Correction du Partiel de Synthèse de Compilation

EPITA – Promo 2005 – **Tous documents autorisés** *

Septembre 2003

Une copie concise, bien orthographiée, dont les résultats sont clairement exposés, sera toujours mieux notée qu'une copie qui aura demandé une quelconque forme d'effort de la part du correcteur.

Certaines questions suggèrent volontairement une mauvaise réponse. Dans ce cas il vous faut expliquer brièvement la bonne façon de résoudre le problème évoqué par la question.

1 Partie frontale

1. Définir la partie frontale d'un compilateur.

   Correction: *Front-end :* tout ce qui dépend du langage source.

2. On cherche à ignorer les commentaires Tiger dans notre reconnaisseur lexical engendré par Flex. Montrer l'expression régulière qui permet de le faire.

   Correction: Les commentaires Tiger étant imbriqués, il ne peut pas exister d'expression régulière pour en rendre compte. Le plus simple est de faire appel aux *start conditions* (qui permettent de créer des "sous-scanners") de Lex/Flex, et de compter le nombre de commentaires ouverts.

3. Montrer le code Bison, **avec les actions écrites en C++ avec STL**, permettant d'analyser une liste de zéro ou plusieurs `exp` séparées par des points-virgules. Bien entendu, on ne détaillera pas la définition de `exp` lui-même.

   Correction: Il est nécessaire d'introduire un non terminal supplémentaire, disons `some_exps` :

```
exps:
  /* nothing. */  { $$ = new std::list<exp> (); }
| some_exps       { $$ = $1;                    }
;

some_exps:
  exp               { $$ = new std::list<exp> (); $$->push_back ($1); }
| some_exps ';' exp { $$ = $1; $$->push_back ($3);                    }
;
```

   Sinon, le langage est nécessairement différent est reconnaît probablement des choses telles que "`, exp`", ou "`exp,`", ou bien n'accepte pas "".
   On aura pris soin de rendre la règle récursive à gauche, pour des questions de performances.

4. Qu'est-ce qu'un AST ?

*"Tout document autorisé" signifie que notes de cours, livres, annales, etc. sont explicitement consultables pendant l'épreuve. Le zèle de la part des surveillants n'a pas lieu d'être, mais dans un tel cas me contacter au 01 53 14 59 42.

5. On désire optimiser statiquement, au niveau Tiger, des programmes tels que

   ```
   if 0 = 1 - 2 / 2 then print ("foo\n") else print ("bar\n")
   ```

   Expliquer comment vous allez faire dans l'architecture `tc` Illustrer le fonctionnement de votre méthode en prenant appui sur l'exemple.

   Correction: Bien entendu il s'agira d'une tache supplémentaire. Bien entendu, elle dépend de celle du contrôle de type, de façon que d'une part on ait bel et bien un AST sur lequel travailler, mais aussi que celui-ci soit sain.
   Cette tâche sera implémentée sous la forme d'un visiteur. Puisque manifestement seule la hiérarchie `Exp` de l'AST est concernée par cette transformation, on tirera partie du `DefaultVisitor`.
   Chaque méthode `visit` commence par visiter ses fils avant de travailler sur le noeud lui-même (on parle de *bottom-up*, ou post-fixe, soit la traversée archétypale). Pour certaines des `visit`, le comportement quand certains fils sont des constantes change. Par exemple pour les `IfExp`, selon que le test est positif ou non, le noeud courant doit être remplacé par la partie `then` ou `else`.
   "Doit être remplacé par"...

6. Quel problème significatif la question précédente rencontre avec l'architecture courante de la partie frontale de `tc`.

   Correction: Remplacer un noeud de l'`ast` par un nouveau noeud n'est pas faisable simplement : c'est très simple au niveau du parent, mais impossible au niveau du noeud lui-même (puis que cela reviendrait à vouloir changer la nature de `this` !).

7. Proposer des solutions.

   Correction: Une solution consisterait à équiper nos AST pour que ce soit possible, à l'image de `Tree` dans lequel nous avons `replace_by`. Une autre consisterait à faire en sorte de chaque fois qu'un père visite un fils, il récupère dans un champ du visiteur la nouvelle valeur de ce fils (exactement au même titre que le `TypeVisitor` récupère le type dans `_type`, et le `TranslateVisitor` récupère la traduction dans `_exp`). Il lui reste à mettre à jour ce fils.
   Enfin, on peut imaginer une version qui évite les effets des bords : le visiteur *produit un nouvel* AST. On introduirait dans la hiérarchie des visiteurs une nouvelle racine, disons `CloneVisitor` qui ne fait que dupliquer. Puis, notre visiteur propagateur de constante ne ferait que spécialiser les certaines `visit` de ce visiteur.

