 =====
(etc.)
```

Un exemple vous est donné dans le dossier test-suite pour la version "full" mais à vous de trouver le résultat pour 3 registres et 5 registres. Une fonction d'impression vous est aussi fournie.

3 Procédure

3.1 Données de test

Des données des tests se trouvent dans le dossier test-suite, séparées par visiteur. Pour tester le code de Fibonacci, il faut insérer dans le fichier `Machine-Code-Emulator/examples/fibb.asm` vos résultats bloc 1 et 2.

Il n'y a pas de fichier `expected` pour les tests sur 3 et 5 registres mais un exemple autre vous est fourni pour que vous ayez une référence pour "full" (sans contrainte de registre).

3.2 Utilisation du simulateur

Pour cette partie, ouvrez un terminal bash dans le dossier `Machine-Code-Emulator/`.

Le simulateur a été écrit en Python3 et nécessite l'utilisation des bibliothèques Arpeggio et NumPy. Si elles ne sont pas installées, vous pouvez les installer avec les commandes suivantes :

```
pip3 install --user arpeggio==1.5
pip3 install --user numpy
```

Au début de la méthode `simulate` du fichier `simulator.py`, l'environnement est créé avec les contraintes du nombre de registres.

Pour exécuter le code machine, lancez la commande suivante :

```
python3 run_tests.py
```

Le code vous demandera alors quel nombre de la suite de Fibonacci vous voulez, vous devrez le rentrer manuellement. Voici un exemple d'output CORRECT avec le nombre 10 :

```
| Please enter the number of the fibonacci suite to compute: |
10
55
| END |

State of simulation:
Registers: [55, 34, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Memory:
[[ 10   0  21  34 610 987   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0]
```

```
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
Test of <<examples/fibb.asm>> Succeeded
```

Tous les fichiers présents dans le dossier examples seront lancés.

Attention, lancer la commande ainsi met une contrainte de 15 registres à votre programme (ce qui est assez pour lancer votre code avant d'implémenter le respect des contraintes). Pour ajouter la contrainte de nombre de registres à votre simulation, ajouter le nombre max de registre après la commande (ex: 3) :

```
python3 run_tests.py 3
```

Remarque : vous pouvez constater que ce simulateur a été écrit avec un analyseur lexical et syntaxique. Le fichier `.peg` décrit la grammaire et un ensemble de visiteurs qui prépare et effectue la simulation.

3.3 Tests automatiques

Pour cette partie, ouvrez un terminal bash dans le dossier `Machine-Code-Emulator/`.

Si vous ne voulez pas manuellement modifier le contenu du fichier `Machine-Code-Emulator/examples/fibb.asm` à chaque tests, vous pouvez aussi utiliser le script automatisé `auto_run_test.py` avec la commande suivante :

```
python3 auto_run_tests.py
```

Celui-ci tests dans l'ordre :

- le fonctionnement du code machine dans contrainte de registre;
- le fonctionnement du code machine avec 3 registres;
- le fonctionnement du code machine avec 5 registres;
- le fonctionnement du code machine avec au moins une réduction de registre (vous arrivez à moins de 11 registres pour les fichier 3 registres. Ce tests est nécessaire pour le cas où vous n'avez pas réussi à faire les contraintes 3 et 5 registres et que vous voulez montrer que vous avez quand même fait une réduction. (voir Barème)

Vous devrez donc rentrer 4 fois des valeurs de Fibonacci. Vous pouvez commenter les parties de code que vous ne voulez pas tester dans le script directement.

4 Rapport

Pour ce TP, on vous demande de rédiger un rapport clair et bref. Celui-ci devra contenir les informations suivantes :

- Description détaillée de votre implémentation, en particulier la partie coloration de graphe, génération du code intermédiaire et réduction complémentaire de code.

- Diagramme UML représentant les différentes partie de votre implémentation.
- Description de vos propres tests : On vous demande de générer vous-même quelques tests et de justifier pourquoi ceux-ci prouve que votre code est valide. Ces tests doivent être ajouté dans test-suite pour vous afin de pouvoir être vérifié, mais pour la remise, ils devront être fourni dans un dossier à part.

Le rapport ne devrait pas faire plus de 2 pages + graphe UML. Soyez concis.

5 Barème

Le TP est évalué sur 50 points, les points étant distribué comme suit :

- Implémentation Life variable (IN et OUT) et next-use : 5 points;
- Implémentation language machine sans contrainte de registres : 15 points;
- Implémentation language machine avec contrainte de registres : 15 points;
- Explication de l'implémentation + UML : 10 points;
- Vos propres tests, pertinants et justifiés dans le rapport : 5 points.

ATTENTION : la couleur des tests sur IntelliJ n'a pas d'importance. Les tests fournis ne sont là qu'à titre d'exemples. Pour les code concernant Fibonacci, la vérification est fait avec le script Python.

6 Remise

Le devoir doit être fait en **binôme**. Remettez sur Moodle une archive nommée *log3210-tp5-matricule1-matricule2.zip* contenant les fichiers suivant:

- PrintMachineCodeVisitor.java (si vous créez de nouvelles classes, ajoutée les dans ce visiteur)
- rapport-matricule1-matricule2.pdf
- test/*.ci

uniquement le fichier `PrintMachineCodeVisitor` ainsi qu'un fichier `README.md` (si vous le désirez) contenant tous commentaires concernant le projet. Attention : si vous ne respectez pas les contraintes, veuillez indiquer dans le README les quelles, notamment le nombre de registre !

L'échéance pour la remise est le **25 Avril 2020 à 23 h 55**.

Une pénalité de 10 points (50%) s'appliquera par jour de retard. Une pénalité de 4 points (20%) s'appliquera si la remise n'est pas conforme aux exigences (nom du fichier de remise, fichiers à rendre).

Si vous avez des questions, veuillez me contacter sur Moodle ou sur mon courriel : doriane.olewicki@gmail.com.