

## Examen de 2<sup>nd</sup>e session

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 (12 points)

On revisite dans cet exercice la structure de données de *trie* vue en cours, permettant de stocker un ensemble de chaînes de caractères. Le but de l'exercice est de programmer d'autres fonctions utilitaires sur les tries. On considère la version des *tries* représentant des ensembles (et non pas des maps), il n'y a donc pas de valeur stockée dans les nœuds mais uniquement un booléen indiquant si la chaîne se terminant sur le nœud considéré est dans l'ensemble ou non. On rappelle le type OCaml correspondant :

```
type trie = Node of bool * (char * trie) list
```

De plus le deuxième argument d'un `Node(...)` est une liste de couples triés par caractères croissants. Par exemple, l'ensemble {"abc", "abd", "abde", "abf", "ac"} peut être représentée par le *trie* dessiné à la figure 1. Dans cette figure, les carrés représentent des valeurs `Node(...)` dont le booléen est **false** (carrés blancs) ou **true** (carrés noirs). Les listes de caractères représentent le deuxième argument d'un `Node`, la flèche se trouvant en dessous représentant récursivement le *trie* associé à ce caractère.

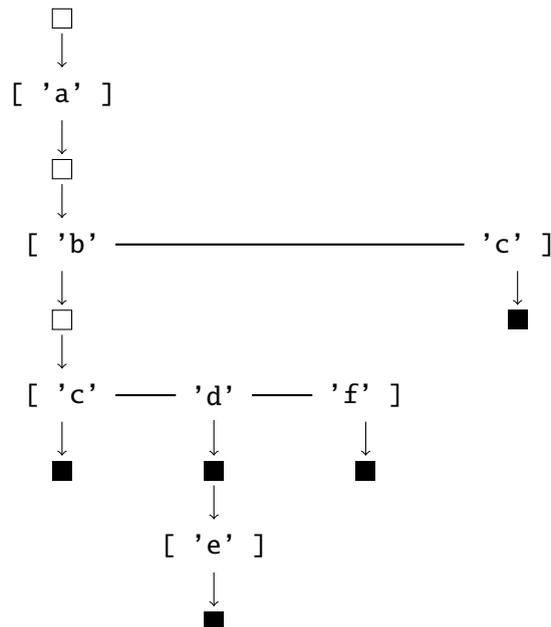


FIGURE 1 – Représentation graphique d'un *trie*

## Questions

- (1 point) Donner une valeur OCaml de type `trie` qui représente l'ensemble {"ab"}.
- (2 points) Écrire une fonction `length : trie -> int` qui renvoie la taille du *trie*, c'est à dire le nombre de chaîne de caractères différentes qui y sont stockées.

**Indication** : on remarque qu'il suffit de parcourir le *trie* en comptant combien on trouve de booléen vallant `true`. Cette fonction suivra le schéma des fonctions sur les *tries* vues en cours, c'est à dire qu'elle sera de la forme

```
let rec length t =
  match t with
  ...

and length_list l =
  match l with
  ...
```

La fonction auxiliaire `length_list` permettant d'appliquer récursivement la fonction `length` en parcourant la liste de paires caractère, *trie*.

- (2 points) On rappelle le code la fonction `implode : char list -> string` :

```
let implode l =
  String.concat ""
  (List.fold_left (fun acc c -> (String.make 1 c)::acc) [] l)
```

Cette dernière prend en argument une liste de caractères et renvoie la chaîne de caractères correspondante. Par exemple `implode ['a'; 'b'; 'c']` renvoie "abc".

Écrire le code d'une fonction `iter : (string -> unit) -> trie -> unit` telle que `iter f t` applique la fonction `f` à toutes chaînes de caractères de l'ensemble représenté par le *trie* `t`.

**Indication** : on rappelle que l'on peut parcourir récursivement le *trie* en maintenant un paramètre accumulant la liste des caractères rencontrés. En arrivant sur un booléen `true` il suffit de renverser cette liste et d'appeler `implode` pour obtenir la chaîne correspondante.

