Programmation fonctionnelle

Cours n°3 : Listes et chaînes de caractères en Racket

Licence 1 Informatique, Université Paris 8

3 Octobre 2022

I. Listes et fonctions de listes

Listes

- Une **liste** est une structure de données permettant de regrouper des données de manière à pouvoir y accéder librement à tout instant.

- En Racket, l'affichage d'une liste est réalisé entre parenthèses précédées d'un symbole **quote** (ou '), afin d'éviter toute ambiguïté avec l'utilisation habituelle des parenthèses.

- On peut créer des listes avec la fonction **list**, ou tout simplement déclarer une liste avec le symbole quote.

```
1 #lang racket
2 (list "a" "b" "c")        '("a" "b" "c")
3 (list 1 2 3)              '(1 2 3)
4 '(1 2 3)                  '(1 2 3)
```

Fonctions de listes

- **list?** permet de tester si le symbole manipulé est une liste;
- **length** permet d'accéder à la longueur d'une liste, c'est-à-dire son nombre d'éléments;
- **empty?** ou plus fréquemment *null?* permet de tester si une liste est vide;
- **first** ou plus fréquemment **car** permet d'accéder au premier élément de la liste;
- **rest** ou plus fréquemment **cdr** permet d'accéder au reste, c'est-à-dire la liste privée de son premier élément;
- L'appel **(cons a L)** prend en arguments un symbole quelconque a et une liste L, et ajoute a au début de L.
- L'appel **(append L1 L2)** prend en arguments deux listes, et renvoie une liste concaténant $L1$ et $L2$. Attention à ne pas confondre **cons** et **append** !
- L'appel **(list-ref L i)** permet de renvoyer l'élément de la liste en position i. Attention, bien se rappeler que l'indexation des éléments d'une liste commence à 0 !

Fonctions de listes

- Si on appelle **car** ou **cdr** sur une liste vide, cela renverra une erreur !
- Si on appelle **cdr** sur une liste à un seul élément, cela renverra une liste vide : '() ou empty.
- Une remarque culturelle : si **first** et **rest** semblent plus intuitifs, **car** et **cdr** sont des fonctions héritées de Lisp en référence à la manipulation des registres en tant qu'emplacement mémoire : CAR est l'abréviation de *Content of the Adress part of the Register* et CDR de *Content of the Decrement part of the Register*.
- Racket propose un raccourci pour enchaîner les appels de **car** et **cdr**, par exemple:
  - **(cadr L)** correspond au premier élément du reste de la liste;
  - **(cddr L)** correspond au reste du reste de la liste;
  - On peut cumuler ces enchaînements (lorsqu'ils ont un sens !) avec 3 appels au maximum, donc pas de **(cddddr L)** par exemple.

Fonctions de listes

- Quelques exemples :

```
 1 #lang racket
 2 (list? '(1 2))                                          #t
 3 (list? (first '(1 2)))                                  #f
 4 (empty? (list 1 2 3))                                   #f
 5 (car (list 1 2 3))                                      1
 6 (cdr (list 1 2 3))                                      '(2 3)
 7 (cons 1 '(2 3 4))                                       '(1 2 3 4)
 8 (append '(1 2) '(3 4 5))                                '(1 2 3 4 5)
 9 (define a 1) (define b 2) '(a b) (list a b)             '(a b)
10 (list-ref (list 1 2 3) 0)                               '(1 2)
11 (list-ref (list 1 2 3) 2)                               1
                                                           3
```

- Notez que dans les lignes 10 et 11, on ne peut passer comme deuxième argument de **list-ref** que les indices accessibles dans la liste, à savoir 0 (correspondant à la valeur 1), 1 (correspondant à la valeur 2) et 2 (correspondant à la valeur 3). Si on essaye **(list-ref (list 1 2 3) 3)**, Racket nous renverra un message d'erreur.

```
list-ref: index too large for list
index: 3
in: '(1 2 3)
```

