DS 2 du 19 avril 2024 - 2 heures 30

Calculatrice, téléphone et documents interdits

Les programmes non commentés ou dont le fonctionnement n'est pas expliqué ne seront pas relus. Une présentation raisonnablement aérée (avec une identation convenable des structures de choix ou de répétition), ainsi que des résultats soulignés ou encadrés, sont des éléments de l'évaluation... C'est vous qui voyez...

Exercice 1 [Tri par sélection d'un tableau]
Sur un tableau que l'on considère non vide dans tout l'exercice, le principe du tri par sélection est le suivant :

- rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
- rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
- continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

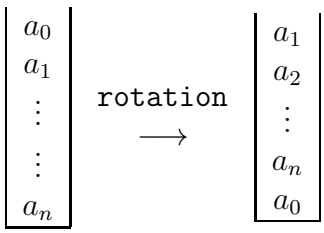
Cet algorithme très simple à programmer, mais pas forcément très efficace, est un algorithme « **en place** », c'est à dire que le tableau est directement modifié (il n'y a pas de copie).

1. Écrire une fonction `indice_mini : 'a array -> int -> int` telle que `indice_mini tab deb` renvoie l'indice du plus petit élément du tableau `tab` en ne s'intéressant qu'aux indices supérieurs où égaux à `deb`. En cas de doublons, on renvoie l'indice du premier élément convenable. Par exemple :

```
# indice_mini [|3;1;2;6;4;5|] 0;;
- : int = 1
# indice_mini [|3;1;2;6;4;5|] 3;;
- : int = 4
```

2. Écrire une fonction `trie : 'a array -> 'a array` qui reçoit un tableau et envoie ce tableau trié selon l'algorithme présenté.
3. Pour un tableau de n éléments, combien de comparaison d'élément sont réalisés lors de l'appel de la fonction `trie`

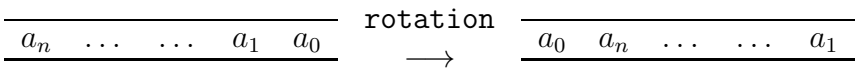
Exercice 2 [Rotation d'une pile, puis d'une file] On appelle rotation d'une pile, une fonction qui modifie une pile selon le principe suivant :



Pour cet exercice, on redonne les signatures de certaines fonctions des modules `Stack` et `Queue`

- `Stack.create : unit -> 'a Stack.t`
- `Stack.pop : 'a Stack.t -> 'a`
- `Stack.push : 'a -> 'a Stack.t -> unit`
- `Stack.is_empty : 'a Stack.t -> bool`
- `Queue.create : unit -> 'a Queue.t`
- `Queue.take : 'a Queue.t -> 'a`
- `Queue.add : 'a -> 'a Queue.t -> unit`
- `Queue.is_empty : 'a Queue.t -> bool`

1. a. Décrire (en français) un algorithme qui réalise la rotation d'une pile.
   b. Donner sa complexité (en terme de `push` et `pop`).
   c. Traduire cet algorithme en OCaml.
2. Reprendre les 3 questions précédentes avec la structure de file :



Exercice 3 [Connecteur de Sheffer] Le connecteur de Sheffer `nand`, noté $|$, est le connecteur binaire de la logique propositionnelle défini par $\boxed{F|G = \neg F \vee \neg G}$ où F et G sont des formules propositionnelles. On considère des formules propositionnelles construites avec des variables propositionnelles $a, b, c, \ldots$, les connecteurs binaires $\vee$(disjonction), $\wedge$(conjonction), $\rightarrow$(implication), le connecteur unaire de négation et le connecteur de Sheffer.

