TP n°5 - Piles et files

Récupérer le fichier `tp05_debut.ml` présent sur le site de classe et qui contient la définition des types pour les deux premières parties et une proposition de trame pour la dernière partie.

Travail à réaliser pour la séance de mise en commun de la semaine du 4 mai :

- Pour tous, au minimum : partie 1 et partie 2.1 ;
- Pour ceux qui souhaitent aller plus loin, et découvrir une nouvelle structure de données : partie 2.2 (plutôt difficile) ;
- Pour ceux qui sont tentés par un exercice un peu plus ludique : partie 3.

1 Évaluation d'expressions arithmétiques postfixées *(Exercice 1 du chapitre 7)*

On écrit habituellement les expressions arithmétiques sous forme *infixe*, en faisant figurer les opérateurs entre leur deux opérandes. Néanmoins, cette notation est ambiguë si on ne définit pas les priorités entre opérateurs : $1+2\times 3$ peut représenter $(1+2)\times 3$ ou $1+(2\times 3)$. Il faut alors introduire des règles de priorité ou des parenthèses.

La notation *postfixée* consiste à écrire d'abord les opérandes, puis leur opérateur ; par exemple $3+4$ s'écrit « 3 4 + » ; $(2+4)\times 3$ s'écrit « 2 4 + 3 × ». L'avantage de cette notation est que les expressions sont alors non ambiguës : pas besoin de parenthèses ni de règles de priorité.

L'évaluation d'une telle expression est réalisée à l'aide d'une pile. On lit l'expression de gauche à droite : lorsqu'on lit un entier, on l'empile ; lorsqu'on lit un opérateur, on dépile deux éléments (ses deux opérandes), on effectue l'opération et on empile le résultat.
A la fin de l'évaluation, la pile ne contient plus qu'un élément, qui est le résultat de son évaluation.

Pour représenter une telle expression, on va utiliser une liste mêlant des opérateurs et des entiers. On définit donc le type `lexeme` suivant :

```
type lexeme =
  | Entier of int
  | Operateur of char
;;
```

Pour les opérateurs, on se limitera aux opérateurs binaires `+, -, *, /` sur les entiers.

Pour la pile, on utilisera le module `Stack`, dont on rappelle quelques fonctions :

- `Stack.create : unit -> 'a Stack.t` crée une pile vide ;
- `Stack.push : 'a -> 'a Stack.t -> unit` ajoute un élément dans une pile ;
- `Stack.pop : 'a Stack.t -> 'a` supprime l'élément en haut d'une pile et le renvoie ;
- `Stack.is_empty : 'a Stack.t -> bool` teste si une pile est vide.

Q1 Écrire une fonction `evalue : lexeme list -> int` qui prend en argument une expression en notation postfixée sous forme de liste de lexèmes et qui renvoie le résultat de cette expression.
Par exemple, `evalue [Entier 2; Entier 4; Operateur '+'; Entier 3 ; Operateur '*']` renverra 18.

2 Implémentation d'une file

2.1 Implémentation avec deux piles

On considère désormais le type `file` suivant :

```
type 'a file = {
    entree : 'a Stack.t;
    sortie : 'a Stack.t
  };;
```

L'ajout d'un élément se fera dans la pile `entree`, la suppression d'un élément dans la pile `sortie`.

Q2 Écrire une fonction `creer_file : unit -> 'a file` qui renvoie une file vide.

Q3 Écrire une fonction `est_vide : unit -> 'a file` qui renvoie `true` si la liste est vide, `false` sinon.

Q4 Écrire une fonction `enfiler : 'a -> 'a file -> unit` qui ajoute un élément dans la file.

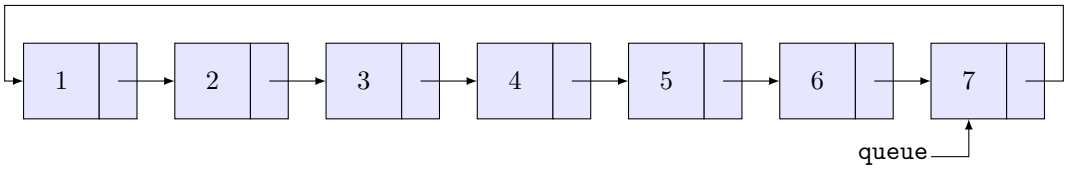
Q5 Écrire une fonction `defiler : 'a file -> 'a` qui supprime et renvoie l'élément en début de file.

Q6 Tester les fonctions précédentes :

- créer une file vide ;
- lui ajouter les entiers de 1 à 5 ;
- supprimer et afficher 3 éléments de la file ;
- lui ajouter les entiers 6 et 7 ;
- vider la file en affichant ses éléments.

2.2 Pour aller plus loin : implémentation à l'aide d'une liste chaînée circulaire

Une liste chaînée circulaire est une liste chaînée dont le dernier élément pointe vers le premier.



Afin de représenter une telle liste, on définit les types suivants :

```
type 'a cellule = {valeur: 'a ; mutable suivant: 'a cellule};;
type 'a liste = Nil | Cellule of 'a cellule;;
```

Nous allons utiliser une liste chaînée circulaire pour représenter une file, ce qui amène à définir le type suivant :

```
type 'a file = {mutable queue : 'a liste};;
```

On pointera sur le dernier élément de la file, autrement dit le dernier élément à être entré. Le premier élément à sortir sera donc l'élément suivant.

Par exemple, la liste représentée ci-dessous représentera la file $1-2-3-4-5-6-7$.

Q7 Écrire une fonction `creer_file : unit -> 'a file` qui renvoie une file vide.

Q8 Écrire une fonction `est_vide : unit -> 'a file` qui renvoie `true` si la liste est vide, `false` sinon.

Q9 Écrire une fonction `un_seul_element : 'a file -> bool` qui renvoie `true` si la file ne contient qu'un élément, `false` sinon.
La file ne contient qu'un seul élément si elle est non vide et que sa cellule de queue a pour élément suivant elle-même...

Q10 Écrire une fonction `enfiler : 'a -> 'a file -> unit` qui ajoute un élément dans la file.

Q11 Écrire une fonction `defiler : 'a file -> 'a` qui supprime et renvoie l'élément en début de file.

Q12 Tester vos fonctions avec le même protocole que dans l'exercice précédent.

3 Feu de forêt

On considère un damier représentant une forêt partiellement en feu. Au départ, une seule case est en feu, puis le feu se propage. La probabilité p que le feu se propage d'une case en feu aux cases voisines est constante.
On considère que le damier est un carré de côté 400 et que l'incendie naît au centre du carré.

L'algorithme de propagation utilise une structure de file.

```
Allumer le centre du damier.
Insérer le centre dans la file des cases en attente (précédemment vide).
Tant que la file n'est pas vide,
    extraire une case de la file.
    Pour chacune des cases voisines,
        Si la case n'est pas en feu,
            tirer au sort la propagation avec la probabilité p,
            Si le feu se propage,
                allumer la case
                l'ajouter à la file.
```

Q13 Compléter la fonction `propagation` qui simule graphiquement l'évolution de l'incendie. Elle prend en argument la probabilité p ; chaque case sera représentée par un pixel.

Q14 Tester différentes valeurs de p et trouver expérimentalement la plus petite valeur de p pour laquelle les chances de propagation du feu jusqu'aux bords semblent non nulles.