Exercice 5 du TD-TP n°2

- **Exercice :** Donnez les résultats en Racket des instructions suivantes:
  - (car '(1 2 3 4))
  - (cdr '(1 2 3 4))
  - (car (cdr '(1 2 3 4)))
  - (cdr '(1 (2 3 4)))
  - (cons 1 (2 3 4))
  - (cons '(1 2) '(3 4))
  - (null? '())
  - (null? '(1 2))
  - (cdr (car (cdr '(a (b c) (d e)))))
  - (cons 2 (cdr (cdr '(0 0 1 0 0))))
  - (cdr (car '(1 2 3 4))
  - (null? (cdr '(1 2 3)))
  - (null? (car '(1 2 3)))

Attention aux données de vos listes !

- Jusqu'ici, nous avons manipulé des listes contenant des entiers. Mais on peut imaginer des listes contenant des symboles de types différents, par exemple des booléens, des strings ou encore... des listes !
- Rien n'empêche de déclarer

```
1 #lang racket
2 (define L '('(1 5) 2 #t #\a "abc" (list 1 2 #t)))
3 L
```

Dans ce cas, L est une liste dont le premier élément est la liste $'(1\ 5)$, le 2ème élément est l'entier 2, le troisième élément est le booléen $\#t$, etc.

- Dans ce cas, si on teste alors de nouveau **(list? (car L))**, cela renverra maintenant $\#t$, puisque le premier élément de L est bien une liste !
- Autre remarque, le premier argument de **cons** peut aussi être une liste ! Dans ce cas, on ajoute cette liste comme premier élément de la liste en second argument.

```
1 #lang racket
2 (define L '(1 2 #t '(1 2)))
3 (cons (list "a" 2) L)
```

```
'(("a" 2) 1 2 #t '(1 2))
```

D'autres fonctions

- Par extension de la fonction **cons**, la fonction **list*** permet d'ajouter plusieurs éléments dans une liste.

```
1 #lang racket
2 (list* 9 10 11 (list 1 2 3))      '(9 10 11 1 2 3)
```

- Par réciprocité de la fonction **list-ref** qui permet de lire une valeur, la fonction **list-set** permet d'écrire une valeur et donc remplacer un élément dans une liste.

```
1 #lang racket
2 (define A (list 1 2 3 4 5))
3 (list-ref A 1)                    2
4 (define B (list-set A 1 22))
5 (list-ref B 1)                    22
```

- Ici, la ligne 4 définit une nouvelle liste dans laquelle on remplace l'élément de A d'indice 1 (donc la valeur 2) par 22. D'où le fait que l'élément d'indice 1 de B est 22.

Application d'une fonction à une liste

- On va définir fonction **apply-fun** qui applique une fonction **fun** à chaque élément d'une liste L.

```
1  #lang racket
2  (define apply-fun
3    (lambda (L fun)
4      (if (empty? L)
5          '()
6          (cons (fun (car L)) (apply-fun (cdr L) fun)))))
7
8  (define add1
9    (lambda (x)
10     (+ x 1)))
11
12 (apply-fun (list 1 2 3 4 5) add1)
```

Ici, on applique la fonction **add1** à notre liste; cela a pour effet d'incrémenter toutes les valeurs de L de 1.

- Voici un exemple classique d'une fonction récursive sur une liste. Pour appliquer **fun** à tous les éléments de la liste récursivement, on va appliquer la fonction au premier élément, et l'ajouter avec **cons** en tant que premier élément de l'appel récursif sur le reste.
- Qui dit récursivité dit cas d'arrêt, ici lorsque la liste est vide, on ne peut appliquer **fun** sur rien donc on renvoie une liste vide.