L'objet de l'exercice est de montrer que pour toute formule F de la logique propositionnelle, il existe une formule F^*, ne contenant que des variables propositionnelles et le connecteur de Sheffer, telle que F et F^* soient logiquement équivalentes

1. Construire la table de vérité d'une formule $x|y$, puis de $x|x$.
2. Montrer que l'on peut exprimer $\neg x$ et $x \vee y$ à l'aide du connecteur de Sheffer.
3. Montrer qu'il existe une formule sémantiquement équivalente à $x \rightarrow y$ ne contenant que les variables propositionnelles x et y et deux occurrences du connecteur de Sheffer.
4. De même, montrez qu'il existe une formule équivalente à $x \wedge y$ ne contenant que les variables propositionnelles x et y et des occurrences du connecteur de Sheffer.
5. Des questions précédentes, déduire un algorithme permettant de transformer toute formule F de la logique propositionnelle en une formule F^* ne contenant que des occurrences du connecteur de Sheffer et des variables propositionnelles présentes dans F. Appliquer cet algorithme à la formule $x \wedge (y \vee z)$
6. On note $\sigma(F)$, la taille de F. *On dit qu'une formule F est équilibrée lorsqu'elle est sans connecteur de négation et qu'elle vérifie :*
   - *Si F est une variable propositionnelle, elle est équilibrée ;*
   - *Si $F = G \star H$ avec $\star \in \{\vee, \wedge, \rightarrow, |\}$ alors G et H sont équilibrées et $\sigma(G) = \sigma(H)$.*

   Montrer que si F est équilibrée alors il existe $k \in \mathbb{N}$ tel que $\sigma(G) = 2^k - 1$ et préciser ce que représente k ?
7. On note $\sigma^*(n)$ la taille au pire de la transformée F^* d'une formule F équilibrée de taille n.
   a. Justifier que $\sigma^*(0) = 0$ et que $\forall n \in \mathbb{N}^*,\ \sigma^*(n) = 4\sigma^*(n-1) + 3$
   b. En déduire $\sigma^*(n)$ pour tout entier naturel n.

Exercice 4 [Roller Splat]

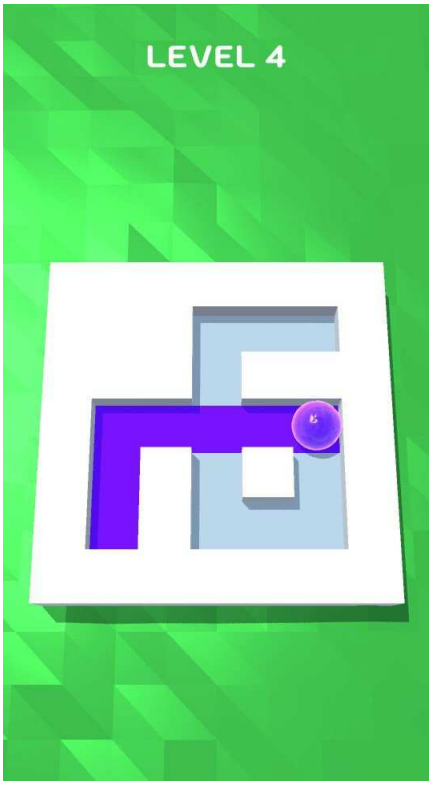
Le but du jeu Roller Splat est de repeindre un plateau de jeu à l'aide d'une bille de peinture. Lorsque l'on choisit une direction

0 : Haut, 1 : Droite, 2 : Bas, 3 : Gauche,

la bille avance jusqu'à toucher un mur. Dans cet exercice, on modélise le plateau de jeu par une matrice (tableau de tableaux) avec la convention suivante :

0 case vide, 1 : mur, 2 : case peinte, 3 : case où se trouve la bille.

Par exemple, le plateau ci-contre est modélisé par la déclaration :




```
let plateau = [|
      [|1; 1; 1; 1; 1; 1; 1|];
      [|1; 1; 1; 0; 0; 0; 1|];
      [|1; 1; 1; 0; 1; 1; 1|];
      [|1; 2; 2; 2; 2; 3; 1|];
      [|1; 2; 1; 0; 1; 0; 1|];
      [|1; 2; 1; 0; 0; 0; 1|];
      [|1; 1; 1; 1; 1; 1; 1|] |];;
```

A noter enfin que l'on peut repasser sur une case peinte et que le pourtour du plateau de jeu est toujours entouré de murs.

