Programmation Fonctionnelle

Séance 4 : Motifs

Alexandros Singh
Université Paris 8

22 octobre 2023

Problème

Implémentez une fonction en Racket qui renvoie `#t` si son entrée est égale à 1 sinon elle renvoie `#f`.

Problème

Implémentez une fonction en Racket qui renvoie `#t` si son entrée est égale à 1 sinon elle renvoie `#f`.

```
(define check-if-one
  (lambda (x)
    (if (eq? x 1)
        #t
        #f)))
```

Motifs simples

La technique de la correspondance des motifs nous permet de tester si une valeur (ou une expression qui s'évalue à une certaine valeur) correspond à un des motifs proposés. En Racket, dans le cas le plus simple, ceci est réalisé en utilisant la forme suivante :

```
(match exp_0
  [pat_1 exp_1]
  [pat_2 exp_2]
  ...
  [pat_k exp_k])
```

Qui vérifie si `expr_0` correspond à l'un des motifs `pat_1`, `pat_2`, etc, dans l'ordre. Dès qu'une correspondance est trouvée, l'expression qui accompagne le motif est renvoyée comme résultat du calcul. Un motif spécial, `_`, peut être utilisé pour faire correspondre n'importe quelle valeur.

En utilisant ce `match`, nous pouvons maintenant donner une implémentation alternative à la fonction `check-if-one` :

```
(define check-if-one-m
  (lambda (x)
    (match x
      [1 #t]
      [_ #f])))
```

Problème

Implémentez une fonction qui prend en entrée deux arguments `x` et `y` et qui effectue les opérations suivantes :

- Si `y` est 0, alors elle retourne `x` multiplié par 10.
- Si `y` is 1, alors elle retourne `x` plus 10.
- Sinon, il renvoie `x`.

Problème

Implémentez une fonction qui prend en entrée deux arguments `x` et `y` et qui effectue les opérations suivantes :

- Si `y` est 0, alors elle retourne `x` multiplié par 10.
- Si `y` is 1, alors elle retourne `x` plus 10.
- Sinon, il renvoie `x`.

```
(define contrived-func
  (lambda (x y)
    (match y
      [0 (* x 10)]
      [1 (+ x 10)]
      [_ x])))
```

Problème

Implémentez une fonction qui prend deux arguments et renvoie true s'ils sont égaux, sinon elle renvoie false.

Problème

Implémentez une fonction qui prend deux arguments et renvoie true s'ils sont égaux, sinon elle renvoie false.

```
(define check-if-eq
  (lambda (x y)
    (if (eq? x y)
      #t
      #f)))
```

Problème

Implémentez une fonction qui prend deux arguments et renvoie `#t` s'ils sont égaux, sinon elle renvoie `#f`.

```
(define check-if-eq-m
  (lambda (x y)
    (match x
      [_ #:when (eq? x y) #t]
      [_ #f])))
```

Les motifs peuvent être accompagnés de conditions ! Par exemple, ici, le premier motif correspond à n'importe quelle valeur (`_`) sous réserve que `x` soit égal à `y`, tandis que le second motif correspond à n'importe quelle valeur (sans aucune condition).

Nous pouvons donc mettre à jour notre “syntaxe” pour `match` en :

```
(match val-expr clause ...)
clause  =   [pat body ...+]
        |   [pat #:when cond-expr body ...+]
```

Motifs

Jusqu'à présent, nous avons utilisé des valeurs de base (telles que des nombres) à la place de `val-expr` dans la syntaxe précédente, mais nous pouvons également utiliser des expressions plus sophistiquées telles que :

```
(define check-if-eq-m-2
 (lambda (x y)
   (match (cons x y)
     [(cons a a) #t]
     [_ #f])))
```

Ici, nous comparons la paire construite à partir de `x,y` à une paire contenant deux fois la même valeur, auquel cas nous renvoyons `#t`, ou à n'importe quelle autre valeur, auquel cas nous renvoyons `#f`.

