Informatique

Cours de compilation INF 553

24 mars 2004

Les deux premières parties de cet examen traitent de l'ajout de diverses constructions au langage accepté par le compilateur `zyva` du cours. Les deux parties du problème sont indépendantes et, au sein de chaque partie, les questions sont de difficulté croissante. Une troisième partie propose quelques questions indépendantes de toutes les autres.

Première addition, le « if » sans « else »

Dans l'instruction conditionnelle, **if** *expression* **then** *instruction* **else** *instruction*, il est pratique de rendre optionnelle la partie **else** *instruction*. Cela permet par exemple d'écrire la fonction valeur absolue ainsi :

```
function abs(x : integer) : integer ;
begin
    if x < 0 then
      x := -x ;
    abs := x
end ;
```

Question 1. On modifie l'analyseur syntaxique de `zyva` en insérant la ligne

```
| IF expression THEN instruction {...}
```

entre les lignes 81 et 82 du fichier `parser.mly` (donné en annexe). Donner l'action de la nouvelle règle de l'analyseur (les ... ci-dessus). Pour le type des arbres de syntaxe abstraite, voir le fichier `pp.mli` (donné en annexe).

Question 2. Voici un compte-rendu de la compilation de l'analyseur modifié :

```
# ocamlyacc parser.mly
1 shift/reduce conflict.
```

Et voici l'extrait pertinent du fichier de diagnostics de `ocamlyacc` :

```
124: shift/reduce conflict (shift 138, reduce 22) on ELSE
state 124
instruction : IF expression THEN instruction . ELSE instruction  (21)
instruction : IF expression THEN instruction .  (22)

ELSE  shift 138
SEMI  reduce 22
END  reduce 22
```

a) Démontrer que la nouvelle grammaire est ambiguë en exhibant une instruction pouvant être interprétée de deux façons.

b) Expliciter l'interprétation effectuée par l'analyseur shift/reduce produit par `ocamlyacc` (sachant qu'entre shift et reduce, `ocamlyacc` choisit shift).

Question 3. Réécrire la grammaire de sorte qu'elle ne soit plus ambiguë et produise les mêmes arbres de syntaxe abstraite que l'analyseur `ocamlyacc` lors de la question précédente. Seules sont demandées les règles de la nouvelle grammaire, on ne donnera pas les actions.

Indication : On peut lever l'ambiguïté en divisant le non-terminal *instruction* en deux non-terminaux.

```
instruction:
  fermé
| ouvert
;
```

Les nouveaux non-terminaux sont tels que le non-terminal *fermé* ne produit jamais de mot se terminant par **THEN** *instruction*, et que le non-terminal *ouvert* produit ces mots.

Seconde addition, l'instruction « return »

On se propose maintenant d'effectuer les retours de fonction et de procédure à l'aide de l'instruction **return**, comme en Java ou en C.

L'instruction **return** permet par exemple une écriture assez naturelle de la fonction suivante de vérification que tous les éléments d'un tableau sont non-nuls.

```
function tousNonNuls (t:array of integer ; n:integer):boolean ;
var i:integer ;
begin
  i := 0 ;
  while i < n do begin
    if t[i] = 0 then
      return false
    else
      i := i+1
  end ;
  return true
end ;
```

La sémantique informelle de l'instruction « **return** » est la suivante.

> *Une fonction transmet son résultat au programme appelant par l'instruction « **return*** expression ». *L'execution de cette instruction entraîne l'évaluation de l'expression, puis le retour au programme appelant.*
>
> *Une procédure retourne au programme appelant par l'instruction « **return** » (sans argument). Aucune valeur n'est transmise.*

On notera donc que la fonction `tousNonNuls` ne parcourt le tableau `t` que jusqu'à trouver son premier élément nul.

Pour rendre compte de cette nouvelle instruction, la définition des arbres de syntaxe abstraite est étendue par l'ajout de la ligne suivante à la fin du fichier `pp.mli`.

```
| Return of expression option
```

Note : Dans toute cette partie on suppose que les programmes pseudo-pascal sont bien typés, en particulier le corps d'une fonction ne contient pas « **return** » sans argument et le corps d'une procédure ne contient pas « **return** *expression* ».

Question 4. Modifier les analyseurs lexical et syntaxique de `zyva` (fichiers `lexer.mll` et `parser.mly`, donnés en annexe) pour intégrer cette nouvelle instruction.