1. Écrire une fonction `dim : 'a array array -> int * int` qui reçoit une matrice et renvoie le couple (l, c) correspondant au nombre de lignes et de colonnes de ce tableau.
2. Écrire une fonction `gagne : int array array -> bool` qui reçoit une matrice et renvoie un booléen indiquant si la partie est gagnée ou non.
3. Écrire une fonction `case : int array array -> int * int` qui reçoit une matrice et renvoie un couple de la forme (ligne, colonne) indiquant où la bille se peinture se trouve.
4. Pour simplifier la suite du code, on souhaite réaliser la table de correspondance suivante :

Français	Numérique	Vecteur $(\Delta_{ligne}, \Delta_{colonne})$
Haut	0	$(-1, 0)$
Droite	1	$(0, 1)$
Bas	2	$(1, 0)$
Gauche	3	$(0, -1)$

   a. Écrire une fonction `dir2num : int * int -> int` qui converti un couple de direction en la valeur numérique correspondante de $\{0, 1, 2, 3\}$ en **utilisant un filtrage**.
   b. Écrire une fonction `num2dir : int -> int * int` qui produit l'effet inverse, on pourra utiliser une table de correspondance à l'aide d'un tableau de couples d'entiers ne manière à n'utiliser ni `if`, ni filtrage. On appelle ce tableau **une table de hachage**.
5. Écrire une fonction `depPossible : int array array -> int * int -> int -> bool` qui reçoit la matrice du plateau de jeu, une case sous la forme d'un couple (ligne, colonne) ainsi qu'une direction de l'ensemble $\{0, 1, 2, 3\}$ et renvoie un booléen, indiquant si le déplacement depuis la case donnée et dans la direction donnée est possible (i.e : il n'y a pas de mur).
6. Écrire une fonction `nouvelle : int array array -> int -> unit` qui reçoit la matrice représentant le plateau et la direction souhaitée et modifie le plateau. Cette fonction est une fonction d'impression, elle modifie le plateau, mais ne renvoie "rien".

Exercice 5

1. Écrire une fonction récursive `colle : 'a list -> 'a list list -> 'a list list` qui associe à une liste d'éléments (l'alphabet) et à une liste de listes constituées d'éléments de l'alphabet, la liste de listes obtenues en ajoutant en tête l'un des éléments de l'alphabet. On souhaite obtenir par exemple

```
# colle [1;2;3;4] [[5;6];[7];[]];;
- : int list list = [[1; 5; 6]; [1; 7]; [1];
[2; 5; 6]; [2; 7]; [2]; [3; 5; 6]; [3; 7]; [3];
[4; 5; 6]; [4; 7]; [4]]
```

2. Écrire une fonction récursive `mots : 'a list -> int -> 'a list list` qui à une liste `alphabet` et un entier `n`, associe la liste de listes correspondant à tous les mots de longueur `n` construits avec les éléments de l'alphabet. On souhaite obtenir par exemple :

```
mots [1;2] 4;;
- : int list list = [[1; 1; 1; 1]; [1; 1; 1; 2];
       [1; 1; 2; 1]; [1; 1; 2; 2]; [1; 2; 1; 1];
       [1; 2; 1; 2]; [1; 2; 2; 1]; [1; 2; 2; 2];
       [2; 1; 1; 1]; [2; 1; 1; 2]; [2; 1; 2; 1];
       [2; 1; 2; 2]; [2; 2; 1; 1]; [2; 2; 1; 2];
       [2; 2; 2; 1]; [2; 2; 2; 2]]
```

Quelques pistes pour la correction

Correction 1 1. Voici une solution impérative où on conserve dans la référence `i`, l'indice du plus petit élément :

```
let indice_mini t deb =
  let i = ref deb in
     for j = deb + 1 to Array.length t - 1 do
       if t.(j) < t.(!i) then i := j
     done;
     !i
;;
```

Ou une solution récursive toujours élégante en OCaml (même si la structure de tableau n'est pas celle qui s'y prête le plus).

```
let rec indice_mini t = function
        | deb when deb = Array.length t - 1 -> deb
        | deb -> let i = indice_mini t (deb+1) in
                         if t.(i) < t.(deb) then i else deb
;;
```

2. Une solution utilisant une référence pour l'échange des valeurs :

```
let trie t =
  let temp = ref 0 in
    for i = 0 to Array.length t - 2 do (* le dernier est forcément bon *)
      let j = indice_mini t i in
        temp := t.(i);
        t.(i) <- t.(j);
        t.(j) <- !temp;
     done;
  t
;;
```

une autre sans référence (un identificateur local est crée à chaque tour de boucle :

```
let trie t =
  for i = 0 to Array.length t - 2 do (* le dernier est forcément bon *)
    let j = indice_mini t i in
      let temp = t.(i) in t.(i) <- t.(j); t.(j) <- temp;
  done;
  t
;;
```

Remarque : les fonctions renvoient le tableau comme demandé, mais de tout manières, celui-ci a été modifié par effet de bord, il n'est donc en pratique pas utile de le renvoyer.

