Master d'informatique	**module TEP**	**Université Pierre et Marie Curie**
Documents autorisés	*Durée 2 heures*	**Année 2009-2010**

Examen du 16 novembre 2009

1 - λ-calcul et codage de structures : listes

On cherche à coder les listes à partir des couples et d'une valeur marquant la fin de liste appelée `nil`.

On utilisera les définitions des couples (créateur et accesseurs) suivantes :

- make_couple = $\lambda xyf.f\ x\ y$
- fst = $\lambda z.z(\lambda xy.x)$
- snd = $\lambda z.z(\lambda xy.y)$

Une liste est soit une liste vide (`nil`), soit un paire pointée (couple) contenant deux éléments : la tête et la queue qui est elle aussi une liste.

On codera `nil` par l'identité ($\lambda x.x$). Cette valeur ne pourra pas être utilisée comme tête de liste.

1. Ecrire le terme L correspondant à la liste [A ;B ;C] où A, B et C sont trois termes quelconques.
2. Ecrire le terme is_nil qui teste si une liste est vide en retournant $True$ ($\lambda xy.x$) si c'est le cas, et le premier élément de la liste sinon. Appliquer ce terme à L et à `nil`.
3. Ecrire la fonction `nth` qui prend une liste et un entier, et retourne le ième élément de la liste s'il existe. On pourra utiliser les entiers de Church pour coder les entiers, ou tout autre codage de votre convenance.
4. Ecrire la fonction `longueur` :

   - $longueur(nil) = 0$
   - $longueur(make\_couple\ x\ l) = 1 + longueur(l)$

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

2 - Typage : fonctionnelles et mutables en Objective Caml

On se donne le type suivant :

```
type 'a ensemble = {mutable racine : 'a ; mutable succ : 'a -> 'a}
```

et la fonction `next`, de type `'a ensemble -> 'a` qui retourne la racine d'un ensemble et calcule la nouvelle racine :

```
let next e =
  let r = e.racine in
  let n = e.succ r in
    e.racine <- n ; r ;;
```

Donner le type, s'il existe, des déclarations et expressions Objective Caml suivantes et indiquer les erreurs de typage, quand elles ont lieu, en précisant la raison ainsi que les valeurs que retournent les expression bien typées :

1.
```
let my_tl l = match l with [] -> [] | h::t -> t;;
let ens2 = {racine = []; succ = my_tl};;
ens2.racine <- [1; 2; 3];;
next ens2;;
ens2.racine <- [true; false];;
next ens2 ;;
```

2.
```
let app f x = f x;;
let ens3 = {racine = []; succ = app my_tl};;
ens3.racine <- [1; 2; 3];;
next ens3;;
ens3.racine <- [true; false];;
next ens3 ;;
```

3.
```
let rec longueur l = match l with [] -> 0 | _::t -> 1 + longueur t;;
let f g c = match c with ([],_) -> c | (h::t, l2) -> (t, (g h)::l2);;
let ens4 = {racine = ([],[]); succ = f longueur };;
ens4.racine <- [true; false];;
next ens4;;
ens4.racine <- [2;3;4];;
next ens4;;
```

3 - Explicitation de la liaison tardive

On cherche à écrire son propre mécanisme de sélection de code (dispatch) lors de l'appel d'une méthode pour des langages objets à la Java ou Objective Caml. On utilisera un langage (ou sous-ensemble d'un langage) sans surcharge. On cherche à expliciter le choix de la méthode à exécuter. On utilisera deux opérateurs :

- cc(o) qui donne la classe de construction de l'objet o,
- equalClass qui prend deux classes et retourne vrai si elles sont égales et faux sinon.

On calculera à chaque appel la classe de construction de l'objet receveur (sur lequel la méthode s'applique).

