Correction du Partiel CMP2

EPITA – Apprentis Promo 2010
Tous documents (notes de cours, polycopiés, livres) autorisés.
Tous dispositifs de calcul automatisé (calculatrice, téléphone portable, ordinateur, etc.) interdits.

Juin 2008 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain. Le *best of* est tiré des copies des étudiants.

Une copie synthétique, bien orthographiée, avec un affichage clair des résultats, sera toujours mieux notée qu'une autre demandant une quelconque forme d'effort de la part du correcteur. Une argumentation informelle mais convaincante, sera souvent suffisante.

1 Incontournables

Il n'est pas admissible d'échouer sur une des questions suivantes : **chacune induit une pénalité sur la note finale**. Répondez sur les feuilles de QCM qui vous sont remises (n'oubliez pas d'y inscrire votre nom ou login).

1. `sizeof(getc())` est une expression valide en C. A. Vrai/B. Faux ?

   Correction: C'est faux. `sizeof` est un opérateur C qui attend un argument *statique* (c'est-à-dire, dont la valeur est connue à la compilation), alors que `getc` est une fonction, dont l'évaluation est dynamique (c'est-à-dire, effectuée à l'exécution).

2. On peut coder jusqu'à 256 valeurs différentes avec un octet signé. A. Vrai/B. Faux ?

   Correction: C'est vrai. Le signe importe peut ; ce qui compte, c'est qu'avec un octet (= 8 bits) on peut coder jusqu'à $2^8 = 256$ valeur différentes.

3. `type_t *x;` ne peut-être qu'une déclaration de variable en C. A. Vrai/B. Faux ?

   Correction: Faux : ce peut être aussi une multiplication, si `type_t` et `x` sont des variables de type numérique (`int`, `float`, etc.).

4. En C++, on ne peut pas redéfinir un opérateur << en dehors d'une classe (c'est-à-dire, en tant que fonction non-membre). A. Vrai/B. Faux ?

   Correction: Bien sûr que c'est possible ; il fallait donc répondre « Faux » ici.

2 Parallélisation du compilateur

Les machine modernes disposent de processeurs multi-cœur, et on souhaite tirer parti de ce parallélisme facilement accessible dans notre compilateur Tiger. Nous allons limiter la portée de l'exercice au *front-end*, car le *back-end* n'a pas été vu en cours.

Les réponses informelles (mais sensées) sont acceptées ; l'objectif est de réfléchir et de faire jouer votre connaissance du compilateur. Répondre simplement « oui » ou « non » n'est pas suffisant.

1. Que peut-on paralléliser dans le scanner et le parser de `tc` ?

   Correction: On ne peut à peu près rien espérer paralléliser dans le scanner et le parser : les données imposent le sens de lecture et de traitement.
   Seule exception notable : les clause `import`, qui peuvent être analysées en parallèle du fichier qui les incluent.

2. Est-ce que la liaison des noms (*binding*) peut bénéficier d'une accélération via la parallélisation ? Comment ? (Bien se rappeler comment l'Abstract Syntax Tree (AST) est parcouru !)

   Correction: Le binding est peu parallélisable : on est dépendant de l'ordre dans lequel les déclarations et les instructions sont introduites. Cependant, certains endroits sont parallélisables :
   – les *corps* (pas les *en-têtes*) des fonctions (resp. des méthodes) contenues dans un même bloc (*chunk*) (resp. dans une même classe) sont indépendants, et peuvent donc être traités séparément.
   – La recherche dans un conteneur (utilisé pour rechercher un symbole dans l'environnement courant lors du binding) est parallélisable, et peut même bénéficier d'une accélération superlinéaire (cf. infra).

3. De même, peut-on paralléliser le *type-checking* ? Comment ? (Même remarque.)

   Correction: La aussi, peu de parallélisation possible en général : les types *utilisés* doivent être *calculés* au préalable. Cependant, plusieurs parties peuvent bénéficier de parallélisme :
   – le type-checking des *corps* des fonctions et des méthodes situées au même niveau (à l'instar de ce qui a été proposé pour le calcul des noms) ;
   – la comparaison de listes de types, dans la vérification des `RecordExp`s, des `CallExp`s et des `SeqExp`s ;
   – le type-checking des opérandes d'une `OpExp` ou d'une `AssignExp` ;
   – idem pour les nœuds fils de `IfExp`, `WhileExp` et `ForExp`.

