Travaux Dirigés Compilation: Feuille 1

Informatique 2ème année. ENSEIRB 2018/2019

—David Janin & Myriam Desainte-Catherine—

Préliminaire : code 3 adresses

Le langage cible utilisé ici est du *code 3 adresses* c'est à dire un sous-ensemble d'instructions C qui ne manipulent que trois adresses au plus. Intuitivement, une adresse correspond à un accès mémoire, en lecture ou en écriture. Dans ce code, on manipule des variables de types simples (`float`, `int`,`bool`, etc...) qu'on peut voir comme des *registres typés* sans limitations sur le nombre de registres utilisables. On pourra faire apparaître le type des registres dans leurs noms comme illustrés dans les exemples ci-dessous.

Un premier exemple d'instruction 3 adresses valide est la conversion de type :

```
ri1 = (int) rf1;
```

avec conversion *explicite* du type `float` vers le type `int`. La conversion de type *implicite* n'est pas autorisée en code 3 adresses.

Un autre exemple d'instruction 3 adresses valide est le calcul binaire homogène :

```
ri1 = ri2 + ri3;
```

les variables de droite pouvant être remplacées par des constantes. Afin de manipuler des pointeurs, on pourra s'autoriser l'instruction 3 adresses de la forme :

```
x = (int) *px;
```

Le type d'un tel accès mémoire devra toujours être explicite. On autorise aussi en code 3 adresses l'application d'un opérateur unaire sur un registre tel que, par exemple, la négation sur les booléens ou l'opposé sur les types numériques.

Dans la définition du code trois adresses, un peu arbitraire, on ne compte pas l'adresse du registre résultat de l'UAL (Unité Arithmétique et Logique), pas plus qu'on ne compte l'adresse du registre résultat d'une conversion.

Afin de coder les contrôles, on autorise aussi les instructions de branchement, conditionnels ou pas, associées à des labels tels que, par exemple :

```
l2: goto l1;
l3: if rb goto l2;
l1: if ! rb goto l3;
```

sur un registre `rb` de type booléen, le test de sa négation étant autorisé. Attention : les labels de branchements `l1`, `l2` ou `l3` ne sont que des symboles qu'on retrouve autant de fois qu'on veut à la suite des branchements (`goto`) mais une et une seule fois à gauche d'une instruction.

On considèrera aussi l'instruction vide qui ne fait rien et on la notera :

```
nop
```

Pour la lisibilité du code 3 adresses, on veillera à aligner les instructions sur une seul colone, les labels de branchements identifiant une ligne se situant à gauche de cette colonne. Aucune autre instruction n'est (pour l'instant) autorisée.

Remarque : oui, le type `bool` n'existe pas en C, mais on le fait apparaitre en code 3 adresses. On conviendra que les comparaisons (par exemple `<=`) renvoient des valeurs de même type, qu'on devra convertir en booléen. De plus, les opérateurs binaires doivent être dupliqués autant que necessaire avec, par exemple, une addition de type `int`, une addition de type `float`, etc.... Cependant, comme ci-dessus, on s'autorisera à simplifier l'écriture des opérateurs en omettant leur type.

Préambule aux exercices de “compilation à la main”

Dans tous les exercices ci-dessous, on s’appuiera sur la structure syntaxique des instructions à compiler qu’on pourra (c’est très recommandé) représenter sous la forme d’arbre de syntaxe “abstraite”.

On cherchera aussi à faire apparaitre des “schémas” de compilation, c’est-à-dire des techniques *localement* applicables, sans *analyse de code*, en *respectant à la lettre* le code source, aussi absurde qu’il puisse paraître. Un compilateur est au service du programmeur. Par exemple le fameux code “espion”

```
float A = 1.0, B = 1.0;
while (((A + 1) - A) -1) == 0 do
  A = 2*A;
while (((A + B) - A) -B) != 0 do
  B = B+1;
printf("Base %i\n",B);
```

pourrait *a priori* être compilé en

```
l : goto l
```

Mais ce serait une erreur comme le montre son execution après compilation avec gcc sans optimisation. Faut-il le rappeler, le type `float` n’est pas le type des réels.