Dernières fonctions de listes

- La fonction **member** permet de savoir si un élément appartient à une liste; si oui, elle retourne la sous-liste commençant par l'élément recherché; sinon la fonction retourne $\#f$.

```
1 #lang racket
2 (member 3 (list 1 2 3 4))          '(3 4)
3 (member 5 (list 1 2 3 4))          #f
```

- La fonction **build-list** permet de construire une liste en appliquant une fonction indexée sur chaque élément.

```
1 #lang racket
2 (build-list 5 values)                                  '(0 1 2 3 4)
3 (define (f x) (* 2 x))                                 '(0 2 4 6 8)
4 (build-list 5 f)
5 (build-list 5 (lambda (x) (* 3 x)))                    '(0 3 6 9 12)
```

La fonction **values** crée une liste contenant les valeurs de 0 jusqu'à $n-1$, où n est le premier argument. On peut changer cette fonction pour appliquer une fonction quelconque à ces valeurs, par exemple multiplier par 2 ou par 3.

- Enfin la fonction de test **equal?** s'applique également aux listes pour tester si elles sont égales.

```
1 #lang racket
2 (define A (list (list 1 2) 3))
3 (define B '((1 2) 3))
4 (equal? A B)                                           #t
```

Liste plate et non plate

- Une liste est plate si elle ne contient pas dans ses éléments des symboles de type liste; visuellement, si on a une seule parenthèse ouvrante et fermante dans la déclaration.

- La fonction **flatten** permet « d'aplatir » une liste, c'est à dire de supprimer toutes les parenthèses hormis celles extérieures.

```
1 #lang racket
2 (define L (list 1 (list 2 (list 3))))
3 (define P '(1 (2 (3))))
4 (define Q (list 1 2 3))
5 (define R (cons 1 (cons 2 (cons 3 '()))))
6 L P Q R
7 (flatten L)
```

```
'(1 (2 (3)))
'(1 (2 (3)))
'(1 2 3)
'(1 2 3)
'(1 2 3)
```

- La liste L est plate, la liste P est équivalente à la liste L. La liste Q est plate. La liste R est équivalente à Q. La liste P aplatie est la même que la liste Q.

Fusion et miroir de listes

- On a déjà vu que la fonction **append** permettait de « fusionner » deux listes.

```
1 #lang racket
2 (define L (list 1 (list 2 (list 3))))
3 (define P '(4 5 6))
4 (append L P)                          '(1 (2 (3)) 4 5 6)
5 (append P L)                          '(4 5 6 1 (2 (3)))
```

Fusion et miroir de listes

- On a déjà vu que la fonction **append** permettait de « fusionner » deux listes.

```
1 #lang racket
2 (define L (list 1 (list 2 (list 3))))
3 (define P '(4 5 6))
4 (append L P)                          '(1 (2 (3)) 4 5 6)
5 (append P L)                          '(4 5 6 1 (2 (3)))
```

- Une autre fonction utile, n'existant pas directement en Racket, est une fonction permettant de renverser une liste, en renvoyant une liste dont les éléments apparaissent dans l'ordre inverse. Voici la fonction :

Fusion et miroir de listes

- On a déjà vu que la fonction **append** permettait de « fusionner » deux listes.

```
1 #lang racket
2 (define L (list 1 (list 2 (list 3))))
3 (define P '(4 5 6))
4 (append L P)                                  '(1 (2 (3)) 4 5 6)
5 (append P L)                                  '(4 5 6 1 (2 (3)))
```

- Une autre fonction utile, n'existant pas directement en Racket, est une fonction permettant de renverser une liste, en renvoyant une liste dont les éléments apparaissent dans l'ordre inverse. Voici la fonction :

```
1 #lang racket
2 (define reverse
3   (lambda (L)
4     (if (empty? L)
5         '()
6         (append (reverse (cdr L)) (list (car L))))))
7
8 (reverse '(1 2 3 4 5))                                  '(5 4 3 2 1)
```

Fusion et miroir de listes

- On a déjà vu que la fonction **append** permettait de « fusionner » deux listes.

```
1 #lang racket
2 (define L (list 1 (list 2 (list 3))))
3 (define P '(4 5 6))
4 (append L P)                                  '(1 (2 (3)) 4 5 6)
5 (append P L)                                  '(4 5 6 1 (2 (3)))
```

