INFO0054 - Programmation fonctionnelle

Répétition 2 : Récursion sur les nombres et sur les listes

Jean-Michel Begon

12 février 2019

Les primitives `cons`, `list`, `append`

Exercice 1.
Donner le résultat de l'évaluation des expressions suivantes :

```
(car (cdr '(a b)))
(cdr (cdr '(a b)))
(cons 'a '())
(cons 'a '(b))
(cons '() '())
(cons '(a) '(b))
(list '(a) '(b))
(append '(a) '(b))
(cons 'a (cons 'b (cons 'c '())))
(car (cons 'a '()))
(cdr (cons 'a '()))
(cons x '(a b))
(cadr '(a b c d))
(cadar '((a b) (c d) (e f)))
(list? (cons 'a 'b))
(list? (cons 'a (cons 'b '())))
(list? (cons (cons 'b '()) 'a))
(list? (cons (cons 'b '()) (cons 'b '())))
```

```
(null? (car '(a)))
(null? (cdr '(a)))
(null? '())
(null? '(a b))
(null? (car '(())))
(null? (car '((()))))
(symbol? (car '(a b c)))
(symbol? (cons '() '()))
(equal? 'a (car '(a b)))
(equal? '(a b c) '(a b c))
(equal? '(a (b c)) '(a b c))
(equal? (cdr '(a c d)) (cdr '(b c d)))
(equal? '(car '((b) c)) (cdr '(a b)))
(cons (car '(a b c))
      (cons
        (car (cdr '(a b c)))
        (cons (car (cdr (cdr '(a b c)))) '())))
```

Exercice 2.
Définir la fonction `prod-list` qui calcule le produit des éléments d'une liste de nombres.

Exercice 3.
Écrire une fonction `prefix-n` à deux arguments, une liste `ls` et un naturel `n`, et qui renvoie la listes des `n` premiers éléments de `ls`.

Exercice 4.
Écrire une fonction `remove-all` à deux arguments `ls` et `n` dont les valeurs sont respectivement une liste et un nombre entier, qui retourne la liste `ls` privée de toutes les occurrences de `n`.

```
(remove-all '(2 7 1 7 3 1) 1) ⇒ (2 7 7 3)
```

Exercice 5.
Écrire une fonction `zip` qui prend en argument `ls = (l_1 l_2 ... l_n)` et `rs = (r_1 r_2 ... r_n)`, deux listes de `n` éléments et qui renvoie la liste `((l_1 r_1) (l_2 r_2) ... (l_n r_n))`, une liste de `n` éléments dont le `i`eme élément est la liste `(l_i r_i)`.

Exercice 6.
Ecrire une fonction renvoyant le minimum d'une liste de nombres.

1 Accumulateurs

Exercice 7.
Soient les fonctions suivantes :

```
(define sum-int
  (lambda (n)
    (sum-int-acc n 0)))

(define sum-int-acc
  (lambda (n m)
    (if (zero? n) m
        (sum-int-acc (- n 1) (+ n m)))))
```

où `sum-int` calcule la somme des `n+1` premiers naturels. Spécifier `sum-int-acc`.

Exercice 8.
Ecrire une fonction `index-of` prenant en argument une liste `ls` et un naturel `n` renvoyant la (première) position de `n` dans `ls`, ou `#f` si `n` n'apparaît pas dans `ls`.

Exercice 9.
Ecrire une version *tail-recursive* des fonctions
- `sum-list`
- `prod-list`
- `prefix-n`
- `zip`

Exercice 10.
Soient les fonctions suivantes :

```
(define sublist-pos
  (lambda (l)
    (reverse (sublist-pos-acc l '()))))

(define sublist-pos-acc
    (lambda (l acc)
        (cond ((null? l) acc)
              ((> (car l) 0) (sublist-pos-acc (cdr l) (cons (car l) acc)))
              (else (sublist-pos-acc (cdr l) acc)))))
```

`sum-pos` renvoie la sous-liste des éléments positifs de la liste de nombre `l`. Spécifier `sum-pos-acc`.