Programmation fonctionnelle

Fiche de TP 9

L3 Informatique 2021-2022

Jeu du nonogram

1 Explications générales

L'objectif de ce sujet[1] est de programmer un résolveur pour le casse-tête du *nonogram* (connu également sous le nom de « picross » ou de « griddler »). Un nonogram est spécifié par une grille rectangulaire de n lignes et de m colonnes, où chaque ligne et colonne est précédée d'une liste non vide d'entiers strictement positifs. Cette liste spécifie comment noircir les cases de la grille selon l'étendue des blocs de cases noircies contiguës. Un bloc est délimité par les bords de la grille ou par des cases vides. Un noircissement de la grille est une solution au casse-tête si toutes les contraintes spécifiées par les listes des lignes et colonnes sont vérifiées.

La figure 1 montre un exemple de grille initialement vide ainsi que sa solution. À titre d'exemple, la liste `[1, 1, 2]` attachée à la première colonne de la grille impose que le remplissage de cette colonne doit être de la forme

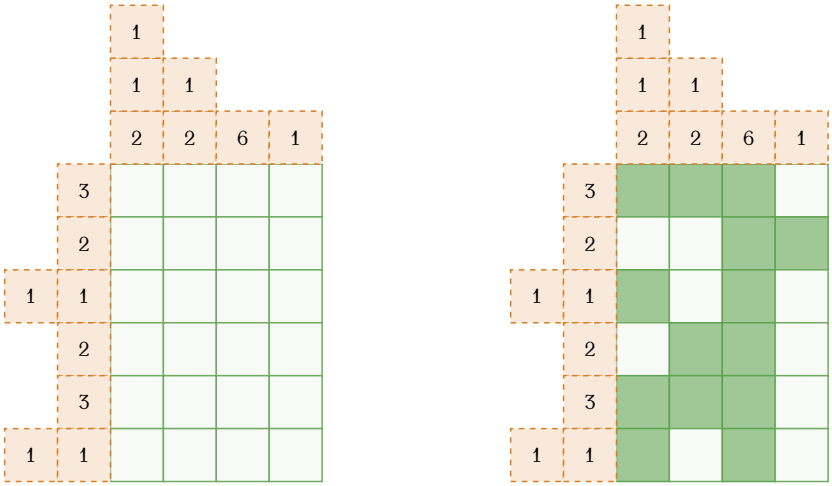
$$\square^* \blacksquare \square^+ \blacksquare \square^+ \blacksquare\blacksquare \square^*$$

où $\square$ est une case non noircie, $\blacksquare$ est une case noircie, « * » est l'étoile de Klenne et « $^+$ » est l'étoile de Kleene propre (ainsi, $\square^+$ signifie une suite non vide de cases non noircies).

2 Recommandations et consignes générales

— Bien lire le sujet avant de commencer à programmer, en particulier la première partie.
— L'élégance dans la programmation sera appréciée, l'inélégance sanctionnée. Cela peut se manifester dans les choix des identificateurs et la qualité des algorithmes sélectionnés.
— Il est impératif de respecter les signatures spécifiées.

1. Ce TP ressemble dans son déroulement, sa présentation et le type des questions posées à l'**examen final** à venir. Il s'agit d'une version révisée d'un examen passé.



(a) Une grille vide munie d'une spécification.

(b) Son unique solution.

FIGURE 1 – Une grille de nonogram 6 × 4 vide et sa solution. Il s'agit de la « petite chèvre ».

— Certaines questions ne sont pas (ou très peu) dépendantes de réponses à des questions précédentes. Il est donc possible de répondre à certaines questions sans avoir fait toutes les précédentes.

— Toutes les fonctions demandées sont illustrées par de nombreux tests. Il est très fortement recommandé de vérifier que les résultats des tests des fonctions programmées sont conformes à ceux donnés. Les exemples sont également utiles pour bien comprendre le comportement de chaque fonction.

3 Travail à accomplir

Le travail consiste à compléter le fichier `Nonogram.ml` fourni en ajoutant sous chaque commentaire de signature de fonction son implantation correspondante. Les réponses aux éventuelles questions qui ne demandent pas de produire du code doivent être fournies en tant que commentaires dans le fichier. Tout ceci est guidé pas à pas dans les exercices suivants. Le fichier `Nonogram.ml` ainsi augmenté est donc l'unique fichier à rendre.