Voici un exemple simple d'une hiérarchie de classes qui définit une classe A sous-classe de Object. La classe A définit 2 méthodes :

- une définition de la méthode `int m(A x)`
- une rédéfinition de la méthode `Object clone()`

Les appels suivants de `m` et `clone` :

```
A a = new A();  /* 1*/
a.m(a);         /* 2 */
a.clone();      /* 3 */
```

deviennent alors

```
/* 1 */
A a = new A();


/* 2 */
Class coa1 = cc(a);
if equalClass(coa1,C_A) { appel de A.m}
```

```
else { erreur }

/* 3 */
Class coa2 = cc(a);
if equalClass(coa2,C_A) { appel de A.clone }
else if equalClass(coa2,C_Object) { appel de Object.clone}
else { erreur }
```

On nomme `C_A` la classe représentant la classe A. On teste l'égalité uniquement des classes ayant une définition de la méthode possédant une signature compatible, comme il n'y a pas de surcharge cela correspond aux classes qui définissent ou redéfinissent cette méthode.

Cette opération d'expansion s'effectue au moment de l'édition de liens pour avoir l'ensemble de la hiérarchie de classes connues.

Soit la hiérarchie de classes suivante :

```
/* hiérarchie */
class A extends Object {
  Object clone() { ... }
  int m(A x) { ... }
  int p(A x, B y) { ... }
}
class B extends A {
  int p(A x , B y) { ... }
  int r(B z) { ...}
}
```

1. Indiquer les expansions de code effectuées, sur les instructions suivantes de la méthode `main` :

   ```
   /* main */
   A a = new A();
   B b = new B();
   A a2 = (A)b;;
   a.m(a);
   b.p(a,b);
   b.m(a);
   a2.p(a,b);
   ```

2. On ajoute une classe `C`, qui étend la classe `B`, qui redéfinie la méthode `m`. Indiquer ce qui doit changer dans l'expansion pour tenir compte de l'ensemble de la hiérarchie de classe.

3. Indiquer les expansions de code effectuées, sur la suite d'instructions suivante :

   ```
   C c = new C();
   B b2 = (B)c;
   a2.m(a);
   b2.m(b2);
   c.m(c);
   ```

4. Indiquer les méthodes appelées sur le code des questions précédentes.

5. Indiquer ce qui changerait à cette explicitation de code si on introduit un mécanisme de surcharge à la Java.

4 - Classes paramétrées en Java

Soit l'interface `Comparable` suivante :

```
public interface Comparable <T> {
 int  compareTo(T o)
}
```

La méthode `compareTo` retourne une valeur négative, nulle ou positive si l'objet receveur est plus petit, égal ou supérieur que l'objet passé en paramètre.

1. Ecrire une classe `Naturel` pour les entiers naturels qui implante l'interface `Comparable <T>`.
2. Ajouter à cette classe une méthode `CompareTo Object o` qui prend un objet en paramètre, et effectue la comparaison précédente.
3. Dans la classe `Main` (contenant une méthode `main`) écrire une méthode statique générique `min` qui calcule le minimum d'un objet de type `ArrayList<T>`.
4. Ecrire une classe `Relatif`, sous-classe de `Naturel`, pour les entiers relatifs en redéfinissant les méthodes `CompareTo`.
5. Indiquer le typage de chaque instruction suivante en précisant s'il y a lieu la raison des erreurs de typage.

```
ArrayList<Naturel> aln = new ArrayList<Naturel>();
aln.add(new Naturel(1)); aln.add(new Naturel(2)); aln.add(new Naturel(3));
int min = aln.min();
ArrayList ale = aln;
int min = ale.min();
ArrayList<Relatif> alr = new ArrayList<Relatif>();
alr.add(new Relatif(1)); alr.add(new Relatif(-2)); alr.add(new Relatif(3));
int min = alr.min();
ArrayList ale2 = alr;
boolean b =  (ale == ale2);
alr.add(new Naturel(4));
ale.add(new Relatif(-3));
ArrayList<Naturel> aln2 = alr;
aln2.add(new Naturel(5));
aln2.add(new Relatif(-5));
```