

# Scheme - TP n° 3

Université Paris I. Maîtrise de Logique. Année 2004-2005.  
Nicolas Stroppa - [stroppa@enst.fr](mailto:stroppa@enst.fr)

## 1 Introduction

Dans ce TP, il s'agira de développer un mini système expert, c'est-à-dire un système d'inférence, qui, à partir d'une base de connaissance et d'une base de règles, produit des énoncés.

La base de règles sera modélisée de la façon suivante.

```
(define base-de-regles
  (list
    (make-regle (si '(cet animal donne du lait))
                (alors '(cet animal est un mammifere)))
    (make-regle (si '(cet animal a des plumes))
                (alors '(cet animal est un oiseau)))
    (make-regle (si '(cet animal vole)
                    '(cet animal pond des oeufs))
                (alors '(cet animal est un oiseau)))
    (make-regle (si '(cet animal est un mammifere)
                    '(cet animal mange de la viande))
                (alors '(cet animal est un carnivore)))
    (make-regle (si '(cet animal est un carnivore)
                    '(cet animal est de couleur fauve)
                    '(cet animal a des taches noires))
                (alors '(cet animal est un guepard)))
    (make-regle (si '(cet animal a des dents pointues)
                    '(cet animal a des griffes))
                (alors '(cet animal est un carnivore)))
    (make-regle (si '(cet animal a des poils)
                    (alors '(cet animal est un mammifere)))
    (make-regle (si '(cet animal est un oiseau)
                    '(cet animal est jaune))
                (alors '(cet animal est un canari)))
  )
```

À partir de cette base, un exemple d'utilisation de votre programme pourra être:

```

Entrez les faits observés.
(
  (cet animal a des poils)
  (cet animal a des dents pointues)
  (cet animal a des griffes)
  (cet animal est de couleur fauve)
  (cet animal a des taches noires) )

(cet animal est un mammifère)
(cet animal est un carnivore)
(cet animal est un guépard)

```

Pour effectuer ce traitement, vous allez séparer votre programme en plusieurs fonctions qui auront chacune une tâche bien identifiée (notion de *modularité*). La fonction principale `main` se chargera de l'initialisation des données et de l'interaction avec l'utilisateur. Elle appellera la fonction `inferer` qui cherchera dans la base de règles les règles dont la partie gauche (si ...) est vérifiée. On ajoutera alors à la base de faits la partie droite de ces règles. Cette information sera également affichée à l'écran. Quand plus aucune règle n'est activable, le programme s'arrête.

## 2 Interaction avec l'utilisateur

Pour interagir avec l'utilisateur, deux fonctions principales sont utilisées. Il s'agit de `write` qui affiche à l'écran la valeur d'une variable et `read`, qui à l'inverse stocke dans une variable la valeur rentrée par l'utilisateur.

Pour comprendre comment ces fonctions se comportent, étudiez le programme suivant:

```

(define a "dfs")
(write a)
(define b '(1 2 3))
(write b)

(define c (read)) ;; Rentrez alors 3
c
(define d (read)) ;; Rentrez alors (3 4 5)
d

```

Écrivez la fonction `main`, qui affiche le message "Entrez les faits observés", lit une liste entrée par l'utilisateur, la stocke dans la variable `base-de-faits`, et affiche le contenu de cette variable.

### 3 Représentation des données

Définissez les fonctions `make-regle`, `si`, `alors`, `conditions`, `consequence` qui permettent d'interagir avec les règles. Celles-ci seront représentées par un couple conditions-conséquence.

L'alternative consiste à définir les règles à l'aide d'une structure, e.g. (`define-struct regle (conditions consequence)`). Après cette déclaration, les fonctions `make-regle`, `regle-conditions`, `regle-consequence` sont automatiquement créées.