—

Il est **impératif** que le fichier `Nonogram.ml` s'interprète sans erreur ni message d'avertissement par la commande `ocaml Nonogram.ml`.

—

Exercice 1. (Outils sur les listes)

Nous commençons par programmer quelques fonctions outils, principalement sur les listes, qui pourront nous servir dans la suite du sujet.

1. Définir une fonction

```
val interval : int -> int -> int list = <fun>
```

paramétrée par deux entiers `a` et `b` et qui renvoie la liste d'entiers `[a; a + 1; ...; b]`.

```
# interval 0 8;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8]
# interval 2 5;;
- : int list = [2; 3; 4; 5]
# interval 3 3;;
- : int list = [3]
# interval (-5) 2;;
- : int list = [-5; -4; -3; -2; -1; 0; 1; 2]
# interval 9 3;;
- : int list = []
```

2. Définir une fonction

```
val interval_f : int -> int -> (int -> 'a) -> 'a list = <fun>
```

paramétrée par deux entiers `a` et `b` et une fonction `f` prenant un entier et renvoyant une valeur. La fonction renvoie la liste `[f a; f (a + 1); ...; f b]`.

```
# interval_f 2 8 (fun x -> 2 * x);;
- : int list = [4; 6; 8; 10; 12; 14; 16]
# interval_f (-3) 3 (fun x -> -x);;
- : int list = [3; 2; 1; 0; -1; -2; -3]
# interval_f 1 5 (fun _ -> 9);;
- : int list = [9; 9; 9; 9; 9]
```

3. Définir une fonction

```
val is_positive_list : int list -> bool = <fun>
```

paramétrée par une liste d'entiers et qui teste si la liste ne contient que des entiers strictement positifs.

```
# is_positive_list [3; 1; 1; 2];;
- : bool = true
# is_positive_list [3; 1; 1; 0; 2];;
- : bool = false
# is_positive_list [3; (-1); 1; 1; 2];;
- : bool = false
```

4. Définir une fonction

```
val is_sum_leq_list : int list -> int -> bool = <fun>
```

paramétrée par une liste d'entiers et un entier `k` et qui teste si la somme des éléments de la liste est inférieure à `k`.

```
# is_sum_leq_list [0; 1; 2] 25;;
- : bool = true
# is_sum_leq_list [0; 1; 2] 3;;
- : bool = true
# is_sum_leq_list [0; 1; 2] 2;;
- : bool = false
# is_sum_leq_list [] 0;;
- : bool = true
# is_sum_leq_list [] (-1);;
- : bool = false
```

5. Définir une fonction

```
val are_equivalent_lists : 'a list -> 'b list -> ('a -> 'b -> bool) -> bool = <fun>
```

paramétrée par deux listes polymorphes `[e1; ...; en]` et `[f1; ...; fm]` et une fonction `p` et qui renvoie `true` si et seulement si `n = m` et tout appel `p ei fi` est vrai pour tout indice `i`.

```
# are_equivalent_lists [1; 2; 3] [1; 2; 3] (=);;
- : bool = true
# are_equivalent_lists [0; 0; 1; 0] [0; 1; 9; 1] (<=);;
- : bool = true
# are_equivalent_lists [[a']; ['b'; 'b']] [[1]; [2; 3]]
      (fun x y -> (List.length x) = List.length y));;
- : bool = true
# are_equivalent_lists [0; 0; 1] [1; 0; 0; 0] (fun x y -> x + y >= 0);;
- : bool = false
```

6. Définir une fonction

```
val cartesian_product_lists : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

paramétrée par deux listes polymorphes `[e1; ...; en]` et `[f1; ...; fm]` et qui renvoie la liste des couples issus du produit cartésien des deux listes. Plus précisément, la fonction doit renvoyer la liste

`[(e1, f1); ...; (e1, fm); (e2, f1); ...; (e2, fm); ...; (en, f1); ...; (en, fm)]`.

