IFT 359 Programmation fonctionnelle

Travail pratique 3

Données mutables et fermetures : Prototypes

Composition des équipes:

- Par équipe de 1 ou 2 personnes
- Une seule réponse sera fournie par équipe.

Barème de correction

- 5 points : Réussite des jeux d'exécution dans turnin
- 2 points : Évaluation du code
    - Définition, utilisation et respect de barrières d'abstraction. (+)
    - Définition de fonctions qui rendent des fonctions. (+)
    - Définitions de fonctions utilitaires. (+)
    - Tous les cas de figure sont traités. (+)
    - Clarté du code. (+)
    - Utilisation de fonctions de haut niveau. (+)
    - Code générique, par exemple dans la fonction dispatch (+)
    - Gestion du nom du prototype
    - Gestion de l'état, gestion des méthodes
    - Code incomplet ou incorrect. (-)
    - Un prédicat rend explicitement #f ou #t. (-)
    - Code inutilement complexe. (-)
-

Tout retard entraînera la note zéro.

Remise

Vous devez remettre votre travail par **turnin** avant le lundi 18 novembre 2023, 23h59 dans le devoir IFT359-TP3.

Le fichier doit être nommé `tp3.rkt`

Votre fichier doit commencer par les lignes suivantes :

```
#lang racket
(provide make-object!)

; ***************** IFT359 / TP3 Groupe 1 ou 2
; ***************** Nom, prénom et matricule du 1er membre de l'équipe
; ***************** Nom, prénom et matricule du 2e membre de l'équipe
```

Votre code ne doit comprendre que les définitions sans aucun appel de fonction pour produire les résultats. Il ne doit pas y avoir non plus d'instructions d'entrée / sortie. Vous pouvez le faire en n'écrivant aucune de ces expressions ou en les mettant en commentaires.

Énoncé

Les langages orientés objets les plus utilisés reposent, pour la grande majorité, sur la notion de classe et d'instance. La classe sert à créer des objets qui sont ses instances. Toutes les instances ont la même structure et le même comportement et elles ne peuvent pas évoluer comme des individus autonomes.

La programmation orientée prototypes est toutefois une autre approche de la programmation par objets qui est beaucoup plus dynamique où chaque objet évolue de manière indépendante. Javascript en est l'exemple le plus connu actuellement. Un objet est créé à partir de zéro ou encore en clonant un objet déjà existant.

L'approche des objets vue en cours est plus près des langages objets à base de prototypes que des langages objets à base de classe. Un objet est une fermeture. Cette fermeture est créée par une fonction, par exemple `make-account`. La fermeture possède toute l'information sur son état (nom et valeur des variables représentant son état), ainsi que l'ensemble des fonctions qui définissent son comportement. Ces fonctions sont activées par des messages.

Dans les TP qui viennent, nous allons généraliser cette implémentation et rendre ces objets plus dynamiques. Les propriétés visées sont

- La mécanique d'invocation des messages doit être généralisée. En particulier, il faut faire disparaitre les conditionnelles et le noms explicites des noms de méthodes. Cela permettra en particulier d'ajouter ou de supprimer des méthodes après sa création (TP3).
- Un objet doit posséder un mécanisme pour utiliser les comportements définis dans d'autres objets. Dans les langages à prototypes, le mécanisme utilisé s'appelle la délégation. L'héritage est son pendant dans les langages à base de classes (TP4 encapsulation).
- Un objet doit pouvoir gérer son comportement et l'adapter dynamiquement en apprenant et en oubliant des comportements (TP4 mémoïsation).

