université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

lundi 16 décembre 2024

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 4. 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 Trie de suffixes (9 points)

On revisite dans cet exercice la structure de données de *trie* vue en cours. On considère la version des *tries* permettant d'associer à chaque chaîne de caractères stockée une valeur (utilisation d'un *trie* comme une *map*). On rappelle le type OCaml correspondant :

```
type 'a trie = Node of ('a option) * (char * trie) list
```

Le paramètre de type `'a` représente le type des valeurs stockées dans le *trie*. On rappelle aussi que le deuxième argument d'un `Node(...)` est une liste de couples triés par caractères croissants.

Le but de l'exercice est de montrer qu'étant donné un texte, stocker tous ses *suffixes* dans un *trie* permet de répondre à des questions intéressantes avec une bonne complexité [1].

Un suffixe d'une chaîne de caractères est une portion de la chaîne commençant à un endroit arbitraire et contenant tous les caractères jusqu'à la fin. Par exemple, le mot `"banana"` possède les suffixes : `"banana"`, `"anana"`, `"nana"`, `"ana"`, `"na"`, `"a"` et `""`. Dans toute la suite, on ignorera le suffixe vide (la chaîne de caractères vide). On considère le *trie* de tous les suffixes non-vides de `"banana"` associés à la position de leur premier caractère dans la chaîne initiale. Ce *trie* est représenté à la figure 1. Dans ce dessin, les nœuds noirs représentent des constructeurs `Node(None, ...)` et les nœuds tels que `3` représentent des constructeurs de la forme `Node(Some 3, ...)`.

Enfin, on suppose écrite la fonction `find_trie : string -> 'a trie -> 'a trie` qui renvoie le *trie* atteignable par la chaîne `s`, ou lève l'exception `Not_found` si un tel *trie* n'existe pas.

```
let find_trie s t =
  let rec find_node i t =
    if i = String.length s then t
    else let Node (_, l) = t in find_list i l
  and find_list i l =
    let ci = s.[i] in
    match l with
      [] -> raise Not_found
    | (d, t) :: ll ->
      if ci > d then find_list i ll
      else if ci = d then find_node (i+1) t
      else raise Not_found
  in
  find_node 0 t
```

1. La version présentée ici est naïve, la version utilisée en pratique est appelée *Suffix Tree* et est due à E. Ukkonen.

