Examen de majeure 2
Langages et compilation

(Durée 2h)

29 mars 1999

On attachera une importance particulière à la clarté, à la précision et la la concision de la rédaction.

1 Expansion en ligne

L'expansion en ligne (*inlining*) consiste à remplacer dans le source des appels à des fonctions par le corps de ces fonctions. On considère la fonction `carré` suivante :

```
function carré (n : integer) : integer;
begin
   carré := n * n
end;
```

1.1

Donner un code SPIM possible pour la fonction `carré` en prenant comme registres conventionnels `a0`, `a1`, `a2`, et `a3` pour les arguments et `v0` pour le résultat.

```
carre:
    mul $v0, $a0, $a0        # v0 <- n n
    j   $ra                  # Retour à l'appelant
```

Réponse :

```
carre:
    mul $v0, $a0, $a0        # v0 <- n n
    j   $ra                  # Retour à l'appelant
```

1.2

On considère la fonction `cube` suivante :

```
function cube (n : integer) : integer;
begin
   cube := carré(n) * n
end;
```

a) Donner un code SPIM possible pour cette fonction (en utilisant les mêmes conventions d'appel), sans faire d'hypothèse sur le code de la fonction `carré` (on indiquera brièvement en commentaires les opérations effectuées par chaque instruction).

Réponse :

```
   cube_f=8
 cube:
     subu $sp, $sp, cube_f      # ajustement de la pile
     sw   $ra, 0($sp)           # sauvegarde de l'adresse de retour
     sw   $a0, 4($sp)           # sauvegarde de n
     jal  carre                 # v0 <- carré(n)
     lw   $a0, 4($sp)           # a0 <- n
     mul  $v0, $a0, $v0         # v0 <- carré(n) * n
     lw   $ra, 0($sp)           # rappel de l'adresse de retour
     addu $sp, $sp, cube_f      # ajustement de la pile
     j    $ra
```

b) Donner le code source résultant de l'expansion en ligne de la fonction `carré` dans la fonction `cube`.

Réponse :

```
function cube (n : integer) : integer;
begin
  cube := (n * n) * n
end;
```

c) Donner un code SPIM pour le code source résultant de l'expansion en ligne.

Réponse :

```
cube:
    mul $v0, $a0, $a0         # v0 <- n * n
    mul $v0, $v0, $a0         # v0 <- (n * n) * n
    j   $ra                   # retour à l'appelant
```

1.3

On s'appuyera sur l'exemple ci-dessus.

a) Quel sont les différentes sources de gain de l'expansion en ligne ?

Réponse : *Le gain immédiat est l'économie des appels de fonction, donc du passage des arguments et la récupération du résultat aux emplacements conventionnels, la sauvegarde de l'adresse de retour, et deux sauts (aller et retour).*

Le gain indirect est la connaissance exacte du code du corps de la fonction carré ce qui permet de savoir qu'elle n'utilise pas les registres `ra`*,* `t0`*,* `t1`*, etc. et d'éviter ainsi des sauvegardes intermédiaires en pile. (Un meilleur placement des temporaires dans les registres peut également résulter du regoupement des ces deux fonctions et de leur compilation simultanée.)*

b) Expliquer comment une partie des gains de l'expansion en ligne, dits indirects, peuvent se retrouver simplement sans effectuer l'expansion en ligne.

Réponse : *Les gains indirects sont décrits à la question précédente.*

On peut retrouver ces gains si l'on connaît effectivement les registres utilisés par la fonction `carré` *pendant la compilation de la fonction* `cube` *(il suffit de compiler* `carré` *avant* `cube`*). On saura alors, par exemple, que les registres temporaires* `t0, t1, ...` *ne sont pas utilisés par la fonction* `carré` *et peuvent l'être par la fonction* `cube` *pour sauvegarder les valeurs intermédiaires et l'adresse de retour, ce qui évitera tous les mouvements de pile dans la fonction* `cube`*.*

Plus généralement, une analyse inter-procédurale permettrait de passer les arguments dans des registres non conventionnels.

c) Quel est le coût de l'expansion en ligne ? Quel compromis proposer ?

