Correction du partiel CMP1

EPITA – Promotion 2012 (ERASMUS)
Tous documents (notes de cours, polycopiés, livres) autorisés.
Calculatrices et ordinateurs interdits.

Juin 2010 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain. Le *best of* est tiré des copies des étudiants, fautes de français y compris, le cas échéant.

Lisez bien les questions, chaque mot est important. Écrivez court, juste, et bien ; servez vous d'un brouillon. Une argumentation informelle mais convaincante est souvent suffisante. Gérez votre temps, ne restez pas bloqués sur les questions les plus difficiles. Une lecture préalable du sujet est recommandée.

Cette correction contient 13 pages. Les pages 1–6 contiennent l'épreuve et son corrigé. Ce document comporte en pages 7–10 une enquête (facultative) sur le cours et le projet Tiger (y répondre sur la feuille de QCM ci-jointe). Enfin, les pages 11–13 sont dévolues aux annexes.

1 Bison

Voici une grammaire au format EBNF (Extended BNF).

```
exps ::= [ exp { ',' exp }]
```

1. Qu'exprime cette grammaire ?

   Correction: Une suite arbitrairement longue de `exp` séparés par des ',', éventuellement vide. Bien voir que le ',' est ici un séparateur (comme en Pascal ou en Tiger), pas un terminateur (comme en C).

2. Que signifie BNF ?

   Correction: Backus-Naur Form.
   Note : D'après Wikipédia, les règles EBNF doivent être terminées par un point-virgule (`http://en.wikipedia.org/wiki/Extended_Backus_Naur_Form`). Je n'ai pas suivi ce style ici pour ne pas introduire de confusion avec la grammaire figurant dans le manuel de référence du compilateur Tiger, qui n'utilise pas non plus de point-virgule terminateur.

   Best-of: BNF signifie B-N format où je ne me souviens plus du B et du N.

3. On souhaite implémenter un parser pour cette grammaire avec Bison. Or ce dernier ne sait pas traiter des grammaires EBNF. Traduisez la grammaire précédente en une grammaire Bison.

Correction: Prendre soin d'utiliser la récurrence à gauche.

```
exps   :
  /* Empty. */
| exps.1
;

exps.1 :
  exp
| exps.1 ',' exp
;
```

4. Complétez votre proposition en incluant les actions sémantiques, permettant de construire un arbre de syntaxe abstraite (AST). Le type du nœud correspondant au symbole `exp` est `exp_t`. Vous êtes libres (de réfléchir) quant au choix du type de nœud pour `exps`.

Correction: On décide d'utiliser une liste de la bibliothèque standard du C++ pour représenter une `exps`.

```
exps   :
  /* Empty. */ { $$ = new std::list<exp_t*>; }
| exps.1        { $$ = $1; }
;

exps.1 :
  exp           { $$ = new std::list<exp_t*> (1, $1); }
| exps.1 ',' exp { $$ = $1; $$->push_back ($3); }
;
```

5. Quelle(s) règle(s) faut-il ajouter à votre grammaire Bison pour faire de la reprise sur erreur ?

Correction:

```
exps.1 :
  error ',' exp  { $$ = new std::list<exp_t*> (1, $3); }
;
```

2 Tiger

Considérez le bout de code Tiger ci-dessous.

```
type tree = { left : tree, val : int, right : tree }

function node (l : tree, v : int, r : tree) : tree =
 tree { left = l, val = v, right = r }

function leaf (v : int) : tree =
 tree { left = nil, val = v, right = nil }

var t := node (node (leaf (1),
                     2,
                     leaf(3)),
               4,
               node (leaf (5),
                     6,
                     leaf (7)))
```

1. À quoi peut bien servir `tree` ?

   Correction: À implémenter un arbre binaire (d'entiers), par exemple pour stocker un ensemble de valeurs un peu comme `std::set` en C++.

2. Quel nom pourrait-on donner à `node` et `leaf` ?

   Correction: Celui de *constructeurs* (externes), puisque ces routines servent à construire des `node` et `leaf` respectivement.

