ENSEIRB-Matmeca
2019-2020

1ère année - Filière Info
PG104

TD n°8 - Modules, macros et formes impératives

Exercice 1: Programmation modulaire

Considérons les deux implémentations suivantes pour un *type abstrait de données* (TAD) représentant un ensemble d'entiers positifs ordonnés. La première implémentation (à gauche) utilise les listes, tandis que la seconde (à droite) utilise des vecteurs contenus dans des structures.

```racket
#lang racket

;; Represent a set as a simple list
(define set? list?)

(define (set-empty) '())

(define set-length length)

(define (set-mem set s)
  (cons? (member s set)))

(define (set-add set s)
  (match set
    ['()         (list s)]
    [(cons x xs)
      (cond [(< s x) (cons s set)]
            [(= s x) set]
            [else    (cons x (set-add s xs))])]))

;; set-list.rkt
```

```racket
#lang racket

;; Represent a set as a struct containing a vector
(struct vset (size tab))
(define SIZE 10)

(define set? vset?)

(define (set-empty) (vset 0 (make-vector SIZE -1)))

(define set-length vset-size)

(define (set-mem set x)
  (number? (vector-member x (vset-tab set))))

(define (set-add set x)
  (if (or (set-mem set x) (= (set-length set) SIZE))
      set
      (let* ([size (vset-size set)]
             [tab  (vset-tab set)])
        (vector-set! tab size x)
        (vector-sort! tab < 0 (add1 size)) ;; bad
        (vset (add1 size) tab))))

;; set-vector.rkt
```

1. Quelles sont les fonctions définies dans ces deux fichiers ? Quelle assurance ce code donne-t'il que les deux fichiers implémentent le même type abstrait de données ?

Supposons vouloir écrire un ensemble de tests fonctionnels permettant de vérifier les deux implémentations. Si elles implémentent bien le même TAD, elles sont interchangeables, et il devrait être envisageable de réutiliser les mêmes tests.

2. Expliquer pourquoi cela implique d'implémenter les tests dans un fichier différent.

Afin de tester ce code, nous proposons d'utiliser une bibliothèque de tests unitaires de Racket appelé RackUnit (cf. `https://docs.racket-lang.org/rackunit/`). Avec cette bibliothèque, les tests peuvent prendre la forme suivante :

```
#lang racket
(require rackunit)
(require rackunit/text-ui)

(define all-tests
  (test-suite
   "Tests␣for␣a␣set␣implementation"

   (test-case
    "Empty␣set␣has␣size␣zero"
    (let* ([set (set-empty)])
      (check-equal? (set-length set) 0)))

   (test-case
    "Adding␣to␣empty␣set␣yields␣size␣one"
    (let* ([set (set-empty)])
      (check-equal? (set-length (set-add set 666)) 1)))

   (test-case
    "Integer␣added␣to␣empty␣set␣is␣found␣back"
    (let* ([set (set-empty)])
      (check-true (set-mem (set-add set 666) 666))
      (check-false (set-mem (set-add set 666) 667))))

))

(printf "Running␣tests\n")
(run-tests all-tests)

;; set-test.rkt
```

3. Décrire à l'aide d'un diagramme les liens de dépendance entre les 3 fichiers `set-list.rkt`, `set-vector.rkt` et `set-test.rkt`.

Le langage Racket dispose de deux mot-clés **`require`** et **`provide`** permettant d'indiquer qu'un fichier dépend d'un identificateur ou exporte un autre identificateur. Typiquement, ces deux mot-clés s'utilisent de la manière suivante :

```
(require "brain.rkt")                ;; file name
(provide make-thought derive-thought) ;; list of identifiers
```

4. Ajouter les dépendances précédentes pour relier les 3 fichiers. Comment se fait le choix de l'implémentation testée ?

L'ajout des tests renforce la confiance que l'on peut avoir dans ces implémentations, mais apporte relativement peu de vérification comparé à un système de type classique. Pour mitiger cela, le langage Racket utilise un système de *contrats*, à savoir des vérifications effectuées dynamiquement lors de l'appel de fonctions. Par exemple :

```
(+ "one" "two") ;; +: contract violation
                ;;   expected: number?, given: "one"
```

Typiquement, un contrat prend la forme d'un ensemble de prédicats que l'on peut passer sur les paramètres d'entrée et de sortie des fonctions :

```
(provide (contract-out [set-length (→ set? (and/c number? positive?))]))
```

La flèche → indique que l'identifiant `set-length` est une fonction, qui prend un paramètre vérifiant `set?` et qui renvoie un résultat vérifiant `(and/c number? positive?)`. La page `https://docs.racket-lang.org/reference/data-structure-contracts.html` décrit quelques fonctions permettant de manipuler des contrats.

