mardi 19 décembre 2023

Université Paris-Saclay — Faculté des sciences d'Orsay

PFA
L3 Info./LDD3 Mag. Info

Examen

Durée 2h00, tiers temps additionnel 40 minutes

Consignes l'examen dure 2h00 et est sur 20 points. Seules les notes personnelles sont autorisées. Les téléphones portables doivent être **éteints** et **rangés**.

Le sujet se termine sur la page 3. Un aide mémoire OCaml est disponible à la page 5 (vous pouvez le détacher du sujet pour le consulter pendant votre lecture). Le barème est indicatif et proportionnel à la difficulté des exercices.

1 Cordes

On se propose de travailler sur la structure de données de *cordes* (*ropes* en anglais). Une corde est une structure de données utilisée pour représenter des suites de caractères, dans lesquelles on souhaite pouvoir faire des modifications arbitraires, tout en évitant de faire un trop grand nombre de copies. Une corde est une forme particulière d'*arbre binaire de recherche*. Les feuilles stockent des chaînes de caractères, alors que les nœuds internes représentent une concaténation entre ces chaînes. On stocke dans chaque nœud la hauteur de l'arbre, le nombre total de caractères stockés et les deux sous-arbres. Les cordes sont représentées par le type OCaml suivant :

```
1  type rope =
2      Str of string                        (* feuille *)
3    | Node of int * int * rope * rope   (* hauteur, taille,
4                                            arbre gauche et droit *)
```

Par exemple, la chaîne `"La programmation fonctionnelle, c'est super!"` peut être représentée par l'arbre de la figure 1. Dans cette figure, les nœuds internes représentent le constructeur `Node` (contenant la hauteur, le nombre de caractères et les deux sous-arbres). Par exemple, le sous-arbre gauche de la racine est un nœud de concaténation. La totalité du texte qu'il contient est de longueur 21 (la chaîne `"La programmation fonc"` est de taille 21). Et la hauteur de cet arbre est de 1 (en prenant la convention qu'une feuille est de hauteur 0). De même, le sous-arbre droit de la racine est de hauteur 2 (car l'un de ses sous-arbre est de hauteur 1) et la taille totale du texte qu'il contient est 23. La racine est de hauteur 3 et sa taille (qui correspond à la longueur total de notre texte) est de 44. Le but de l'exercice est d'écrire un ensemble de fonctions permettant de manipuler des cordes.

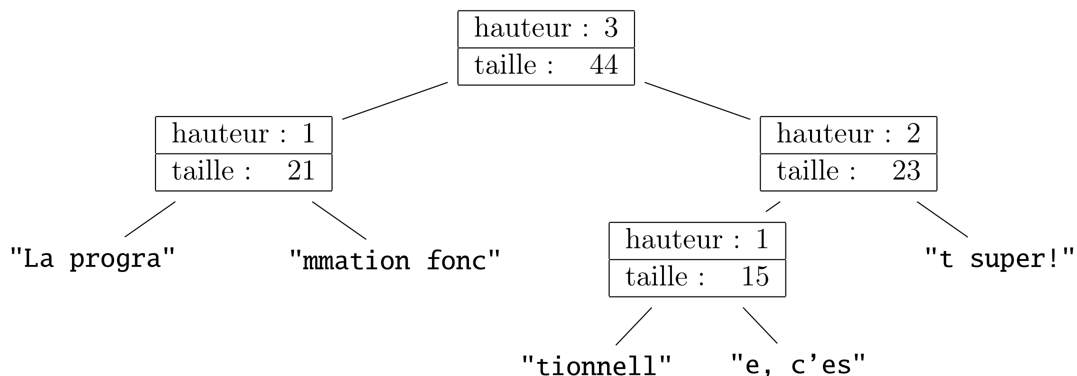

Figure 1 – Une corde pour la phrase d'exemple

Comme tout arbre binaire de recherche, les fonctions n'auront une bonne complexité que si l'arbre est équilibré. On ignore ce problème dans un premier temps.

