Correction du Partiel CMP1

EPITA – Apprentis promo 2010
Tous documents (notes de cours, polycopiés, livres) autorisés

Février 2008 (1h00)

Correction: Le sujet et sa correction ont été écrits par Akim Demaille et 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**.

1. La surcharge de fonctions en C++ est un mécanisme qui dépend des types à l'exécution. Vrai ou faux ?

   Correction: C'est faux: c'est un mécanisme statique, qui dépend des types *statique* des arguments effectifs, résolu à la compilation.

2. L'utilisation de fonctions virtuelles en C++ est incompatible avec la compilation séparée. Vrai ou faux ?

   Correction: Faux. Le modèle de compilation du C++ permet la compilation séparée. L'implémentation des méthodes polymorphes repose habituellement sur un mécanisme de *tables de fonctions virtuelles*, et sont extensibles sans intrusion.

3. Parmi ces propositions, laquelle (lesquelles) est (sont) un pré-requis pour le typage statique fort dans un langage orienté objet ?

   (a) une liaison des noms résolue à la compilation ;

   (b) l'absence d'instructions de transtypage (*cast*) ;

   (c) l'absence de méthodes polymorphes abstraites (appelées fonctions membres virtuelles pures en C++) ;

   (d) la présence automatique d'une surclasse au sommet de toute hiérarchie de classes (`Object`, par exemple).

Correction: Seule les réponses 3a et 3b sont valides.

- 3a est bien sûr un pré-requis fort : sans liaison des noms statique, pas de typage statique fort ;
- avec les casts, on introduit une une faille dans le système de typage ;
- la présence de méthodes polymorphes abstraites ne pose aucun problème du point de vue du typage : dit grossièrement, le polymorphisme d'inclusion garantit que les objets manipulés fournissent les interfaces attendues ;
- 3d est indépendant de la sûreté du typage : on pourrait par exemple s'en passer dans Tiger.

4. Quelle est le type (de Chomsky) du langage engendré par

$$S \to X|Y \qquad X \to Zp \qquad Y \to o \qquad Z \to pS$$

Correction: Ce langage est $p^n o p^n, n \in \mathbb{N}$ bien connu pour être hors-contexte (type 2), et non rationnel.

Best-of:

- C'est un langage irrationnel.
- Cette grammaire n'est pas de type généraliste car elle comporte des terminaux. [...] Enfin, c'est une grammaire rationnelle linéaire à gauche.

2 Langages, grammaires et compilation

1. Quel est l'avantage de l'algorithme GLR sur LALR ?

Correction: Toutes les grammaires hors-contexte sont acceptées, ce qui permet de ne pas avoir à contourner les conflits dus aux limitations de LALR(1).

Best-of:

- LALR a besoin de passer par LL dans certaines situations.

2. Pourquoi est-ce aussi un désavantage dangereux ?

Correction: Mais on se retrouve à accepter des conflits sans nécessairement les comprendre. Autant LALR(1) est pénible, mais quand ça compile, il ne reste plus de doute (autre, bien sûr, que des bugs dans la grammaire), autant en GLR on peut découvrir tardivement des ambiguïtés pas prévues.

Best-of:

- Parce que Chuck Norris est en opposition sur la case 42.

3. Que signifie AST en anglais et en français ?

Correction: Abstract syntax tree, arbre de syntaxe abstraite. Il s'agit surtout de vérifier le féminin en français : c'est bien sûr la syntaxe qui est abstrait, pas l'arbre (bien concret lui).

4. Donner la syntaxe abstraite (soit sous la forme d'une grammaire, soit sous la forme d'une hiérarchie orientée objet typée) de la grammaire suivante.

```
<term> ::= <term> ( <term> )        -- Application
         | λ <var> . <term>         -- Abstraction
         | <var>                    -- Variable
<var>  ::= <identifier>
```

On utilise -- pour introduire des commentaires qui bien sûr n'appartiennent pas à la grammaire.