3. Si le tableau possède n éléments,
   - Le premier élément est comparé aux $n-1$ autres $\longrightarrow n-1$ comparaisons.
   - Le second aux $n-2$ autres $\longrightarrow n-2$ comparaisons.
   - ...
   - L'avant dernier au dernier $\longrightarrow 1$ comparaison.

   Soit au total : $(n-1)+(n-2)+...+2+1 = \dfrac{n(n-1)}{2}$ comparaisons.

Correction 2 1. a. Voici une idée : soit `p` la pile sur laquelle on veut réaliser une rotation. On utilise une variable tampon `x` pour la tête et une autre pile `q` pour conserver `p`.

Étape	`p`	`q`	`x`
1 situation de départ	1 2 3 4		?
2 on place la tête de pile dans `x`	2 3 4		1
3 On vide `p` dans `q`		4 3 2	1
4 On empile `x` dans `p`	1	4 3 2	1
5 On vide `q` dans `p`	2 3 4 1		1

b. Comptons...

Étape	nombre de `push`	nombre de `pop`
2	0	1
3	$n-1$	$n-1$
4	1	0
5	$n-1$	$n-1$

Soit au total $4n-2$ opérations.

c. Traduction de l'algorithme (on ne renvoie pas la pile, celle-ci est modifiée par effet de bord) :

```
let rotation p =
  if not (Stack.is_empty p) then begin
    let x = Stack.pop p and q = Stack.create() in
      while not (Stack.is_empty p) do
        Stack.push (Stack.pop p) q;
      done;
      Stack.push x p;
      while not (Stack.is_empty q) do
        Stack.push (Stack.pop q) p;
      done;
  end;
;;
```

2. Pour la file, il n'y a rien à faire (pour tourner dans ce sens !). Le programme semble assez clair pour ne pas avoir à l'expliquer :

```
let rotation f = Queue.add (Queue.take f) f;;
```

Correction 3 1. On a directement

x	$x\|x$
V	F
F	V

et

x	y	$x\|y$
V	V	F
V	F	V
F	V	V
F	F	V

d'où l'expression `nand` : $x|y \equiv \neg(x \wedge y)$.

2. De la question précédente, on déduit que :
   - $\neg x \equiv x|x$
   - $x \vee y \equiv \neg(\neg x) \vee \neg(\neg y) \equiv (\neg x)|(\neg y) \equiv (x|x)\big|(y|y)$
3. $x \to y \equiv \neg x \vee y \equiv \neg(x \wedge \neg y) \equiv x|\neg y \equiv x\big|(y|y)$
4. Avec la même idée $x \wedge y \equiv \neg(\neg x \vee \neg y) \equiv \neg(x|y) \equiv (x|y)\big|(x|y)$
5. Peut définir l'algorithme par induction : si on note φ la fonction qui converti la formule au format demandé
   - Si F est une variable : $\varphi(F) = F$
   - Si F est de la forme $\neg G$ alors $\varphi(F) = \varphi(G)\big|\varphi(G)$.
   - Si F est de la forme $G_1 \vee G_2$ alors $\varphi(F) = \big(\varphi(G_1)\big|\varphi(G_1)\big) \,\Big|\, \big(\varphi(G_2)\big|\varphi(G_2)\big)$.
   - Si F est de la forme $G_1 \wedge G_2$ alors $\varphi(F) = \big(\varphi(G_1)\big|\varphi(G_2)\big) \,\Big|\, \big(\varphi(G_1)\big|\varphi(G_2)\big)$.
   - Enfin, si F est de la forme $G_1 \to G_2$ alors $\varphi(F) = \varphi(G_1)\Big|\, \big(\varphi(G_2)\big|\varphi(G_2)\big)$.
6. On raisonne par récurrence sur le nombre de connecteurs.
   - Si F ne contient pas de connecteur, alors $\sigma(F) = 0 = 2^0 - 1$.
   - Pour $k \in \mathbb{N}$, supposons le résultat vrai pour toute formule équilibrées à moins de k connecteurs. Si F est équilibrée et contient $k+1$ connecteurs alors F s'écrit $f = G \star H$ avec $\star \in \{\vee, \wedge, \to, |\}$ et G et H sont équilibrées avec $\sigma(G) = \sigma(H)$. Alors le nombre de connecteurs de G et H est inférieur ou égal à k donc par propriété de récurrence, on a $\sigma(G) = \sigma(H) = 2^\ell - 1$, finalement $\sigma(F) = \sigma(G) + \sigma(H) + 1 = 2^{\ell+1} - 2 + 1 = 2^k - 1$ avec $k = \ell + 1$.

