Correction du partiel CMP1 de S2

EPITA – Promo 2015
Tous documents (notes de cours, polycopiés, livres) autorisés
Calculatrices, ordinateurs, tablettes et téléphones interdits.

Avril 2013 (1h00)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain.

Best-of: Le *best-of* est tiré des copies des étudiants.

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 bloqué 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 7–9 une enquête (facultative) sur le cours et le projet Tiger (y répondre sur la feuille de QCM qui vous est fournie). Enfin, les pages 10–13 sont dévolues aux annexes.

1 Const

À la différence du C++, Tiger est dépourvu de mot-clef `const`. On se propose dans cet exercice de l'ajouter au langage et de passer en revue les différentes parties de notre compilateur afin de déterminer quels sont les aménagements nécessaires.

1.1 Préliminaires

1. À quoi sert `const` en C++ ? (Il y a plusieurs réponses possibles, une seule vous est demandée.)

   Correction:
   – Créer des constantes, au sens strict.
   – Créer des alias (via des pointeurs ou des références) de variables existantes, en restreignant leur accès à la simple lecture.
   – Appliqué à une méthode, garantir que le code de celle-ci ne modifiera pas l'objet cible.
   – Par extension des points précédents, permettre certaines optimisations (propagation de constantes, élimination de constantes, évaluation partielle, etc.).
   Attention, `const` ne sert pas à « figer » une zone mémoire. Dans le code ci-dessous, `i` et `j` désignent la même zone mémoire, mais le fait que `j` soit qualifié de `const` ne donne aucune garantie sur la constance de `i` !

```
int i = 42;
const int& j = i;
```

Best-of:
– `const` permet d'interdire la réallocation d'une variable.
– `const` sert à définir une variable de manière unique.
– `const` sert à indiquer qu'une fonction n'a pas d'effet de bord.

2. Parmi les lignes suivantes, quelles sont celles qui sont valides en C++ ? Justifiez votre réponse.
   (a) `int i = 42; const int* const p = &i;`
   (b) `int i = 42; const int const * p = &i;`
   (c) `int i = 42; int const * p = &i;`
   (d) `int i = 42; int const * const p = &i;`

Correction: Seule la ligne 2b est incorrecte ; les autres sont justes. Rappelons la règle du placement du mot clef `const` : celui-ci s'applique à ce qui précède, sauf lorsqu'il est placé en tête ; dans ce cas, il qualifie ce qui suit. Dans le cas de la ligne 2b, les deux `const`s s'appliqueraient donc tous les deux à `int`, ce que le compilateur refuse.

3. Même question.
   (a) `int i = 42; const int& const p = i;`
   (b) `int i = 42; const int const & p = i;`
   (c) `int i = 42; int const & p = i;`
   (d) `int i = 42; int const & const p = i;`

Correction: Ici, une règle supplémentaire s'applique : une référence est implicitement un pointeur constant déguisé. En conséquence, `const` ne peut s'appliquer à `&`. Seule la réponse 3c est donc valide ici.

Best-of: Et là, c'est le drame.

4. Soit la classe suivante :

```
struct C
{
  void m1 () {}        // (0).
  void m1 () const {}  // (1).
  void m2 () {}        // (2).
  void m3 () const {}  // (3).
};
```

Pour chacun des cas suivants, indiquer quelle méthode est appelée (0, 1, 2 ou 3) ou s'il y a une erreur de compilation, le cas échéant.

   (a) `C o; o.m1();`

Correction: Réponse : 0 (`o` est non constant).

   (b) `const C o = C(); o.m1();`

Correction: Réponse : 1 (`o` est constant).

   (c) `const C o = C(); o.m2();`

Correction: Erreur de compilation : `m2` est une méthode non constante, et ne peut être appelée avec une cible constante (`o`).

   (d) `C o; o.m3();`

Correction: Réponse : 3 (on peut passer un objet mutable à une méthode constante).

(e) `const void* o = new C(); o->m1();`

Correction: Erreur de compilation : le type de l'objet pointé par `o` est perdu lors de la conversion vers `void*` : le compilateur est donc incapable de savoir que `o` pointe sur un objet de type `C`.

(f) `const C o; C(o).m1();`

Correction: Réponse : 0. On crée ici une copie de `o` de type `C` (non `const`) ; C'est la méthode `m1` non `const` qui est donc appelée. On rappelle que la syntaxe `C(o)` est un appel au constructeur par copie de `C`, qui est ajouté automatiquement par le compilateur.

1.2 Partie frontale (*front end*)

Cette partie s'intéresse aux modifications à apporter au front end Tiger pour prendre en compte `const`. Nous ne considérerons ce nouveau mot-clef que comme qualificatif de types (le cas des méthodes « constantes » n'est pas à traiter).