Correction:

```
<term> ::= Application ( <term>,  <term> )
         | Abstraction ( <var>,  <term>)
         | Variable ( <var> )
<var>  ::= NamedVariable ( <identifier> )
```

ou encore

```
<term> ::= Application ( <term>,  <term> )
         | Abstraction ( <identifier>,  <term>)
         | Variable ( <identifier> )
```

qui est moins bon: moins typé.
En terme de hiérarchie, ce serait quelque chose comme

```
/Term/
   Application (Term, Term)
   Abstraction (Var,  Term)
   Variable (Var)
Var (Symbol)
```

5. Pourquoi un langage comme le C inclut les "prédéclarations", et pas un autre comme Java ?

Correction: Les prédéclarations permettent de faire une simple passe sur l'AST, alors que les langages comme Java et Tiger nécessitent plusieurs passes, ce qui implique d'avoir à conserver une grande part du programme (typiquement sous la forme d'un AST), quelque chose qui est bien trop coûteux au temps de C, et de parfaitement admissible à celui de Java.

Best-of:

- [...] Tiger et Java nécessitent plusieurs passes, ce qui implique d'avoir une copie du programme, qui est extrêmement coûteux en C mais pas en Java.
- Parce qu'en C on a un langage compilé et en Java un langage interprété.

6. Étant donné que le programme Tiger suivant est valide, expliquer quelle politique de gestion mémoire le compilateur doit utiliser pour les types.

```
let
 var box :=
  let
   type box = { val: string }
  in
   box { val = "42\n" }
  end
in
```

```
 print (box.val)
end
```

Correction: Très manifestement le compilateur aura besoin d'information à propos du type `box` après être sorti de la portée qui le définit. Par conséquent les descriptions des types ne peuvent pas être désaloué simplement au sortir de la portée qui les définit. Une solution simple est de désalouer à la fin de la passe de typage.

7. Soient deux fractions $\frac{a}{b}, \frac{c}{d}$, à quelle surprise s'expose-t-on si l'on programme $\frac{a}{b} \leq \frac{c}{d}$ comme `(a/b) <= (c/d)`.

Correction: Les flottants ne sont pas nos amis, et il est possible que `(1/3) < (1/3)` s'évalue à vrai.

Best-of: Diviser par 0 ; que a/b et c/d ne soient pas du même type.

3 Un peu de C++

1. Qu'affiche le programme suivant :

```
#include <iostream>
struct Foo
{
 void m1 () const
 {
  std::cout << "Foo::m1()" << std::endl;
 }
 virtual void m2 () const
 {
  std::cout << "Foo::m2()" << std::endl;
 }
};
struct Bar : public Foo
{
 void m1 () const
 {
  std::cout << "Bar::m1()" << std::endl;
 }
 virtual void m2 () const
 {
  std::cout << "Bar::m2()" << std::endl;
 }
};
int main ()
{
 const Bar& a = Bar (); a.m1 (); a.m2 ();
 const Foo& b = Bar (); b.m1 (); b.m2 ();
 const Foo& c = Foo (); c.m1 (); c.m2 ();
}
```

Correction:

```
Bar::m1()
Bar::m2()
Foo::m1()
Bar::m2()
Foo::m1()
Foo::m2()
```

2. Écrire un *class template* `Pair` qui stocke deux éléments, `a` et `b`, d'un même type, avec un constructeur prenant les deux valeurs.

Correction: Commencez par remarquer que j'ai écrit "**un** class template", et non pas "une" : c'est normal, puisqu'ici le nom est "template" et l'adjectif "class". Puisque l'on dit "un template", on dit "un class template" de la même façon qu'on dit "un patron de classe" et non une "patron de classe".
Par exemple:

```
template < typename T >
class Pair
{
public:
  Pair (const T &a, const T &b): _a (a), _b (b) {}
private:
  T _a, _b;
};
```

3. Utiliser la classe précédente pour déclarer une variable `pi` qui contienne la paire $(3, 14159)$.

Correction:

```
Pair <int> pi (3, 14159);
```

4. Qu'est-ce qu'un `const_iterator` dans STL ?

Correction: Un itérateur en lecture seulement, i.e., un itérateur qui pointe vers les éléments du conteneur en lecture seulement. Si `i` est un `const_iterator`, alors `*i` est `const`.

Best-of:
- C'est un iterator qui pointe toujours sur le même élément, et n'est pas modifiable.

5. Écrire une fonction `product` qui prenne une liste STL de `float` par référence, et en retourne le produit des éléments. On prendra garde à la constance et au `namespace`.

Correction:

```
float
product (const std::list<float> &l)
{
  float res = 1.f;
  for (std::list<float>::const_iterator i = l.begin ();
    i != l.end (); ++i)
    res *= *i;
  return res;
}
```