Université Pierre et Marie Curie — Année 2000-2001
DEUG MIAS première année — **Examen septembre**
Informatique-1 — 5 septembre 2001
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.
- Le barême (total sur 55) n'est donné qu'à titre indicatif.

Le but de ce problème est d'écrire une fonction qui teste si l'expression booléenne qui lui est donnée comme argument est une tautologie (*i.e.* une expression qui est toujours vraie). Pour la clarté de l'énoncé, nous l'avons divisé en parties.

Le langage des expressions booléennes est le langage des expressions booléennes simples vu en TD. Sa grammaire, sa barrière syntaxique et la barrière des environnements sont données en annexe, dans la dernière page que vous pouvez détacher.

1 Recherche de l'ensemble des variables ayant une occurrence dans une expression

Nous avons besoin de l'ensemble des variables qui apparaissent dans une expression donnée et, dans un premier temps, nous voudrions manipuler des ensembles en général.

1.1 Manipulation d'ensembles d'éléments

Définition : une liste $(l_1\ l_2 \ldots l_i \ldots l_j \ldots l_n)$ est sans répétition lorsque deux éléments de la liste sont toujours différents, autement dit, formellement, lorsque $\forall i \forall j : 1 \leqslant i < j \leqslant n, l_i \neq l_j$.

Nous représenterons un ensemble par une liste sans répétition. Par exemple, l'ensemble d'entiers naturels $\{1,5,2\}$ est représenté par la liste `(1 5 2)` et l'ensemble vide est représenté par la liste vide.

Question 1 (6 points) :
Écrire une définition de la fonction `union-un` de spécification :

```
;;; union-un : α * Ensemble[α] -> Ensemble[α]
;;;  où  Ensemble[α] = LISTE[α]/sans répétition/
;;; (union-un el L) renvoie l'ensemble égal à l'ensemble "L" union "{el}"
;;; (ainsi, lorsque "el" appartient à "L", elle renvoie "L").
```

Exemples d'application :

```
(union-un 3 '(2 1)) → (2 1 3))
(union-un 3 '(2 3 1)) → (2 3 1))
```

```
(define (union-un el L)
  (if (pair? L)
      (if (equal? el (car L))
          L
          (cons (car L) (union-un el (cdr L))))
      (list el)))
```

Question 2 (5 points) :

Écrire une définition de la fonction `union` de spécification :

```
;;; union : Ensemble[α] * Ensemble[α] -> Ensemble[α]
;;;   où  Ensemble[α] = LISTE[α]/sans répétition/
;;; (union L1 L2) renvoie "L1 union L2"
```

Exemple d'application :

`(union '(3 5 2) '(2 1)) → (2 1 5 3))`

```
(define (union L1 L2)
  (if (pair? L1)
      (union-un (car L1) (union (cdr L1) L2))
      L2))
```

1.2 Recherche de l'ensemble des variables ayant une occurrence dans une expression

Définition : une variable a une occurrence dans une expression booléenne si, et seulement si, elle « apparait », au moins une fois, dans l'expression. Par exemple, l'ensemble des variables qui ont une occurrence dans l'expression `((non a) et (b ou (@V et a)))` est l'ensemble `{a, b}`.

Question 3 (10 points) :

En utilisant obligatoirement les fonctions de la barrière d'abstraction donnée en annexe, écrire une définition de la fonction `ensemble-vars` de spécification :

```
;;; ensemble-vars : ExprBoolSimple -> Ensemble[Variable]
;;;   où  Ensemble[Variable] = LISTE[Variable]/sans répétition/
;;; (ensemble-vars exp) renvoie l'ensemble des variables ayant une
;;; occurrence dans l'expression donnée "exp".
```

```
(define (ensemble-vars exp)
  (cond ((atomique? exp)
         (if (constante? exp)
             '()
             (list exp)))
        ((negation? exp)
         (ensemble-vars (neg-sous-exp exp)))
        ((or (conjonction? exp) (disjonction? exp))
         (union (ensemble-vars (bin-sous-expG exp))
                (ensemble-vars (bin-sous-expD exp))))
        (else (error 'ensemble-vars  ; ou erreur
                     "expression mal formée"))))
```

2 Fonctions sur les environnements

Dans toute la suite, nous considèrerons que
- l'ensemble des valeurs booléennes manipulées par les expressions booléennes est l'ensemble $\{0,\ 1\}$, 0 étant la valeur fausse et 1 étant la valeur vraie et nous noterons `Booleen` le type correspondant ;
- un environnement associe des valeurs booléennes à des variables (et uniquement aux variables, les constantes n'étant pas dans l'environnement initial) et qu'il est implanté par une liste d'associations *(variable valeur)*.

