IFIPS – Cycle Ingénieur 2e année Informatique
Compilation 2006–2007

Travaux Dirigés n^{o}4
Analyse sémantique

1 Surcharge à la Java

Dans Java, la surcharge est résolue statiquement en ne prenant en compte que le type des arguments de la méthode. À cause du sous-typage, plusieurs méthodes peuvent s'appliquer aux mêmes arguments. L'ambiguité est levée s'il existe parmi toutes les méthodes applicables exactement une méthode dont le profil est plus petit que tous les autres. Le profil d'une méthode à n arguments est formé de la classe dans laquelle la méthode est déclarée et des types des arguments. Un profil $(C, \tau_1, \ldots, \tau_n)$ est plus petit qu'un profil $(D, \sigma_1, \ldots, \sigma_n)$ si $n = p$, que C est une sous-classe de D et que chaque τ_i est un sous-type de σ_i.

Parmi les exemples suivants, dire s'ils sont correctement formés et si oui indiquer à quelle déclaration fait référence chaque utilisation de méthode.

1.
```
class A {
    void g(A a) { g(a,a); }
    void g(A a, A b) {g(a); }
    void g(B b, B a) {g(a); }
}
class B extends A {
    void g(B b) { g(b,b); }
    void g() { g(this); }
    void g(int n) { g(new A()); }
    }
```
2.
```
class A { }
class B extends A {
    void g() { g(this,this); }
    void g(A a, B b) { }
    void g(B b, A a) { }
}
```
3.
```
class A {
    void f(int x) { }
}
class B extends A {
    void f(double x) { }
}
class Main {
    static void main() { new B().f(2); }
}
```
4.
```
class A {
    void f(B b) {  }
```

```
}
class B extends A {
    void f(A a) {  }
}
class Main {
    static void main(A a) { a.f(a); }
}
```

2 Typage d'un fragment de Java

On considère le fragment très simple de Java ne comprenant que des expressions entières et booléennes et des instructions uniquement formées d'affectations, de tests, de blocs et de variables locales. La syntaxe abstraite de ce fragment est la suivante :

$$\begin{array}{lcl} e & ::= & x \mid n \mid e\texttt{+}e \mid \texttt{-}e \mid e\texttt{==}e \mid e\texttt{>}e \\ s & ::= & x\texttt{=}e \mid \texttt{if}\ (e)\ s\ \texttt{else}\ s \mid \{s; \dots s;\} \mid \texttt{int}\ x\texttt{=}e \end{array}$$

On se donne des classes Java pour représenter cette syntaxe abstraite, de la manière suivante :

```
abstract class Expr { ... }
class Eident extends Expr { public String id; ... }
class Econst extends Expr { public int val; ... }
class Eplus  extends Expr { public Expr e1,e2; ... }
class Eneg   extends Expr { public Expr e1; ... }
class Eequal extends Expr { public Expr e1,e2; ... }
class Egt    extends Expr { public Expr e1,e2; ... }

abstract class Statement { ... }
class Sassign extends Statement { String id; Expr e; ... }
class Sif     extends Statement { Expr e1; Statement e2,e3; ... }
class Sbloc   extends Statement { Vector sl; ... }
class Svar    extends Statement { String id; Expr e; ... }
```

et on suppose défini un visiteur pour cette syntaxe abstraite, sous la forme d'une interface `Visitor` contenant une méthode `void visit(`C `e)` pour chaque classe C ci-dessus (elle-même étant munie d'une méthode `void accept(Visitor v)`).

On suppose églement donnée une classe `Table` réalisant des tables de hachage avec chaînage externe (analogues à celles vues en cours mais ici sous la forme d'un ensemble et non d'un dictionnaire) et présentant l'interface suivante :

```
class Table {
  Table();
  void put(String s);
  boolean contains(String s);
  void remove(String s);
}
```

Écrire une classe `Typage` réalisant le typage de ce fragment de Java sous la forme d'un visiteur.