1 Compilation des expressions

▸**Exercice 1.** *Donner une majoration du nombre d’adresses nécessaire à l’évaluation des instructions C suivantes :*

```
x = (y1 + y2) + y3;

y = *(px++);

*(px+3) = *(px + y) + z;
```

Proposer pour chacune des lignes de codes ci-dessus une compilation en code trois adresses. Indication : on pourra réutiliser verbatim les noms variables apparaissant dans ces expressions, dites variables utilisateurs, *en ajoutant autant de* registres compilateurs *que souhaités.*

▸**Exercice 2.** *Compiler les instructions :*

```
x = y1 + (y2 * (y3 + (y4 * y5)));

x = (((y1 + y2) * y3) + y4) * y5;

x = ((y1 + y2) * (y3 + y4)) + ((y3 + y1) * (y2 + y5));
```

Combien de registres utilisez vous ? Sont-ils necessaires ? Peut-on facilement automatiser leur optimisation ?

2 Compilation des instructions de contrôle

▸**Exercice 3.** *Proposer un “schéma” de compilation en code 3 adresses pour les instructions de contrôle apparaissant dans les exemples suivants :*

```
if x then y = 0;

if b then x = 1 else x = 2;

while (x < 100) do x = x + 1;

repeat x = x + 1 while x < 100;

do x = x + 2 until x == 100;
```

On utilisera bien entendu des branchements conditionels.

3 Compilation "paresseuse" des booléens

La compilation d'une expression booléenne peut toujours être représentée par un triplet

(`p`, `lt`, `lf`)

où `p` est un programme (en code 3 adresses) qui branche sur `lt` si le booléen est évalué à *vrai* et qui branche sur `lf` si le booléen est évalué à *faux*. Cette représentation est dite "paresseuse" puisqu'elle permet facilement de mettre en oeuvre les principes suivants :

(1) dans une conjonction $e1 \wedge e2$ il n'est pas necessaire d'évaluer $e2$ lorsque $e1$ est faux,

(2) dans une disjonction $e1 \vee e2$ il n'est pas necessaire d'évaluer $e2$ lorsque $e1$ est vrai.

Cette "optimisation", qui est en fait une sémantique asymétrique des booléens, est mise en oeuvre dans le langage *C*.

▸**Exercice 4.** *En supposant l'expression booléenne* `b` *compilée comme ci-dessus, proposer un schéma de traduction des flots de contrôles :*

```
if b then p1;

if b then p1 else p2;

while b do p1;

do p1 until b;
```

▸**Exercice 5.**

(1) Donner une représentation "paresseuse" des booléens constants true *et* false.

(2) Donner une représentation "paresseuse" d'une variable de type booléen.

(3) Comment combiner des représentations paresseuses (`p1`, `lt1`, `lf1`*) et (*`p2`, `lt2`, `lf2`*) de deux expressions booléennes pour obtenir une représentation paresseuse de :*

(a) la négation de la première,

(b) leur conjonction,

(c) leur disjonction.

(4) En déduire un schéma récursif de traduction des expressions booléennes en représentation paresseuse qui s'appuient sur la syntaxe des arbres de ces expressions booléennes.

Note historique : cette propriété des booléens est utilisé par Alonzo Church dans son codage des booléens (et des conditionelles) en λ-calcul en posant $true = \lambda x \lambda y.x$ et $false = \lambda x \lambda y.y$. On construit alors facilement le code de la fonction $ifte$ (if-then-else) qui, appliquée à trois arguments b, x et y vaut x si b est vrai et qui vaut y si b est faux.

4 Optimisation du nombre de registres

▸**Exercice 6.** *Proposer un algorithme de calcul du nombre de registres nécessaire à l'évaluation d'une expression arithmétique et de production de code associé.*

Indication : analyser la liberté dont on dispose pour générer le code 3 adresses (sans supposer que les opérateurs binaires sont associatifs) et l'exploiter au maximum. On observera que cette tâche necessite deux passes, c'est à dire deux lectures du code source.