```
# cartesian_product_lists [1; 2] [3; 4; 5];;
- : (int * int) list = [(1, 3); (1, 4); (1, 5); (2, 3); (2, 4); (2, 5)]
# cartesian_product_lists [1; 2; 3] [4; 5];;
- : (int * int) list = [(1, 4); (1, 5); (2, 4); (2, 5); (3, 4); (3, 5)]
# cartesian_product_lists [1; 2] [];;
- : (int * 'a) list = []
# cartesian_product_lists [] [];;
- : ('a * 'b) list = []
```

Exercice 2. (Le type `grid` et les autres types de base)

Nous représentons l'état vide ou noirci d'une case par le type somme `state` défini par

```
type state = Empty | Black
```

Une case est un couple d'entiers `(i, j)` représentant une case indexée comme une entrée de matrice. En d'autres termes, `i` représente le numéro de la ligne de la case et `j` représente le numéro de la colonne. La case la plus en haut à gauche est `(1, 1)`. Ainsi, nous considérons le type `square` pour représenter les cases défini par

```
type square = int * int
```

Finalement, une grille est l'information de ses dimensions (nombre de lignes et nombre de colonnes) et de l'état (vide ou noirci) de chacune de ses cases. Nous utilisons ainsi le type enregistrement `grid` défini par

```
type grid = {
    nb_rows : int;
    nb_columns : int;
    states : square -> state
}
```

pour représenter une grille. Le champ `states` est une fonction qui, pour chaque case de la grille dont les dimensions sont spécifiées par les champs `nb_rows` et `nb_columns`, renvoie son état.

Les fonctions suivantes :

```
val state_to_string : state -> string = <fun>
```

```
val grid_to_string : grid -> string = <fun>
```

qui renvoient respectivement une chaîne de caractères représentant un état et une grille sont fournies. Elles vont nous permettre de mettre en place les tests qui suivent.

1. Définir une fonction

   ```
   val empty_grid : int -> int -> grid = <fun>
   ```

   paramétrée par deux entiers strictement positifs `n` et `m` et qui renvoie une grille à `n` lignes et `m` colonnes de cases toutes vides.

   ```
   # empty_grid 5 3;;
   - : grid = {nb_rows = 5; nb_columns = 3; states = <fun>}
   # print_string (grid_to_string (empty_grid 5 3));;
   |-|-|-|
   |-|-|-|
   |-|-|-|
   |-|-|-|
   |-|-|-|
   - : unit = ()
   ```

2. Définir une fonction

```
val fill_square : grid -> square -> grid = <fun>
```

paramétrée par une grille et une case et qui renvoie la grille obtenue en noircissant la case spécifiée. Dans le cas où la case n'appartient pas à la grille, ou quand celle-ci est déjà noircie, la fonction renvoie la grille entrée.

```
# let g = empty_grid 5 3;;
val g : grid = {nb_rows = 5; nb_columns = 3; states = <fun>}
# let g = fill_square g (1, 1);;
val g : grid = {nb_rows = 5; nb_columns = 3; states = <fun>}
# print_string (grid_to_string g);;
|O|-|-|
|-|-|-|
|-|-|-|
|-|-|-|
|-|-|-|
- : unit = ()
# let g = fill_square g (5, 3);;
val g : grid = {nb_rows = 5; nb_columns = 3; states = <fun>}
# print_string (grid_to_string g);;
|O|-|-|
|-|-|-|
|-|-|-|
|-|-|-|
|-|-|O|
- : unit = ()
```

3. Définir une fonction

```
val fill_squares : grid -> square list -> grid = <fun>
```

paramétrée par une grille et une liste de case et qui renvoie la grille obtenue en noircissant toutes les cases spécifiée par la liste.

```
# print_string (grid_to_string (fill_squares (empty_grid 4 3) []));;
|-|-|-|
|-|-|-|
|-|-|-|
|-|-|-|
- : unit = ()
# print_string (grid_to_string (fill_squares (empty_grid 4 3)
       [(1, 2); (1, 3); (4, 3)]));;
|-|O|O|
|-|-|-|
|-|-|-|
|-|-|O|
- : unit = ()
```

4. Définir une fonction

```
val row : grid -> int -> state list <fun>
```

paramétrée par une grille et un entier `i` et qui renvoie la liste des états des cases de la ligne d'indice `i` de la grille. Les éléments de la liste doivent apparaître par indices croissants par rapport à leur emplacement d'origine dans la grille. Dans le cas où `i` ne correspond à aucune ligne de la grille, la fonction renvoie la liste vide.

