Examen de majeure 2
Compilation

(Durée 2h)

23 mars 2001

On attachera une importance particulière à la clarté, à la précision, mais aussi à la ***concision*** *de la rédaction. Les trois exercices sont indépendants. La plupart des questions sont également indépendentes dans les exercices 1 et 2.*

1 Interruptions de boucles

Nous ajoutons à Pseudo-Pascal une construction **break** qui peut être utilisée à l'intérieur d'une boucle **while** comme dans le langage `C` : l'exécution de l'instruction **break** à l'intérieur d'une boucle **while** interrompt l'exécution de cette boucle et continue avec l'instruction qui suit immédiatement la boucle (c'est une erreur si l'opération **break** apparaît en dehors d'une boucle **while**).

1.1 Exemple d'utilisation de la construction **break**.

a) Donner une fonction ou une procédure Pseudo-Pascal qui utilise l'instruction **break**.

b) Donner un programme équivalent (qui effectue le même calcul) sans utiliser l'instruction **break**.

1.2 On ajoute à l'arbre de syntaxe abstraire de Pseudo-Pascal (type `Pp.instruction` dans le cours) un nœud `Break`. On donne le squelette d'un interprète du langage Pseudo-Pascal définit par deux fonctions récursives :

```
val instruction : environnement -> Pp.instruction -> unit
val expression : environnement -> Pp.expression -> valeur
```

dont on rappelle la structure ci-dessous :

```
let rec instruction env = function
  | While (e, i1) ->
      while bool (expression env e)
      do instruction env i done
  | ...

and expression env = function
    ...
```

où `bool : valeur -> bool` convertit son argument en un booléen si c'est possible et lance un erreur d'exécution sinon.

Décrire une modification très simple du cas `While` et l'ajout du cas `Break` dans la fonction `instruction` afin de traiter la construction **break**.

1.3 Décrire la compilation en code intermédiaire des instructions `While` et `Break` (et d'autres modifications éventuelles à apporter à la fonction de compilation). On utilisera de préférence une description mathématique à l'aide des fonctions $[\![\ ]\!]^e{}_\rho$ et $[\![\ ]\!]^s{}_\rho$ utilisés dans le cours et que l'on pourra raffiner.

2 Exercice sur la canonisation

Un des buts principaux de la canonisation du code intermédiaire est d'extraire les expressions `Call` emboîtées, pour n'avoir au plus qu'une seule expression `Call` dans une instruction et, le cas échéant, figurant en tête. La canonisation d'une expression retourne une séquence d'instructions extraite et une expression résiduelle. La canonisation d'une instruction retourne une séquence d'instructions.

2.1 Dans la canonisation du code intermédiaire, il est parfois utile de savoir si une expression commute avec une instruction.

a) Donner un exemple de canonisation du code intermédiaire qui utilise la commutation.

b) Donner une expression et une instruction qui ne commutent pas.

c) Que doit-on faire dans ce cas là ?

d) Expliquer pourquoi le code produit peut être moins bon lorsqu'il n'y a pas commutation.

2.2 Dans le cours nous avons choisi un prédicat de commutation très simple, en disant qu'une expression constante commute avec toute autre instruction. Nous considérons ici une définition plus fine : une expression commute avec une instruction si, à la fois,

– les temporaires lus par l'expression ne sont pas écrits par l'instruction, et

– soit l'expression ne lit pas en mémoire, soit l'instruction n'écrit pas en mémoire.

On rappelle le code intermédiaire :

```
type exp =                          and stm =
    Const of int                    | Label of label
  | Name of label                   | Move_temp of temp * exp
  | Temp of temp                    | Move_mem of exp * exp
  | Mem of exp                      | Exp of exp
  | Bin of binop * exp * exp        | Seq of stm list
  | Call of frame * exp list        | Jump of exp
                                    | Cjump of relop * exp * exp
                                        * label * label
```

a) Écrire en Ocaml une fonction `lus_par_exp` de type `exp -> temp list` qui prend une expression canonique (sans `Call`) du code intermédiaire et retourne la liste des temporaires lus par celle-ci (on admet les répétitions dans le résultat).

b) Écrire en Ocaml une fonction `écrits_par_instr` de type `stm -> temp list` qui prend une instruction canonique (sans saut, sans séquence et sans `Call` emboîtés) du code intermédiaire et qui retourne la liste des temporaires écrits par celle-ci.

c) Comment peut-on prendre en compte simplement l'écriture et la lecture en mémoire dans les fonctions `lus_par_exp` et `écrits_par_stm` ?