2.1 Idée de l'algorithme

Pour savoir si une expression est une tautologie (*i.e.* une expression qui est toujours vraie), on peut construire sa table de vérité : dans les premières colonnes, on met les différentes valeurs de vérités possibles pour les variables et dans la dernière colonne, on met la valeur de vérité de l'expression pour ces valeurs. Par exemple, pour voir que « ((non (a et ((non a) ou b))) ou b) » est une tautologie, on construit sa table de vérité :

a	b	((non (a et ((non a) ou b))) ou b)
0	0	1
1	0	1
0	1	1
1	1	1

et on conclue que la formule est bien une tautologie puisqu'il n'y a que des « 1 » dans la dernière colonne.
Autre exemple : pour voir que « (non (c et (b et (non a)))) » n'est pas une tautologie, on construit sa table de vérité :

a	b	c	(non (c et (b et (non a))))
0	0	0	1
1	0	0	1
0	1	0	1
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	0
1	1	1	1

et on conclue que la formule n'est pas une tautologie puisqu'il y a un « 0 » dans la dernière colonne. Noter que dans cet exemple il est inutile de calculer la dernière ligne puisque, dès qu'il y a un « 0 », on peut répondre que la formule donnée n'est pas une tautologie.

2.2 Mise en œuvre en Scheme

Pour informatiser ce procédé (rappelons que le type *Booleen* est le type $\{0,\ 1\}$, et non le type *bool* des booléens Scheme) :
- pour « remplir » les premières colonnes
  - tout d'abord, nous définirons une fonction `environnement` qui, étant données une liste de variables et une liste de valeurs, construit l'environnement sous la forme d'une liste d'associations *(variable valeur)* (question 4) ;
  - ensuite, nous définirons trois fonctions qui travaillent sur les listes de valeurs :
    - la première pour construire une liste dont toutes les valeurs sont fausses, autrement dit pour construire la première ligne de la table de vérité (question 5),
    - la deuxième pour tester si une liste ne contient que des valeurs vraies, autrement dit pour savoir si la liste correspond à la dernière ligne de la table de vérité (question 6),
    - la troisième pour construire une liste de valeurs booléennes à partir d'une autre, autrement dit pour définir les valeurs contenues dans les premières colonnes de la ligne qui suit une ligne donnée (question 7),
- tout sera alors mis en place pour définir la fonction `tautologie?` (question 8).

Question 4 (8 points) :

Donner une définition de la fonction `environnement` dont la spécification est :

```
;;; environnement : LISTE[Variable] * LISTE[Booleen] -> Environnement
;;; ERREUR lorsque les listes "L-var" et "L-val" ne sont pas de même longueur
;;; ERREUR lorsqu'un des éléments de "L-val" n'est pas un booléen
;;; (environnement L-var L-val) rend l'environnement qui associe à chaque
;;; variable de la liste "L-var" le booléen de même rang dans la liste
;;; "L-val". Par exemple, (environnement '(a b) '(1 0)) rend
;;; l'environnement qui associe 1 à a et 0 à b.
```

```
(define (environnement L-var L-val)
  (if (and (pair? L-var) (pair? L-val))
      (env-ajout (car L-var)
                 (car L-val)
                 (environnement (cdr L-var) (cdr L-val)))
      (if (or (pair? L-var) (pair? L-val))
          (error 'environnement                    ;ou erreur
                 "les deux listes ne sont pas de même longueur")
          (env-initial))))
```

Question 5 (6 points) :

Donner une définition de la fonction `liste-tous-faux` spécifiée par :

```
;;; liste-tous-faux : nat -> LISTE[Booleen]
;;; (liste-tous-faux n) rend la liste de booléens, de longueur "n", dont tous
;;; les éléments sont faux
```

```
(define (liste-tous-faux n)
  (if (= n 0)
      '()
      (cons 0 (liste-tous-faux (- n 1)))))
```

Question 6 (5 points) :

Donner une définition de la fonction `liste-tous-vrais?` spécifiée par :

```
;;; liste-tous-vrais? :  LISTE[Booleen] -> bool
;;; (liste-tous-vrais? L) rend #t si, et seulement si, tous les éléments
;;; de la liste "L" sont vrais.
```

```
(define (liste-tous-vrais? L)
  (if (pair? L)
      (and (= 1 (car l)) (liste-tous-vrais? (cdr L)))
      #t))
```

Question 7 (5 points) :

Pour construire la liste de valeurs qui suit une liste donnée, on peut remplacer, en commençant au début de la liste, tous les 1 par 0 et cela jusqu'à ce que l'on trouve un 0 qui est remplacé par 1.