4. Comment peut-on paralléliser la traduction vers la représentation intermédiaire ?

   Correction: À ce stade, on dispose de toutes les informations de nom et de types sur l'AST. Une forme de parallélisme simple et efficace consiste à traiter en parallèle les fonctions situées au même niveau (les variables non locales empêchent un traitement parallèle inter-niveaux).
   Par la suite, chaque fonction sera compilée sous forme d'un fragment HIR indépendant des autres. La canonisation de ces fragments en LIR est donc facilement parallélisable.

5. Intuitivement, on se dit qu'une tâche nécessitant un temps d'exécution t sur un processeur mono-cœur s'effectuera *au mieux* en un temps $\frac{t}{n}$ sur un processeur à n cœurs (tous les cœurs étant identiques, bien entendu) : c'est une *accélération linéaire*. Il existe cependant certaines opération qui peuvent bénéficier d'une accélération encore meilleure, c'est-à-dire s'exécutant en un temps *inférieur ou égal* à $\frac{t}{n}$; on dit alors que l'accélération est *superlinéaire*.
   À votre avis, quels types de traitements rentrent dans ce cas ? Pourquoi ?

Correction: Deux façons de bénéficier d'une accélération superlinéaire :
(a) effectuer moins de travail « de façon disproportionnée » ;
(b) exploiter de façon disproportionnée plus de ressources.
Un exemple qui tombe dans la première possibilité est la recherche parallélisée. En effet, partager la recherche d'une (seule) occurrence d'un élément dans un conteneur en n tâches prendra *au plus* un temps $\frac{t}{n}$ (où t est le temps d'une recherche séquentielle usuelle), car dès que l'une des tâches a trouvé l'élément recherché, le traitement complet peut s'arrêter.
Le second point peut s'illustrer par l'utilisation de plus de mémoire cache lorsque l'on fait travailler en parallèle plusieurs cœurs d'un même processeur (tous les cœurs n'ayant pas accès à la totalité du cache), ce qui permet d'accélérer de façon générale l'accès en mémoire. On utilise ainsi plus de ressources grâce à la parallélisation.
Voir à ce sujet les excellents articles de Herb Sutter (secrétaire du comité de la norme ISO/ANSI C++, expert C++ et architecte logiciel chez Microsoft) dans Dr. Dobbs Journal :
– http://www.ddj.com/hpc-high-performance-computing/206100542
– http://www.ddj.com/hpc-high-performance-computing/206903306

6. Quelle(s) tâche(s) parallélisable pourraient tirer parti d'une accélération superlinéaire dans notre compilateur Tiger ?

Correction: La recherche en parallèle est un très bon exemple de traitement pouvant bénéficier d'une accélération superlinéaire. La recherche d'un élément dans un conteneur est par exemple utilisée lors de la résolution des noms à l'étape du *binding*. On peut envisager une implémentation des tables de symboles autorisant une recherche en parallèle superlinéaire (cf. réponse précédente).

3 Pointeurs

Dans cet exercice, on s'intéresse à l'adjonction de *pointeurs* au langage Tiger.

1. Au cours de l'enseignement que vous avez suivi, vous avez été mis en garde plusieurs fois contre les dangers que peuvent présenter les pointeurs dans un langage comme le C ou le C++. Rappelez ces problèmes que peuvent poser les pointeurs.

Correction:
– Le langage n'impose pas à un pointeur d'être initialisé avant d'avoir été utilisé.
– Le fait qu'on puisse initialiser un pointeur avec autre chose qu'un objet existant (une valeur numérique représentant une adresse, par exemple) signifie qu'un pointeur peut référencer et/ou être utilisé pour modifier une zone mémoire indépendante « saine », *a priori* inaccessible dans la portée courante.
– L'arithmétique des pointeurs et les opérateurs de transtypage (cast) permettent de modifier la valeur d'un pointeur sans aucune garantie quant à la validité du résultat.

2. Néanmoins, les pointeurs représentent une fonctionnalité utile dans un langage de programmation. Que peuvent-ils apporter ?

Correction:
- Ils permettent de simuler un passage par référence, dans un langage qui n'en dispose pas (le C, par exemple).
- Plus généralement, ils permettent de simuler… des références, dans un langage qui n'en dispose pas (le C, par exemple).
- L'implémentation de *call-backs* via des pointeurs sur fonction (ou sur méthode).
- L'allocation/manipulation de la mémoire bas niveau « faite main » afin d'optimiser.
- L'arithmétique des pointeurs permet des parcours en mémoire très rapides, particulièrement utiles pour implémenter des itérations sur des conteneurs (comme des tableaux).