1. **Spécifications lexicales.** Que faut-il ajouter aux spécifications lexicales du langage pour supporter `const` ?

   Correction: Il suffit d'ajouter un nouveau mot-clef (réservé) `const`. Le terme *token* ne convient pas ici : il s'agit de spécification, pas d'implémentation.

2. **Scanner.** Comment étendre le scanner, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`scantiger.ll`' pour supporter ces nouvelles spécifications lexicales ?

   Correction: Quelque chose de ce genre dans la section des règles lexicales (entre les deux occurrences de `%%`, placé avant la règle reconnaissant les identifiants du langage) :

   ```
   "const"  return parse::parser::make_CONST(tp.location_);
   ```

   où `tp` est le *parser driver* (contenant entre autres la *location* courante). Il faudra également penser à définir un nouveau token `CONST` dans '`parsetiger.yy`' (cf. réponse à la question 5).

3. **Spécifications syntaxiques.** Que faut-il ajouter aux spécifications syntaxiques du langage pour supporter `const` ?

   Correction: L'extension la plus simple et qui conserve tout le pouvoir d'expression de `const` (au détriment de constructions plus lourdes par endroits) consiste à augmenter la règle de production `type-id` pour qu'elle accepte les identifiants précédés de `const`.

   ```
   ⟨type-id⟩ ::= id
               | "const" id
   ```

   Certes, cela ne permet pas des écritures telles que :

   ```
   type t = const array of int
   ```

   mais il suffit de remplacer par :

   ```
   type u = array of int
   type t = const u
   ```

   pour contourner cette limitation. Il bien sûr possible de supporter la première écriture, en modifiant un peu plus la grammaire, et au prix de quelques modifications pour lever les ambiguïtés introduites. Dans le reste de la correction, on considère que c'est l'extension la plus simple (la première décrite ci-avant) qui est adoptée.

Best-of:
– Il suffit de rajouter une règle.
– Spécifier la précédence de `const`.

4. **Syntaxe abstraite.** Faut-il effectuer des modifications ou des ajouts dans la syntaxe abstraite du langage pour supporter `const` ? Si oui, lesquelles ?

Correction: Oui, il faut pouvoir matérialiser `const` dans la syntaxe abstraite. Plusieurs solutions s'offrent à nous.
– En ajoutant un attribut booléen (par exemple `const_`), à `ast::NameTy`, mis à `false` par défaut, et à `true` lorsque le type est précédé de `const`. Il faut équiper la classe : ajout d'accesseur(s), modification de constructeurs, etc.
– En ajoutant une classe `ast::ConstNameTy`, symétrique à `ast::NameTy`. Pour des raisons de factorisation, on peut faire dériver ces deux classes d'une même classe abstraite.
Pour des raisons de simplicité, nous choisissons la première solution pour le reste de la correction, mais toutes les réponses valides ont été acceptées.

5. **Parser.** Comment étendre le parser, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`parsetiger.yy`' pour supporter ces nouvelles spécifications syntaxiques ?

Correction: Les ajouts nécessaires sont :
– la définition du token `CONST`, déjà mentionnée à la question 2 :

```
%token CONST "const"
```

– la règle de production évoquée à la question 3 :

```
typeid:
  ID        { $$ = new ast::NameTy (@$, $ID, false); }
  // New rule.
| "const" ID { $$ = new ast::NameTy (@$, $ID, true); }
;
```

où le constructeur de `ast::NameTy` est étendu pour prendre un booléen représentant la constance du type.

6. **Liaison des noms.** Que faut-il changer dans le `Binder` vis-à-vis de `const` ?

Correction: Rien, à condition que les visiteurs par défaut (`ast::DefaultVisitor`, `ast::DefaultConstVisitor`) aient bien été écrits. En effet, l'ajout de `const` ne change pas la façon dont les *noms* sont traités ; c'est lors de l'analyse des types que la constance sera vérifiée.

7. **Vérification des types.** Selon vous, le support de `const` requiert-il des changements au niveau de la vérification des types du compilateur ? Justifiez votre réponse.

Correction: Réponse courte : oui, bien entendu, puisqu'il faut interdire les affectations à des l-values constantes (c'est-à-dire, dont le type contient `const`).

Voici une liste des modifications suggérées :

– Concernant les types du langage, le plus simple est d'équiper la classe `type::Named` d'un champ `const_`, à l'instar de `ast::NameTy` (cf. question 4).

