

Programmation impérative et fonctionnelle avec OCaml

TP 5

Objectifs :

- Manipulation de listes fonctionnelles.
- Modularité :
 - fichiers d’interfaces ;
 - compilation séparée ;
 - alias de module.
- Parcours de structure de données récursive.

1 File d’attente fonctionnelle

On pourrait utiliser une liste pour représenter une file d’attente (structure de données *First In, First Out* ou FIFO). L’opération d’ajout d’un élément consiste alors à le concaténer en fin de liste, et l’opération de retrait à récupérer la tête de liste (ou l’inverse). Cependant, l’opération d’ajout est très coûteuse puisqu’il faut parcourir la liste entièrement (complexité linéaire à chaque ajout).

On peut améliorer la **complexité amortie** (i.e. la complexité d’une *séquence* d’opérations d’ajout et de retrait sur la file) avec la technique suivante :

- la file est constituée de **deux** listes désignées **Entrée** et **Sortie** ;
- l’ajout d’un élément à la file consiste à l’ajouter en tête de la liste **Entrée** ;
- le retrait d’un élément de la file consiste à retirer l’élément **en tête** de la liste **Sortie** **si elle n’est pas vide**, sinon il faut inverser la liste **Entrée**, puis remplacer la liste **Sortie** par cette liste inversée et enfin remplacer la liste **Entrée** par la liste vide – on peut ensuite retirer l’élément en tête de la liste **Sortie** comme dans le cas où elle n’est pas vide.

On obtient ainsi une structure de données avec ajout et retrait en temps constant, sauf quand la liste **Sortie** devient vide (et dans ce cas l’opération de renversement de la liste a une complexité linéaire). Pour une séquence arbitraire de n ajouts et de n retraits, on aura besoin d’inverser la liste **Entrée** à chaque fois que **Sortie** est vide : chaque inversion prend un temps linéaire avec la taille de **Sortie**, et la somme de ces temps d’inversion est proportionnelle à n . La **complexité amortie** de chaque opération est donc **constante**.

1. Définir le type **t** des files d’attente contenant n’importe quel type de donnée à l’aide d’un **tuple** dans un fichier **fifo.ml**.
2. Définir une exception indiquant qu’une file est vide.
3. Définir la file vide **empty**.
4. Définir la fonction **is_empty** qui prend en paramètre une file et renvoie un booléen.
5. Définir la fonction **add** d’ajout d’un élément dans la file qui prend un élément et une file en paramètres et renvoie la nouvelle file.
6. Définir la fonction **take** qui prend une file en paramètre et renvoie l’élément en tête de file **et** la nouvelle file. On lèvera l’exception définie précédemment si la file est vide.
7. Écrire le fichier d’interface **.mli** correspondant **en cachant l’implémentation** du type des files (*type abstract*).

2 Pile fonctionnelle

Réutiliser l'implémentation de pile fonctionnelle vue en cours (p. 155) en changeant le nom de la fonction `pop` en `take`.

Dans les fichiers d'implémentation `lifo.ml` et d'interface `lifo.mli` :

1. Ajouter la fonction `add`.
2. Ajouter la fonction `is_empty`.
3. Ajouter la fonction `size`.

Les modules `Fifo` et `Lifo` doivent posséder la même interface (hormis la fonction `size`).

3 Utilisation des files et des piles

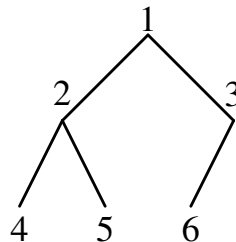
On va parcourir une structure d'arbre binaire en utilisant soit une file soit une pile.

1. Dans un nouveau fichier `tree.ml`, définir le type des arbres binaires constitués soit d'un nœud associé à une donnée quelconque, soit de l'arbre vide.
2. On peut accéder à un module `X` à l'aide d'un **alias** avec la notation suivante :

```
module M = X
```

Renommer le module `Fifo` en `Set`.

3. Écrire la fonction `iter` sur les arbres binaires équivalente à la fonction `List.iter` sur les listes en utilisant la structure de donnée fournie par le module `Set` :
 - on initialise la structure en ajoutant la racine de l'arbre à la structure vide ;
 - si la structure n'est pas vide, on retire un nœud, on traite la donnée associée à l'aide de la fonction passée en paramètre à `iter` puis on appelle récursivement l'itérateur sur la structure dans laquelle on aura ajouté les fils du nœud traité.
4. Appliquer cet itérateur à l'arbre suivant :



pour afficher l'entier contenu à chaque nœud.

5. Utiliser le module `Lifo` à la place du module `Fifo` et comparer les résultats obtenus.