LOG3210 - Éléments de langages et compilateurs
TP4 : Langage machine

Chargée de cours
Sara Beddouch

Chargés de laboratoire
Hubert Boucher - G1
Thierry Beaulieu - G2
Christopher Salem - G3

Automne 2024

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

Dans ce TP, vous devez convertir du code intermédiaire en code machine. À la fin du TP, afin de tester votre implémentation, vous devrez utiliser le code machine généré pour calculer 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` :

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

Langage de ce TP

Dans les TPs 1 à 3, on lisait un fichier de code (dossier `data`) que l'on souhaitait transformer en code intermédiaire (dossier `expected`).

Pour le TP 4, on part cette fois du code intermédiaire généré au TP 3 (voir dossier `data`), et on souhaite transformer ce code en code machine (dossier `expected`). Nous n'aurons donc à lire que des lignes de code intermédiaire, c'est-à-dire uniquement des entiers et des assignations.

Exercice 1: Remplir la variable `CODE`

Dans ce premier exercice, il faut remplir la variable `CODE`. Chaque ligne de code intermédiaire doit être enregistrée dans un nouvel objet `MachineCodeLine` et stocké dans cette variable.
Ainsi, `a=b+c` dans le code intermédiaire donnera `MachineCodeLine("ADD", a, b, c)`.
Aidez-vous du `Langage.jjt` pour comprendre comment accepter les enfants!

Exercice 2: Implémenter l'algorithme des variables vives

Dans ce deuxième exercice, il faut implémenter l'algorithme des variables vives (voir annexe A).

Exercice 3: Implémenter l'algorithme next-use

Pour ce troisième exercice, il faut implémenter l'algorithme des next-use (voir annexe B).

Exercice 4: Générer le code machine

Les générations de variables vives et next-use vous aiderons à générer le code machine en respectant les limitations de registre (le nombre de registre est limité à `MAX_REGISTERS_COUNT`). 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 Utilisation du simulateur

3.1 Tests manuels

Dans cette partie, on va pouvoir essayer le code machine que l'on a généré.

1. Ouvrez un terminal, et naviguez vers le dossier `simulator`.
2. Le simlateur a été écrit en Python, et nécessite l'utilisation de deux bibliothèques à installer (si ce n'est pas déjà fait):

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

3. Pour tester par exemple le code à 3 registres, il faut aller dans le fichier `simulator/examples/fibb.asm`, et identifier ces deux TODO:

```
// TODO:: PUT THE BLOCK 1 HERE !
// ...
// TODO:: END THE BLOCK 1 HERE ABOVE !
```

```
// TODO:: PUT THE BLOCK 2 HERE !
// ...
// TODO:: END THE BLOCK 2 HERE ABOVE!
```

4. Comme les commentaires indiquent, il faut maintenant copier/coller les codes machines générés dans le fichier `.asm`:

- test-suite/PrintMachineCodeTest/result/block_1_3.ci → BLOCK 1
- test-suite/PrintMachineCodeTest/result/block_2_3.ci → BLOCK 2

Ceci est l'exemple pour tester le code machine à 3 registres. Pour tester les 5 registres et illimités, il faudra copier/coller les results `*_5` et `*_full`.

5. Maintenant, pour tester pour 3 registres:

```
python3 run_tests.py 3  # ou 5 pour les 5 registres.
```

ou pour la version sans la contrainte de registres:

```
python3 run_tests.py
```

6. Le code vous demandera alors quel nombre de la suite de Fibonacci vous voulez calculer. 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]

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

Remarque

Vous pouvez constater que ce simulateur a été écrit avec un analyseur lexical et syntaxique! Dans le code Python, le fichier `.peg` décrit la grammaire, et un ensemble de visiteurs préparent et effectuent la simulation.

3.2 Tests automatiques

Pour les tests automatiques, placez-vous comme précédemment dans le dossier `simulator/`.

Si vous ne voulez pas manuellement modifier le contenu du fichier `simulator/examples/fibb.asm` à chaque test, vous pouvez utiliser le script automatisé `auto_run_test.py` avec la commande suivante :

```
python3 auto_run_tests.py
```

Celui-ci teste dans l'ordre :

- le fonctionnement du code machine sans contrainte de registres;
- le fonctionnement du code machine avec 3 registres;
- le fonctionnement du code machine avec 5 registres.

Vous devrez donc rentrer 3 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 20 points, distribués comme suit :

Fonctionnalités	
Variables vives	/3
Next-use	/3
Langage machine avec registres illimités	/8
Langage machine avec limite de registres (3 ou 5)	/6
Pénalités	
Retard (par jour de retard)	-10
Non-respect des consignes de remise (nom du fichier,...)	-4
Code de mauvaise qualité	-2
Total	**/20**

Le laboratoire sera principalement évalué avec le simulateur. Assurez-vous que le code fonctionne avec!

5 Remise

L'échéance pour la remise du TP4 est le `3 Décembre 2024 à 23:55`.

Remettez sur Moodle une archive nommée `log3210-tp4-matricule1-matricule2.zip` (tout en minuscule), contenant **uniquement** les 2 fichiers suivants:

- `PrintMachineCodeVisitor.java`;
- `README.md` (facultatif, à fournir si vous avez des commentaires à ajouter sur votre projet).

Le devoir doit être fait en **binôme**. Si vous avez des questions, veuillez nous contacter sur Discord via le canal de votre équipe (`#equipe-00`), sans nous mentionner! Les chargés vous répondront dès que possible, s'ils ont du temps libre en dehors des séances!

A Algorithme: Variables vives

- `IN` := `Life_IN`
- `OUT` := `Life_OUT`
- Le `Life_OUT` de la dernière ligne devra contenir les variables dans l'expression return (`ReturnStmt`).

```
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]]
}
```

B Algorithme: Next-use

- `IN` := `Next_IN`
- `OUT` := `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)
   }
  }
}
```