Problème

Implémentez une fonction qui renvoie `#t` lorsqu'elle reçoit une paire de nombres dont la somme est égale à 10, sinon elle renvoie `#f`.

```
(define check-pair-sum
  (lambda (p)
    (cond
      [(eq? (+ (car p) (cdr p)) 10) #t]
      [else #f])))
```

Problème

Implémentez une fonction qui renvoie `#t` lorsqu'elle reçoit une paire de nombres dont la somme est égale à 10, sinon elle renvoie `#f`.

```
(define check-pair-sum-m
  (lambda (p)
    (match p
      [(cons a b) #:when (eq? (+ a b) 10) #t]
      [_ #f])))
```

Problème

Implémentez une fonction qui, à partir d'une liste, effectue l'une des opérations suivantes :

- Si la liste est vide, elle renvoie la chaîne de caractères "List has no elements !"
- Si la liste a une longueur comprise entre 1 et 3, elle renvoie le dernier élément de la liste.
- Sinon, il renvoie "La liste est trop grande !"

```
(define return-last-elem
  (lambda (l)
    (cond
      [(eq? (length l) 0) "List has no elements!"]
      [(eq? (length l) 1) (car l)]
      [(eq? (length l) 2) (cadr l)]
      [(eq? (length l) 3) (caddr l)]
      [else "List is too big!"])))
```

Problème

Implémentez une fonction qui, à partir d'une liste, effectue l'une des opérations suivantes :

- Si la liste est vide, elle renvoie la chaîne de caractères "List has no elements !"
- Si la liste a une longueur comprise entre 1 et 3, elle renvoie le dernier élément de la liste.
- Sinon, il renvoie "La liste est trop grande !"

```
(define return-last-elem-m
  (lambda (l)
    (match l
      [_ #:when (null? l) "List has no elements!"]
      [(list a) a]
      [(list a b) b]
      [(list a b c) c]
      [_ "List is too big!"])))
```

Problème

Implémentez une fonction qui, étant donné une liste de nombres, renvoie `#t` si la somme des éléments de la liste est égale à 6, sinon elle renvoie `#f`.

```
(define sum-is-six?
 (lambda (l)
   (match l
     [_ #:when (eq? (foldr + 0 l) 6) #t]
     [_ #f])))
```

Problème

Implémentez une fonction qui, étant donné une liste `l` de nombres, fait ce qui suit :

- Si `l` est vide, elle renvoie "La liste n'a pas d'éléments !"
- Si `l` a un seul élément, elle renvoie l'élément multiplié par 10.
- Sinon, il renvoie la différence des deux premiers éléments plus le produit de tous les autres éléments de la liste.

Motifs

Les différents cas d'utilisation des expressions textttmatch que nous avons vus jusqu'à présent respectaient cette syntaxe :

```
(match val-expr clause ...)
clause  =   [pat body ...+]
        |   [pat #:when cond-expr body ...+]
```

où pat était :

```
pat = id
    | _
    | (cons pat pat)
    | (list lvp ...)
```

Racket propose de nombreux autres types de motifs, voir
`https://docs.racket-lang.org/reference/match.html`.
Par exemple, un type de motif particulièrement pratique est :

```
(list-rest lvp ... pat)
```

Qui associe les premiers éléments d'une liste à des identifiants (commençant par `lvp`) et fait correspondre le reste de la liste à un motif final `pat`. Consultez la page ci-dessus pour découvrir d'autres types de motifs utiles.

Problème

Implémenter une fonction qui, étant donné une liste `l` de nombres, effectue les opérations suivantes :

- Si `l` est vide, elle renvoie `"List has no elements!"`.
- Si `l` a un seul élément, il renvoie l'élément multiplié par 10.
- Sinon, il renvoie la différence des deux premiers éléments plus le produit de tous les autres éléments de la liste.

Problème

Implémenter une fonction qui, étant donné une liste `l` de nombres, effectue les opérations suivantes :

- Si `l` est vide, elle renvoie `"List has no elements!"`.
- Si `l` a un seul élément, il renvoie l'élément multiplié par 10.
- Sinon, il renvoie la différence des deux premiers éléments plus le produit de tous les autres éléments de la liste.

```
(define head-plus-tailprod
  (lambda (l)
    (match l
      [_ #:when (null? l) "List has no elements!"]
      [(list a) (* 10 a)]
      [(list-rest h t) (+ h (foldr * 1 t))])))
```