Réponse : *L'expansion en ligne duplique le code du corps de la fonction. De façon générale, il faut limiter l'expansion en ligne en tenant compte du rapport entre la taille du code source et celle du code résultant de l'expansion en ligne et éventuellement du gain en rapidité d'exécution.*

Pour de petites fonctions auxiliaires, comme ci-dessus, le code résultant de l'expansion est plus petit que le code d'un appel de fonction (qui inclue le code protocolaire de placement des arguments, et de récupération du résultat). On a donc simultanément un gain en espace et en rapidité d'exécution. De tels appels peuvent être systématiquement mis en ligne.

À l'opposé de grosses fonctions ne seront jamais mises en ligne, sauf peut-être si elles ne sont appelées qu'une seule fois, ou si elles sont dans à l'interieurs de boucles critiques les plus internes.

Les fonctions récursives sont un cas particulier pour lesquelles l'expansion en ligne n'est pas possible. (On peut en revanche dérouler la récursion un petit nombre de fois.)

1.4

L'expansion en ligne naïve ne fonctionne pas correctement. Considérons la fonction :

```
function quatre (n : integer) : integer;
begin
   quatre := carré (carré (n*n))
end;
```

a) Expliquer sur cet exemple un problème lié à l'expansion naïve.

Réponse : *La traduction naïve donne :*

```
function quatre (n : integer) : integer;
begin
   quatre := ((n * n) * (n * n)) * ((n * n) * (n * n))
end;
```

L'argument de `carré` *(par exemple* `(n*n)` *mais aussi* `carré (n*n)`*) est évalué deux fois (au toral* `(n*n)` *est évalué quatre fois) ce qui est inefficace.*

b) En déduire un exemple montrant que l'expansion naïve peut aussi être incorrecte.

Réponse : *Il suffit d'inclure un effet de bord dans le passage d'argument.*

```
function trace (n : integer) : integer;
begin
  write (x);
  trace := x
end;
function erronée () : integer;
begin
   erronée := carré (trace x)
end;
```

La fonction `erronée` *évalue* `trace x` *une seule fois. Après l'expansion en ligne naïve,* `trace x` *sera évalué deux fois et imprimera deux fois la valeur de* `x`*, changeant ainsi la sémantique du programme.*

c) Donner une meilleure version de l'expansion de `carré` dans la fonction `quatre` qui suggère un schéma général et correct.

Réponse : *Pour n'évaluer les arguments qu'une seule fois, on place leur valeur dans des variables auxillaires. Pour pouvoir emboîter les appels imbriqués de façon systématique, on place également le résultat de chaque appel dans des variables auxillaires.*

Pour cela, on introduit pour chaque fonction expansée en ligne autant de variables nouvelles que d'arguments, plus une variable nouvelle pour le résultat. On prend soin de renommer les variables *du corps de la fonction expansée en ligne de façon cohérente.*

```
function quatre (n : integer) : integer;
var n1 : integer; var carré1 : integer;
var n2 : integer; var carré2 : integer;
begin
    n1 := n;                (* argument de l'appel le plus interne *)
    carré1 := n1 * n1;      (* résultat du premier appel *)
    n2 := carré1;           (* argument du deuxième appel *)
    carré2 := n2 * n2;      (* argument de l'appel externe *)
    quatre := carré2
end;
```

Précisons que les variables auxilliaires ne pénalisent pas (en général) le programme, car elles seront associées à des temporaires qui pourront si besoin partager le même registre. Au pire, dans des cas pathologiques, la séquentialisation pourrait éliminer quelques opportunités de simplification.