```
# let g = fill_squares (empty_grid 5 3)
       [(1, 1); (1, 2); (1, 3); (3, 1); (3, 3); (4, 3)];;
val g : grid = {nb_rows = 5; nb_columns = 3; states = <fun>}
# print_string (grid_to_string g);;
|O|O|O|
|-|-|-|
|O|-|O|
|-|-|O|
|-|-|-|
- : unit = ()
# row g 1;;
- : state list = [Black; Black; Black]
# row g 4;;
- : state list = [Empty; Empty; Black]
# row g 6;;
- : state list = []
```

5. Définir une fonction

```
val column : grid -> int -> state list <fun>
```

paramétrée par une grille et un entier `j` et qui renvoie la liste des états des cases de la colonne d'indice `j` de la grille. Les éléments de la liste doivent apparaître par indices croissants par rapport à leur emplacement d'origine dans la grille. Dans le cas où `j` ne correspond à aucune colonne de la grille, la fonction renvoie la liste vide.

```
# let g = fill_squares (empty_grid 5 3)
       [(1, 1); (1, 2); (1, 3); (3, 1); (3, 3); (4, 3)];;
val g : grid = {nb_rows = 5; nb_columns = 3; states = <fun>}
# print_string (grid_to_string g);;
|O|O|O|
|-|-|-|
|O|-|O|
|-|-|O|
|-|-|-|
- : unit = ()
# column g 1;;
- : state list = [Black; Empty; Black; Empty; Empty]
# column g 3;;
- : state list = [Black; Empty; Black; Black; Empty]
```

```
# column g 4;;
- : state list = []
```

Exercice 3. (Le type `profile`)

Le *profil* d'une ligne ou d'une colonne d'une grille est la suite $(p_1, \ldots, p_k)$ d'entiers strictement positifs telle que chaque p_i est le nombre de cases noircies de chaque bloc de longueur maximale de cases noircies de la ligne ou de la colonne. Par exemple, une colonne de contenu

■■■□■□■□□■■□□□□

a pour profil la suite $(3, 1, 1, 2)$. Toute ligne ou colonne constituée uniquement de cases vides possède la suite vide comme profil.

Nous représentons les profils à l'aide du type `profile` défini par

```
type profile = int list
```

Notons bien que le traitement des profils qui va suivre est indépendant du fait qu'ils proviennent d'une ligne ou d'une colonne.

1. Définir une fonction

   ```
   val profile : state list -> profile = <fun>
   ```

   paramétrée par une liste d'états et qui renvoie le profil de cette liste vue comme désignant le contenu d'une ligne ou d'une colonne d'une grille.

   ```
   # profile [Black; Black];;
   - : int list = [2]
   # profile [Black; Black; Empty];;
   - : int list = [2]
   # profile [Empty; Black; Black; Empty];;
   - : int list = [2]
   # profile [Empty; Black; Empty; Empty; Black; Black; Empty];;
   - : int list = [1; 2]
   # profile [];;
   - : int list = []
   ```

2. Un profil p est *compatible* avec un profil p' s'il existe un contenu de ligne ou de colonne ayant p pour profil qui peut, en uniquement noircissant certaines de ses cases, former un contenu de ligne ou de colonne ayant p' pour profil.

Par exemple, le profil $p := (2,1,1)$ est compatible avec le profil $p' := (4,1,2,2)$ car nous pouvons considérer le contenu

□□□■■□□■□■□□□□□□□□

qui a p pour profil, et nous pouvons obtenir en noircissant certaines cases le contenu

□■■■■□□■□■■□□■■□□□

qui a pour profil p'. En revanche, le profil $p := (3,1)$ n'est pas compatible avec le profil $p' := (1,1,1,2)$ car il n'est pas possible de supprimer le bloc de cases noircies de taille 3 dans tout contenu ayant p pour profil. De même, observons que le profil $p := (3,1)$ n'est pas compatible avec le profil $p' := (4)$. À l'inverse, le profil $p := (3,1)$ est compatible avec le profil $(4,1)$.

Définir une fonction

