7 Listes

7.1 Définition

La fonction `list` permet de créer des listes [37].

```
1 (list "a" "b" "c")
2 (list 1 2 3)
```

```
'("a" "b" "c")
'(1 2 3)
```

L'affichage des listes est réalisé entre parenthèses précédées d'une quote, afin de retirer toute ambiguité possible avec l'utilisation habituelle des parenthèses qui annoncent un appel de fonction.

```
1  (length (list 1 2 3))
2  (empty? (list 1 2 3))
3  (list-ref (list 1 2 3) 0)
4  (first (list 1 2 3))
5  (rest (list 1 2 3))
6  (define L empty)
7  (cons 1 L)
8  (cons 2 (cons 1 L))
9  L
10 (define a 1)
11 (define b 2)
12 '(a b)
13 (list a b)
14 (list? (first '(1 2)))
15 (list? (rest '(1 2)))
```

```
3
#f
1
1
'(2 3)
'(1)
'(2 1)
'()
'(a b)
'(1 2)
#f
#t
```

La première ligne présente la fonction `length` retournant le nombre d'éléments contenus dans une liste ; la ligne 2 présente le prédicat `empty?` permettant de savoir si une liste est vide ; la ligne 3 présente la fonction `list-ref` retournant l'élément présent à une position spécifiée dans une liste ; la ligne 4 présente la fonction `first` retournant le premier élément d'une liste [38] ; `first` d'une liste

37. Racket est un dialecte du langage LISP, abréviation de « LISt Processor », ayant une relation privilégiée avec les listes.

38. Parfois cet élément sera une liste, comme dans `(first '((2)))` qui retourne `'(2)`.

vide retourne une erreur ; la ligne 5 présente la fonction `rest` retournant une liste privée de son premier élément [39] ; `rest` d'une liste vide retourne une erreur ; `rest` d'une liste contenant un seul élément retourne la liste vide ; la ligne 6 présente la définition d'une liste vide ; les lignes 7 et 8 présentent la fonction `cons` permettant de construire une nouvelle liste par ajout d'un élément au début d'une liste ; la ligne 9 présente la liste L non modifiée ; les lignes 10 à 13 présentent la différence entre la quote et la fonction `list` pour créer une liste [40] ; enfin le prédicat `list?` permet de savoir si un paramètre est une liste.

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

```
1 (list* 9 10 11 (list 1 2 3))
```

```
'(9 10 11 1 2 3)
```

Par réciprocité à 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 ; pour une liste de `N` éléments, les indices varient de 0 à `N-1`.

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

```
2
22
```

Les fonctions `first` et `rest` étaient à l'origine appelées `car` et `cdr` ; ces fonctions sont toujours présentes [41]. Dans le contexte de la manipulation des listes, `first` et `rest` sont plus intutifs.

```
1 (car (list 1 2 3))
2 (cdr (list 1 2 3))
```

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

39. Parfois cette liste sera la liste vide, comme dans `(rest '((2)))` qui retourne `'()`.

40. La fonction `quote` existe également ; elle précise que la parenthèse ouvrante qui suit n'annonce pas une fonction, donc n'est pas sujette à interprétation ; `(list 1 2)` est équivalent à `'( 1 2)` et à `(quote( 1 2))` ; `'(1 2)` définit une liste à deux éléments ; `'(1 2 (3 4))` définit une liste à trois éléments, dont le troisième est également une liste à deux éléments ; `'(1 2 '(3 4 5))` définit une liste à trois éléments, dont le troisième est une liste à 2 éléments (qui sont `quote` et une liste à trois éléments (qui est la liste composée de `3`, `4` et `5`) ; l'utilisation de `quote` dans la définition de liste implique donc une attention particulière afin d'éviter les confusions.

41. Le nom de ces fonctions est hérité de la première version de Lisp, en référence à la manipulation des registres en tant qu'emplacement mémoire et non pas en tant que registre d'architecture des ordinateurs ; CAR est l'abréviation Contents of the Adress part of the Register et CDR de Content of the Decrement part of Register number.

