Examen final de compilation

Énsiie, semestre 3

18 janvier 2011

Durée : 1h45.
Tout document personnel autorisé (pas de prêt entre voisins).
Ce sujet comporte quatre exercices indépendants, qui peuvent être traités dans l'ordre voulu. Il contient 4 pages.
Le barème est donné à titre indicatif, il est susceptible d'être modifié. Le total est sur 30 points.

Exercice 1 : Analyse syntaxique *(8 points)*

On considère la grammaire $G_1 = (\{a; b\}, \{S; A\}, S, P_1)$ et $G_2 = (\{c; d; e; f\}, \{C; D; E\}, C, P_2)$ avec

$$P_1 = \begin{cases} S \to aAa \\ S \to bAba \\ A \to b \\ A \to \epsilon \end{cases} \qquad P_2 = \begin{cases} C \to Dd \\ C \to Ee \\ C \to f \\ D \to Cc \\ E \to Cc \end{cases}$$

(ϵ représente le mot vide).

1. La grammaire G_1 est-elle dans LL(1), LR(0), LR(1) ? À chaque fois, une réponse détaillée est attendue.
2. Quand la grammaire n'appartient pas à une de ces classes, donner une grammaire appartenant à cette classe et reconnaissant le même langage. On justifiera le fait que la grammaire appartient à la classe et que le même langage est reconnu.
3. Mêmes questions pour G_2.

Exercice 2 : Pile d'appel *(8 points)*

On considère un langage source du type Pascal, dans lequel on peut définir des fonctions imbriquées. On utilise la convention d'appel MIPS. On suppose de plus que toutes

les variables locales des fonctions sont allouées sur la pile, et non pas stockées dans des registres physiques.

Soit le programme suivant :

```
function f (x : integer) : integer;
  var y : integer;
  function g (a, b, c, d, e : integer) : integer;
    var u : integer;
    function h (y, z : integer) : integer;
      var t : integer;
      begin
        t := x + z; ④
        h := t + y;
      end;
    begin
      u := h(a, e) ⑤ * u;
      g := i(b, y, c, x, d);
    end;
  function i (j, k, l, m, n : integer) : integer;
    begin
      y := j + k; ②
      i := x + l - m;
    end;
begin
  y := 3 * x + 1; ①
  y := i(x, y, x, y, x) ③;
  f := g(y, x, y, x, y) ⑥ + 2;
end;
```

1. Quels registres (parmi `$a0`–`$a3` et `$ra`) ont besoin d'être sauvegardés par `f` (respectivement par `g`, par `h` et par `i`) ? On supposera par la suite qu'ils sont sauvegardés sur la pile d'appel. On rappelle que le lien statique d'une fonction à n arguments est passé comme un $n + 1^{\text{e}}$ argument.
2. Donner la forme détaillée des trames des fonctions `f`, `g`, `h` et `i`.
3. Une fonction appelle `f(1)` avec comme adresse de retour ®. À chacun des points ① à ⑥, dessiner la pile d'appel et donner le contenu des registres `$a0`–`$a3`, `$v0` et `$ra`. Attention, un même point peut être atteint plusieurs fois. On emploiera l'optimisation des appels terminaux. On pourra utiliser les chiffres entourés pour les adresses de retour. On mettra *indéf* dans le cas où une valeur ne peut être connue.
4. Traduire la fonction `h` en instructions MIPS, y compris la création et destruction de sa trame et le retour à l'appelant. (On utilisera des pseudo-registres en cas de besoin de stocker des calculs intermédiaires.)

Exercice 3 : Allocation de registres *(5 points)*

On considère le bout de programme Pseudo-Pascal suivant :

```
neutre := 0;
abs := neutre;
opp := -x;
while opp > x do
begin
  tmp := x;
  x := opp;
  opp := tmp;
end;
abs := x;
```

1. Donner le graphe de flot de contrôle correspondant, en gardant une syntaxe Pseudo-Pascal.
2. Calculer la durée de vie des variables.
3. En déduire le graphe d'interférence, avec les arêtes de préférence.
4. 2-colorier le graphe à l'aide de l'algorithme de George et Appel, en détaillant son déroulement. Si un spill est nécessaire, on choisira la variable la moins utilisée dans le programme.

Exercice 4 : *Reaching definitions (9 points)*

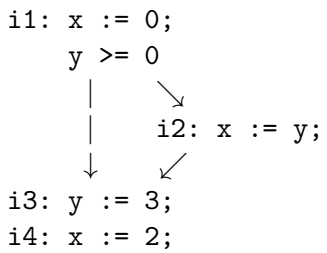
Pour certaines optimisations, il est parfois nécessaire de connaître à quels endroits du programme a été définie une variable. On appelle ces instructions des *reaching definitions* :

Définition 1. *Les* reaching definitions *à un point de programme p sont les instructions i définissant une variable v et telles qu'il existe un chemin de i à p le long duquel v n'est pas redéfinie.*

$$Reach(p, v) = \{i : i \text{ définit } v \text{ et il existe un chemin de } i \text{ à } p \text{ qui ne redéfinit pas } v\}$$

Cet exercice a pour but d'écrire un algorithme permettant de calculer les reaching definitions à partir du graphe de flot de contrôle.

1. Considérer le graphe de flot de contrôle suivant :



Donner $\mathit{Reach}((\mathit{avant}, i), \texttt{x})$ et $\mathit{Reach}((\mathit{après}, i), \texttt{x})$ pour $i =$ `i1`, `i2`, `i3` et `i4`.

2. Quelle est la relation entre les *reaching definitions* au point situé avant une instruction i et celles situées après les instructions précédant immédiatement i ?
3. Soit une instruction i : `op %v, ...` définissant la variable `%v`. Justifier les relations suivantes :

$$\begin{aligned} \mathit{Reach}((\mathit{après}, i), \texttt{\%v}) &= \{i\} \\ \mathit{Reach}((\mathit{après}, i), w) &= \mathit{Reach}((\mathit{avant}, i), w) \qquad \text{pour } w \neq \texttt{\%v} \end{aligned}$$

4. On pose $\mathcal{I}\mathit{nstr}$ l'ensemble des instructions du programme et $\mathcal{V}\mathit{ar}$ l'ensemble des variables. Dans quel espace de fonctions évolue *Reach* ? En déduire que *Reach* peut être calculée à l'aide d'un point fixe.
5. Donner un algorithme permettant de calculer *Reach*. On donnera une version optimisée à l'aide d'une liste de travail. On pourra utiliser un prédicat prédéfini `definit(i, v)` qui retourne vrai ssi l'instruction `i` définit la variable `v`.