```
val are_profiles_compatible : profile -> profile -> bool = <fun>
```

paramétrée par deux profils `p` et `p'` et qui teste si `p` est compatible avec `p'`. Cette fonction doit simplement se baser sur l'algorithme suivant. Il prend en entrée les deux profils $p := (p_1, \ldots, p_\ell)$ et $p' := (p'_1, \ldots, p'_{\ell'})$ et

(A) si p est la suite vide, `vrai` est renvoyé ;

(B) si p n'est pas la suite vide et p' est la suite vide, `faux` est renvoyé ;

(C) si $p_1 \leqslant p'_1$, on pose $x := p'_1 - p_1 - 1$ et

  (1) si $x \leqslant 0$, le résultat de l'algorithme appliqué à $(p_2, \ldots, p_\ell)$ et $(p'_2, \ldots, p'_{\ell'})$ es t renvoyé ;

  (2) sinon, le résultat de l'algorithme appliqué à $(p_2, \ldots, p_\ell)$ et $(x, p'_2, \ldots, p'_{\ell'})$ est renvoyé ;

(D) sinon, le résultat de l'algorithme appliqué à $(p_1, \ldots, p_\ell)$ et $(p'_2, \ldots, p'_{\ell'})$ est renvoyé.

```
# are_profiles_compatible [] [];;
- : bool = true
# are_profiles_compatible [] [1; 3; 1; 2];;
- : bool = true
# are_profiles_compatible [1] [];;
- : bool = false
# are_profiles_compatible [1] [1; 3; 1; 2];;
- : bool = true
# are_profiles_compatible [1; 2] [1; 3; 1; 2];;
- : bool = true
# are_profiles_compatible [1; 4] [1; 3; 1; 2];;
- : bool = false
# are_profiles_compatible [3] [1; 3; 1; 2];;
- : bool = true
# are_profiles_compatible [4] [1; 3; 1; 2];;
- : bool = false
# are_profiles_compatible [1; 2; 3] [1; 4; 4];;
- : bool = true
```

Exercice 4. (Le type `spec`)

Une *spécification* de jeu de nonogram est la donnée d'une liste de profils pour les lignes de la grille candidate et d'une liste de profils pour les colonnes. Nous représentons ainsi une spécification à l'aide du type `spec` défini par

```
type spec = {
   rows : profile list;
   columns : profile list
}
```

Une grille est *solution* d'une spécification si toutes les lignes et colonnes de la grille possèdent comme profils les profils correspondants de la spécification. On supposera dans la suite que toutes les spécifications sont *valides*, c'est-à-dire qu'il existe au moins une grille solution de la spécification.

1. Définir une fonction

   ```
   val dimensions_spec : spec -> int * int = <fun>
   ```

   paramétrée par une spécification et qui renvoie le couple renseignant sur le nombre de lignes et de colonnes des grilles sur lesquelles la spécification est définie.

   ```
   # dimensions_spec {rows = [[3]; [2]; [1; 1]; [2]; [3]; [1; 1]];
          columns = [[1; 1; 2] ; [1; 2]; [6]; [1]]};;
   - : int * int = (6, 4)
   ```

2. Définir une fonction

   ```
   val is_solution : spec -> grid -> bool = <fun>
   ```

   paramétrée par une spécification et une grille et qui teste si la grille est solution de la spécification.

   ```
   # let sp = {rows = [[3]; [2]; [1; 1]; [2]; [3]; [1; 1]];
          columns = [[1; 1; 2] ; [1; 2]; [6]; [1]]};;
   val sp : spec =
     {rows = [[3]; [2]; [1; 1]; [2]; [3]; [1; 1]];
      columns = [[1; 1; 2]; [1; 2]; [6]; [1]]}
   # let g = fill_squares (empty_grid 6 4)
          [(1, 1); (1, 2); (1, 3); (2, 3); (2, 4); (3, 1); (3, 3);
          (4, 2); (4, 3); (5, 1); (5, 2); (5, 3); (6, 1); (6, 3)];;
   val g : grid = {nb_rows = 6; nb_columns = 4; states = <fun>}
   # print_string (grid_to_string g);;
   |O|O|O|-|
   |-|-|O|O|
   |O|-|O|-|
   |-|O|O|-|
   ```

