m2103

TM (5) Projet calculette : compilation

Table des matières

1 Présentation

Dans cette partie, on apprendrez comment **compiler** du code, c'est-à-dire construire une suite d'instructions de bas niveau qui, en s'exécutant, produit le résultat décrit par le code source.

On verra que cette compilation peut se faire en ajoutant une seule méthode aux arbres syntaxiques.

Pré-requis : pour faire ce travail, vous devez avoir, dans le code de calculette, mis en place la fabrication des arbres de syntaxe abstraite par l'analyseur.

2 Un exemple

Exemple, à partir du code source

```
y = 3*x + 1
```

on produira le code

```
li   r1,3
ld   r2,x
mul  r1,r2
li   r1,2
add  r1,r2
st   r1,y
```

écrit en *langage d'assemblage* pour une machine hypothétique.

3 Rappels

- En mémoire, un programme est une suite de bits qui codent les instructions que le processeur va exécuter.
- Ces instructions ont des formats spécifiques : un certain nombre de bits pour le code opération, des zones pour noter les numéros des registres, les adresses des opérandes, etc.
- Pour décrire ces instructions, les programmeurs utilisent une notation symbolique, avec des **mnémoniques** pour les opérations. En effet, on comprend mieux

```
li r5,33
```

qui veut dire "charger la valeur `33` dans le registre `r5`", (`li` = load immediate) qu'une instruction binaire `1000010100010001`.

- La traduction est faite par un programme qu'on appelle **assembleur**.
- On dit que le code source est écrit en "langage d'assemblage".

On parle aussi couramment, par métonymie, de

- "programmation en assembleur" (le mot *assembleur* désigne en réalité le **programme de traduction**),
- "programmation en langage machine" (qui est en fait le **résultat de la traduction**).

4 La machine et son langage d'assemblage

On va prendre comme exemple une machine simple, avec une architecture RISC :

- la machine dispose de registres `r1`,`r2`, `r3` … en nombre assez grand (typiquement, 32).
- il y a quelques types d'instructions :
  - chargement d'une valeur immédiate dans un registre,
  - lecture / écriture d'un registre depuis/vers la mémoire,
  - opérations arithmétiques entre 2 registres.

Instruction	Signification	Description
`li` registre,valeur	*load immediate*	charge la valeur dans le registre
`ld` registre,adresse	*load*	transfert de mémoire à registre
`st` registre,adresse	*store*	transfert de mémoire à registre
`add` reg1,reg2	*add registers*	calcule reg1+reg2, résultat dans reg1
`sub` reg1,reg2	*subtract registers*	calcule reg1-reg2, résultat dans reg1
`mul` reg1,reg2	*multiply registers*	calcule reg1*reg2, résultat dans reg1
`div` reg1,reg2	*divide registers*	calcule reg1/reg2, résultat dans reg1

5 Exercices

Ces exercices vous permettront

- de vous familiariser avec les instructions,
- d'acquérir une intuition sur la manière de procéder pour traduire les expressions et affectations.

1. Que fait la séquence suivante ?

```
ld r1,A
ld r2,B
st r1,B
st r2,A
```

Comment faire : supposer qu'au départ A et B contiennent des valeurs `a` et `b`, et que le contenu des registres est inconnu. Puis remplir le tableau en exécutant pas-à-pas :

instruction	A	B	r1	r2
	a	b	????	????
ld r1,A				
ld r2,B				
st r1,B				
st r2,A				

Ici c'est simple, on peut le faire de tête, mais quand il y plus de registres c'est un effort dont on peut se dispenser, alors qu'il est si simple de prendre un papier…

2. Que fait la séquence suivante ?

```
                A=   B=   X=   Y=   r1=   r2=   r3=   r4=
ld  r1,A
ld  r2,X
mul r1,r2
ld  r3,B
ld  r4,Y
mul r3,r4
sub r1,r3
st  r1,Z
```

3. Écrire un code qui fait la même chose, mais en économisant les registres.

4. Fournir une traduction de l'expression

   ```
   (a*x+2)/(y-3)
   ```

qui dépose le résultat dans le registre `r1`.

6 Traduction d'une expression

L'objectif est de modifier notre calculette pour qu'elle affiche aussi la traduction de l'expression en langage d'assemblage.

Plus précisément, un texte avec une suite d'instructions en langage d'assemblage qui **amène le résultat dans le registre** `r1`.

Exemple, si on tape `a*x + b`, on obtiendra quelque chose comme `ld r1,a ; ld r2,x ; mul r1,r2 ; ld r2,b ; add r1,r2`.