Fonctions de base (10 points)

1. (0.75 point) Donner une valeur OCaml de type `rope` qui représente le sous-arbre droit de la racine, pour l'arbre de la figure 1.
2. (1 point) Donner une fonction `height : rope -> int` qui renvoie la hauteur de l'arbre donné (attention, on veut juste renvoyer la hauteur stockée, **on ne demande pas de la recalculer en parcourant l'arbre**). On utilise la convention qu'une feuille est de hauteur 0.
3. (1 point) Donner une fonction `length : rope -> int` qui renvoie le nombre de caractères de l'arbre donné (attention, on veut juste renvoyer la longueur stockée, **on ne demande pas de la recalculer en parcourant l'arbre**). La fonction `String.length` renvoie la taille d'une chaîne.
4. (1.25 point) Écrire une fonction `node : rope -> rope -> rope` qui construit un nœud entre deux cordes données, c'est à dire qui construit `Node(`h`,`l`, r1, r2)` où h et l sont calculés à partir de `t1` et `t2`.
5. (2 points) Écrire une fonction `collect : rope -> string list` qui renvoie la liste de toutes les chaînes se trouvant aux feuilles. Les chaînes doivent se trouver dans l'ordre du texte.
6. (0.5 point) Écrire une fonction `to_string : rope -> string` qui convertit une corde en chaîne de caractères (on pourra utiliser la fonction `collect` puis concaténer toutes les chaînes ensemble grâce à `String.concat`).
7. (2 points) Écrire une fonction `get : rope -> int -> char` tel que `get r i` renvoie le `i`$^{\text{ème}}$ caractère de la corde `r`. Vous pouvez supposer que `i` est un indice valide (i.e. positif ou nul et strictement inférieur à la longueur totale). La fonction s'écrit récursivement :
   — si `r` est une feuille, il suffit de renvoyer le `i`$^{\text{ème}}$ caractère de la chaîne
   — sinon, c'est un nœud constitué de deux sous-cordes `r1` et `r2`. Si `i` est inférieur à la longueur de `r1`, on peut recherche dans `r1`. Sinon on recherche dans `r2`. Attention, dans ce deuxième cas, il faut retrancher la longueur de `r1`.

   Par exemple, pour prendre le caractère d'indice 30 dans la corde représentée à la figure 1, Il faut prendre le $30 - 21 = 9^{\text{ème}}$ caractère du sous-arbre droit (et procéder récursivement) pour trouver au final `','`.
8. (1.5 point) Écrire une fonction `iter : (char -> unit) -> rope -> unit` telle que `iter f r` appelle la fonction `f` tour à tour sur tous les caractères de la corde `r` dans l'ordre donné.

Extraction de texte (4.5 points)

On considère la fonction `String.sub : string -> int -> int -> string` des chaînes de caractères OCaml. L'appel `String.sub s i len` renvoie la sous-chaîne de `s` commençant à l'indice `i` et de longueur `len`. Par exemple

```
String.sub "ABCDEFGHIJKLMNO" 8 5
```

renvoie `"IJKLM"`. On souhaite implémenter la même opération mais pour des cordes.

1. (3 points) Écrire une fonction `sub : rope -> int -> int -> t` qui est telle que `sub r i len` renvoie la corde correspondants aux caractères compris entre le `i`$^{\text{ème}}$ (inclus) et le `i+len`$^{\text{ième}}$ (exclus) de `r`. On suppose que les l'indice `i` et la longueur `len` sont valides.

   Par exemple, pour la corde de la figure 1, `sub r 17 13` renvoie la corde contenant le texte `"fonctionnelle"`. On pourra procéder récursivement :
   — si la corde est une feuille, utiliser `String.sub` (cf. question 5).
   — sinon on a trois cas :
   — `i + len` est inférieur ou égal à la longueur de la corde gauche, on s'appelle récursivement sur la corde gauche

— `i` est supérieur ou égal à la longueur de la corde gauche, on s'appelle récursivement sur la corde droite

— sinon le texte cherché est à cheval sur les deux cordes. Si on appelle `n` la longueur entre `i` la taille de la corde gauche alors, on va chercher récursivement à gauche la portion de texte commençant en `i` de longueur `n`, dans le sous-arbre droit la portion de texte commençant à l'indice 0 et de longueur `len - n` et on réunit les deux cordes ainsi obtenues.

Pour reprendre notre exemple, on souhaite calculer `sub r 17 13`. Comme la taille $17 + 13 = 30$ est supérieur à la longueur du sous-arbre gauche (21 caractères), on va chercher la sous-chaîne de taille $21 - 17 =$ commençant à l'indice 17 dans le sous-arbre gauche (`Str "fonc"`) et la sous-chaîne de taille $13 - 4 = 9$ commençant à l'indice 0 dans le sous-arbre droit. Cette dernière est elle-même à cheval sur deux sous-arbres, donnant `Node(...,Str "tionnell", Str "e")`.