2 Langage de parenthèses

Sur l'alphabet $\Sigma = \{g, d\}$ on considère la grammaire G_1 (resp. G_2) définie par l'axiome S_1 (resp. S_2) et les règles de production ci-dessous à gauche (resp. à droite) :

$$\begin{array}{l} S_1 \to E \\ E \to gEd \\ E \to EE \\ E \to \epsilon \end{array} \qquad\qquad \begin{array}{l} S_2 \to E \\ E \to gEdE \\ E \to \epsilon \end{array}$$

2.1

a) Donner deux dérivations de $ggdgdd$ l'une dans G_1 et l'autre dans G_2.

Réponse : *Dérivation de ggdgdd.*

$$\underline{S_1} \to \overline{\underline{E}} \to \overline{g\underline{E}d} \to g\overline{\underline{E}E}d \to \overline{gg\underline{E}d}Ed \to ggd\underline{E}d \to ggd\overline{g\underline{E}d}d \to ggdgdd$$

$$\underline{S_2} \to \overline{\underline{E}} \to \overline{gEd\underline{E}} \to g\underline{E}d \to \overline{gg\underline{E}dE}d \to ggd\underline{E}d \to ggd\overline{g\underline{E}dE}d \to ggdgd\underline{E}d \to ggdgdd$$

b) Les grammaires G_1 et G_2 sont-elles ambigües ? LL(1) ? LR(1) ?

Réponse : *La grammaire G_1 est ambigüe car, par exemple, on peut reconnaître le mot vide par des dérivations appartenant à des arbres syntaxiques (ou arbres de dérivations) différents :*

$$S_1 \to E \to \left\{ \begin{array}{l} E \to \epsilon \\ E \to \epsilon \end{array} \right. \qquad\qquad S_1 \to E \to \epsilon$$

Donc elle n'est pas LR(1) ni a fortiori LL(1).

La grammaire G_2 est LL(1). Pour le vérifier, nous calculons pour chaque règle $\alpha \longrightarrow \beta$ les ensembles $first(\beta)$, $follow(\alpha)$, et nous vérifions si $\alpha \longrightarrow^* \epsilon$; finalement nous construisons la relation de transition de domain $V \times \Sigma$. Les calculs sont résumés dans la table suivante :

α	β	$first(\beta)$	$follow(\alpha)$	$\alpha \longrightarrow^* \epsilon$	g	d
S	E	g	g	yes	$\surd$	
E	$gEdE$	g	d	yes	$\surd$	
	ϵ					$\surd$

La fonction de transition donnée par les $\surd$ dans la sous-matrice en bas à droite ne comporte qu'un élément par case ; elle définie donc bien une fonction. La grammaire est donc LL(1). Par conséquence, elle est aussi LR(1) et n'est pas ambigüe.

2.2

La grammaire G_1 définit le langage des parenthèses L. Peut-on reconnaître L par une expression régulière ? par un lexeur ? (Justifier très brièvement chaque réponse).

Réponse : *Le langage des parenthèses L ne peut pas être reconnu par une expression régulière. Sinon, il serait reconnaissable par un automate fini. Or, les ensembles $L_n = \{w \mid g^n w \in L\}$ des suffixes des mots de L commençants par g^n sont deux à deux distincts (au moins un mot est dans l'un mais pas dans l'autre). Il faudrait donc un nombre d'états arbitraires à un automate pour disntinguer l'ensemble des configurations possibles au cours de la reconnaissance d'une expression.*

L peut être reconnu par un lexeur mais seulement en utilisant la puissance du langage hôte (qui est un complet en général) dans les actions du lexeur pour compter les parenthèses ouvertes.

2.3

Montrer que le langage reconnu par G_2 est un sous ensemble de L.