3. Citer les différentes phases de la partie frontale (*front end*) d'un compilateur comme `tc`.

   Correction: (i) scanner, (ii) parser et construction de l'Abstract Syntax Tree (AST), (iii) liaison des noms, (iv) typage, (v) traduction vers une représentation intermédiaire.

4. Quelle différence faites-vous entre un arbre de dérivation et un arbre de syntaxe abstraite (ou AST) ?
5. Donnez une représentation de l'AST du programme Tiger ci-dessus. Aidez-vous de l'annexe A à la fin de ce document.
6. Sur le schéma de la question précédente, indiquez les liens entre sites d'utilisation et sites de définition à l'aide de flèches partant des premiers et allant vers les seconds.
7. Le type `tree` est défini récursivement ; décrivez le mécanisme de la partie chargée du calcul des liaisons dans le compilateur qui autorise une telle définition récursive.
8. Écrivez une fonction Tiger `print_tree` prenant un `tree` en argument et affichant ses éléments comme dans la sortie ci-dessous (avec `t` passé en argument) :

   ```
   ␣␣␣␣1
   ␣␣2
   ␣␣␣␣3
   4
   ␣␣␣␣5
   ␣␣6
   ␣␣␣␣7
   ```

   Le niveau d'indentation (marqué par des paires d'espace) représentant la « profondeur » de chaque valeur d'entier par rapport à la racine (l'élément passé en argument à `print_tree`). L'annexe B peut vous être utile.

Correction: Une solution possible, utilisant la récursivité :

```
function print_tree (t : tree) =
 let
   function print_rec (t : tree, indent : string) =
     if (t <> nil) then
       (
         print_rec (t.left,  concat (indent, "  "));
         print (indent); print_int (t.val); print ("\n");
         print_rec (t.right, concat (indent, "  "))
       )
 in
   print_rec (t, "")
 end
```

9. Écrire une fonction `incr` prenant un `tree` en argument appliquant cette fonction à tous les `int` contenus dans le `tree`, et renvoyant un `tree` structurellement équivalent à celui d'entrée, mais dont tous les éléments entiers ont été incrémentés de 1. Par exemple, 'print_tree (incr (t))' afficherait

```
␣␣␣␣2
␣␣3
␣␣␣␣4
5
␣␣␣␣6
␣␣7
␣␣␣␣8
```

Correction:

```
function incr (t : tree) : tree =
 if t = nil then
   nil
 else
   node (incr (t.left), t.val + 1, incr (t.right))
```

10. On aimerait pouvoir généraliser la fonction `incr` de la question précédente et écrire une fonction `map` prenant un `tree` en argument et une fonction de type `int` → `int` (prenant un entier, renvoyant un entier), appliquant cette fonction à tous les `int` contenus dans le `tree`, et renvoyant un `tree` structurellement équivalent à celui d'entrée, mais portant les résultats de la fonction appliquée.
    (a) Pourquoi n'est-ce pas possible en l'état avec notre langage Tiger ?
    (b) En modifiant légèrement l'énoncé du problème, il est possible de s'en sortir grâce à la programmation orientée objet (toujours en Tiger). Comment ?

        Correction: En utilisant des objets-fonctions (ou *functors*), c'est-à-dire des objets se comportant comme des fonctions. En C++, c'est d'autant plus simple à utiliser que l'on peut surcharger l'opérateur « parenthèses » (`operator()`) pour utiliser un objet-fonction comme une fonction. Tiger ne fournit pas un tel sucre syntaxique, mais on peut toujours bénéficier de la fonctionnalité en utilisant un nom de méthode classique (comme `eval`, `go`, `apply`, coderun, etc.).
        La fonction `map` prendra donc un objet-fonction `f` en second argument, en lieu et place de la fonction initialement souhaitée. Pour garantir la réutilisabilité de `map`, on prendra soin de donner à `f` un type abstrait, de sorte que l'on puisse passer des instances de types différents en argument. Voir la correction de la question suivante pour plus de détails.

    (c) En utilisant la réponse à la question 10b, écrivez la fonction `map`, y compris ce qu'il faut ajouter pour que l'ensemble fonctionne.