3. Beaucoup de langages modernes ont banni les pointeurs, et leur ont préféré un substitut plus sûr. Quel est-il ? En quoi ce remplaçant est-il plus sûr ?

   Correction: Ce sont les références, qui, dans la plupart des langages qui les fournissent, permettent d'éviter les problèmes de la question 1.

   Best-of:

   Ce remplaçant [les références] est plus sûr, car le type est connue à la compilation.

4. Le substitut aux pointeurs de la question précédente remplace avantageusement ceux-ci dans la majorité des cas. Citez un cas où les pointeurs permettent de faire « plus de choses ». (N.B. : Cette question recoupe – à dessein – la question 2, et il se peut que votre réponse ait déjà été mentionnée à cette précédente question, mais ce n'est pas grave.)

   Correction: L'arithmétique des pointeurs, par exemple, pour itérer rapidement sur un conteneur.

5. **Grammaire.** Comment étendez-vous la grammaire de `tc` pour lui ajouter le support des pointeurs ?

Correction:

– Les pointeurs introduisent de nouveaux types : il faut donc un mot clef permettant de construire des tels types, par exemple :

```
type int_ptr = pointer on int
```

– Il nous faut également un moyen de créer un pointeur à partir d'un objet du type pointé, par exemple un opérateur unaire `ref` :

```
type int_ptr = pointer on int
var a := 42
var p : int_ptr := ref a
```

– Symétriquement, nous devons pouvoir déréférencer un objet de type pointeur, par exemple avec un opérateur unaire `deref` :

```
type int_ptr = pointer on int
var a := 42
var p : int_ptr := ref a
var b := deref p
```

Il faudra veiller à gérer la priorité de ces opérateurs vis-à-vis des opérateurs existants ; pour garder une sémantique proche de celle du C et du C++, on veillera à donner à `ref` et `deref` des priorités élevées (en C et C++, la priorités de ces opérateurs est située juste en dessous de celle de `()`, `[]`, `->` et `.`).

En conclusion, cela donnerait les extensions de grammaire suivantes :

```
ty → pointer on type−id

lvalue → ref lvalue
lvalue → deref exp
```

(On a choisi d'étendre la production `lvalue` plutôt que `exp`, car cela permet d'écrire directement dans un emplacement référencé/déréférencé).

Il est aussi envisageable de doter le langage d'un nouveau mot-clef (`null` par exemple) afin d'initialiser un pointeur à une valeur invalide, ou bien d'utiliser simplement 0 car on souhaite pouvoir convertir nos pointeurs vers des entiers et vice versa.

6. **AST.** Comment modifiez-vous les classes de la syntaxe abstraite ? (*Hint* : Des diagrammes UML sont parfois plus parlant que du code ou qu'une longue explication.)

   Correction: Il faut ajouter des nœuds pour chacune des constructions de la question précédente. Les voici, en résumé, insérés dans la hiérarchie actuelle :

```
/Ast/
 /Ty/
   PointerTy   (const NameTy& base_type)

 /Exp/
   /Var/
     RefVar    (const Var& var)
     DerefVar  (const Exp& exp)
```

7. **Noms.** Que faut-il changer dans le calcul de liaisons des noms ?

   Correction: Rien : les noms introduits (types et variables) sont déjà gérés par le `Binder`.

8. **Typage.** Que faut-il ajouter et/ou changer dans l'étape de vérification des types ?

Correction:

– Ajouter une nouvelle classe à la hiérarchie des types :

```
/Type/
  Pointer (const Type& type)
    const Type& type_get () const
    bool compatible_with (const Type& other) const;
```

Ne pas oublier de redéfinir la méthode `compatible_with` pour que deux variables de deux types pointeurs *différents*, mais pointant directement ou indirectement sur le *même type* soient compatibles. C'est-à-dire, le code suivant doit pouvoir compiler et fonctionner :

```
let
  type int_ptr = pointer on int
  type int_alias = int
  type int_alias_ptr = pointer on int_alias
  var a := 42
  var b := 51
  var pa : int_ptr := ref a
  var pb : int_alias_ptr := ref b
in
  deref a = deref b
end
```

