Master d'informatique | **module TEP** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2005-2006**

Examen du 13 novembre 2006

1 - λ-calcul pur : listes

Le codage suivant des listes s'inspire des entiers de Church :
- `[]` = $\lambda gy.y$
- `[a1; ...; an]` = $\lambda gy.ga_1(...(ga_n y)...)$

1. Ecrire le λ-terme correspondant à la liste `[1;2]` avec `1` et `2` entiers de Church.
2. Ecrire les fonctions `::` (constructeur), `car` (tête de liste) et `cdr` (queue de liste) sur des listes non vides qui vérifient les propriétés suivantes à la manière des couples :
   - `car (a :: b)` donne `a`
   - `cdr (a :: b)` donne `b`

   On utilisera le constructeur `::` sous forme infixe à la manière d'O'Caml : `a :: b` correspond à la liste dont `a` est la tête et `b` la queue.
3. Ecrire la fonction `longueur` :
   - $longueur([]) = 0$
   - $longueur(x :: l) = 1 + longueur(l)$

   On utilisera les entiers de Church et les opérations associées si nécessaire.

2 - Typage : expressions expansives

Soit les types suivants :

```
type 'a t =
  F of ( 'a -> 'a )
| V of 'a
| R;;
```

Indiquez, quand ils existent, les types dans les déclarations suivantes, en indiquant dans chaque cas pourquoi sont-ils ou non généralisés. Si le typage échoue indiquez en la raison.

1. let fg = F (fun x -> x) ; ;
2. let fv = V [] ; ;
3. let fr = R ; ;
4. ```
   let f1 t x = match t with
       F f -> V (f x)
     | V y -> V x
     | R -> V x;;
   ```
5. let f2 = f1 (F (fun x -> x)) [] ; ;
6. let f3 = f1 R R ; ;
7. let f4 = f1 (F f1) ; ;
8. let f5 = f1 (V f1) ; ;
   ```

3 - Héritage et sous-typage

Soit le programme O'Caml suivant :

```
class a (x:int) =
  object(self : 'a)
    val mutable lx = x
    method getx () = lx
    method eq (z : 'a) = lx == z#getx()
  end;;

class b x (y:int) =
  object (self)
    inherit a x as super
    val mutable ly = y
    method gety () = ly
    method eq (z : 'a) = (super#eq(z)) && (ly == z#gety())
  end;;

class c (x:int) (y:int)  =
  object(self:'a)
    val mutable la = x
    val mutable lb = y
    method getx () = la
    method gety () = lb
    method eq (z : 'a) = (la == z#getx()) && (lb == z#gety())
  end;;

let f x y = x#getx() + y#gety();;
let g x y = if x#eq(y) then x else y;;
let va = new a 8;;
let vb = new b 2 4;;
let vc = new c 1 9;;
let va1 = (vb :> a);;
let va2 = (vc :> a);;
let r1 = f va;;
let r2 = f vb;;
let r3 = g va ;;
let r4 = g va (vb :> a);;
let r5 = g vb vc;;
let r6 = g vb;;
```

1. Indiquez les relations d'héritage et de sous-typage des trois classes `a`, `b` et c
2. Indiquez, quand ils existent, les types dans des déclarations globales du programme précédent. Si le typage échoue indiquez en la raison.

4 - Surcharge et liaison tardive en Java

Dans le cadre de traitements extensibles sur les formules logiques du calcul propositionnel on utilise le modèle de conception « Visiteur ». On se donne tout d'abord les classes suivantes pour la description des formules :

Formule.java

```
class Formule {
    void accepte(Visiteur v) {v.visite(this);}
}
```

Opbin.java

```
class OpBin extends Formule {
  Formule fg, fd;
  Formule sous_formule_g(){return fg;}
  Formule sous_formule_d(){return fd;}
}
```

Constante.java

```
class Constante extends Formule {
  boolean  b;

  Constante(boolean b) {this.b = b;}
  Constante() {this.b = false;}

  boolean valeur(){return b;}

%  void  accepte(Visiteur v) {v.visite(this);}
}
```

Et.java et Ou.java

```
class Et extends OpBin {
  Et(Formule fg, Formule fd){
    this.fg = fg; this.fd = fd;
  }
%  void accepte(Visiteur v){v.visite(this);}
}

class Ou extends OpBin {
  Ou(Formule fg, Formule fd){
    this.fg = fg; this.fd = fd;
  }
%  void accepte(Visiteur v){v.visite(this);}
}
```

Non.java

```
class Non extends Formule {
  Formule f;
  Non(Formule f) {this.f = f;}
  Formule sous_formule(){return f;}

%  void accepte(Visiteur v) {v.visite(this);}
}
```

Var.java

```
class Var extends Formule {
  String v;
  Var(String v){this.v = v;}
  String ident(){return v;}
%  void accepte(Visiteur v){v.visite(this);}
}
```

et la classe abstraite `Visiteur.java` :

```
abstract class Visiteur {
  abstract void visite(Constante c);
  abstract void visite(Non n);
  abstract void visite(Et e);
  abstract void visite(Ou o );
  abstract void visite(Var v );
}
```

Un premier réflexe quand on lit ces classes est de vouloir factoriser la méthode `accepte` dans les classes de la hiérarchie `Formule` et dérivées. Pour cela on remplace la classe abstraite Formule par la classe concrète suivante :

```
class Formule {
  void accepte(Visiteur v) {v.accepte(this);}
}
```

et l'on supprime les définitions de la méthode `accepte` dans les classes dérivées de Formule.

1. Dans ce cadre indiquez en le justifiant ce qui se passe à la compilation.
2. Proposez une solution permettant d'implanter cette factorisation. Comparez votre proposition avec la solution initiale (sans factorisation).

5 - Sous-typage et classes paramétrées

Soit le programme suivant :

```
import java.util.*;

class P {
    int x;
    void set(int x){this.x=x;}
    int get(){return x;}
}

class PC extends P {
    String c;
    void setColor(String c){this.c = c;}
    String getColor(){return c;}
}


class ex {

  public static void main(String[] args) {
      P p = new P();
      PC pc = new PC();

   Vector<P> ap = new Vector<P>(5);
   Vector<PC> bp = new Vector<PC>(5);
   Vector<P> cp = (Vector<P>)bp;
   pc.setColor("ROUGE");

   // Q1
   ap.add(0,p);
   bp.add(0,p);

   // Q2
   ap.add(1,pc);
   bp.add(1,pc);

   // Q3
   cp.add(2,p);
   cp.add(3,pc);

   // Q4
   System.out.println(bp.elementAt(0).getColor());
   System.out.println(bp.elementAt(1).getColor());
   System.out.println(bp.elementAt(2).getColor());
   System.out.println(bp.elementAt(3).getColor());
  }
}
```

1. Indiquez en les expliquant les deux erreurs de compilation de ce programme.
2. Indiquez en quoi cela serait dangereux de pouvoir compiler et donc exécuter ce programme.