- Une autre fonction utile, n'existant pas directement en Racket, est une fonction permettant de renverser une liste, en renvoyant une liste dont les éléments apparaissent dans l'ordre inverse. Voici la fonction :

```
1 #lang racket
2 (define reverse
3   (lambda (L)
4     (if (empty? L)
5         '()
6         (append (reverse (cdr L)) (list (car L))))))
7
8 (reverse '(1 2 3 4 5))                                  '(5 4 3 2 1)
```

- Encore une fonction récursive sur une liste. Imaginons qu'on sache renverser le reste de la liste *L*, alors il faut rajouter le premier élément à la fin de ce résultat. Cependant, **cons** n'autorise de rajouter que des éléments au début. Pas le choix, il faut donc utiliser un **append** mais pour cela il faut utiliser non pas **(car L)** mais une liste contenant le premier élément, d'où le **(list (car L))**.

Combinaison de fonctions et de listes

- La fonction **map** prend en arguments une fonction et une liste d'éléments sur lesquels s'applique cette fonction, retournant la liste modifiée en mémoire par application de la fonction. Ceci implique un lien entre le type des paramètres de la fonction et le type des éléments de la liste.
- La fonction **andmap** retourne un *et logique* de l'application d'un prédicat sur les éléments d'une liste. Ainsi, le résultat sera $\#t$ si le prédicat est évalué à vrai sur tous les éléments, et $\#f$ sinon.
- La fonction **ormap** a la même fonction avec un *ou logique*. Elle retournera donc $\#t$ si l'un des éléments de la liste évalue le prédicat à vrai, et $\#f$ sinon.

```
1 #lang racket
2 (map sqrt (list 1 4 9))                              '(1 2 3)
3 (map (lambda(x) (* x x)) (list 0 2 4))               '(0 4 16)
4 (andmap integer? (list 1 2 3 4))                     #t
5 (andmap integer? (list 1 2 3 4 #t))                  #f
6 (ormap integer? (list #t '(1 3) 2))                  #t
7 (ormap (lambda(v) (< v 10)) (list 12 13 25))         #f
```

II. Chaînes de caractères

Chaînes de caractères

- Les caractères **ASCII** (American Standard Code for Information Interchange, norme informatique de codage de caractères) sont définis par les trois symboles « # », « \ » et le caractère lui-même.
  - Ainsi, « #\a » est le caractère *a*.
  - Le caractère espace se note « #\space »;
  - Le caractère nouvelle ligne se note « #\newline »
- Une chaîne de caractères est appelée **string**; dans une string, nouvelle ligne devient « \n ».
- Une string définit une suite de caractères et est encadrée par le caractère double quote, noté ".
- Lorsqu'on manipule une string, on va vouloir accéder à certains de ses caractères : attention, en Racket les éléments sont numérotés à partir de 0. Ainsi, le premier élément d'une string sera appelé en position 0, le deuxième élément en position 1, etc.

Strings en Racket

- Voici plusieurs fonctions utilisables sur des chaînes de caractères en Racket, ainsi que les résultats de l'exécution :

```
1  #lang racket
2  "abcde"
3  (string #\a #\b #\c #\d #\e)
4  (make-string 5 #\a)
5  (string-append "a" "bc")
6  (string-length (string-append "a" (string #\b #\c)))
7  (define s (string #\a #\b #\c #\d #\e))
8  (string-set! s 0 #\A)
9  (string-set! s 4 #\a)
10 s
11 (string-ref s 1) (substring s 1 3) (substring s 1)
```

```
"abcde"
"abcde"
"aaaaa"
"abc"
3
"Abcda"
#\b
"bc"
"bcda"
```

Strings en Racket

- Les lignes 1 à 3 présentent 3 méthodes différentes de définition d'une string : la première est non modifiable, alors que les deux suivantes le sont.

- La ligne 5 construit une string en concaténant bout à bout deux strings, avec **string-append**. La ligne 6 présente la fonction **string-length** permettant de connaître la taille d'une chaîne de caractères.