2. (1.5 point) Écrire une fonction `insert : t -> int -> t -> t` telle que `insert r i s` insère la corde `s` à la position `i` (en utilisant la fonction de la question précédente) dans `r`. Une façon simple est de partager `r` en deux à la position `i`, puis de reformer une corde en concatenant dans le bon ordre les trois cordes ainsi obtenues.

Équilibrage (5.5 points)

On s'intéresse maintenant à produire des arbres équilibrés. Une propriété (que l'on ne démontre pas) est qu'une arbre de hauteur h est équilibrée si elle contient au moins F_{h+2} caractères, où F_n est les $n^{\text{ième}}$ nombre de Fibonacci. On rappelle que cette fonction est définie par la relation de récurrence :

$$\begin{array}{lcl} F_0 & = & 0 \\ F_1 & = & 1 \\ F_n & = & F_{n-1} + F_{n-2} \text{ pour } n \geq 2 \end{array}$$

1. (1.5 point) Écrire une fonction `fib_array n` qui crée un tableau de taille `n` et telle que la case `i` contient le `i`$^{\text{ème}}$ nombre de Fibonacci.

   Indication on ne demande pas d'écrire la fonction Fibonacci de façon récursive, mais simplement d'initialiser les cases 0 et 1 du tableau à F_0 et F_1 puis de remplir le reste des cases en utilisant les cases précédentes.

2. (1 point) On suppose qu'un tableau global `fib` a été initialisé par

```
let fib = fib_array 80
```

   Écrire une fonction `is_balanced : rope -> bool` qui renvoie `true` si et seulement si la corde donnée en argument est équilibrée.

3. (2 points) Écrire une fonction `merge : string array -> int -> int -> rope`. Cette dernière prend en argument un tableau de chaînes de caractères, un indice de début i (inclus) et un indice de fin j (exclus) et crée une corde dont les feuilles sont toutes les chaînes comprises entre i et j dans le tableau. On procède de façon récursive :

   — si `i - j = 1`, on fabrique une feuille avec la chaîne à l'indice `i` ;

   — si `i - j = 2`, on fabrique deux feuilles avec les chaînes aux indices `i` et `i+1` et on les réunit par un nœud ;

   — sinon on calcule l'indice `mid` se trouvant entre `i` et `j`. On fusionnes récursivement les feuilles se trouvant entre `i` et `mid` d'une part et entre `mid` et `j` d'autre part et on les réunit par un nœud.

4. (1 point) Écrire une fonction `balance : rope -> rope` qui rééquilibre une corde si besoin. Si la corde donnée en argument n'est pas équilibrée, alors on crée la liste de ses feuilles (avec `collect`), on transforme cette liste en tableau avec la fonction prédéfinie

```
Array.of_list : 'a list -> 'a array
```

   et on appelle `merge` sur les bons indices.

Aide-mémoire OCaml

Cet aide mémoire rappelle les types de bases en OCaml ainsi que les fonctions utilitaires associées ainsi que leurs types. Attention, toutes ces fonctions ne sont pas forcément utiles pour les exercices. Dans la suite, lorsqu'une fonction est marquée comme « opérateur binaire » (par exemple `+`) cela signifie qu'il faut l'écrire `a op b`. Sinon c'est une fonction qu'il faut appeler avec `op a b`.

Entiers

Le type `int` représente des entiers signés. Les constantes entières s'écrivent simplement `0`, `42`, `-233`. Les opérations et fonctions sur les entiers sont :

`+ : int -> int -> int` addition entre deux entiers (opérateur binaire).
`- : int -> int -> int` soustraction entre deux entiers (opérateur binaire).
`* : int -> int -> int` multiplication entre deux entiers (opérateur binaire).
`/ : int -> int -> int` division **entière** entre deux entiers (opérateur binaire).
`mod : int -> int -> int` reste dans la division entière (opérateur binaire).
`int_of_string : string -> int` conversion d'une chaîne en entier. Lève une exception si la chaîne n'est pas au bon format.
`int_of_float : float -> int` conversion d'un flottant en entier (la partie décimale est tronquée).

