LOG3210 - Élément de langages et compilateur
TP4 : 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 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. Remplissez la liste `CODE`. Chaque ligne de code intermédiaire correspondra à un `MachLine`. Vous pouvez utiliser le constructeur `MachLine(op, assign, left, right)`. (Je vous recommande de stocker dans les différents champs les valeurs du code intermédiaire, par exemple "a=b+c" donne MachLine("ADD", a, b, c))

2. Implémentez le code variable vive pour bloc de base. IN et OUT du code sont redéfinis à Life_IN et Life_OUT. Le Life_OUT de la dernière ligne devra contenir les variables dans l'expression return (`ReturnStmt`).

   Rappel de l'algorithme :

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

OUT[lastNode] = Returned_Values
for(i=nodeList.size()-1; i >= 0; i--) {
   if (i < (nodeList.size() - 1))
      OUT[nodeList[i]] = IN[nodeList[i+1]];

   IN[nodeList[i]] = (OUT[nodeList[i]] - DEF[nodeList[i]])
                                 union REF[nodeList[i]];
}
```

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 nodeList) {
IN[node] = {}
OUT[node] = {}
}

for(i=nodeList.size()-1; i >= 0; i--) {
   if (i < (nodeList.size() - 1))
      OUT[nodeList[i]] = IN[nodeList[i+1]];

   for ((v,n) in OUT[nodeList[i]] ) {
      if (not DEF[nodeList[i]].contains(v))
         IN[nodeList[i]] = IN[nodeList[i]] union {(v, n)}
   }

   for(ref in REF[nodeList[i]])
      IN[nodeList[i]] = IN[nodeList[i]] union {(ref, current_line_number)}
}
```

4. Générez le code machine. Les générations de variable vive et next-use vous aiderons à générer le code machine en respectant les limitations de registre (le nombre de registre est limité à `REG`). Vous êtes encouragé à faire des réductions de code, par exemple en cachant les lignes de code inutile (ex: ADD R0, #0, R0). Consultez les expected fournis pour voir le format de l'output attendu.

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 sans 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 Barème

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

- Implémentation Life variable : 3 points;
- Implémentation Next-use : 3 points;
- Implémentation language machine sans contrainte de registres : 8 points;
- Implémentation language machine avec contrainte de registres (3 et 5 registres) : 6 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.

5 Remise

Le devoir doit être fait en **binôme**. Remettez sur Moodle une archive nommée *log3210-tp4-matricule1-matricule2.zip* contenant les fichiers suivant:

- PrintMachineCodeVisitor.java (si vous créez de nouvelles classes, ajoutez les dans ce visiteur)
- README.md (facultatif)

L'échéance pour la remise est le **7 Décembre 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 discord ou sur mon courriel : doriane.olewicki@gmail.com.