- (2 points) **En utilisant la fonction `iter`** définie précédemment, écrire une fonction

$$\text{fold} : ('a \rightarrow 'b \rightarrow 'b) \rightarrow \text{trie} \rightarrow 'b \rightarrow 'b$$

telle que si le *trie* `t` représente l'ensemble  $[s_1; \dots; s_n]$ , alors `fold f t init` calcule

$$f s_n (\dots(f s_2 (f s_1 \text{init})))$$

**Indication** : on pourra initialiser une référence avec la valeur `init` puis écrire une fonction mettant à jour cette référence et l'utiliser comme `f`. On n'oubliera pas de renvoyer le contenu de la référence après l'appel à `iter`.

- (2 points) Donner une fonction `longuest : trie -> int option` telle que `longuest t` renvoie :
  - `None` si `t` est vide
  - `Some n` où `n` est la longueur de la plus grande chaîne stockée dans `t` sinon.
- (3 points) Écrire une fonction `subset : trie -> trie -> bool` telle que `subset t1 t2` renvoie vrai si et seulement si la *trie* `t1` est un sous-ensemble du *trie* `t2`. **Votre fonction devra utiliser un espace au plus proportionnel à la hauteur du trie le plus petit.** Ainsi, il est **interdit** de convertir les deux *tries* en listes puis comparer ces dernières.

**Indications** : on pourra explorer récursivement les deux *tries* en même temps. Si on pose  $t_1 \equiv \text{Node}(b_1, l_1)$  et  $t_2 \equiv \text{Node}(b_2, l_2)$  alors,  $t_1$  est un sous-ensemble de  $t_2$  si et seulement si :

- $b_1 \leq b_2$  (avec `false < true`)
- la liste  $l_1$  est incluse dans la liste  $l_2$ .

La liste  $l_1$  est incluse dans  $l_2$  si pour tout couple  $(c, t)$  apparaissant dans  $l_1$ , il existe un couple  $(c, t')$  apparaissant dans  $l_2$  et le *trie*  $t$  est récursivement inclus dans  $t'$ .

Pour implémenter correctement la fonction `subset` cette dernière sera de la forme :

```

let rec subset t1 t2 =
  match t1, t2 with
  ...

and subset_list l1 l2 =
  match l1, l2 with
  ...

```

La fonction `subset_list`, mutuellement récursive avec `subset` permet de tester l'inclusion des deux listes.

## 2 Foncteurs et arbres binaires de recherche (8 points)

On suppose que le module `StringSet` est crée par :

```

module StringSet = Set.Make(String)

```

où `String` est le module des chaînes de caractères de la bibliothèque standard et `Set.Make` est le *foncteur* de la bibliothèque standard permettant de créer des ensembles dont les éléments sont donnés par un module (ici `String`) implémentant l'interface `COMPARABLE` (la syntaxe des modules et l'interface `COMPARABLE` sont décrit dans l'aide mémoire). La signature du module `StringSet` est donné à la figure 2 avec quelques commentaires. On fera l'hypothèse que le module ne contient pas d'autres fonctions que

```

module StringSet : sig
  type elt = string          (* le type des éléments *)
  type t                    (* le type des ensembles *)

  val empty : t              (* l'ensemble vide *)
  val is_empty : t -> bool   (* test si l'ensemble est vide *)
  val add : elt -> t -> t    (* ajout d'une valeur *)
  val mem : elt -> t -> bool (* teste la présence d'une valeur *)

  val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a

  (* on suppose qu'il n'y a pas d'autres fonctions. *)
end

```

FIGURE 2 – La signature du module `StringSet`

celles de la figure 2. La fonction `fold` est telle que si  $t$  est un ensemble contenant les chaînes  $s_1, \dots, s_n$ , `fold f t init` est équivalent à

$$(f s_n (\dots(f s_2 (f s_1 \text{init}))))$$

avec  $s_1 < s_2 < \dots < s_n$ . Enfin, on rappelle que la structure de données utilisées en interne est celle des arbres binaires de recherche équilibrés et que la comparaison des chaînes de caractères (`String.compare : string -> string -> int`) les compare lexicographiquement (« l'ordre du dictionnaire »). Ainsi, l'ajout, le test d'appartenance et la suppression d'un élément dans un ensemble coût  $O(\log(n))$  où  $n$  est la taille de l'ensemble.