```
|O|O|O|-|
|O|-|O|-|
- : unit = ()
# is_solution sp g;;
- : bool = true
# let g = fill_square g (2, 1);;
val g : grid = {nb_rows = 6; nb_columns = 4; states = <fun>}
# is_solution sp g;;
- : bool = false
```

Exercice 5. (Le résolveur)

Nous allons utiliser les types et fonctions introduits précédemment pour mettre en place une fonction résolvant une spécification de jeu. Pour cela, un algorithme de recherche exhaustif (ou « brute force ») va être mis en place.

1. Définir une fonction

```
val next_square : grid -> square -> square = <fun>
```

paramétrée par une grille et une case c et qui renvoie la case suivante à c. Celle-ci est définie comme étant la case située immédiatement à sa droite si celle-ci existe dans la grille. Sinon, la case suivante à c est la case la plus à gauche de la ligne immédiatement en dessous de celle de c. Cette fonction appelée avec la case la plus en bas à droite de la grille (qui est donc la dernière) renvoie ainsi une case située en dehors de la grille.

```
# let g = empty_grid 5 4;;
val g : grid = {nb_rows = 5; nb_columns = 4; states = <fun>}
# next_square g (1, 1);;
- : int * int = (1, 2)
# next_square g (1, 2);;
- : int * int = (1, 3)
# next_square g (2, 4);;
- : int * int = (3, 1)
# next_square g (5, 4);;
- : int * int = (6, 1)
```

2. Définir une fonction

```
val is_square : grid -> square -> bool = <fun>
```

paramétrée par une grille et une case et qui teste si la case fait partie de la grille.

```
# is_square (empty_grid 4 3) (1, 1);;
- : bool = true
# is_square (empty_grid 4 3) (0, 1);;
- : bool = false
# is_square (empty_grid 4 3) (2, 3);;
```

```
- : bool = true
# is_square (empty_grid 4 3) (2, 4);;
- : bool = false
# is_square (empty_grid 4 3) (5, 4);;
- : bool = false
```

3. **(Bonus)** Définir une fonction

```
val solve_brute_force : spec -> bool * grid = <fun>
```

paramétrée par une spécification et qui renvoie le couple `(true, g)` si la spécification admet la grille `g` comme solution et `(false, g)` dans le cas contraire où `g` peut être dans ce cas-ci une grille quelconque.

Pour ce faire, nous suivons l'algorithme prenant en entrée une spécification s qui s'articule de la façon suivante :

(A) Soient n et m les dimensions de s ;

(B) soit g la grille vide de taille $n \times m$;

(C) soit c la case $(1,1)$ de g ;

(D) renvoyer $\mathcal{A}(g,c)$.

Cet algorithme utilise le sous-algorithme récursif $\mathcal{A}(g,c)$ prenant en entrée une grille g et une case c, défini de la façon suivante :

(SA) si g est solution de s, renvoyer le couple (vrai, g) ;

(SB) sinon, si c est une case en dehors de g, renvoyer le couple (faux, g) ;

(SC) sinon,

- (S1) soit g' la grille obtenue en noircissant la case c dans g ;
- (S2) soit p le profil de la ligne de g' sur laquelle se trouve c ;
- (S3) soit q le profil de la colonne de g' sur laquelle se trouve c ;
- (S4) soit c' la case suivante de c dans g' ;
- (S5) si les profils p et q sont compatibles avec ceux correspondants de s :
  - (Sa) si $\mathcal{A}(g',c') = (\text{vrai}, g'')$, renvoyer (vrai, g'') ;
  - (Sb) sinon, renvoyer $\mathcal{A}(g,c')$;
- (S6) sinon, renvoyer $\mathcal{A}(g,c')$.