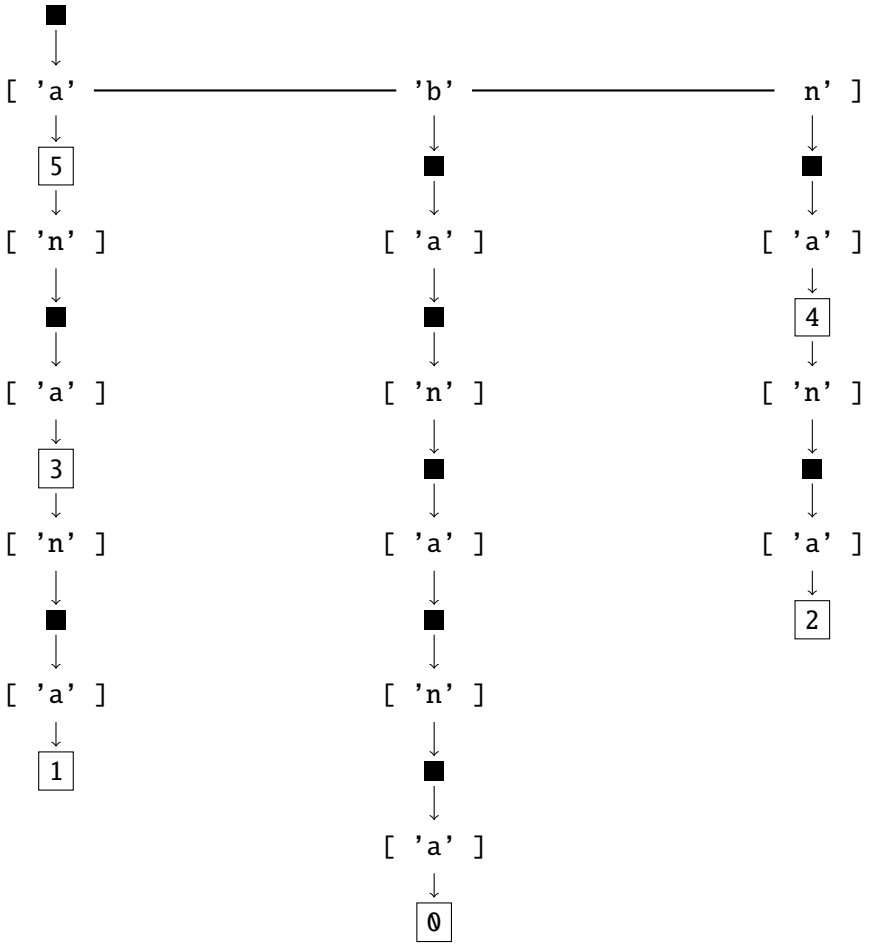

FIGURE 1 – Représentation graphique du *trie* contenant tous les suffixes non-vides de `"banana"`

Questions

1. (1 point) Donner une valeur OCaml de type `int trie` contenant une unique chaîne de caractères `"na"` associée à la valeur `2`.
2. (2.5 points) Écrire une fonction `add_from : string -> int -> 'a -> 'a trie -> 'a trie` telle que `add_from s i v t` ajoute au *trie* `t` la valeur `v` en l'associant au suffixe de la chaine `s` commençant en `i`. Par exemple, `add_from "banana" 2 42 t` associe dans le *trie* `t` la valeur `42` à la chaîne `"nana"` (on commence au caractère d'indice 2).

   Indication : C'est une simple variante triviale de la fonction `add s v t` vue en cours, dans laquelle on commence directement au `i`ème caractère plutôt qu'à 0.
3. (1.5 point) Écrire une fonction `create_suffix_trie : s -> int trie` qui renvoie le *trie* contenant tous les suffixes de la chaîne `s`, chacun associé à l'indice de son premier caractère dans `s`.

   Indication : vous pouvez soit utiliser une boucle **`for`** et une référence, soit utiliser une fonction récursive.
4. (2.5 point) écrire une fonction `collect : 'a trie -> 'a list` qui renvoie la liste de toutes les valeurs stockées dans le *trie* passé en argument. Par exemple, si on appelle cette fonction sur le *trie* de la figure 1, elle renvoie

   `[1, 3, 5, 0, 2, 4]`

   L'ordre n'est pas significatif (c'est à dire qu'on n'impose pas d'ordre, il suffit que la liste renvoyée contienne bien toutes les valeurs).

5. (2 points) Une très belle propriété du *trie* de suffixes est qu'il permet de déterminer très rapidement toutes les positions où se trouvent une chaîne `m` donnée. Pour cela il suffit d'extraire le *trie* atteignable par `m`, puis de renvoyer toutes les valeurs qui se trouvent en dessous (à n'importe quelle profondeur). Par exemple, la chaîne `"ana"` apparaît 2 fois dans le texte de départ, aux position `1` et `3`. La chaîne `"ban"` apparaît une fois, en position `0`.

   Écrire une fonction `find_all : string -> int trie -> int list` telle que `find_all m t` renvoie la liste de toutes les positions où se trouvent la chaîne `m` dans le texte de départ. La liste renvoyée doit être triée par ordre croissant de valeurs. Si la chaîne n'est pas présente dans le texte de départ, la fonction renvoie la liste vide.

   Indication : cette fonction est juste une combinaison judicieuse des fonctions précédemment écrites, elle n'est pas récursive.

2 Sudoku (11 points)

On se propose de résoudre le problème du *Sudoku* en utilisant la méthode de retour arrière (*backtracking*). On rappelle le principe de ce jeu. Étant donné une grille telle que celle donnée à la figure 2, on souhaite remplir les cases vides par des chiffres de 1 à 9 en respectant les contraintes suivantes :

— un chiffre apparaît exactement une fois dans chaque ligne de la grille
— un chiffre apparaît exactement une fois dans chaque colonne de la grille
— un chiffre apparaît exactement une fois dans chaque bloc de 3×3