– Les affectations doivent être vérifiées : une valeur de type constant ne peut être une l-value. La méthode `type::TypeChecker::operator() (ast::AssignExp & e)` doit donc s'assurer que l'opérande gauche de l'opérateur ':=' est de type non constant.

– On souhaite aussi propager `const` à l'intérieur des types agrégats (d'un enregistrement à ses champs, d'un tableau à ses cases, d'un objet à ses attributs). L'exemple qui suit illustre ce que l'on voudrait interdire :

```
let
  type rec = { x : int }
  var a : const rec := rec { x = 51 }
in
  /* a.x := -1 */ /* Forbidden, as 'a' is of type 'const rec',
                     so 'a.x' is a 'const int'.  */
end
```

– Il faut aussi propager la constance lors de l'initialisation. L'utilisation d'une valeur constante comme r-value lors de l'initialisation d'une variable (ou d'un argument formel) ou lors de l'affectation d'une l-value (de type non constant) doit être interdite, au moins en ce qui concerne les valeurs manipulées par référence (tableaux, enregistrements, objets, etc.). Considérons l'exemple suivant :

```
let
  type rec = { x : int }
  var a : const rec := rec { x = 51 }
  var b : rec := a /* Danger: 'b' is a mutable alias of 'a'! */
in
  /* a.x := 42 */ /* Forbidden, 'a' is const. */
  b.x := -1       /* Allowed: 'b' is mutable, so 'a.x = -1'!! */
end
```

L'initialisation de 'b' pose problème, puisqu'elle ôte implicitement la constance de 'a'. On souhaite donc proscrire ce genre de constructions.

Par ailleurs, il faut autoriser les comparaisons entre deux valeurs dont les types ne diffèrent que d'un `const`.

Beaucoup de copies avancent que `const` n'a rien à voir avec le typage, mais ne disent pas quand la vérification de la constance devrait être effectuée. On rappelle par ailleurs qu'un exemple de vérification de constance a été vu dans le *type-checker* du compilateur Tiger : il s'agit des indices de boucles `for`, qui doivent être accessible seulement en lecture.

8. **Représentation intermédiaire et traduction de Tiger vers Tree.** Y a-t-il des changements à apporter au langage intermédiaire Tree ou au visiteur chargé de la traduction vers la représentation intermédiaire ? Justifiez votre réponse.

   Correction: Réponse courte : non. En effet, une fois passée l'analyse sémantique, `const` n'a plus de rôle à jouer.

   Réponse plus développée : cependant, on pourrait cependant décider d'*annoter* les sous-arbres correspondant aux nœuds étiquetés par un type `const` dans l'AST, afin d'autoriser certaines optimisations dans le *middle end*. Il est aussi possible d'effectuer ces optimisations en amont (avant la traduction), notamment pour des raisons de simplicité. Cependant, les appliquer sur la représentation intermédiaire présente un intérêt : ces optimisations sont indépendantes des langages sources et cibles du compilateur et donc ré-applicables dans d'autres contextes que la compilation Tiger → MIPS (ou Tiger → IA-32).

   Best-of: Oui probablement car Tigrou vie dans la foret.

2 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. N'y passez pas plus de dix minutes.

Cette épreuve

Q.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.
e. Plus de 120 minutes.

Q.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

Q.3 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.

Q.4 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.

Q.5 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.

Q.6 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 de travail par semaine).
c. Supportable (environ une heure par semaine).
d. Légère (quelques minutes par semaine).

Les formateurs

Q.7 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.

Q.8 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

Q.9 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.

Q.10 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 par semaine).
d. Légère (une ou deux heures par semaine).
e. J'ai été dispensé du projet.

Q.11 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 12-18 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

Q.12 C++.

Q.13 modélisation orientée objet et *design patterns*.

Q.14 anglais technique.

Q.15 compréhension du fonctionnement des ordinateurs.

Q.16 compréhension du fonctionnement des langages de programmation.

Q.17 travail collaboratif.

Q.18 outils de développement (contrôle de version, systèmes de construction, débogueurs, générateurs de code, etc.)

Questions 19-29 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.

Q.19 Rush `.tig` : mini-projet à écrire en Tiger.

Q.20 `PTHL/TC-0`, Scanner & Parser.

Q.21 `TC-1`, Scanner & Parser, Tâches, Autotools.

Q.22 `TC-2`, Construction de l'AST et pretty-printer.

Q.23 `TC-3`, Liaison des noms et renommage.

Q.24 `TC-4`, Typage et calcul des échappements.

Q.25 `TC-D (option)`, Désucrage (boucles `for`, comparaisons de chaînes de caractères).

Q.26 `TC-I (option)`, Mise en ligne du corps des fonctions.