Flottants

Le type `float` représente des nombre flottants. Les constantes flottantes s'écrivent en notation scientifique `0.5`, `42e3`, `-233.8e-20`. Les opérations et fonctions sur les flottants sont :

`+. : float -> float -> float` addition entre deux flottants (opérateur binaire).
`-. : float -> float -> float` soustraction entre deux flottants (opérateur binaire)
`*. : float -> float -> float` multiplication entre deux flottants (opérateur binaire)
`/. : float -> float -> float` division entre deux flottants (opérateur binaire)
`** : float -> float -> float` puissance entre deux flottants (opérateur binaire)
`float_of_string : string -> float` conversion d'une chaîne en flottant. Lève une exception si la chaîne n'est pas au bon format.
`float : int -> float` conversion d'un flottant en entier (la partie décimale est tronquée).
`sqrt : float -> float` racine carrée d'un flottant.

Booléens

Le type `bool` représente des booléens. Les constantes booléennes sont **`true`** et **`false`**. Les opérations et fonctions sur les booléens sont :

`&& : bool -> bool -> bool` « et » logique entre deux booléens (opérateur binaire).
`|| : bool -> bool -> bool` « ou » logique entre deux booléens (opérateur binaire).
`not : bool -> bool` négation d'un booléen.

Chaînes de caractères

Le type `string` représente des chaînes de caractères. Les chaînes de caractères constantes sont délimitées par des guillemets :`"Hello, world !"`. De façon usuelle, la séquence d'échappement `\n` représente un retour à la ligne. Les opérations et fonctions sur les chaînes sont :

s`.[`i`]` accède au $i^{\text{ème}}$ caractère de la chaîne s. Les chaînes ne sont pas modifiables et les indicies commencent à 0.
`^ : string -> string -> string` concaténation entre deux chaînes (opérateur binaire).

`String.length : string -> int` longueur d'une chaîne.
`String.trim : string -> string` renvoie une copie de la chaîne où les blancs (espaces, tabulations, retours à la ligne) en début et en fin de chaîne ont été supprimés.
`String.split_on_char : c -> string -> string list` découpe une chaîne donnée en une liste de chaînes, en utilisant le caractère donné comme séparateur.
`String.concat : string -> string list -> string` concatène une liste de chaînes en les séparant par la chaîne de séparation donnée.

Affichage

La fonction `Printf.printf fmt arg1 arg2 ... argn`, permet d'afficher `n` arguments en utilisant la chaîne de format `fmt`. Cette dernière est une chaîne de caractères contenant des séquences spéciales :
`%d` Affichage d'un entier.
`%s` Affichage d'une chaîne.
`%f` Affichage d'un flottant.

Exemple : `Printf.printf "Salut, mon nom est %s et j'ai %d ans" "Toto" 42` affiche :

```
Salut, mon nom est Toto et j'ai 42 ans
```

Comparaisons

En OCaml les opérations de comparaison permettent de comparer n'importe quelles valeurs du même type :
`<, <=, >, >=, =, <> : 'a -> 'a -> bool` comparaisons entre deux valeurs (respectivement, inférieur, inférieur ou égal, supérieur, supérieur ou égal, égal et différent), (opérateur binaire).
`compare : 'a -> 'a -> int` comparaison générique : `compare x y` renvoie un entier négatif si `x < y`, nul si `x = y` et positif si `x > y`.
`min : 'a -> 'a -> 'a` renvoie la plus petite de deux valeurs du même type.
`max : 'a -> 'a -> 'a` renvoie la plus grande de deux valeurs du même type.

n-uplets

Les n-uplets ou produits sont délimités par des parenthèses et des virgules. Dans le cas particulier des paires, deux fonctions `fst` et `snd` permettent d'accéder à la première et seconde composante. Dans les autres cas, on peut utiliser un **`let`** multiple :

```
1   let p1 = (10, 12)
2   let p2 = (-1, false)
3   let t3 = ("A", "B", 24)
4
5   let x = fst p1 (* 10 *)
6   let y = snd p2 (* false *)
7   let a, b, n = t3  (* a vaut "A", b vaut "B" et n vaut 24 *)
```

Définitions de types

La directive **`type`** `t = ...` permet de définir un type OCaml nommé `t`. Ce type peut être :