Correction: Commençons par définit une classe « abstraite » définissant l'interface d'un functor, dont le code est contenu dans une méthode `eval` Tiger ne permet pas de définir de méthode abstraite (« virtuelle pure » en C++), aussi devons-nous donner un code fictive pour respecter le typage. On pourrait envisager d'insérer un appel à `exit` (avec un argument différent de 0) pour déclencher une erreur au *run time*, et ainsi « interdire » l'usage de cette méthode abstraite.

```
class Functor
{
 /* Dummy (identity). */
 method eval (i : int) : int = i
}
```

Ensuite, on peut définir des sous-classes de `Foreign` définissant des objets-fonctions concret, par exemple incrémentant leur argument ou les élevant au carré. On rappelle que toutes les méthodes sont virtuelles (polymorphes) en Tiger, donc aucun mot-clef (`virtual`, `deferred`, etc.) n'est nécessaire pour bénéficier du polymorphisme.

```
class Incr extends Functor
{
 method eval (i : int) : int = i + 1
}

class Square extends Functor
{
 method eval (i : int) : int = i * i
}
```

Enfin, nous pouvons écrire `map` de façon récursive, en utilisant la méthode `eval` du functor `f` passé en argument.

```
function map (t : tree, f : Functor) : tree =
 if t = nil then
  nil
 else
  node (map (t.left, f),
        f.eval (t.val),
        map (t.right, f))
```

11. **Bonus** Le type du champ `val` d'un `tree` est fixe : il s'agit toujours d'un `int`. C'est dommage, car peu réutilisable ; imaginons que l'on ait besoin d'une structure similaire à `tree`, mais avec des `string` (ou tout autre type) à la place. Il y a au moins deux solutions à ce problème. La première solution, que l'on nommera $\mathcal{A}$, est déjà implémentable dans le langage Tiger utilisé à l'EPITA [1], mais présente des défauts. La seconde solution, meilleure du point de vue du typage et des performances, et que l'on nommera $\mathcal{B}$, serait classique à implémenter en C++, mais n'est pas réalisable en l'état en Tiger.

    (a) Quelles sont ces deux solutions ?

    (b) Les deux solutions $\mathcal{A}$ et $\mathcal{B}$ de la question précédentes s'appuient chacune sur un type de polymorphisme. Quel est celui correspondant à $\mathcal{A}$? et celui relatif à $\mathcal{B}$?

    (c) Réécrivez les définitions de `tree`, `node` et `leaf` en utilisant la solution $\mathcal{A}$.

1. On rappelle que le langage Tiger utilisé à l'EPITA est différent de celui que propose Andrew Appel, et comporte certaines additions.

3 À propos de ce cours

Pour terminer cette épreuve, nous vous invitons à répondre à un petit questionnaire. Les renseignements ci-dessous ne seront bien entendu pas utilisés pour noter votre copie. Ils ne sont pas anonymes, car nous souhaitons pouvoir confronter réponses et notes. En échange, quelques points seront attribués pour avoir répondu. Merci d'avance.

Sauf indication contraire, vous pouvez cocher plusieurs réponses par question. Répondez sur la feuille de QCM qui vous est remise. N'y passez pas plus de dix minutes.

Cette épreuve

1. Sans compter le temps mis pour remplir ce questionnaire, combien de temps ce partiel vous a-t-il demandé (si vous avez terminé dans les temps), ou combien vous aurait-il demandé (si vous aviez eu un peu plus de temps pour terminer) ?
   A Moins de 30 minutes.
   B Entre 30 et 60 minutes.
   C Entre 60 et 90 minutes.
   D Entre 90 et 120 minutes (estimation).
   E Plus de 120 minutes (estimation).
2. Ce partiel vous a paru
   A Trop difficile.
   B Assez difficile.
   C D'une difficulté normale.
   D Assez facile.
   E Trop facile.

Le cours

3. Vous avez pris des notes
   A Aucune.
   B Sur papier.
   C Sur ordinateur à clavier.
   D Sur ardoise.
   E Sur le journal du jour.
4. Quelle a été votre implication dans les cours CMP1 ?
   A Rien.
   B Bachotage récent.
   C Relu les notes entre chaque cours.
   D Fait les annales.
   E Lu d'autres sources.
5. Ce cours
   A Est incompréhensible et j'ai rapidement abandonné.
   B Est difficile à suivre mais j'essaie.
   C Est facile à suivre une fois qu'on a compris le truc.
   D Est trop élémentaire.

