1 Palindromes

Une chaîne de caractères est un palindrome si elle est symétrique par rapport à son milieu. Les chaînes “ABCBA” et “00100100” sont des palindromes. Pour tester si une chaîne est un palindrome, on utilise l’algorithme récursif suivant :

Soit `str` est une chaîne à vérifier. On teste si le premier caractère de `str` et le dernier caractère de `str` sont identiques. Si c’est le cas, on compare le deuxième caractère de `str` avec le caractère en avant-dernière position dans la chaîne, et ainsi de suite. Le processus se termine lorsque l’on parvient au milieu de la chaîne (auquel cas, on peut conclure que la chaîne est effectivement un palindrome) ou bien lorsque deux caractères à comparer sont différents (on peut en conclure alors que la chaîne n’est pas un palindrome).

En Caml, une chaîne est une suite de caractères entre guillemets (`"`). Il existe un certain nombre de fonctions prédéfinies permettant de manipuler des chaînes. Nous vous proposons d’en utiliser deux :

– La fonction `String.length : String -> int` qui donne la taille de la chaîne passée en argument.
– La fonction `String.get : String -> int -> char`. Lors de l’appel `String.get str n`, cette fonction renvoie le caractère situé à l’indice `n` de la chaîne `str`.

Notons que l’indice des caractères d’une chaîne commence à `0`. Ainsi l’indice du dernier caractère d’une chaîne `str` est `(String.length str) - 1`.

Pour implanter l’algorithme ci-dessus, vous utiliserez la fonction Caml
`estPalindrome : string->boolean`
définie de la manière suivante :

```
let estPalindrome str =
      estPalindromeMinMax str 0 (String.length str - 1)
```

où l’appel `estPalindromeMinMax str m n` teste si une sous-chaîne entre l’indice `m` et `n` de la chaîne `str` est un palindrome.

Écrivez la définition Caml de la fonction récursive `estPalindromeMinMax : string -> int -> int -> boolean`.

2 Typage et écriture d’expressions

1. Donner le type des expressions suivantes :

```
(2,3);;
( 2, (3. ,3.0));;
((3. , 2., 1.), (true, false));;
function (x,y) -> x*10 + y;;
function g -> function (x,y,z) -> g x & g y & g z ;;
function f->function (x,y) -> f x + f y;;
function g-> function (x1,y1,z1)-> function (x2,y2,z2) ->
( g (x1,x2), g (y1,y2), g (z1,z2));;
```

2. Donner une expression de chacun des types ci-dessous :
   – $float * float \rightarrow float$
   – $float * (float \rightarrow float)$

– $int \rightarrow int \rightarrow int * int$
– $(int * int \rightarrow bool) \rightarrow bool * bool$

3. Soit la fonction f définie par $f(x, y) = x + y$
   Donner deux fonctions CAML $f1$ et $f2$, respectivement de type $int * int \rightarrow int$ et $int \rightarrow int \rightarrow int$, modélisant la fonction f.
   Quels sont les appels à $f1$ et $f2$ permettant de calculer $2 + 3$?
   Avec laquelle des deux fonctions est-il possible de dériver la fonction inc de type $int \rightarrow int$ dont le résultat est d'ajouter 1 à son argument ?

3 Fonctions de base sur les listes

Dans cet exercice, on programme quelques fonctions de base sur les listes, fonctions qui pré-existent dans la bibliothèque de fonctions Caml, à partir du schéma suivant donné en cours :

```
let rec f_s_l =
    function
       [] -> <cas de la liste vide>
     | <tete>::<queue> -> expression(<tete>, f_s_l <queue>) ;;
```

où expression représente les traitements à effectuer sur la tête de liste et la queue à l'aide du rappel récursif de la fonction.

1. Ecrire les fonctions :
   (a) `longueur : 'a list -> int`, qui à une liste associe son nombre d'éléments,
   (b) `concat : 'a list -> 'a list -> 'a list`, qui à partir de 2 listes en produit une troisième constituée des 2 mises bout à bout
   (c) `renverse : 'a list -> 'a list`, qui à partie d'une liste `(`a_1`, `a_2`,...,`a_n`)` retourne une liste `(`a_n`, ..., `a_2`, `a_1`)`,
   (d) `appartient : 'a -> 'a list -> bool`, qui à une liste `L` et une valeur `v` associe vrai ou faux selon que `v` est dans `L` ou non.
2. Ecrire une fonction `map : ('a -> 'b) -> 'a list -> 'b list`, qui à une fonction `f` et une liste `[`a_1` ; `a_2` ;...; `a_n`]` associe la liste formée de `f(`a_1`), f(`a_2`), ..., f(`a_n`)`.
   Profils de fonctions de la bibliothèque Caml sur les listes utilisables ultérieurement :
   `List.length : 'a list -> int`, longueur d'une liste
   `(@) : 'a list -> 'a list -> 'a list`, concaténation de 2 listes en une
   `List.hd : 'a list -> 'a`, premier élément de la liste
   `List.tl : 'a list -> 'a list`, liste privée de son premier élément
   `List.rev : 'a list -> 'a list`, retourne une liste inversée
   `List.mem : 'a -> 'a list -> bool`, teste l'appartenance d'un élément à une liste
   `List.map : ('a -> 'b) -> 'a list -> 'b list`, retourne une liste comme map ci-dessus
   `List.for_all : ('a -> bool) -> 'a list -> bool`, traduit le quel que soit
   `List.exists : ('a -> bool) -> 'a list -> bool`, traduit le il existe

4 Aggrégation de résultats

La fonction Ocaml `List.fold_left` permet d'aggréger un résultat en parcourant une liste à partir d'une valeur intiale et d'une fonction qui utilise la valeur aggrégée courante et l'élement de la liste que l'on considère. La liste est parcourue de la gauche vers la droite. Le type de `List.fold_left` est le suivant `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>`.

Considérons maintenant l'exemple suivant d'une liste d'entiers `l` sur laquelle on applique `List.fold_left` en utilisant la fonction `somme` et 0 comme valeur initiale. Cette opération permet de sommer les éléments de `l`

```
let somme x y = x+y;;
let l = [1; 2; 3; 4; 5];;
List.fold_left somme 0 l;;
-: int = 15
```

Nous pouvons décrire l'exécution de cette fonction de la manière suivante :

```
somme (somme (somme (somme (sommme 0 1) 2) 3) 4) 5
           somme (somme (somme (somme 1 2) 3) 4) 5
                     somme (somme (somme 3 3) 4) 5
                               somme (somme 6 4) 5
                                         somme 10 5
```

Ecrivez une fonction `accumule` correspondant à la fonction `List.fold_left`. Cette fonction aura pour paramètres :

1. une fonction `f`
2. une valeur initiale `initval`
3. et une liste `l`

Approche : Le corps de la fonction `accumule` sera composé d'une fonction récursive `boucle` ayant pour paramètres la valeur d'accumulation `accuval` et la liste `l`.