2 Partie centrale

1. Définir la partie centrale d'un compilateur.

   Correction: *Middle-end :* tout ce qui ne dépend ni du langage source, ni du langage cible. En pratique, il est possible qu'il reste des "call-back" ou d'autres formes de paramétrisation en fonction du langage source et/ou cible (par exemple `$v0` dans `tc`), mais le code est essentiellement générique.

2. Pour les expressions Tree suivantes, montrer ce que les règles de canonisation doivent produire. Détailler les cas où des résultats différents pourraient être obtenus selon des conditions de commutabilité.

   `move (temp t, eseq (s, e))`

   Correction: `seq (s, move (temp t, e))` dans tous les cas.

3. `sxp (eseq (s, e))`

Correction: `seq (s, sxp (e))` dans tous les cas.

4. `sxp (call (f, e1, e2, eseq (s, e3)))`

Correction: Si `s` commute avec `e1` et `e2`, alors

```
seq (s,
     sxp (call (f, e1, e2, e3)))
```

Si `s` commute avec `e2` mais pas `e1`, alors

```
seq (move (t1, e1),
     s,
     sxp (call (f, t1, e2, e3)))
```

Sinon

```
seq (move (t1, e1),
     move (t2, e2),
     s,
     sxp (call (f, t1, t2, e3)))
```

On pourrait être tenté dans le cas où `s` commute avec `e1` mais pas `e2`, de produire :

```
seq (move (t2, e2),
     s,
     sxp (call (f, e1, t2, e3)))
```

mais dans ce cas, remarquez que vous avez échangé l'ordre d'évaluation entre `e1` et `e2` : la moindre des choses est de s'assurer que c'est valide.

3 Partie terminale

Dans cette section, on considère une machine hypothétique, dont le jeu d'instruction est suffisamment clair pour ne pas avoir à être défini, et qui possède trois registres, `r1`, `r2`, `r3`. Les deux premiers sont sous la responsabilité de l'appelant, et le dernier sous celle de l'appelé.

1. Définir la partie terminale d'un compilateur.

   Correction: *Back-end :* tout ce qui est dédié à la cible.

2. "Désassembler" le programme suivant, avant allocation des registres, en un programme C ou Tiger :

```
enter: n  <- r1
       c  <- r3
       r  <- 1
loop:  if n <= 1 jump exit
       r  <- r * n
       n  <- n - 1
       goto loop
exit:  r1 <- r
       r3 <- c
       return
```

   Correction: Ce programme est la traduction d'un fact exprimé comme suit :

```
int fact (int n)
{
  int r = 1;
  while (1 < n)
    {
      r *= n;
      n -= 1;
    }
  return r;
}
```

```
function fact (n : int) : int =
  let
    var r := 1
  in
    while (1 < n)
      (r := r * n; n := n - 1);
    r
  end
```

   Le programme Tiger ne peut contenir de boucle `for`, qui d'une part introduise nécessaire une variable supplémentaire pour itérer, et d'autre part, étant donnée la sémantique forte du `for`, il nous faudrait au moins deux tests. Mais tout cela c'est de toutes façons sans compter que les boucles Tiger sont croissantes de pas 1.

3. D'après ce programme, détailler les conventions d'appel de cette machine, et en particulier, pour chacun des registres dire lesquels servent au passage d'argument, et au retour de résultat ?

   Correction: Il est sûr que `r1` sert pour le premier argument, et aussi pour retourner une valeur. Le registre `r2` pourrait également servir au passage d'arguments, voire au retour de « grosses » valeurs, mais ce n'est pas visible dans le cas présent. En tout cas, puisque `r3` est sous la responsabilité de l'appelé, il ne peut clairement pas servir ni aux arguments, ni aux retours.

4. Étant données les conventions d'appel, le sens du programme, expliquer quelles sont les variables vives lors du `return`.

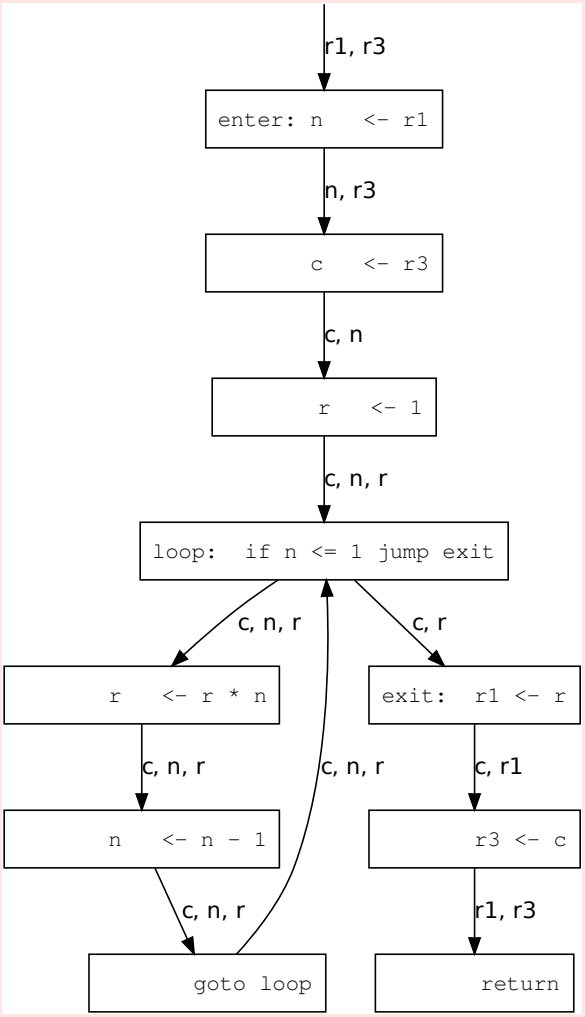
   Correction: Puisqu'elle est sous la responsabilité de l'appelé, `r3` est clairement vive. Puisqu'elle sert au retour de valeur, `r1` l'est aussi.