   D'après ce qui précède, k est la hauteur de l'arbre représentant la formule logique.
7. a. Si F est de taille 0, c'est une variable elle est égale à sa transformée. Donc $\sigma^*(0) = 0$.
   Sinon, au pire, un connecteur $\star \in \{\vee, \wedge, \to, |\}$ donne 3 connecteurs de Sheffer, en plus des connecteurs de Sheffer engendrés par les arguments de ceux-ci. Donc $\sigma^*(n) = 4\sigma^*(n-1)^* + 3$.
   b. La suite $\big(\sigma^*(n)\big)_{n\in\mathbb{N}}$ est arithmético–géométrique, on trouve après résolution : $\sigma^*(n) = 4^n - 1$.

Correction 4 1. en utilisant `length` :

```
let dim p = Array.length p, Array.length p.(0);;
```

2. On parcours tout le tableau, si on trouve un 0, c'est que ce n'est pas gagné :

```
let gagne p = let g = ref true and l,c = dim p in
   for i = 0 to l-1 do
      for j = 0 to c-1 do
         if p.(i).(j) = 0 then g := false
      done
   done;
   !g
;;
```

3. Même principe pour trouver la balle, mais on cherche un 3 :

```
let case p = let rep = ref (0,0) and l,c = dim p in
   for i = 0 to l-1 do
      for j = 0 to c-1 do
         if p.(i).(j) = 3 then rep:= (i,j)
      done
   done;
   !rep
;;
```

4. Les 2 fonctions :

```
let dir2num = function
        | (-1,0) -> 0
        | (0,1) -> 1
        | (1,0) -> 2
        | _ -> 3
;;

let num2dir n = let table = [|(-1,0);(0,1);(1,0);(0,-1)|] in table.(n);;
```

5. on teste la présence d'un mur :

```
let depPossible p case d =
        let l,c = case and dl, dc = num2dir d in
                p.(l+dl).(c+dc) <> 1
;;
```

6. On avance jusqu'à toucher un mur

```
let nouvelle p d =
        let c = ref (case p) and dl, dc = num2dir d in
        while depPossible p !c d do
                p.(fst !c).(snd !c) <- 2;
                c:= fst !c + dl, snd !c + dc
        done;
        p.(fst !c).(snd !c) <- 3
;;
```

Correction 5

1. Une première solution récursive :

```
let rec colle l1 l2 = match (l1, l2) with
          [], _ -> []
        | [h1], [] -> []
        | [h1], (h2::t2) -> (h1::h2) :: (colle [h1] t2)
        | h1::t1, l2 -> (colle [h1] l2) @ (colle t1 l2)
;;
```

Une autre, non récursive mais qui utilise une fonction auxiliaire récursive (pour conserver `l2` en mémoire) :

```
let colle l1 l2 =
        let rec colleR la lb =
                match la, lb with
                        | [], _     -> []
                        | h::t, [] -> colleR t l2
                        | l, h::t  -> (List.hd l::h)::colleR l t
        in colleR l1 l2
;;
```

2. Une solution :

```
let rec mots alphabet = function
        | 0 -> [[]]
        | n -> colle alphabet (mots alphabet (n-1))
;;
```