De même, les expressions de référencement et déréférencement doivent avoir des types qui sont compatibles avec n'importe quel type pointeur référençant le même type d'objet :

```
var a := 3       /* A has type 'int'. */
var p := ref a   /* P has type 'pointer on int'. */
var b := deref p /* B has type 'int'. */
```

– Étendre le `TypeChecker` pour prendre en compte
  – `PointerTy`, qui introduit une nouvelle instance de `type::Pointer` ;
  – `RefVar`, qui produit une l-value de type pointeur du fils de ce nœud ;
  – `DerefVar`, qui produit une l-value du type pointé du fils de ce nœud.

– De plus, on veut que les types pointeurs et entiers soient compatibles, afin d'avoir une arithmétique sur les pointeurs. Il va donc falloir modifier/ajouter la méthode `compatible_with` des classes `type::Int` et `type::Pointer` pour qu'elle réponde vrai lorsqu'elles reçoivent un argument de l'autre type. Ainsi, le code suivant doit être valide :

```
var p := 0  /* Warning, null pointer. */
```

9. **Traduction.** L'ajout des pointeurs à `tc` a-t-il une incidence sur la traduction vers la représentation intermédiaire ?

   Correction: Oui, bien sûr : il va falloir traduire les noeuds `RefVar` et `DerefVar` en langage TREE. Cette traduction est immédiate, puisque ces deux opérateurs correspondent respectivement à récupérer l'adresse d'une l-value (le code de traitement de `SimpleVar`, sans déréférencement), et à déréférencer une adresse avec l'opérateur MEM.

10. **Question bonus.** À votre avis, est-ce que les pointeurs nécessitent une modification du *back-end* (génération de code, allocation de registres, etc.) ? Si oui, laquelle/lesquelles ? Si non, pour quelle(s) raison(s) ?

Correction: Non, aucune modification n'est nécessaire dans le back-end : les pointeurs ont été complètement traduits lors de la phase de traduction, et le code TREE émis n'a pas de particularité supplémentaire.

4 À propos de ce cours

Pour terminer cette épreuve, nous vous invitons à répondre à un petit questionnaire.

Les renseignements ci-dessous ne serons 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 les feuilles de QCM qui vous sont remises. N'y passez pas plus de dix minutes.

Le cours

5. Quelle a été votre implication dans le cours (CCMP & TYLA) ?
   - **A** Rien.
   - **B** Bachotage récent.
   - **C** Relu les notes entre chaque cours.
   - **D** Fait les annales.
   - **E** Lu d'autres sources.
6. Le 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.
7. 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.
8. La charge générale du cours (relecture de notes, compréhension, recherches supplémentaires, etc.) est
   - **A** Légère (quelques minutes par semaine).
   - **B** Supportable (environ une heure de travail par semaine).
   - **C** Lourde (plusieurs heures par semaine).
   - **D** Telle que je n'ai pas pu suivre du tout.

Les formateurs

9. L'enseignant de CCMP & TYLA
   - **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.

10. 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

11. 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.
12. La charge générale du projet Tiger est
    A (J'ai été dispensé du projet.)
    B Légère (une ou deux heures par semaine).
    C Supportable (plusieurs heures de travail par semaine).
    D Lourde (plusieurs jours de travail par semaine).
    E Telle que je n'ai pas pu suivre du tout.
13. 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 14-20 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

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

Questions 22-33 Comment furent les étapes du projet (ne pas répondre à celles que vous n'avez pas faites). Répondre selon la grille suivante. (Une seule réponse attendue par question.)

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

21. Mini-projet en Tiger (LZW)

22. `TC-0`, Scanner & Parser.
23. `TC-1`, Scanner & Parser en C++, Autotools.
24. `TC-2`, Construction de l'AST.
25. `TC-3`, Liaison des noms.
26. `TC-4`, Typage.
27. Désucrage des constructions objets (transformation Tiger → Panther)
28. `TC-5`, Traduction vers représentation intermédiaire.

29. Option `TC-E`, Calcul des échappements.
30. Option `TC-A`, Surcharge des fonctions.
31. Option `TC-D`, Suppression du sucre syntaxique (boucles `for`, comparaisons de chaînes de caractères).
32. Option `TC-B`, Vérification dynamique des bornes de tableaux.
33. Option `TC-I`, Mise en ligne du corps des fonctions.