7.2 Parcours de liste

Pour réaliser un parcours de liste, on peut définir une fonction `apply-fun` qui applique une fonction `fun` à chaque élément.

```
1 (define (apply-fun l fun)
2   (if (empty? l) empty
3     (cons (fun (first l)) (apply-fun (rest l) fun))
4   ))
5 (apply-fun (list 1 2 3 4 5) add1)
```

```
'(2 3 4 5 6)
```

En appliquant (ligne 5) la fonction `add1` à chaque élément d'une liste, on obtient une liste incrémentée.

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

```
1 (member 3 (list 1 2 3 4))
2 (member 5 (list 1 2 3 4))
```

```
'(3 4)
#f
```

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

```
1 (build-list 5 values)
2 (define (f x) (* 2 x))
3 (build-list 5 f)
4 (build-list 5 (lambda (x) (* 3 x)))
```

```
'(0 1 2 3 4)
'(0 2 4 6 8)
'(0 3 6 9 12)
```

Le premier élément est associé à l'indice 0 et les indices sont ensuite incrémentés ; avec le mot clé `values` à la place de la fonction indexée, on obtient les valeurs des indices.

Le prédicat `equal?` présenté à la section 4.4 s'applique également aux listes.

```
1 (define A (list (list 1 2) 3) )
2 (define B '((1 2) 3) )
3 (equal? A B)
```

```
#t
```

La comparaison réalisée à la ligne 3 montre que l'utilisation d'une quote pour la déclaration d'une liste (comme à la ligne 2) permet de déclarer des listes en utilisant simplement une parenthèse ouvrante ; avec une quote suivi de deux parenthèses ouvrantes imbriquées, on obtient une liste de liste.

7.3 Liste plate et non plate

Une liste plate est une liste dont les éléments ne sont pas eux-même des listes.

```
(define L (list 1 (list 2 (list 3))))
(define P '(1 (2 (3))) )
(define Q (list 1 2 3))
(define R (cons 1 (cons 2 (cons 3 empty))))
L
P
Q
R
(flatten L)
```

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

La liste L est non plate ; la liste P est équivalente à L ; la liste Q est plate ; la liste R est équivalente à Q ; la fonction `flatten` permet d'aplatir une liste.

7.4 Fusion de listes

La fonction `append` permet de fusionner des listes ; les éléments de chacunes des listes passées en paramètre sont insérés dans l'ordre dans la liste retournée.

```
(define L (list 1 (list 2 (list 3))))
(define P (list 1 2 3))
(append L P)
```

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

La fonction `revl`, définie ci-dessous, inverse le contenu d'une liste.

```
(define (revl l)
  (if (empty? l) empty
    (append (revl (rest l)) (list (first l)) )
  ))
(revl (list 1 2 3 4 5))
```

```
'(5 4 3 2 1)
```

7.5 Combinaison de fonctions et de listes

La fonction `map` prend une fonction et une liste d'éléments sur lesquels s'applique cette fonction, retournant à une modification de la liste par application de la fonction passée en paramètre, ce qui implique un lien entre le type des paramètres de la fonction et le type des éléments de la liste en paramètre.

```
1 (map sqrt (list 1 4 9))
2 (andmap integer? (list 1 2 3 4))
3 (define (inf10 v) (< v 10))
4 (ormap inf10 (list 2 4 6 8 10))
```

```
'(1 2 3)
#t
#t
```

La fonction `andmap` retourne un et-logique de l'application d'un prédicat sur les éléments d'une liste ; la fonction `ormap` en retourne le ou-logique.

La fonction `foldl` applique une fonction passée en paramètres aux éléments d'une liste ; dans ce cas, le retour de la fonction conditionne le format du résultat de l'exécution de `foldl`.

```
1 (foldl (lambda (e v) (+ v (* e e))) 0 (list 1 2 3))
2 (foldl + 0 (map sqr (list 1 2 3)))
```

```
14
14
```

La fonction appliquée par `foldl` est l'addition et la valeur retournée est initialisée à zéro.

En utilisant `foldl`, la fonction récursive enveloppée `(revl L)` se définit en une seule ligne.

```
1 (define (revl L) (foldl cons empty L))
```