5. Proposer un ensemble de contrats pour le type abstrait de données manipulé ici.
   De quelle manière peut-on les implémenter ici ?

Exercice 2: Génération de nombres premiers

Sans utiliser de variable globale, écrire une fonction `gen-prime` sans arguments qui retourne à chaque appel le nombre premier suivant celui de l'appel précédent à `gen-prime`.

Exemple :

```
* (gen-prime)
2
* (gen-prime)
3
* (gen-prime)
5
* (gen-prime)
7
```

Exercice 3: Expansion de macros

Pour chacun des exemples suivants, deviner l'expansion qui va être réalisée sur la macro, puis utiliser `expand-once` et `expand` pour observer l'expansion réelle.

1. Premier élément d'une liste : `first`

```
(define-syntax-rule (first l)
  (car l))

(first '(1 2 3))
(first '(f (g x y)))
```

2. Second élément d'une liste : `second`

```
(define-syntax-rule (second l)
  (first (cdr l)))

(second '(1 2 3))
(second '(f (g x y)))
```

3. Troisième élément d'une liste : `third`

```
(define-syntax-rule (third l)
  (caddr l))

(third '(1 2 3))       ;; → 3
(third '(f (g x y) z)) ;; → z
```

Exercice 4: Macro double

Définir une macro `double` qui double son argument (à travers une affectation `set!`) :

```
(let ((x 1))
  (double x)
  x) ; -→ 2
```

La tester sur des exemples.
Avertissement : n'utiliser la fonction `expand` que depuis la REPL.

Exercice 5: Conditionnelle protégée

Écrire une macro `cond-raise` qui ajoute au `cond` "classique" la levée d'une exception lorsqu'aucun des cas gérés par la conditionnelle ne s'applique.
Exemple :

```
(define (suspicious-choice n)
    (cond-raise
     [(zero? n) 'zero]
     [(= 1 n)   'un]
     [(= 2 n)   'deux]))

(suspicious-choice 1) ;; → un
(suspicious-choice 5) ;; → Error : cond-raise : no match
```

Exercice 6: Conversion `let` en `lambda`

1. Définir une *fonction* `transform-let-to-lambda` qui transforme une expression `letl` avec la syntaxe suivante en une lambda-expression. Le schéma général est le suivant :

```
(transform-let-to-lambda
   '(letl ([p1 a1] ... [pK aK])
          forme1
          ...
          formeN))
```

doit donner

```
((lambda (p1 ... pK)
  forme1
  ...
  formeN)
  a1 ... aK)
```

   Vous pouvez découper le travail en plusieurs fonctions auxiliaires.
   Est-il possible de remplacer le symbole `letl` par `let` ?

2. Écrire une *macro* `letm` qui évalue la lambda-expression précédente :

```
(letm
 ([x 45] [y (+ 3 4)])
 (print x)
 (cons x y)))
;; expands to ((lambda (x y) (print x) (cons x y)) 45 (+ 3 4))
;; prints 45, → (45 . 7)
```

Exercice 7: Records Scheme

En Scheme, il est possible de définir des types de données usuellement appelés *records* ou enregistrements. Leur équivalent en C est défini par le mot-clé `struct`. En Scheme, on les définit par la macro `define-struct`. On peut aussi plus simplement utiliser la construction suivante :

```
(struct furniture (material number-of-legs number-of-doors))
```

Cette macro définit un type de données *furniture* qui peut alors être utilisé dans la suite du code. En plus de ce type de données, des fonctions sont automatiquement créées :

— pour créer des éléments de type *furniture* :

```
(define chair (furniture "Wood" 3 0))
```

— pour accéder aux champs de l'enregistrement :

```
(furniture-number-of-legs chair) ;; → 3
```

— qu'il est alors possible de modifier :

```
(add1 (furniture-number-of-legs chair)) ;; → 4
```

— et un prédicat pour tester l'appartenance à ce type :

```
(furniture? chair) ;; → #t
```

Le langage étant ce qu'il est, il est possible de personnaliser ces fonctions, par exemple en spécifiant une fonction constructeur ou une fonction pour l'affichage.

Écrire un type de données `rational` pour représenter les nombres rationnels, ainsi que les fonctions `ration-sum` et `ration-prod`.