TP3 : **Abstraire et généraliser la structure d'un objet et le traitement des messages**

Le canevas actuel pour créer un objet est le suivant

```
(define (make-account amount)
  (let ( ; définition de l'état de l'objet
         [my-balance amount])
    ; définition du comportement de l'objet
    (define (balance)  my-balance)
    (define (withdraw amount)
      (if (>=  my-balance amount)
          (begin (set!  my-balance (-  my-balance amount))
                 my-balance)
          "Insufficient funds"))
    (define (deposit amount)
      (set!  my-balance (+  my-balance amount))
      my-balance)
    ; fonction responsable du traitement des messages
    (define (dispatch m)
      (cond ((eq? m 'balance) balance)
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            (else (error "Unknown request: MAKE-ACCOUNT" m))))
    ; l'objet est représenté par une fermeture, la fonction dispatch,
    ; qui contient son état, son comportement
    ;              et la fonction qui traite les messages
    ; on active l'objet
```

```
    ; en invoquant la fonction qui traite les messages
    dispatch))
```

Il existe trois barrières principales à l'évolution dynamique de l'objet :

- Ce n'est pas possible d'ajouter ou de supprimer des variables représentant l'état de l'objet sans modifier le code
- Ce n'est pas possible d'ajouter ou de supprimer des méthodes représentant le comportement de l'objet sans modifier le code
- Les sélecteurs des comportements sont codés en dur dans la fonction `dispatch`, ce qui empêche l'ajout ou la suppression de méthodes.

Le but de ce TP est de pallier ces limitations. Pour cela, vous devez implémenter les fonctions suivantes

Création des objets		
`(make-object! name . props)`	Rend une fermeture (la fonction `dispatch`).	`name` est un symbole. `props` est la p-liste des noms de variable d'état (symbole) et de leur valeur.
`(get-name)`	Rend le nom de l'objet.	`name` est un symbole.
`(set-name! new-name)`	Modifie le nom de l'objet. Rend le nom de l'objet.	
Gestion de l'état		
`(get-state var)`	Rend la valeur de la variable d'état '`undefined` si la variable n'existe pas	`var` est un symbole.
`(set-state! var value)`	Modifie la valeur de la variable d'état	`var` est un symbole. Si la variable d'état n'existe pas, elle est créée.
`(delete-state! var)`	Supprime la variable d'état	`var` est un symbole.
Gestion des comportements		
`(understand? selector)`	Prédicat qui indique si l'objet comprend le message	`selector` est un symbole.
`(add-method! selector method)`	Ajoute ou remplace un comportement Rend le nom de la méthode ajoutée ou remplacée	`selector` est un symbole. `method` est une fonction.
`(delete-method! selector )`	Supprime le comportement. Rend le nom de la méthode supprimée.	`selector` est un symbole.

- Les objets sont nommés. L'attribut `name` est codé en dur dans l'objet. Il est accédé via les méthodes `get-name` et `set-name!`. Il n'est pas accédé via les méthodes `get-state`, `set-state!` et `delete-state!`.
- L'état initial de l'objet est fourni en argument lors de l'appel de la fonction `make-object!`.
- Pour gérer un état et des comportements qui peuvent évoluer, vous devrez sans doute les conserver dans un ou deux dictionnaires dont les clés seront les symboles qui servent d'identificateur (nom de la variable ou sélecteur).
- À la création de l'objet, vous aurez sans doute un certain nombre d'initialisation à faire.
- Vous devrez réécrire la fonction `dispatch` pour la rendre générique et faire disparaitre la conditionnelle habituellement utilisée qui repose sur le nom des méthodes codé en dur.
- Lorsqu'un message ne sera pas compris, la méthode `dispatch` rendra '`doesNotUnderstand`.
- Vous verrez dans le jeu d'essai que dans certaines méthodes, le premier paramètre est `self`. Cela permet à un objet de se renvoyer des messages lorsque c'est nécessaire. Il faut donc ajouter l'objet qui reçoit le message en tant que premier argument comme dans l'exemple suivant :

```
(Eve 'add-method! 'saluer
                  (lambda(self object)
                     (string-append "Bonjour " (object 'get-name)
                                    ", je m'appelle " (self 'get-name) )))
(Eve 'saluer Eve Adam)
```