Question 5. L'existence d'une instruction explicite pour retourner de la fonction en cours met en évidence une erreur de programmation : la fin du corps de la fonction est atteinte sans qu'aucune valeur ne soit renvoyée. Une telle erreur est par exemple possible si on modifie la fonction `tousNonNuls` en supprimant le ligne numéro 11.

Écrire (en Caml) une fonction `retourneToujours` qui prend en argument le corps d'une fonction et qui renvoie un booléen. Le booléen vaudra `true` si l'on a pu détecter que l'exécution du corps se termine nécessairement par **return** *expression* (quand elle termine), et `false` autrement.

```
val retourneToujours : Pp.instruction -> bool
```

Attention : Une fonction `retourneToujours` qui répond toujours `false` est conforme à la spécification demandée. Toutefois, il est clair qu'une telle fonction ne fait aucun effort et ne constitue pas une réponse valable.

Note : Ceux qui ne se sentent pas à l'aise en Caml peuvent répondre autrement, en définissant (par induction) un prédicat sur les instructions.

Question 6. Modifier la compilation vers le code intermédiaire pour qu'elle traite la nouvelle instruction **return**.

On modifiera la seule fonction `cinstruction` en supposant qu'elle est maintenant de la forme :

```
(*
  Le nouveau type de cinstruction est :
  (access, Frame.frame) Env.environment -> Frame.frame -> Pp.instruction -> Code.stm
*)
let rec cinstruction env f i = match i with ...
```

Où `f` est le frame de la fonction (ou procédure) en cours de compilation.

Indications : Vous disposez en annexe de la définition du code intermédiaire `code.mli`, du source `trans.ml` du compilateur vers le code intermédiaire, et de l'interface `frame.mli` du module `Frame`, dont voici un extrait utile.

```
val frame_result : frame -> temp option
   (* Renvoie le temporaire choisi pour retourner
   le résultat d'une vraie fonction, None pour une procédure *)
val frame_return : frame -> label
   (* Renvoie l'étiquette choisie pour l'épilogue (fin de la sous-routine) *)
```

Question 7. Cette question aborde deux points complémentaires de la question 5. Les réponses peuvent être relativement brèves (mais précises).

a) Cela a-t-il un sens de procéder à une analyse du corps des procédure, comme la question 5 procédait à une analyse du corps des fonctions ?

b) Indiquer comment résoudre la question 5 à l'aide d'une analyse de durée de vie pratiquée sur le code intermédiaire.

Question 8. Soit f une fonction quelconque de pseudo-pascal. Dans le corps de f, l'instruction « **return** *f(...)* » est un appel récursif terminal, dont la compilation peut être largement optimisée. Grossièrement, l'optimisation consiste à remplacer l'appel terminal par un saut vers une cible bien choisie dans le corps de f.

a) Identifier précisément la « *cible bien choisie* ». On supposera par la suite que l'étiquette définissant cette cible est en place dans le code intermédiaire (de la fonction en cours de compilation) et accessible à l'aide d'une nouvelle fonction `frame_bien_choisie` du module `Frame`.

b) Donner un code intermédiaire que peut produire la compilation optimisant l'appel récursif terminal de cette fonction pseudo-pascal.

```
function fact(n,r:integer):integer ;
begin
    if n = 0 then
      return r
    else
      return fact(n-1, n*r)
end ;
```

ATTENTION : Il faut respecter la sémantique : le programme optimisé doit produire le même résultat que le programme non-optimisé.

NOTE : Les temporaires contenant les arguments seront notés t_n et t_r, le temporaire contenant le résultat sera noté t_f. Vous pouvez donner le code intermédiaire dans le style adopté dans la section 7.3 du poly.

c) Modifier la fonction `cinstruction` de la question 6 pour qu'elle procède à l'optimisation des appels récursifs terminaux.

Bonus

Cette partie pose quelques questions assez diverses. En y répondant brièvement, vous pouvez faire état de votre compréhension du sujet de la compilation.

Question 9. En dehors des bienfaits que le contrôle de type apporte au programmeur, expliquer pourquoi les compilateurs comportent presque toujours une phase de contrôle des types.