```
# let sp = {rows = [[3]; [2]; [1; 1]; [2]; [3]; [1; 1]];
       columns = [[1; 1; 2] ; [1; 2]; [6]; [1]]};;
val sp : spec =
  {rows = [[3]; [2]; [1; 1]; [2]; [3]; [1; 1]];
   columns = [[1; 1; 2]; [1; 2]; [6]; [1]]}
# let (b, g) = solve_brute_force sp;;
val b : bool = true
```

```
val g : grid = {nb_rows = 6; nb_columns = 4; states = <fun>}
# print_string (grid_to_string g);;
|O|O|O|-|
|-|-|O|O|
|O|-|O|-|
|-|O|O|-|
|O|O|O|-|
|O|-|O|-|
- : unit = ()
```

Note : bien entendu, l'algorithme brute force a été choisi pour des raisons de simplicité et de faisabilité dans le cadre du temps limité de l'examen dont ce sujet est tiré. Il ne permet pas de résoudre des grosses spécifications. Il existe des algorithmes bien plus efficaces.

La documentation de tous les modules CAML est accessible à l'adresse suivante.

https://ocaml.org/api/

4 Annexes

Le fichier `Nonogram.ml` :

```
(*****************************************************************)
(*************************** Exercise 1 ***************************)
(*****************************************************************)

(* 1.1. *)
(* val interval : int -> int -> int list = <fun> *)

(* 1.2. *)
(* val interval_f : int -> int -> (int -> 'a) -> 'a list = <fun> *)

(* 1.3. *)
(* val is_positive_list : int list -> bool = <fun> *)

(* 1.4. *)
(* val is_sum_leq_list : int list -> int -> bool = <fun> *)

(* 1.5. *)
(* val are_equivalent_lists : 'a list -> 'b list -> ('a -> 'b -> bool)
      -> bool = <fun> *)

(* 1.6 *)
(* val cartesian_product_lists : 'a list -> 'b list -> ('a * 'b) list
      = <fun> *)

(*****************************************************************)
(*************************** Exercise 2 ***************************)
(*****************************************************************)

(* Given type. *)
type state = Empty | Black

(* Given type. *)
type square = int * int

(* Given type. *)
type grid = {
    nb_rows : int;
    nb_columns : int;
    states : square -> state
}

(* Given function. *)
let state_to_string c =
    match c with
        |Empty -> "-"
        |Black -> "O"

(* Given function. *)
```

```
let grid_to_string grid =
    let index_rows = interval 1 grid.nb_rows
    and index_columns = interval 1 grid.nb_columns in
    List.fold_left
        (fun res i ->
            let line = List.fold_left
                (fun res j ->
                    res ^ (state_to_string (grid.states (i, j))) ^ "|")
                "|"
                index_columns
            in
            res ^ line ^ "\n")
        ""
        index_rows

(* 2.1. *)
(* val empty_grid : int -> int -> grid = <fun> *)

(* 2.2 *)
(* val fill_square : grid -> square -> grid = <fun> *)

(* 2.3. *)
(* val fill_squares : grid -> square list -> grid = <fun> *)

(* 2.4. *)
(* val row : grid -> int -> state list = <fun> *)

(* 2.5. *)
(* val column : grid -> int -> state list = <fun> *)

(*****************************************************************)
(***************************** Exercise 3 ****************************)
(*****************************************************************)

(* Given type. *)
type profile = int list

(* 3.1. *)
(* val profile : state list -> profile = <fun> *)

(* 3.2. *)
(* val are_profiles_compatible : profile -> profile -> bool = <fun> *)

(*****************************************************************)
(***************************** Exercise 4 ****************************)
(*****************************************************************)

(* Given type. *)
type spec = {
    rows : profile list;
    columns : profile list
```

```
}

(* 4.1. *)
(* val dimensions_spec : spec -> int * int = <fun> *)

(* 4.2. *)
(* val is_solution : spec -> grid -> bool = <fun> *)

(*****************************************************************)
(***************************** Exercise 5 ****************************)
(*****************************************************************)

(* 5.1. *)
(* val next_square : grid -> square -> square = <fun> *)

(* 5.2. *)
(* val is_square : grid -> square -> bool = <fun> *)

(* 5.3. *)
(* val solve_brute_force : spec -> bool * grid = <fun> *)
```