### 4 Interrogation de la base

Définissez les fonctions (`sont-verifiees conditions`) (`est-verifiee condition`) qui permettent de tester des conditions. Définissez également `ajoute-aux-faits` qui ajoute une conséquence aux faits si celle-ci n'est pas déjà présentes dans la base de faits. Définissez maintenant la fonction `teste-regle` qui vérifie qu'une règle peut être appliquée. Une règle peut être appliquée si toutes ses conditions sont vérifiées par les faits et si sa conséquence n'est pas dans les faits.

### 5 La glue

Traiter maintenant la fonction `inferer` qui s'occupe de gérer l'inférence à l'aide des fonctions précédemment définies.

## 6 Éléments de corrigé

```
;; Solution 1.
(define-struct regle (conditions consequence))

;; Solution 2.
;;(define-struct regle (id conditions consequence))

;; Pour rendre la syntaxe des règles plus claires.
(define (si . a) a)
(define (alors a) a)

;; Liste de règles
(define base-de-regles
  (list
    (make-regle (si '(cet animal donne du lait))
               (alors '(cet animal est un  mammifere)))
    (make-regle (si '(cet animal a des plumes))
               (alors '(cet animal est un  oiseau)))
    (make-regle (si '(cet animal vole)
                   '(cet animal pond des oeufs))
               (alors '(cet animal est un  oiseau)))
    (make-regle (si '(cet animal est un  mammifere)
                   '(cet animal mange de la viande))
               (alors '(cet animal est un  carnivore)))
    (make-regle (si '(cet animal est un  carnivore)
                   '(cet animal est de couleur fauve)
                   '(cet animal a des taches noires))
               (alors '(cet animal est un  guepard)))
    (make-regle (si '(cet animal a des dents pointues)
                   '(cet animal a des griffes))
               (alors '(cet animal est un  carnivore)))
    (make-regle (si '(cet animal a des poils)
                   (alors '(cet animal est un  mammifere)))
               (alors '(cet animal est un  mammifere)))
    (make-regle (si '(cet animal est un  oiseau)
                   '(cet animal est jaune))
               (alors '(cet animal est un  canari)))
  )
)

;; Liste de faits
(define base-de-faits '())
```

```

;; Solution 2.
;;(define regles-activeses '())

;; On rentre les faits, puis on lance l'inférence
(define (main)
  (display "Entrez les faits observés")
  (newline)
  (set! base-de-faits (read))
  (inferer))

;; On parcourt l'ensemble des règles.
;; Si un règle a été appliquée, on relance le parcours.
(define (inferer)
  (if (parcourir-regles base-de-regles) (inferer) '()))

;; On teste chaque règle.
(define (parcourir-regles regles)
  (if (null? regles) #f
      (or (teste-regle (car regles))
          (parcourir-regles (cdr regles)))))

;; On regarde si la règle peut être appliquée.
;; Pour cela il faut que les conditions soient vérifiées
;; et que la conséquence ne soit pas déjà présente
;; dans les faits.
(define (teste-regle regle)
  (and (sont-verifiees (regle-conditions regle))
       (ajoute-aux-faits (regle-consequence regle))))

(define (sont-verifiees conditions)
  (if (null? conditions)
      #t
      (and (est-verifiee (car conditions) base-de-faits)
           (sont-verifiees (cdr conditions)))))

(define (est-verifiee condition faits)
  (if (null? faits)
      #f
      (or (equal? (car faits) condition)
          (est-verifiee condition (cdr faits)))))

;; ajoute-aux-faits regarde si la conséquence à ajouter
;; aux faits est déjà dans ceux-ci. Si oui, elle renvoie faux.

```

```
;; Sinon, elle affiche la conséquence, l'ajoute aux faits et
;; renvoie vrai.
(define (ajoute-aux-faits consequence)
  (if (member consequence base-de-faits)
      #f
      (begin
        (set! base-de-faits (cons consequence base-de-faits))
        (write consequence)
        (newline)
        #t
        )))

;; Un exemple de base de faits:
;((cet animal a des poils)
;(cet animal a des dents pointues)
;(cet animal a des griffes)
;(cet animal a des taches noires)
;(cet animal est de couleur fauve))
```