- La ligne 7 à 11 présentent comment définir et utiliser une string nommée *s* avec différentes fonctions. La fonction **string-set!** permet de modifier un caractère dans une string en précisant la position de celui-ci et le nouveau caractère à placer.

- La fonction **string-ref** permet de récupérer le caractère d'une string à une position donnée; la fonction **substring** permet de récupérer une partie d'une string, précisant la position de départ à une position d'arrivée.

Valeur des caractères ASCII

- Les caractères correspondant à des valeurs entières définies selon la norme ASCII. La fonction **char->integer** permet de convertir un caractère ASCII en sa valeur numérique; et inversement la fonction **integer->char** permet de convertir une valeur numérique en caractère.

```
1 #lang racket
2 (integer->char 10)          #\newline
3 (integer->char 97)          #\a
4 (char->integer #\A)         65
5 (char->integer #\Z)         90
```

Symboles et valeurs associées

- Un symbole est associé à une valeur. Il est possible d'avoir plusieurs symboles associés à une valeur :

```
1 #lang racket
2 (define s (string #\h #\e #\e #\l #\o))
3 (define s2 (string-copy s))
4 (define s3 s)
5 (string-set! s 2 #\l)
6 s                                          "hello"
7 s2                                         "heelo"
8 s3                                         "hello"
```

- Dans cet exemple, *s* et *s*3 sont des symboles associés à une même valeur; modifier la valeur associée à *s* modifie également la valeur associée à *s*3. En revanche, *s*2 ne change pas, grâce à la fonction **string-copy**.

Modification d'une string

- La fonction **string-copy!** permet de recopier une partie d'une string dans une autre string: par exemple,

```
1 #lang racket
2 (define s (string-append (make-string 10 #\a) "bc"))
3 (define s1 (string #\A #\B))
4 (string-copy! s 0 s1 0 2)
5 s
```

remplace le premier caractère de s par la sous-string de $s1$ contenant les caractères 1 et 2. Ceci renvoie donc *ABaaaaaaaabc*.

- La fonction **string-fill!** permet de remplacer tous les caractères d'une string par un caractère.

```
1 #lang racket
2 (define s (string-append (make-string 10 #\a) "bc"))
3 (string-fill! s #\d)
```

renvoie *dddddddddddd*.

Comparaison de strings

- Les comparaisons entre strings utilisent les comparaisons caractère par caractère, et sont réalisées avec les fonctions **string=?**, **string<?**, **string<=?** et symétriquement **string>?**, **string>=?**.

- La première fonction compare si deux strings sont identiquement égales; la deuxième compare si une string est inférieure à l'autre dans **l'ordre lexicographique**, qui est l'ordre du dictionnaire, etc.

- Pour comparer des caractères, on peut utiliser la fonction **equal?**, qui est plus généraliste. La fonction **equal?** peut même être utilisée pour des strings, mais on préfèrera **string?** pour plus de clarté et plus de vigilance.

```
1 #lang racket
2 (equal? #\a #\b)                          #f
3 (equal? (string #\a #\b #\c) "abc")       #t
4 (string=? "abc" "abd")                    #f
5 (string<? "abc" "bd")                     #t
6 (string<=? "abc" "abb")                   #f
7
```

Remplacement de caractères, conversion de nombres

- La fonction **string-replace** permet de remplacer un groupe de caractères d'une string par une autre :

```
1 #lang racket
2 (string-replace "abcdefghi" "def" "DEF")
```

renvoie la string *"abcDEFghi"*.

- La fonction **number->string** permet de convertir un nombre (en base 10) en string; et la fonction **string->number** permet de convertir un string en nombre (en base 10).

```
1 #lang racket
2 (number->string 121)                    "121"
3 (string->number "00125")                125
4 (string->number "abc")                  #f
```

- Pour que **string->number** renvoie un nombre, il faut que les caractères de la string passée en paramètres soient numériques; sinon, cela renverra un booléen $\#f$ comme ci-dessus.