Question 10. Dans le cas d'une compilation vers le MIPS, quel nombre *minimum* de registres peut-on autoriser l'allocation de registres à employer ? Quels registres choisiriez vous et quelles conventions d'appel (la cible reste le MIPS), afin de mimimiser les modifications à apporter à `zyva` ?

Question 11. Expliquer l'intérêt (marginal) qu'il peut y avoir à rendre, lors de la selection, l'instruction du code intermédiaire :

$$\texttt{Move\_temp}\,(t_1, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Temp}\ t_2, \texttt{Const}\ 0))$$

par l'instruction MIPS `move` t_1, t_2 au lieu de l'instruction `add` t_1, t_2, `0`.

Corrigé du contrôle INF 553

Question 1. Il n'y a pas besoin de modifier le type des instructions de l'arbre de syntaxe abstraire, car on dispose déjà d'une instruction qui ne fait rien (la séquence vide).

`| IF expression THEN instruction` { *If ($2, $4, Sequence [])* }

Question 2.

a) En notant E une expression et I une instruction, l'instruction :

if E_1 **then** **if** E_2 **then** I_1 **else** I_2

Peut être interprétée des deux façons suivantes :

1. Le **else** se rapporte au **if** le plus externe,
2. ou au **if** le plus interne.

Plus précisément, l'indentation permet d'exprimer les deux interprétations possibles.

if E_1 **then**
  begin **if** E_2 **then** I_1 **end**
else
  I_2

Ou encore :

if E_1 **then**
  begin **if** E_2 **then** I_1 **else** I_2 **end**

On peut noter que la seconde interprétation est l'interprétation usuelle. Le **else** se rapporte au **if** englobant le plus proche.

b) Pour ce qui est de l'analyseur LALR, **ocamlyacc** « *résout* » les conflits shift/reduce en donnant la priorité à shift. L'analyseur produit adoptera donc la seconde interprétation : ayant reconnu **if** E_2 **then** I_1 et confronté à **else**, il choisi d'empiler ce dernier lexème. Ce qui conduira, après reconnaissance de I_2, à appliquer d'abord la production complète de la conditionnelle, puis la production tronquée.

Question 3. Ce n'est pas si facile qu'il y paraît mais on peut trouver en suivant strictement l'indication.

```
instruction:
  ferm^^e9
| ouvert
;

ferm^^e9:
  IF expression THEN ferm^^e9 ELSE ferm^^e9
| WHILE expression DO ferm^^e9
/* R^^e8gles reprises de la d^^e9finition de instruction du parser.mly original */
| IDENT COLONEQUAL expression
| BEGIN bloc END
| READ LPAREN IDENT RPAREN
| WRITE LPAREN expression RPAREN
| WRITELN LPAREN expression RPAREN
| array_expression LBRACKET expression RBRACKET  COLONEQUAL expression
;

ouvert:
  IF expression THEN instruction
```

```
| IF expression THEN ferm^^e9 ELSE ouvert
| WHILE expression DO ouvert
;
```

Question 4. Modifions d'abord l'analyseur syntaxique. Après la ligne 16, ajout d'un lexème.

```
%token RETURN
```

Puis, ajout de deux règles à la définition des instructions (disons après la ligne 87).

```
| RETURN { Return None }
| RETURN expression { Return (Some $2) }
```

En ce qui concerne l'analyseur lexical, on peut se contenter d'ajouter un mot-clé à la table de hachage `keywords` (par exemple après la ligne 10).

```
add_keyword "return" RETURN ;
```

Question 5. Le code suivant exploite l'hypothèse de bon typage, en évitant de vérifier si `Return` a bien un argument `Some` *e*.

```
let rec retourneToujours i = match i with
| Return _ -> true
| Sequence (i::rem) ->
    retourneToujours i || retourneToujours (Sequence rem)
| Sequence [] -> false
| If (_, inst, insf) ->
    retourneToujours inst && retourneToujours insf
| _ -> false
```

Question 6. Dans ma nouvelle définition de la fonction `cinstruction`, je n'ai indiqué que les cas du filtrage de l'instruction `i` qui changent :

```
let rec cinstruction env f i = match i with
| Sequence is ->
(* Comme avant, sauf qu'il faut ajouter l'argument f apr^^e8s env *)
    Seq (List.map (cinstruction env f) is)
| If (e, inst, insf) -> (* Idem *)
| While (e, i) ->       (* Idem *)
| Procedure_call ... | Seti ... (* Pas de changement *)
(* Seuls cas r^^e9ellement nouveaux *)
| Return None -> Jump (Frame.frame_return f)
| Return (Some e) ->
    let ce = cexpr env e in
    let tr =
      match Frame.frame_result f with
      | Some t -> t
      | None -> assert false in
    Seq [Move_temp (tr, ce) ; Jump (Frame.frame_return f)]
```