6.1 Méthode `getCode`

On va donc ajouter une méthode `getCode()`, qui retourne le code correspondant à une expression, avec les instructions qui laissent la valeur de l'expression **dans le registre** `r1`.

Cette méthode `getCode()` :

- s'applique à l'arbre de syntaxe abstraite (obtenu à la phase précédente)
- prend naturellement place dans l'interface des arbres de syntaxe abstraite.

Pour traduire une expression, il faut traduire ses sous-expressions. La traduction sera récursive, de façon parfaitement naturelle.

6.2 Le calcul d'une expression complexe

Le traduction de `(a*x+2)/(5*y-3)` comporte 3 étapes

1. construire le code qui met le résultat de `a*x+2` dans le registre `r1` ;
2. construire le code qui met le résultat de `5*y-3` dans un *autre* registre `rx` ;
3. concaténer et ajouter l'instruction `div r1,rx`

Quel registre choisir pour `rx` ?

La réponse est `r2`. Pourquoi ?

- le calcul de la partie gauche utilise éventuellement plusieurs registres pour des calculs intermédiaires.
- mais quand il est terminé, le résultat occupe `r1`, les autres registres sont disponibles.
- on peut faire le calcul de la partie droite en utilisant les autres registres, à partir de `r2`.

Et de même,

- pour calculer la partie droite `5*y-3` dans `r2`, on calculera `5*y` dans `r2`, 3 dans `r3` et on fera la différence.
- pour calculer `5*y` dans `r2`, on mettre 5 dans `r2`, y dans `r3` et on multipliera.

En résumé, à chaque étape, on construit le code qui amène le résultat dans le registre numéro n, en se servant des registres n+1, n+2, pour les calculs intermédiaires.

La méthode `getCode()` (appelée par la calculette) fait donc appel à une autre méthode, paramétrée par le **numéro du registre** destination.

Remarque : on suppose ici qu'on dispose de registres en nombre suffisant (on manipule rarement des expressions à plus de 32 niveaux !).

7 Réalisation

7.1 Mise en place de l'interface

1. Dans le code de l'interface, ajouter :

```
interface Expr {
    // ...
    default public String getCode() {
        return getCode(1) ;
    }

    String getCode(int numRegistre) ;
```

2. Laisser `Netbeans` ajouter des "stubs" la méthode `getCode(int)`.

3. Dans le code de la calculette, ajoutez l'affichage à partir de l'expression obtenue par analyse.

7.2 Constantes

Dans la classe qui représente les constantes, la méthode `getCode` retourne simplement une instruction "`li`"

```
class ExprConstante .....   // quel que soit le nom que vous lui avez donné
{
    int valeur;

    @Override
    String getCode(int numeroRegistre) {
        return String.format("li   r%d,%d", numeroRegistre, valeur);
    }
    // ...
}
```

1. Ajoutez ce code
2. Testez. En donnant la valeur "33", vous devez obtenir le code "`li r1,33`".

7.3 Variables

Ce n'est pas plus compliqué. Produire "`ld  rx,nom`".

7.4 Addition

Pour obtenir le code d'une somme dans le registre n

- trouver le code qui met la valeur de la partie gauche dans le registre n ;
- trouver le code qui met la valeur de la partie droite dans le registre n+1 ;
- concaténer les deux avec `add rn,r(n+1)`

Idem pour les autres opérations.

7.5 Affectation

pour le code qui affecte une variable V avec une expression E en utilisant le registre n

- calculer la valeur de E dans le registre n
- ajouter une instruction pour ranger le résultat.

La valeur de l'expression reste disponible dans le registre. Ce qui fait qu'on pourra l'utiliser comme dans :

```
a = (b = 10) + (c = 20)
```

qui devrait se traduire par

```
li  r1,10
st  r1,b
li  r2,20
st  r2,c
add r1,r2
st  r1,a
```

8 Synthèse

- La compilation se ramène à un calcul sur les arbres de syntaxe abstraite.
- C'est un calcul récursif, ce qui est naturel sur des AST définis eux-mêmes de façon inductive.
- En partant de cette idée, on peut écrire un compilateur complet. Le code pour une boucle "tant que", c'est
  - le code de la condition, avec un saut à la fin si elle est fausse
  - le code du corps de la boucle
  - un saut pour revenir au début.
- La difficulté, passée sous silence, est l'optimisation. Ici, l'instruction `a = 2+2` sera traduite par :

```
li  r1,2
li  r2,2
add r1,r2
st  r1,a
```

qui est correct, mais on peut certainement faire mieux !