Un produit nommé est défini en donnant pour chaque étiquette le type des valeurs associées :

```
1   type point_colore = { x : float; y : float; couleur : string }
```

On peut créer des valeurs enregistrement avec des accolades et accéder aux champs avec la notation `.f` :

```
let prouge = { x = 0.5; y = 10.3; couleur = "rouge" }

(* Fonction pour afficher un point coloré *)
let pr_point p = Printf.printf "<x = %f, y = %f, couleur = %s>"
       p.x p.y p.couleur
```

Un type somme est défini en donnant la liste de cas possibles :

```
type val_carte = Roi | Dame | Valet | Val of int
```

On peut créer des valeurs de ces types en utilisant les constantes ou en leur donnant un argument. On peut tester les valeurs en utilisant l'opérateur de filtrage :

```
let dix = Val 10
let as = Val 1

(* Fonction pour afficher une valeur de carte *)
let pr_val v = match v with
     Roi -> Printf.printf "%s"  "Roi"
   | Dame -> Printf.printf "%s"  "Dame"
   | Valet -> Printf.printf "%s"  "Valet"
   | Val (1) -> Printf.printf "%s"  "As"
   | Val (n) -> Printf.printf "%d"  n
```

Exceptions

En OCaml, les exceptions sont des objets particuliers permettant de signaler une erreur. On peut définir une exception avec la directive `exception E of ...` :

```
exception MonErreur of string (* un message *)
```

On peut « lever » une exception, c'est à dire signaler une erreur au moyen de la fonction prédéfinie `raise` :

```
let err_arg_invalide () = raise (MonErreur "argument invalide")
```

Une exception non rattrapée interrompt immédiatement le programme. On peut rattraper une exception avec la construction `try with` :

```
try
  18 + (f 42)   (* f peut lever une exception *)
with
   Not_found -> (* si l'exception Not_found est levée par f *)
                10
 | MonErreur msg ->
                12
```

Si on veut signaler une erreur avec un message, la fonction prédéfinie `failwith` permet de lever une exception avec ce message en argument.

```
if x < 0.0 then
    failwith "valeur négative interdite"
else
    sqrt x
```

Listes

Le type OCaml des listes est `'a list` et permet de représenter une collection ordonnée de valeurs du type `'a`. Les listes constantes sont délimitées par des crochets et des points-virgules : `[1; 2; 42; -5]`. La liste vide est représentée par `[]`. Les opérations et fonctions sur les listes sont :

`:: : 'a -> 'a list -> 'a list` ajout en tête de liste : `1 :: l` (opérateur binaire).

`@ : 'a list -> 'a list -> 'a list` concaténation de deux listes.

`List.rev : 'a list -> 'a list` renvoie la liste avec les éléments dans l'ordre inverse.

`List.iter : ('a -> unit) -> 'a list -> unit` `List.iter f l` applique `f` à tous les éléments de `l`. La fonction `f` ne renvoie pas de résultat (par exemple elle fait un affichage).

`List.map : ('a -> 'b) -> 'a list -> 'b list` `List.map f l` applique `f` à tous les éléments de `l` et renvoie la liste des images par `f`.

`List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` `List.fold_left f init l` applique la fonction de combinaison `f` à `init` et tous les éléments de `l` dans l'ordre. Si

```
l = [ v1; v2; ... ; vn]
```

alors

```
List.fold_left f init l = (f ... (f init v1) v2) ... vn)
```

`List.filter : ('a -> bool) -> 'a list -> 'a list` `List.filter f l` renvoie la liste de tous les éléments de `l` pour lesquels `f` renvoie **`true`**.

`List.assoc : 'a -> ('a * 'b) list -> 'b` `List.assoc a l` renvoie la seconde composante de la première paire dans `l` qui possède `a` comme première composante. La fonction lève l'exception `Not_found` si une telle paire n'existe pas.

`List.sort : ('a -> 'a -> int) -> 'a list -> 'a list` `List.sort f l` renvoie une copie triée de `l` selon la fonction de comparaison `f`. Cette dernière suit les mêmes conventions que la fonction prédéfinie `compare`.

