Université Pierre et Marie Curie
DEUG MIAS première année
Informatique-1

Année 2002-2003
Examen septembre
1 septembre 2003
2 heures

Remarques :

- Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Scheme.
- Les questions peuvent être résolues de façon indépendante. Il est possible, voire même utile, pour répondre à une question, d'utiliser les fonctions qui sont l'objet des questions précédentes.
- La clarté des réponses et la présentation des programmes seront appréciées. Toutes les fonctions apparaissant dans les réponses doivent être accompagnées de leur spécification (lorsqu'elle n'est pas donnée dans le sujet).
- La note tiendra compte de l'efficacité de vos programmes.
- Le barème (total sur 50) n'est donné qu'à titre indicatif.

Exercice I : fonctions sur les listes

Question 1 (4 points) :

Écrire une définition et une expression de test de la fonction `tous-egaux-a?` de spécification :

```
;;; tous-egaux-a? : LISTE[alpha] * alpha -> bool
;;; (tous-egaux-a? L elt) rend #t ssi tous les éléments de la liste «L» sont égaux
;;; à la valeur de «elt». La fonction «tous-egaux-a?» rend, bien sûr, #t lorsque la
;;; liste «L» est vide.
```

Question 2 (4 points) :

En utilisant la fonction `tous-egaux-a?`, écrire une définition de la fonction `tous-egaux?` spécifiée par :

`;;;` *tous-egaux? : LISTE[alpha] -> bool*
`;;;` *(tous-egaux? L) rend #t ssi tous les éléments de la liste «L» sont égaux.*
`;;;` *La fonction «tous-egaux?» rend, bien sûr, #t lorsque la liste «L» est vide.*

Quelle est la valeur de l'application `(tous-egaux? '(a))` ?

Question 3 (5 points) :

Écrire une définition de l'itérateur `verifient-tous?` spécifié par :

`;;;` *verifient-tous? : (alpha -> bool) * LISTE[alpha] -> bool*
`;;;` *(verifient-tous? test? L) rend #t ssi tous les éléments de la liste «L»*
`;;;` *satisfont «test?». La fonction «verifient-tous?» rend bien sûr #t lorsque*
`;;;` *la liste donnée est vide.*

Par exemple :

```
(verifient-tous? odd? '(1 3 5))  → #T
(verifient-tous? odd? '(1 2 5))  → #F
(verifient-tous? odd? '())  → #T
```

Question 4 (4 points) :

En utilisant l'itérateur `verifient-tous?` précédent, écrire une spécification, une définition et une expression de test du prédicat `toutes-non-vides?` qui a pour donnée une liste de listes et qui vérifie que tous les éléments de la liste sont des listes non vides.

Question 5 (5 points) :

En utilisant l'itérateur `verifient-tous?` précédent, écrire une autre définition de la fonction `tous-egaux-a?`.

Exercice II : typage d'une expression

Le but de cet exercice est de typer des expressions, complètement parenthésées et constantes. Dans un premier temps, nous considèrerons que les opérateurs sont des opérateurs binaires et que les expressions sont écrites en infixé. Plus précisément, le langage considéré comporte les constantes numériques (par exemple, 45, 67.8...), les constantes booléennes (`#t`, `#f`), les opérateurs arithmétiques binaires (`+`, `*`, `-`, `/`), les opérateurs logiques (`et`, `ou`) et les opérateurs de comparaison (<, =, >). On supposera que les comparaisons peuvent être effectuées sur des nombres ou des booléens (on prendra `#f` strictement inférieur à `#t`), mais pas entre un nombre et un booléen.

Question 1 (4 points) :

Écrire une définition de la fonction `op-comparaison?` spécifiée par :

`;;;` *op-comparaison? : Symbole -> bool*
`;;;` *(op-comparaison? s) rend #t ssi «s» est un opérateur de comparaison (<, =, >)*

Par exemple :

```
(op-comparaison? '<)  →  #T
(op-comparaison? '+)  →  #F
(op-comparaison? 'a)  →  #F
```

Dans la suite, on pourra aussi faire appel aux fonctions `op-logique?` et `op-arithmetique?` de spécification :

```
;;; op-logique? : Symbole -> bool
;;; (op-logique? s) rend #t ssi «s» est un opérateur logique (et, ou)
;;; op-arithmetique? : Symbole -> bool
;;; (op-arithmetique? s) rend #t ssi «s» est un opérateur arithmétique (+, *, -, /)
```

On définit le type d'une expression de la façon suivante :