Q.27 `TC-B (option)`, Vérification dynamique des bornes de tableaux.

Q.28 `TC-A (option)`, Surcharge des fonctions.

Q.29 `TC-O (option)`, Désucrage des constructions objets.

A Grammaire du langage Tiger

Cette grammaire utilise le formalisme Extended Backus Naur Form (EBNF), où '[' et ']' sont utilisés pour représenter zéro (0) or une (1) occurrence du terme encadré, tandis que '{' et '}' désignent un nombre arbitraire de répétitions (y compris zéro).

```
⟨program⟩ ::=
    ⟨exp⟩
  | ⟨decs⟩

⟨exp⟩ ::=
  # Literals.
    "nil"
  | integer
  | string

  # Array and record creations.
  | ⟨type-id⟩ "[" ⟨exp⟩ "]" "of" ⟨exp⟩
  | ⟨type-id⟩ "{" [ id "=" ⟨exp⟩ { "," id "=" ⟨exp⟩ } ] "}"

  # Object creation.
  | "new" ⟨type-id⟩

  # Variables, field, elements of an array.
  | ⟨lvalue⟩

  # Function call.
  | id "(" [ ⟨exp⟩ { "," ⟨exp⟩ } ] ")"

  # Method call.
  | ⟨lvalue⟩ "." id "(" [ ⟨exp⟩ { "," ⟨exp⟩ } ] ")"

  # Operations.
  | "-" ⟨exp⟩
  | ⟨exp⟩ ⟨op⟩ ⟨exp⟩
  | "(" ⟨exps⟩ ")"

  # Assignment.
  | ⟨lvalue⟩ ":=" ⟨exp⟩

  # Control structures.
  | "if" ⟨exp⟩ "then" ⟨exp⟩ [ "else" ⟨exp⟩ ]
  | "while" ⟨exp⟩ "do" ⟨exp⟩
  | "for" id ":=" ⟨exp⟩ "to" ⟨exp⟩ "do" ⟨exp⟩
  | "break"
  | "let" ⟨decs⟩ "in" ⟨exps⟩ "end"

⟨lvalue⟩ ::= id
  | ⟨lvalue⟩ "." id
  | ⟨lvalue⟩ "[" ⟨exp⟩ "]"

⟨exps⟩ ::= [ ⟨exp⟩ { ";" ⟨exp⟩ } ]
```

```
⟨decs⟩ ::= { ⟨dec⟩ }
⟨dec⟩ ::=
  # Type declaration.
    "type" id "=" ⟨ty⟩
  # Class definition (alternative form).
  | "class" id [ "extends" ⟨type-id⟩ ] "{" ⟨classfields⟩ "}"
  # Variable declaration.
  | ⟨vardec⟩
  # Function declaration.
  | "function" id "(" ⟨tyfields⟩ ")" [ ":" ⟨type-id⟩ ] "=" ⟨exp⟩
  # Primitive declaration.
  | "primitive" id "(" ⟨tyfields⟩ ")" [ ":" ⟨type-id⟩ ]
  # Importing a set of declarations.
  | "import" string

⟨vardec⟩ ::= "var" id [ ":" ⟨type-id⟩ ] ":=" ⟨exp⟩

⟨classfields⟩ ::= { ⟨classfield⟩ }
# Class fields.
⟨classfield⟩ ::=
  # Attribute declaration.
    ⟨vardec⟩
  # Method declaration.
  | "method" id "(" ⟨tyfields⟩ ")" [ ":" ⟨type-id⟩ ] "=" ⟨exp⟩

# Types.
⟨ty⟩ ::=
   # Type alias.
     ⟨type-id⟩
   # Record type definition.
   | "{" ⟨tyfields⟩ "}"
   # Array type definition.
   | "array" "of" ⟨type-id⟩
   # Class definition (canonical form).
   | "class" [ "extends" ⟨type-id⟩ ] "{" ⟨classfields⟩ "}"
⟨tyfields⟩ ::= [ id ":" ⟨type-id⟩ { "," id ":" ⟨type-id⟩ } ]
⟨type-id⟩ ::= id

⟨op⟩ ::= "+" | "−" | "*" | "/" | "=" | "<>" | ">" | "<" | ">=" | "<=" | "&" | "|"
```

Les priorités des opérateurs binaires (op), de la plus haute à la plus basse, sont comme suit :

```
* /
+ -
>= <= = <> < >
&
|
```

Les opérateurs de comparaison (<, <=, =, <>, >, >=) ne sont pas associatifs. Tous les autres opérateurs listés dans op sont associatifs à gauche.

B 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     ()
    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)

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

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