	3		8					
		9			7			
		1			3	4	5	7
7				2		6		4
	1		5				9	2
				6		1		
2			9	4			8	
	6	4			5			3
8						7		1

```
type grid = int array array
let example_grid = [|
  [| 0; 3; 0;   8; 0; 0;  0; 0; 0 |];
  [| 0; 0; 9;   0; 0; 7;  0; 0; 0 |];
  [| 0; 0; 1;   0; 0; 3;  4; 5; 7 |];

  [| 7; 0; 0;   0; 2; 0;  6; 0; 4 |];
  [| 0; 1; 0;   5; 0; 0;  0; 9; 2 |];
  [| 0; 0; 0;   0; 6; 0;  1; 0; 0 |];

  [| 2; 0; 0;   9; 4; 0;  0; 8; 0 |];
  [| 0; 6; 4;   0; 0; 5;  0; 0; 3 |];
  [| 8; 0; 0;   0; 0; 0;  7; 0; 1 |];
|]
```

FIGURE 2 – Une grille de *sudoku* et sa représentation en OCaml

On représente une grille de *Sudoku* en OCaml par le type `grid` (un simple alias pour le type des tableaux de tableaux d'entiers). On rappelle que les indices commencent à 0. Par exemple, le nombre 5 se trouvant dans le carré central est sur la ligne 4 et sur la colonne 3 et est donc accessible par : `grid.(4).(3)`. Dans tout l'exercice, vous pouvez supposer que la grille est de taille 9×9, c'est à dire que le tableau externe est de taille 9, et que chaque tableau interne est de taille 9.

Questions

1. (0.5 point) Écrire une fonction `extract_row : grid -> int -> int array` telle que

   ```
   extract_row grid r
   ```

   renvoie la ligne d'indice `r` de la grille passée en argument.
2. (1.5 point) Écrire une fonction `extract_col : grid -> int -> int array` telle que

   ```
   extract_col grid c
   ```

renvoie un tableau contenant tous les entiers de la colonne d'indice `c`

Indication : il faut allouer un tableau de taille 9 et le remplir avec les valeurs de la colonne, puis le renvoyer.

3. (2.5 points) Écrire une fonction `extract_block : grid -> int -> int -> int array` telle que `extract_block grid r c` renvoie un tableau contenant tous les entiers du bloc contenant la case se trouvant ligne `r` colonne `c`. Par exemple, `extract_block grid 5 4` renvoie le tableau de toutes les valeurs du bloc central.

   Indication : il faut allouer un tableau de taille 9 et le remplir avec les valeurs du bloc correspondant.

4. (1 point) La fonction `Array.for_all : ('a -> bool) -> 'a array -> bool` est telle que `Array.for_all f tab` renvoie `true` si et seulement si `f x` vaut `true` pour toutes les valeurs `x` du tableau `tab`. Utiliser cette fonction pour définir une fonction `not_in : 'a array -> 'a -> bool` telle que `not_in tab v` vaut `true` si `v` n'est pas dans le tableau `tab`.

5. (0.5 point) Écrie une fonction `valid : grid -> int -> int -> int -> bool` telle que

   ```
   valid grid r c v
   ```

   renvoie vrai s'il est possible de mettre la valeur `v` dans la case se trouvant ligne `r` et colonne `c` de la grille `grid`. Votre fonction doit vérifier les trois contraintes (unicité dans la ligne, la colonne et le bloc) et aussi que la case en question contient 0 (i.e. n'est pas déjà occupée).

6. (2 points) Écrire une fonction `holes : grid -> (int*int) list` qui renvoie la liste des paires (ligne, colonne) de toutes les cases de la grille contenant un 0.

   Indication : on pourra soit faire un usage judicieux de `Array.fold_left` (cf. aide-mémoire) soit utiliser une double boucle `for` et ajouter les paires trouvées dans une liste stockée dans une référence.

7. (3 points) Écrire une fonction `solve : (grid -> unit) -> grid -> unit` telle que `solve f grid` appelle la fonction `f` sur toutes les grilles solutions de la grille donnée en argument.

   Indication on utilisera la méthode du retour arrière, en considérant la liste des positions où se trouvent un 0.
   - si cette liste est vide, alors on a trouvé une solution
   - sinon, on considère la première position de cette liste et on essaye d'y mettre tour à tour toutes les valeurs valides entre 1 et 9 en s'appelant à chaque essai récursivement sur la suite de la liste.

   On fera attention au fait que la grille est une valeur mutable, et donc on pensera à annuler les modifications faites au retour des appels récursifs.

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 :

```
let p1 = (10, 12)
let p2 = (-1, false)
let t3 = ("A", "B", 24)

let x = fst p1 (* 10 *)
let y = snd p2 (* false *)
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 :

```
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.fold_left` 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)))
```