université de BORDEAUX Collège Sciences et Technologies DEVUIP Service Scolarité	ANNÉE UNIVERSITAIRE 2014/2015 1ÈRE SESSION D'AUTOMNE **Parcours :** IN501 et IM500 **Code UE :** IN5011 **Épreuve :** Programmation 3 **Date :** Mardi 16 décembre 2014 **Heure :** 08h30 **Durée :** 1h30 Documents : autorisés Épreuve de Mme Irène Durand	Département D L Licence

Le barème est donné à titre **indicatif**. Le sujet comporte **??** pages.

Exercice 1 *(3pts)*

Soit la fonction `compress (l)` qui s'applique à une liste d'atomes `l` et qui construit une forme compressée de `l` constituée d'une liste de paires $(c\ .\ a)$ où c est le nombre d'atomes a apparaissant de manière consécutive. Exemples :

```
CL-USER> (compress '())
NIL
CL-USER> (compress '(0))
((1 . 0))
CL-USER> (compress '(1 1 1 2 1 1 4 4))
((3 . 1) (1 . 2) (2 . 1) (2 . 4))
```

1. Implémenter la fonction `compress`.

Exercice 2 *(12pts)*

Un jeu se joue à deux joueurs (numérotés `0` et `1`) sur un plateau de $2 \times$ `*player-nb-places*` cases (`place` en anglais) numérotées à partir de 0. Le plateau est représenté par un tableau à une dimension. Les cases numérotées de 0 à `*player-nb-places*` -1 appartiennent au joueur `0` tandis que les cases numérotées de `*player-nb-places*` à `*size*` $-\ 1$ appartiennent au joueur `1`.
Au départ, chaque case contient un nombre de jetons égal à `*nb-pebbles*`. Par la suite, les cases pourront contenir un nombre quelconque de jetons. Les cases seront souvent repérées par leur numéro.

2. Écrire une fonction `player-places (player)` qui retourne la liste[1] des numéros des cases qui appartiennent au joueur `player`.

Exemples :

```
CL-USER> (player-places 0)
(0 1 2 3 4 5)
CL-USER> (player-places 1)
(6 7 8 9 10 11)
```

1. L'ordre est sans importance.

3. Écrire une fonction `make-game-array (&optional nb-pebbles)` qui crée le tableau initial à la bonne taille avec `nb-pebbles` jetons dans chaque case, `nb-pebbles` ayant par défaut la valeur `*nb-pebbles*`.

Exemples :

```
CL-USER> (defvar *player-nb-places* 6)
*PLAYER-NB-PLACES*
CL-USER> (defvar *size* (* 2 *player-nb-places*))
*SIZE*
CL-USER> (defvar *nb-pebbles* 4)
*NB-PEBBLES*
CL-USER> (defparameter *game-array* (make-game-array))
*GAME-ARRAY*
CL-USER> *game-array*
#(4 4 4 4 4 4 4 4 4 4 4 4)
CL-USER> (setf *game-array* (make-game-array 3))
#(3 3 3 3 3 3 3 3 3 3 3 3)
```

Un joueur peut jouer quand il a au moins une case non vide.

4. Écrire une fonction `player-valid-places (player game-array)` qui retourne la liste des numéros de cases où le joueur `player` peut jouer sur `game-array`.

Exemples :

```
CL-USER> *game-array*
#(1 1 1 0 0 0 1 2 0 0 7 4)
CL-USER> (player-valid-places 0 *game-array*)
(0 1 2)
CL-USER> (player-valid-places 1 *game-array*)
(6 7 10 11)
```

Quand un joueur joue dans l'une de ses cases, il prend tous les jetons de cette case et les distribue un par un dans les cases suivantes et ceci de manière circulaire[2]. La fonction `play-in-place (place game-array)` réalise cette opération. Par commodité, elle retourne le tableau `game-array`. Exemples :

```
CL-USER> *game-array*
#(2 1 0 0 1 0 2 0 4 5 3 3)
CL-USER> (play-in-place 9 *game-array*)
#(3 2 1 0 1 0 2 0 4 0 4 4)
CL-USER> (play-in-place 0 *game-array*)
#(0 3 2 1 1 0 2 0 4 0 4 4)
```

5. Implémenter la fonction `play-in-place`.

On utilise des instances de la classe `strategy` pour représenter une stratégie. Une stratégie a un nom, `strategy-name` (une chaîne de caractères) et une fonction `strategy-fun` qui s'applique à un joueur et un tableau et retourne la case dans laquelle le joueur veut jouer ou `NIL` s'il ne peut pas jouer.