Question 7.

a) Non, cette vérification n'aurait pas de sens car il n'y à ici pas de valeur de retour à transmettre. De fait, si le contrôle atteint la fin du corps d'une procédure, l'épilogue sera exécuté. L'un dans l'autre, on peut expliquer ce comportement, en imaginant qu'il y a une instruction **return** sans argument à la fin du corps.

b) Par définitions même du temporaire vivant, tous les chemins de l'entrée du prologue à la sortie de l'épilogue stipulent une valeur de retour si et seulement si le temporaire choisi pour la valeur de retour n'est pas vivant à l'entrée de la fonction, car l'instruction « **return** *expression* » écrit dans ce temporaire (et seulement elle).

Question 8.
a) La cible bien choisie est l'entrée du corps la fonction. C'est-à-dire, non pas le prologue mais la première instruction qui suit le prologue. On peut parler de point d'entrée « *interne* ».
b) Par exemple, l'étiquette bien choisie étant `Lenter`.

```
Lenter:
  Cjump L15 L16 (= t_n 0)
L16:
  (set t'_n (- t_n 1))
  (set t'_r (* t_n t_r))
  (set t_n t'_n)
  (set t_r t'_r)
  Jump Lenter
L15:
  (set t_f t_r)
```

Où `(set` t e`)` est l'instruction `Move_temp`. On note l'emploi de temporaires additionnels t'_n et t'_r.
c) Toute la difficulté consiste à bien réaliser le passage des arguments. Dans le filtrage de la fonction `cinstruction` on ajoute cette clause *avant* la clause traitant le motif `Return (Some e)`.

```
| Return (Some (Function_call (g, args)))
    when Env.find_definition env g == f ->
      let ts = List.map (fun _ -> Gen.new_temp ()) args in
      let cargs =
        List.map2 (fun t e -> Move_temp (t, cexpr env e)) ts args
      and cmove_params =
        List.map2 (fun at t -> Move_temp (at, Temp t))
            (Frame.frame_args f) ts in
      Seq [Seq cargs ; Seq cmove_params ; Jump (Frame.frame_bien_choisie f)]
```

L'usage des temporaires `ts` permet de calculer les nouvelles valeurs des arguments à l'aide des anciennes, sans eux, le code généré pourrait être incorrect, si par exemple le second argument de l'appel terminal fait référence à la valeur du premier argument. Comme dans l'exemple donné.

Question 9. Pour `zyva`, les valeurs manipulés à l'exécution sont toutes de même taille (un mot machine). Si tel est pas le cas le compilateur a besoin de la taille de la valeur d'une expression arbitraire, en particulier si celle ci-depasse le mot machine, pour par exemple la passer en pile. On obtient facilement cette taille à partir du type en supposant bien entendu que toutes le valeurs d'un type sont de la même taillle. Par ailleurs la connaissance des types est parfois nécessaire pour emettre le code d'une opération (en C l'addition « `+` » peut être l'addition flottante ou entière). Que l'on pense par exemple à des valeurs de type `double` en C vers le Pentium (qui doivent aller dans des registres spécialisés et occupent deux mots mémoire).

Question 10. Deux registres suffisent, car deux registres suffisent pour executer une instruction trois adresses (poly, section 9.1). On peut choisir n'importe quels registres, mais pour minimiser les modifications à `zyva` il convient sans doute de choisir `$ra` (pour conserver le traitement actuel de l'adresse de retour de sous-routine) et le registre de retour `$v0` (pour conserver le traitement actuel de la valeur de retour des fonctions). On peut alors se contenter de modifier le code des primitives à un argument.

Question 11. L'instruction `move` t_1`,` t_2 n'engendre pas nécessairement d'inférence entre t_1 et t_2 (si t_2 est vivant en sortie) et pourra être supprimée si les temporaires t_1 et t_2 se retrouvent au final dans le même registre machine. Ce n'est pas le cas de l'instruction `add` t_1`,` t_2`, 0`.