Enfin, on peut inspecter les listes au moyen de l'opérateur de filtrage :

```
1     (* teste si une liste est de longueur paire *)
2   let rec liste_long_paire l =
3     match l with
4    []   -> true (* liste vide est de longueur 0, pair *)
5  | [ _ ] -> false (* liste à un élément est de longueur 1, imapair *)
6  | _ :: _ :: ll -> liste_long_paire ll (* cas récursif *)
```

Boucles et séquences d'instructions

On a deux types de boucles en OCaml :

```
1   for i = 0 to n do
2     ...
3   done
```

```
1   while c do
2     ...
3   done
```

Dans le boucle **for** les deux bornes sont incluses. Dans la boucle **while**, la condition `c` devrait pouvoir varier et donc utiliser des références ou autre valeur mutable.

On peut mettre en séquence deux expressions OCaml en utilisant le « `;` ». Enfin, on peut grouper des séquences d'instructions au moyen de `begin ... end`.

Modules et foncteurs

Module Un module est définit par la construction `module X = struct ... end`, dans laquelle on trouve des définitions de valeurs et de types :

```
module Point =
struct
  type t = int * int

  let compare (x1, y1) (x2, y2) =
    let c = compare x1 x2 in
    if c = 0 then compare y1 y2 else c
end
```

Type de modules Un type de module est définit par la construction `module type T = sig ... end`, dans laquelle on trouve des déclarations de valeurs et de types :

```
module type COMPARABLE =
sig
  type t

  val compare : t -> t - int
end
```

Foncteur Un foncteur est un module paramétré par un ou plusieurs autres modules. On le définit comme un module normal, mais en donnant en plus des paramètres associés à un type de module. Une fois définit, le foncteur peut être appliqué à un module concret pour obtenir un module résultat :

```
module SetBuilder (E : COMPARABLE) =
struct
  type t = Leaf | Node of (int * t * E.t * t)

  let empty = Leaf
  let singleton e = Node (1, Leaf, e, Leaf)
  (* ...
    La suite peut utiliser E.compare pour comparer
    des valeurs de types E.t
  *)
end

(* On applique le foncteur *)
module PointSet = SetBuilder(Point)
```

Références

Le type `'a ref` représente les références vers des valeurs de type `'a`.

`ref : 'a -> 'a ref` création d'une référence.
r `:=` v mise à jour de la référence r avec la valeur v.
`!` r accès au contenu de la référence r.

Tableaux

Les tableaux sont des collections ordonnées de valeurs. On peut accéder aux indices d'un tableau (à partir de 0) et modifier le contenu des cases.

t`.(`i`)` accès à la $i^{\text{ème}}$ case du tableau t.

t`.(`i`) <-v` accès à la $i^{\text{ème}}$ case du tableau t.

`Array.make : int -> 'a -> 'a array` crée un tableau de la taille donnée, dont toutes les cases contiennent la valeur par défaut fournie.

`Array.length : 'a array -> int` renvoie la longueur du tableau t.

`Array.map : ('a -> 'b) -> 'a array -> 'b array` renvoie le tableau des images par la fonction donnée.

`Array.fold_left : ('a -> 'b -> 'a) -> 'a -> ' b array -> 'a` comme `List.map` mais sur les tableaux :

```
Array.fold_left f init t = (f ... (f init t.(0) t.(1)) ... t.(n-1))
```

Tables de hachage

Le type des tables de hachage est `('a,'b) Hashtbl.t` où `'a` est le type des clés et `'b` le type des valeurs.
`Hashtbl.create : int -> ('a, 'b) Hashtbl.t` crée une table de hachage avec une capacité initiale donnée (on peut utiliser n'importe quelle valeur, la tablea est redimentionnée automatiquement).

`Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` associe la clé à la valeur données dans la table (et ne renvoie rien). Si la clé était déjà présente, l'ancienne valeur est écrasée.

`Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` teste si une clé est présente dans la table.

`Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` renvoie la valeur associée à la clé donnée, ou lève l'exception `Not_found` si la clé est absente.

`Hashtbl.length : ('a, 'b) Hashtbl.t -> int` le nombre d'entrées présentes dans la table.

`Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit` appelle la fonction passée en paramètre sur toues les clés et valeurs de la table.

`Hashtbl.fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) Hashtbl.t -> 'c -> 'c` similaire à `fold_left`. Si la table contient des clés et valeurs (`k1: v1, k2 : v2, ... ,kn : vn`) alors

```
Hashtbl.fold f table init = (f kn vn ... (f k2 v2 (f k1 v1 init)))
```