Donner une définition de la fonction `liste-suivante` spécifiée par :

```
;;; liste-suivante : LISTE[Booleen] -> LISTE[Booleen]
;;; ERREUR lorsque la liste donnée ne comporte pas d'élément égal à 0
;;; (liste-suivante L) rend la liste qui suit, dans la table de vérité,
;;; la liste "L"
```

```
(define (liste-suivante L)
  (if (= 0 (car L))
      (cons 1 (cdr L))
      (cons 0 (liste-suivante (cdr L)))))
```

3 Fonction tautologie ?

Question 8 (10 points) :

La spécification de la fonction `exp-val` (**dont on ne vous demande pas la définition**) étant

```
;;; exp-val : ExprBoolSimple * Environnement -> Booleen
;;; ERREUR lorsque exp n'est pas une expression bien formée ou lorsqu'une
;;; variable n'est pas définie dans l'environnement
;;; (exp-val exp env) rend la valeur de exp dans l'environnement env
```

Exemple d'application :

`(exp-val '((non (a et ((non a) ou b))) ou b) '((a 1) (b 0))) → 1`

donner une définition de la fonction `tautologie?` spécifiée par :

;;; tautologie? : ExprBoolSimple -> bool
;;; (tautologie? exp) renvoie #t ssi "exp" est une tautologie

```
(define (tautologie? exp)
  (let ((L-var (ensemble-vars exp)))
    ;; tautologie-aux-exp? : LISTE[Booleen] -> bool
    ;; (tautologie-aux-exp? L-val) renvoie #t ssi "exp" est vraie pour les valeurs "L-val"
    ;; et pour toutes les listes de valeurs suivantes
    (define (tautologie-aux-exp? L-val)
      (if (= 1 (exp-val exp (environnement L-var L-val)))
          (if (liste-tous-vrais? L-val)
              #t
              (tautologie-aux-exp? (liste-suivante L-val)))
          #f))
    ;; expression définissant tautologie? :
    (tautologie-aux-exp? (liste-tous-faux (length L-var)))))
```

Annexe

Grammaire des expressions booléennes simples

<expBoolSimple> → *<atomique>*
<négation>
<binaire>

<atomique> → *<constante>*
<variable>

<constante> → `@V` VRAI
`@F` FAUX

<négation> → `(non` *<expBoolSimple>* `)`

<binaire> → `(` *<expBoolSimple>* `et` *<expBoolSimple>* `)` CONJONCTION
`(` *<expBoolSimple>* `ou` *<expBoolSimple>* `)` DISJONCTION

Barrière d'abstraction pour les expressions booléennes simples

```
;;; atomique? : ExprBoolSimple -> bool
;;; constante? : ExprBoolSimple -> bool
;;; constante-vrai? : Constante -> bool
;;; constante-faux? : Constante -> bool
;;; negation? : ExprBoolSimple -> bool
;;; conjonction? : ExprBoolSimple -> bool
;;; disjonction? : ExprBoolSimple -> bool

;;; neg-sous-exp : Negation -> ExprBoolSimple
;;; (neg-sous-exp exp) rend exp1 lorsque exp est égale à (non exp1)

;;; bin-sous-expG : Binaire -> ExprBoolSimple
;;; (bin-sous-expG exp) rend expG lorsque exp est égale à (expG et expD) ou à (expG ou expD)

;;; bin-sous-expD : Binaire -> ExprBoolSimple
;;; (bin-sous-expD exp) rend expD lorsque exp est égale à (expG et expD) ou à (expG ou expD)
```

Environnement

```
;;; env-initial : -> Environnement
;;; (env-initial) rend l'environnement vide.
(define (env-initial)
  '())

;;; env-ajout : Variable * Booleen * Environnement -> Environnement
;;; ERREUR lorsque val n'est pas un booléen
;;; (env-ajout var val env) rend l'environnement obtenu en étendant
;;; l'environnement "env" en associant à la variable "var" la valeur "val"
```