- une expression réduite à une constante numérique est de type *Numérique*,
- une expression réduite à une constante booléenne est de type *Booléen*,
- lorsque `op` est un opérateur arithmétique, une expression de la forme `(exp1 op exp2)` est
  - de type *Numérique* lorsque `exp1` et `exp2` sont de type *Numérique*,
  - mal typée sinon (autrement dit, lorsque `exp1` et `exp2` ne sont pas de type *Numérique* ou que l'une d'elles est mal typée),
- lorsque `op` est un opérateur logique, une expression de la forme `(exp1 op exp2)` est
  - de type *Booléen* lorsque `exp1` et `exp2` sont de type *Booléen*,
  - mal typée sinon (autrement dit, lorsque `exp1` et `exp2` ne sont pas de type *Booléen* ou que l'une d'elles est mal typée),
- lorsque `op` est un opérateur de comparaison, une expression de la forme `(exp1 op exp2)` est
  - de type *Booléen* lorsque `exp1` et `exp2` sont de même type,
  - mal typée sinon (autrement dit, lorsque `exp1` et `exp2` ne sont pas de même type ou que l'une d'elles est mal typée).

Voici des exemples d'expressions bien typées :

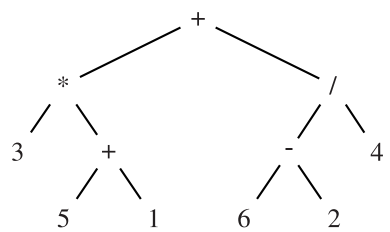
`3`	est de type *Numérique*
`#f`	est de type *Booléen*
`(3 + 5)`	est de type *Numérique*
`(3 < (5 + 6))`	est de type *Booléen*
`(#f < #t)`	est de type *Booléen*
`((3 < 5) et (3 < (5 + 6)))`	est de type *Booléen*

Voici des exemples d'expressions mal typées :

```
(3 + #f)
(#t < (5 + 6))
(3 et 5)
(5 + (3 + #f))
```

Question 2 (4 points) :
Pour chacune des expressions précédentes, expliquer pourquoi elle est mal typée.

Nous représentons ces expressions (bien formées mais bien ou mal typées) par des arbres binaires. Par exemple, l'expression `((3 * (5 + 1)) + ((6 - 2) / 4))` est représentée par l'arbre



Le type `ArbreExprBinConst` est alors le type des arbres binaires qui représentent des expressions bien formées. **Remarque :** dans toute la suite de l'exercice, on manipulera les arbres binaires **uniquement** à travers les fonctions de la barrière d'abstraction (*cf.* carte de référence). On pourra aussi faire appel aux fonctions `ab-feuille` et `ab-feuille?`, de spécification :

;;; *ab-feuille : alpha -> ArbreBinaire[alpha]*
;;; *(ab-feuille e) rend la feuille d'etiquette «e»*

;;; *ab-feuille? : ArbreBinaire[alpha] -> bool*
;;; *(ab-feuille? B) rend vrai ssi «B» est une feuille*

Question 3 (8 points) :

On considère le type `TypeExpression` égal à `{Numerique, Booleen}`, la valeur `Numerique` représentant le type des expressions numériques et la valeur `Booleen` représentant le type des expressions booléennes.
Écrire une définition de la fonction `abe-type` spécifiée par :

`;;;` *abe-type : ArbreExprBinConst -> TypeExpression*
`;;;` *ERREUR lorsque l'expression représentée par «expr» n'est pas bien typée*
`;;;` *(abe-type expr) rend le type de l'expression représentée par «expr»*

Nous considérons maintenant que les opérateurs arithmétiques et logiques peuvent avoir un nombre quelconque d'arguments et nous représentons les expressions par des arbres généralisés. Le type des arbres repésentant de telles expressions est nommé `ArbreExprGenConst`.

Le but de la suite de l'exercice est de définir la fonction `age-type` spécifiée par :

`;;;` *age-type : ArbreExprGenConst -> TypeExpression*
`;;;` *ERREUR lorsque l'expression représentée par «expr» n'est pas bien typée*

`;;;` *(age-type expr) rend le type de l'expression représentée par «expr»*

Question 4 (4 points) :

En utilisant la fonction `age-type`, écrire une définition de la fonction `tous-type?` spécifiée par :

`;;;` *tous-type? : LISTE[ArbreExprGenConst] * TypeExpression -> bool*
`;;;` *ERREUR lorsque l'expression représentée par un des éléments de «LA» est mal typée.*
`;;;` *(tous-type? LA t) rend #t ssi tous les éléments de la liste d'arbres*
`;;;` *d'expression «LA» sont de type «t»*

Question 5 (8 points) :

Écrire une définition de la fonction `age-type`. Indication : on peut utiliser la fonction `tous-type?`.

`;;;` *age-type : ArbreExprGenConst -> TypeExpression*
`;;;` *ERREUR lorsque l'expression représentée par «expr» n'est pas bien typée*
`;;;` *(age-type expr) rend le type de l'expression représentée par «expr»*