2.3 Traiter la mémoire comme un seul bloc est très grossier. En particulier, une lecture en mémoire commute avec une écriture en mémoire lorsque ces deux adresses mémoire sont distinctes.

a) Pourquoi, dans le langage Pascal, peut-on facilement considérer la mémoire comme une partition de zones disjointes ? (Donner quelques exemples d'opérations en mémoire qui peuvent être considérées comme opérant dans des zones différentes)
b) Pourquoi est-ce plus difficile en C ?
c) Est-ce encore possible en Ocaml ?

3 Allocation de registres dans les processeurs du type CISC

Le processeur MIPS, utilisé en cours, est du type RISC. En particulier, les registres sont nombreux et (presque) tous équivalents. Cependant, de vieux processeurs tels que le 68000 distinguent les registres d'adresse (`A0`, ..., `A7`) des registres de donnée (`D0`, ..., `D7`). Pour simplifier, nous considérons un processeur expérimental *CIRC* dont le jeu d'instructions est identique à celui du code SPIM mais dont les registres sont séparés en registres d'adresses (`A0`, ..., `A7`) et registres de données (`D0`, ..., `D7`) et dont certaines occurrences des registres dans les instructions sont contraintes. En particulier, on considère les contraintes suivantes,

- Les opérations arithmétiques (**add**, **mul**, etc.) n'utilisent que des registres de donnée.
- Dans les écritures et lectures en mémoire delà forme **lw** $D, n(A)$ et **sw** $D, n(A)$ le registre D est un registre de donnée, le registre A est un registre d'adresse (et n un entier).
- Les autres instructions sont non contraintes, en particulier **move** R, S peut indépendemment utiliser des registres d'adresses ou de données pour R et S.

On appelle *code machine abstrait* le code de type `Ass.code` dans le cours, abstrait par rapport au choix final des registres et *code machine final* le code émit à la fin de la compilation et pouvant être lu par l'assembleur. Dans l'une ou l'autre expression, on remplace *machine* par SPIM ou CIRC pour préciser la version du cours pour l'architecture SPIM ou la version de cet exercice pour une architecture CIRC.

3.1 Expliquer en quoi l'architecture CIRC pose, a priori, un problème pour l'allocation de registres.

3.2 À la recherche d'une solution.
a) Expliquer succinctement comment on peut exprimer ces contraintes dans le graphe d'interférence.
b) Proposer une modification légère du code SPIM abstrait en un code CIRC abstrait permettant de mettre simplement en œuvre la contrainte ci-dessus ? (Préciser si besoin, informellement et succinctement, quelle autre partie de la chaîne de compilation il faudra modifier)
c) On considère le code intermédiaire `Bin (Plus, Temp t1, Temp t2)`. Décrire le code CIRC abstrait produit pour ce code On ne se préoccupera pas ici du contexte dans lequel l'instruction est placée. (On pourra supposer que les variables `rA` et `rD` sont liées à l'ensemble des registres d'adresses et de données respectivement.)

3.3 À la poursuite de la solution.
a) Donner la traduction du code intermédiaire suivant en code SPIM abstrait (on suppose que le nom associé à l'étiquette `debut` est *"debut"*) :

```
[ Label debut;
  Move_mem (Temp t1, Temp t2);
  Move_temp (t1, Bin (Plus (Temp t1, Temp t3)));
  Jump (debut); ]
```

b) Donner un code SPIM final possible.

c) Quel est le problème lorsqu'on essaye de générer du code pour l'architecture CIRC ? Donner un code CIRC final équivalent au code SPIM final de la question b.

d) Quel code CIRC abstrait aurait permis d'obtenir le code CIRC final de la question c ?

3.4 La démarche de la question 3.3 revient, sur un cas particulier, à une effectuer une transformation du code CIRC de façon à rendre les contraintes solvables.

a) En s'inspirant d'une transformation vue en cours, proposer une mécanisation de cette transformation.

c) Expliquer pourquoi le code résultant sera plus efficace qu'il n'y paraît.

d) Avez-vous un exemple typique qui sera moins efficace que ce que l'on pourrait obtenir manuellement ?