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

Examen du 14 novembre 2005

1 - λ-calcul pur : stratégie d'évaluation

Q1 Soient les λ-termes suivants :
$T = \lambda xy.y$
$\Delta = \lambda x.xx$
$I = \lambda x.x$
$Q = T\ (\Delta\Delta)\ I$

1. Réduisez quand cela est possible le λ-terme Q suivant la stratégie d'évaluation retardée (l'argument est passé tel quel à la fonction) et la stratégie d'évaluation immédiate (l'argument est évalué avant d'être passé à la fonction).
2. Quelle est la stratégie d'évaluation d'Objective Caml ?
3. Le terme Q est-il typage en Objective Caml ? Pourquoi ?

2 - Typage : partage d'environnement et fonctionnelles

1. Donner le type O'Caml, s'il existe, des déclarations O'Caml suivantes :

```
let make_gensym s =
    let c = ref 0 in
    let g facc  = facc s c  in g ;;

let r = make_gensym "TOTO";;

let f_reset s c = c:=0; s;;

let f_new s c = incr c; s^(string_of_int !c);;
```

2. Calculer les valeurs des expressions O'Caml suivantes en justifiant les résultats obtenus :

```
r f_new;;
r f_new;;
r f_new;;
r f_reset;;
r f_new;;
r f_new;;
```

3. Indiquer en le justifiant le type des déclarations et expressions suivantes :

```
let f_mystere s c = c:= !c + 2;  c;;
r f_mystere;;
```

3 - Surcharge et liaison tardive en Java

On cherche à se doter de traitements extensibles sur les formules logiques du calcul propositionnel en utilisant le modèle de conception « Visiteur ».

On se donne tout d'abord les classes suivantes pour la description des formules :

Formule.java

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

Opbin.java

```
abstract 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 );
}
```

1. Construire troix formules simples suivantes :
   (a) f_1 = (FAUX OU (VRAI ET FAUX))
   (b) f_2 = ((NON a) ET b) OU (a ET (NON b))
   (c) f_3 = (((NON a) OU a) OU ((NON a) ET a))

   Les constantes VRAI et FAUX seront représentées par une instance de `Constante` ; les opérateurs NON, ET et OU correspondront à des instances des classes de même nom ; les variables sont représentées par des instances de la classe `Var` dont la variable d'instance contient la chaîne désirée.

   Pour cela il vous est demandé d'écrire une classe `Question1` contenant les trois méthodes statiques de signatures suivantes :

```
static Formule f1();
static Formule f2();
static Formule f3();
```

Les méthodes statiques `f1`, `f2` et `f3` retournent respectivement les formules f_1, f_2 et f_3 décrites ci-dessus.

2. On se donne un visiteur de conversion en chaîne de caractères `VisiteurTS` et du programme `Exemple.java` qui l'utilise. Notez que c'est au *visiteur* de gérer l'accumulation des résultats intermédiaires.

VisiteurTS.java

```
class VisiteurTS extends Visiteur {
  String res;
  VisiteurTS(){res="";}
  String get_res(){return res;}
  void set_res(String s){res=s;}

  void visite(Constante c){res=res+c.valeur();}

  void visite(Non n){
    res=res+"!(";
    n.sous_formule().accepte(this);res=res+")";
  }

  void visite(Et e){
    res=res+"(";
    e.sous_formule_g().accepte(this);
    res=res+" ^ ";
    e.sous_formule_d().accepte(this);
    res=res+")";}

  void visite(Ou o){
    res=res+"(";
    o.sous_formule_g().accepte(this);
    res=res+" v ";
    o.sous_formule_d().accepte(this);
    res=res+")";}

  void visite(Var v){res=res+v.ident();}
}
```

Exemple.java

```
class Exemple {
  public static void main(String[] args) {
    VisiteurTS v1 = new VisiteurTS();
    VisiteurTS v2 = new VisiteurTS();

    Formule f1 = Question1.f1();
    Formule f2 = Question1.f2();
    Formule f3 = Question1.f3();

    f1.accepte(v1);
    System.out.println(v1.get_res());
    f2.accepte(v2);
    System.out.println(v2.get_res());
    v2.set_res("");
    f3.accepte(v2);
    System.out.println(v2.get_res());
  }
}

/* Trace de java Exemple

(false v (true ^ false))
((!(a) ^ b) v (a ^ !(b)))
((!(a) v a) v (!(a) ^ a))
*/
```

Donnez la trace des appels aux différentes méthodes de l'expression `f2.accepte(v2)` de la méthode `main` de la classe `Exemple`.

3. On retrouve la définition de la méthode `accepte` dans les classes concrètes des formules (Constante, ET, Ou, Non et Var). Que se passe-t-il si on la définit de manière concrète dans la classe ancêtre `Formule`. Justifier votre réponse au niveau du typage et de l'exécution.

4. Dans un langage sans surcharge, il devient nécessaire de renommer les méthodes `visite` en fonction du paramètre discriminant. Par exemple `visite(Non)` devient `visite_non(Non)`. Modifier le programe présenté pour tenir compte de ce renommage. Vous pouvez utiliser une pseudo syntaxe objet à la Java ou à la Caml.

4 - Classes paramétrées

Soient les classes O'Caml suivantes :

```
class virtual obs_abs () =
object
  method virtual miseajour : unit -> unit
end;;

class  sujet_gen () =
object
  val mutable lobs = ([] : obs_abs list)
  method  attache o = lobs <- o :: lobs
  method  detache o =
    lobs <- List.filter (fun x -> x <> o) lobs
  method  private notify () =
     List.iter (fun x -> x#miseajour()) lobs
end;;
```

```
class ['a] sujet (x:'a) =
object
  inherit sujet_gen ()
  val mutable etat = x
  method acqEtat  = etat
end;;

class ['a] obs (x : 'a sujet) =
object(self)
  inherit obs_abs ()
  val s = x
  val mutable etat = x#acqEtat
  initializer  s#attache(self :> obs_abs)
  method miseajour () = etat <- s#acqEtat
end;;
```

Quand vous écrivez une classe Java, indiquer s'il y a des "warnings" à la compilation.

1. Donner le type O'Caml des classes `obs_abs`, `sujet_gen`, `['a] sujet`, `['a] obs`.
2. Ecrire quatre classes équivalentes, paramétrée si besoin est, en Java (1.5). Pour les listes vous pouvez utiliser la classe `ArrayList<E>` de l'API Java.
3. Ecrire une méthode `main` équivalente à la fonction `main` O'Caml suivante :

```
let main () =
  let s = new sujet (0.0, 0.0) in
  let o1 = new obs s in
  let o2 = new obs s in
    s#attache(o1);
    s#attache(o2);;
```