5. Expliquer pourquoi les lignes `c <- r3` et `r3 <- c` ont été générées.

Correction: Il faut préserver `r3`, ce qui, sans ces lignes, signifierait simplement que `r3` étant vive sur toutes les arêtes, elle est en conflit avec toutes les temporaires, donc inutilisable par l'allocation des registres. En clair, on compile avec deux registres au lieu de trois. Ici, on se donne la possible de se servir de `r3` quitte à devoir sauver sa valeur initiale sur la pile, puis la restaurer à la fin : `r3` reste disponible pour le corps de la boucle.

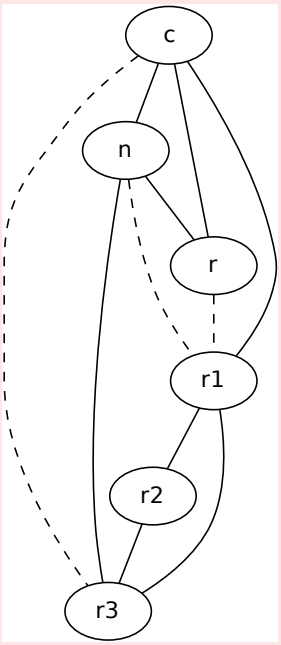
6. Calculer son graphe de contrôle de flux étiqueté sur les arêtes par les temporaires vives.

Correction:



7. Calculer le graphe d'interférence, y compris les fusions (*coalesce*) possibles.

Correction:



8. Est-il possible de le colorer tel quel ?

   Correction: Oui. D'abord on effectue `fusion (c, r3)`, puis on a le choix de `fusion (n, r1)` ou `fusion (r, r1)`. Il ne reste alors plus le choix pour le dernier.

9. Détaillez ce que il faut faire lorsque la coloration de ce graphe échoue. Dans la suite, on considère que si la coloration a effectivement échoué dans le cas présent, alors vous avez effectué les opérations nécessaires pour poursuivre l'allocation des registres.

   Correction: Si la coloration échoue, il faut choisir une cible, et la verser sur la pile (*spill*). Ici, `c` est un bon candidat, n'apparaissant pas dans la boucle. Puis on remplace la temporaire `c` par un accès mémoire `M[m]` [a]), enfin on récrit les lectures/écritures sur `c` pour charger/décharger depuis la pile. Par exemple :

```
enter: n    <- r1
       c1   <- r3
       M[m] <- c1
       r    <- 1
loop:  if n <= 1 jump exit
       r  <- r * n
       n  <- n - 1
       goto loop
exit:  r1 <- r
       c2 <- M[m]
       r3 <- c2
       return
```

   On a gagné le fait que le graphe sera moins contraint (on a remplacé `c` qui avait une gigantesque durée de vie qui le conduisait à être en conflit avec tous les autres en deux temporaires dont la durée de vie est très courte), donc plus coloriable.

   a. En toute exactitude, bien entendu `M[m]` serait plutôt quelque chose comme `[fp + 4]`, 4 étant pris au titre d'exemple : en fait n'importe quel endroit de la pile. Mais dans le cas présent, pour simplifier, cachons l'existence de ce registre supplémentaire.

10. Proposer une coloration du graphe obtenu.

Correction:

c	n	r	Code

r3	r1	r2	`enter: r2 <- 1`
			`loop: if r1 <= 1 jump exit`
			`r2 <- r2 * r1`
			`r1 <- r1 - 1`
			`goto loop`
			`exit: r1 <- r`
			`return`

r3	r2	r1	`enter: r2 <- r1`
			`r1 <- 1`
			`loop: if r2 <= 1 jump exit`
			`r1 <- r1 * r2`
			`r2 <- r2 - 1`
			`goto loop`
			`exit: return`

D'autres colorations sont possibles, mais elles sont moins performantes.

11. Donner le code assembleur final.