Dans toutes les questions suivantes, vous ne pouvez utiliser **que** des fonctions de la figure 2 ou des fonctions définies dans les questions précédentes ou des fonctions de l'aide mémoire. Vous **n'avez pas accès à la représentation interne des ensembles**.

### Questions

1. (1 point) dessiner un arbre binaire de recherche possible pour l'ensemble {"ab", "ac", "bb", "bc"}, sachant que la hauteur de deux sous-arbre diffère d'au plus 1.
2. (1 point) écrire une fonction `singleton : elt -> t` qui crée l'ensemble contenant uniquement l'élément donné.
3. (3 points) écrire une fonction `inter : t -> t -> t` qui calcule l'intersection des deux ensembles donnés en argument. Donner la complexité d'un appel à votre fonction `inter t t1 t2` en fonction de la taille de `t1` et `t2`.
4. (2 points) écrire une fonction `exists : (elt -> bool) -> t -> bool` qui renvoie `true` si et seulement si la fonction prédicat donnée en argument renvoie `true` sur l'un des éléments de l'ensemble.
5. (1 point) écrire une fonction `forall : (elt -> bool) -> t -> bool` qui renvoie `true` si et seulement si tous les éléments de l'ensemble vérifient le prédicat donné en argument. **Il est interdit d'utiliser la fonction fold**, votre fonction `forall` doit uniquement se baser sur la fonction `exists` (et éventuellement des opérations de bases du langage).

## 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 indices 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_couleur = { x : float; y : float; couleur : string }
```

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

```

1  let prouge = { x = 0.5; y = 10.3; couleur = "rouge" }
2
3  (* Fonction pour afficher un point coloré *)
4  let pr_point p = Printf.printf "<x = %f, y = %f, couleur = %s>"
5      p.x p.y p.couleur

```

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

```

1  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 :

```

1  let dix = Val 10
2  let as = Val 1
3
4  (* Fonction pour afficher une valeur de carte *)
5  let pr_val v = match v with
6      Roi -> Printf.printf "%s" "Roi"
7      | Dame -> Printf.printf "%s" "Dame"
8      | Valet -> Printf.printf "%s" "Valet"
9      | Val (1) -> Printf.printf "%s" "As"
10     | 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 ...** :

```

1  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** :

```

1  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** :

```

1  try
2      18 + (f 42) (* f peut lever une exception *)
3  with
4      Not_found -> (* si l'exception Not_found est levée par f *)
5          10
6      | MonErreur msg ->
7          12

```

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

```

1  if x < 0.0 then
2      failwith "valeur négative interdite"
3  else
4      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

$$l = [v_1; v_2; \dots; v_n]$$

alors

$$\text{List.fold\_left } f \text{ init } l = (f \dots (f \text{ init } v_1) v_2) \dots v_n$$

`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 pré-définie `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, impair *)
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 la 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 défini par la construction **module X = struct ... end**, dans laquelle on trouve des définitions de valeurs et de types :

```
1  module Point =
2  struct
3      type t = int * int
4
5      let compare (x1, y1) (x2, y2) =
6          let c = compare x1 x2 in
7          if c = 0 then compare y1 y2 else c
8  end
```

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

```
1  module type COMPARABLE =
2  sig
3      type t
4
5      val compare : t -> t - int
6  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 défini, le foncteur peut être appliqué à un module concret pour obtenir un module résultat :

```
1  module SetBuilder (E : COMPARABLE) =
2  struct
3      type t = Leaf | Node of (int * t * E.t * t)
4
5      let empty = Leaf
6      let singleton e = Node (1, Leaf, e, Leaf)
7      (* ...
8         La suite peut utiliser E.compare pour comparer
9         des valeurs de types E.t
10     *)
11  end
12
13  (* On applique le foncteur *)
14  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*<sup>ème</sup> case du tableau *t*.

*t*.(*i*) <-*v* accès à la *i*<sup>ème</sup> 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 table est redimensionnée 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 toutes 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 (*k*<sub>1</sub> : *v*<sub>1</sub>, *k*<sub>2</sub> : *v*<sub>2</sub>, ..., *k*<sub>*n*</sub> : *v*<sub>*n*</sub>) alors

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