6. Ce cours
   A Ne m'a donné aucune satisfaction.
   B N'a aucun intérêt dans ma formation.
   C Est une agréable curiosité.
   D Est nécessaire mais pas intéressant.
   E Je le recommande.
7. La charge générale du cours en sus de la présence en amphi (relecture de notes, compréhension, recherches supplémentaires, etc.) est
   A Telle que je n'ai pas pu suivre du tout.
   B Lourde (plusieurs heures par semaine).
   C Supportable (environ une heure de travail par semaine).
   D Légère (quelques minutes par semaine).

Les formateurs

8. L'enseignant
   A N'est pas pédagogue.
   B Parle à des étudiants qui sont au dessus de mon niveau.
   C Me parle.
   D Se répète vraiment trop.
   E Se contente de trop simple et devrait pousser le niveau vers le haut.
9. Les assistants
   A Ne sont pas pédagogues.
   B Parlent à des étudiants qui sont au dessus de mon niveau.
   C M'ont aidé à avancer dans le projet.
   D Ont résolu certains de mes gros problèmes, mais ne m'ont pas expliqué comment ils avaient fait.
   E Pourraient viser plus haut et enseigner des notions supplémentaires.

Le projet Tiger

10. Vous avez contribué au développement du compilateur de votre groupe (une seule réponse attendue) :
    A Presque jamais.
    B Moins que les autres.
    C Équitablement avec vos pairs.
    D Plus que les autres.
    E Pratiquement seul.
11. La charge générale du projet Tiger est
    A Telle que je n'ai pas pu suivre du tout.
    B Lourde (plusieurs jours de travail par semaine).
    C Supportable (plusieurs heures de travail par semaine).
    D Légère (une ou deux heures par semaine).
    E J'ai été dispensé du projet.

12. Y a-t-il de la triche dans le projet Tiger ? (Une seule réponse attendue.)
    A Pas à votre connaissance.
    B Vous connaissez un ou deux groupes concernés.
    C Quelques groupes.
    D Dans la plupart des groupes.
    E Dans tous les groupes.

Questions 13-19 Le projet Tiger vous a-t-il bien formé aux sujets suivants ? Répondre selon la grille qui suit. (Une seule réponse attendue par question.)

A Pas du tout.
B Trop peu.
C Correctement.
D Bien.
E Très bien.

13. Formation au C++.
14. Formation à la modélisation orientée objet et aux *design patterns*.
15. Formation à l'anglais technique.
16. Formation à la compréhension du fonctionnement des ordinateurs.
17. Formation à la compréhension du fonctionnement des langages de programmation.
18. Formation au travail collaboratif.
19. Formation aux outils de développement (contrôle de version, systèmes de construction, débogueurs, générateurs de code, etc.)

Questions 20-35 Comment furent les étapes du projet ? Répondre selon la grille suivante. (Une seule réponse attendue par question ; ne pas répondre pour les étapes que vous n'avez pas faites.)

A Trop facile.
B Facile.
C Nickel.
D Difficile.
E Trop difficile.

20. Rush `.tig` : mini-projet en Tiger (Bistromatig).
21. `TC-0`, Scanner & Parser.
22. `TC-1`, Scanner & Parser, Tâches, Autotools.
23. `TC-2`, Construction de l'AST et pretty-printer.
24. `TC-3`, Liaison des noms et renommage.
25. `TC-4`, Typage et calcul des échappements.
26. `TC-5`, Traduction vers représentation intermédiaire.
27. `TC-6`, Simplification de la représentation intermédiaire.
28. `TC-7`, Sélection des instructions.
29. `TC-8`, Analyse du flot de contrôle.
30. `TC-9`, Allocation de registres.

31. Option `TC-D`, Suppression du sucre syntaxique (boucles `for`, comparaisons de chaînes de caractères).
32. Option `TC-I`, Mise en ligne du corps des fonctions.
33. Option `TC-B`, Vérification dynamique des bornes de tableaux.
34. Option `TC-A`, Surcharge des fonctions.
35. Option `TC-O`, Désucrage des constructions objets (transformation Tiger → Panther).