Réponse : *La règle $E \to gEdE$ est une règle dérivée dans G_1 par $E \to \underline{E}E \to \overline{gEd}E$. On ne change pas le langage reconnu par G_1 en lui ajoutant cette règle dérivée (on appelle G'_1 la grammaire résultante). La grammaire G_2 peut être plongée de façon homomorhe dans la grammaire G'_1. En effet, au renommage des non-terminaux près, ses règles sont incluses dans celles de G'_1. Une dérivation de G_2 est donc aussi une dérivation de G'_1 au renommage des non-terminaux près. Ainsi tout mot reconnu par G_2 est aussi reconnu par G'_1 et appartient donc à L.*

2.4

On peut représenter les arbres syntaxiques (ou arbres de dérivations) de la grammaire G_1 par un type concret `Ocaml` :

```
type g1 = G1_S of e1
and e1 =
    G1_gEd of e1
  | G1_EE of e1 * e1
  | G1_0;;
```

a) Donner en `Ocaml` une définition de type `g2` représentant les arbres syntaxiques de la grammaire G_2.

Réponse :

```
type g2 = G2_S of e2
and e2 =
  | G2_gEdE of e2 * e2
  | G2_0;;
```

b) Donner en `Ocaml` l'arbre syntaxique `a2` correspondant à la dérivation de *ggdgdd* dans G_2 que vous avez donnée ci-dessus.

Réponse :

```
let a2 = G2_S (G2_gEdE (G2_gEdE (G2_0, G2_gEdE (G2_0, G2_0)), G2_0));;
```

c) Écrire en `Ocaml` une fonction `mot1` qui prend un arbre syntaxique de type `g1` et qui retourne le mot (chaîne de caractères) du langage L lui correspondant.

Réponse :

```
let rec lemot1 = function
  | G1_gEd e -> "g" ^ (lemot1 e) ^ "d"
  | G1_EE (e, e') -> (lemot1 e) ^ (lemot1 e')
  | G1_0 -> ""
let mot1 (G1_S e) = lemot1 e;;
```

d) Écrire en `Ocaml` une fonction totale `réécrit` qui transforme un arbre syntaxique `a1` de type `g1` en un arbre syntaxique `a2` de type `g2` tel que `a2` représente le même mot que `a1` (la preuve de correction n'est pas demandée dans cette question).

Réponse :

```
let rec réécrit_le  = function
    G1_EE (G1_EE (e, e'), e") -> réécrit_le (G1_EE (e, G1_EE (e', e")))
  | G1_EE (G1_gEd e, e') -> G2_gEdE (réécrit_le e, réécrit_le e')
  | G1_EE (G1_0, e) -> réécrit_le e
  | G1_gEd e -> G2_gEdE (réécrit_le e, G2_0)
  | G1_0 -> G2_0;;
let réécrit (G1_S e) = G2_S (réécrit_le e);;
```

e) Justifier la terminaison de ce programme.

Réponse : *La fonction* `réécrit` *est récursive. Cependant l'argument de chaque appel récursif est un sous-arbre strict de l'argument de l'appel principal, sauf dans le premier cas du filtrage : l'argument de l'appel principal est alors un nœud de la forme* `G1_EE(u,v)` *et l'argument de l'appel récursif est un nœud de même forme* `G1_EE(u',v')` *tel que* `u'` *soit un sous-arbre strict de* `u`*. On ne peut donc passer par le premier cas de filtrage qu'un nombre fini de fois ; ensuite l'argument de l'appel récursif ne peut que décroître strictement.*

f) On admet la correction de la fonction `réécrit`. En déduire que G_2 reconnaît tout L.

Réponse : *La correction assure que la fonction* `mot1` *est égale à la composée de* `mot2` *(définie de façon analogue à* `mot1`*) et de* `réécrit`*. Tout mot w de L peut être reconnu par G_1 (par définition de L). Il existe donc un arbre syntaxique a_1 dont w est l'image par* `mot1`*. L'image de a_1 par la fonction* `réécrit` *est un arbre a_2 de la grammaire G_2 dont l'image par* `mot2` *est le même mot w. Le mot w est donc aussi reconnu par G_2.*