2. La case 0 suit donc la case 11.

6. Définir la classe `strategy` en implémentant les deux opérations suivantes :
   (a) `(defgeneric strategy-name (strategy) (:documentation "returns the name of STRATEGY"))`
   (b) `(defgeneric strategy-fun (strategy) (:documentation "returns the strategy function of STRATEGY"))`
7. Écrire une fonction `make-strategy (strategy-name strategy-fun)` qui retourne une instance de la classe `strategy`. Exemple :

```
CL-USER> (defparameter *s* (make-strategy "random" #'random-fun))
*S*
CL-USER> (strategy-name *s*)
"random"
CL-USER> (strategy-fun *s*)
#<FUNCTION RANDOM-FUN>
```

Exercice 3 *(5pts)*
On souhaite stocker les stratégies dans une *table de hachage*. La *clé* utilisée sera le nom de la stratégie donc une chaîne de caractères.

8. Donner une expression qui construit une telle table de hachage et la mémorise dans la variable globale `*strategies*`.
9. Écrire une fonction `add-strategy (strategy)` qui ajoute la stratégie `strategy` dans la table.

On souhaite que les stratégies disponibles soient **systématiquement** ajoutées à la table. L'utilisateur est donc censé utiliser `add-strategy` à chaque fois qu'il crée une stratégie. Par exemple :

```
(add-strategy
 (make-strategy "random"
                (lambda (player game-array)
                  (let ((places (player-valid-places player game-array)))
                    (if (endp places)
                        nil
                        (nth (random (length places)) places))))))

(add-strategy
 (make-strategy "interactive"
                (lambda (player game-array)
                  (let ((places (player-valid-places player game-array)))
                    (if (endp places)
                        nil
                        (progn
                          (format t "tapez un numéro de case parmi ~A: " places)
                          (read)))))))
```

Dans toutes les fonctions de stratégie, on commence par vérifier si le joueur a une case dans laquelle il peut jouer. Toutes les fonctions de stratégies s'appliquent toujours à un joueur (`player`) et à un tableau (`game-array`). Pour éviter la duplication

```
                (lambda (player game-array)
                  (let ((places (player-valid-places player game-array)))
                    (if (endp places)
                         nil
```

présente dans toutes les stratégies, on souhaite disposer d'une macro `def-strategy (strategy-name &body body)` permettant d'éviter cette duplication et grâce à laquelle on transmettrait uniquement le nom et le code spécifique à la stratégie. On écrirait ainsi

```
(def-strategy "random"
    (nth (random (length places)) places))

(def-strategy "interactive"
    (format t "tapez un numéro de case parmi ~A: " places) (read))
```

Dans le `body` transmis à la macro, l'utilisateur peut utiliser les variables `player, game-array` et `places` qui sont définies implicitement par la macro.

```
CL-USER> (macroexpand-1
          '(def-strategy
            "random"
            (nth (random (length places)) places)))
(ADD-STRATEGY
 (MAKE-STRATEGY "random"
                (LAMBDA (PLAYER GAME-ARRAY)
                  (LET ((PLACES (PLAYER-VALID-PLACES PLAYER GAME-ARRAY)))
                    (IF (ENDP PLACES)
                        NIL
                        (PROGN (NTH (RANDOM (LENGTH PLACES)) PLACES)))))))
T
CL-USER> (macroexpand-1
          '(def-strategy
            "interactive"
            (format t "tapez un numéro de case parmi ~A: " places) (read)))
(ADD-STRATEGY
 (MAKE-STRATEGY "interactive"
                (LAMBDA (PLAYER GAME-ARRAY)
                  (LET ((PLACES (PLAYER-VALID-PLACES PLAYER GAME-ARRAY)))
                    (IF (ENDP PLACES)
                        NIL
                        (PROGN
                         (FORMAT T "tapez un numéro de case parmi ~A: " PLACES)
                         (READ)))))))
T
```

10. Écrire la macro `def-strategy`.

FIN