A Classes de la syntaxe abstraite du langage Tiger

Voici une copie du fichier 'src/ast/README' fourni avec le code de tc.

```
Tiger Abstract Syntax Tree nodes with their principal members.
Incomplete classes are tagged with a '*'.

/Ast/               (Location location)
  /Dec/             (symbol name)
    FunctionDec     (VarDecs formals, NameTy result, Exp body)
      MethodDec     ()
    RuleDec         (Exp match, Exp build)
    TypeDec         (Ty ty)
    VarDec          (NameTy type_name, Exp init)

  /Exp/             ()
*   /Var/
      CastVar       (Var var, Ty ty)
*     FieldVar
      SimpleVar     (symbol name)
      SubscriptVar  (Var var, Exp index)

*   ArrayExp
*   AssignExp
*   BreakExp
*   CallExp
*     MethodCallExp
    CastExp         (Exp exp, Ty ty)
    ForExp          (VarDec vardec, Exp hi, Exp body)
*   IfExp
    IntExp          (int value)
*   LetExp
    MetavarExp      ()
    NilExp          ()
*   ObjectExp
    OpExp           (Exp left, Oper oper, Exp right)
*   RecordExp
*   SeqExp
*   StringExp
    WhileExp        (Exp test, Exp body)

  /Ty/              ()
    ArrayTy         (NameTy base_type)
    ClassTy         (NameTy super, DecsList decs)
    NameTy          (symbol name)
*   RecordTy

  DecsList          (decs_type decs)

  Field             (symbol name, NameTy type_name)

  FieldInit         (symbol name, Exp init)
```

```
Some of these classes also inherit from other classes.

/Escapable/
  VarDec            (NameTy type_name, Exp init)

/Metavariable/      (unsigned id)
  MetavarExp        ()

/Typable/
  /Dec/             (symbol name)
  /Exp/             ()
  /Ty/              ()

/TypeConstructor/
  /Ty/              ()
  FunctionDec       (VarDecs formals, NameTy result, Exp body)
  TypeDec           (Ty ty)
```

B Bibliothèque standard du langage Tiger

Le listing ci-dessous contient les déclarations (annotées) du prélude actuel du compilateur Tiger (`prelude.tih`).

```
/* Print STRING on the standard output. */
primitive print (string: string)
/* Note: this is an EPITA extension. Same as print(), but the output
   is written to the standard error. */
primitive print_err (string: string)
/* Note: this is an EPITA extension. Output INT in its decimal
   canonical form (equivalent to "%d" for printf()). */
primitive print_int (int: int)
/* Flush the output buffer. */
primitive flush ()
/* Read a character on input. Return an empty string on an end of
   file. */
primitive getchar () : string

/* Return the ASCII code of the first character in STRING and -1 if
   the given string is empty. */
primitive ord (string: string) : int
/* Return the one character long string containing the character which
   code is CODE. If CODE does not belong to the range [0..255], raise
   a runtime error: "chr: character out of range." */
primitive chr (code: int) : string
/* Return the size in characters of the STRING. */
primitive size (string: string) : int

/* Note: this is an EPITA extension. Return 1 if the strings A and B
   are equal, 0 otherwise. Often faster than strcmp() to test string
   equality. */
primitive streq (s1: string, s2: string) : int
/* Note: this is an EPITA extension. Compare the strings A and B:
   return -1 if A < B, 0 if equal, and 1 otherwise. */
```

```
primitive strcmp (s1: string, s2: string) : int
/* Return a string composed of the characters of STRING starting at
   the FIRST character (0 being the origin), and composed of LENGTH
   characters (i.e., up to and including the character
   FIRST + LENGTH). */
primitive substring (string: string, start: int, length: int) : string
/* Concatenate FIRST and SECOND. */
primitive concat (first : string, second: string) : string

/* Return 1 if BOOLEAN = 1, else return 0. */
primitive not (boolean : int) : int

/* Exit the program with exit code STATUS. */
primitive exit (status: int)
```