Programmation impérative et fonctionnelle avec OCaml

TP 1

Objectifs :

- Prise en main de l'environnement de développement :
  - terminal avec `bash` ;
  - éditeur `emacs` ;
  - interpréteur `ocaml` ;
  - compilateurs `ocamlc` et `ocamlopt`.
- Programmation impérative :
  - références ;
  - tableaux ;
  - boucles `for`.
- Évaluation de temps de calcul et complexité.

Tableaux extensibles

On veut implémenter des tableaux d'entiers extensibles similaires aux listes du langage Python, avec une stratégie de « sur-allocation » :

- On initialise le tableau « vide » avec une taille s_0 (par exemple $s_0 = 10$), en utilisant la première case pour stocker le nombre d'éléments insérés (0 initialement).
- Quand on veut ajouter un nouvel élément à la fin d'un tableau de taille réelle n :
  - soit le nombre d'éléments est inférieur à la taille du tableau et on ajoute le nouvel élément dans la case libre suivante ;
  - soit le tableau est plein et on crée un nouveau tableau de taille cn, avec c le facteur d'agrandissement du tableau (par exemple $c = 1.125$), puis on recopie l'ancien tableau dans le nouveau et on ajoute le nouvel élément.

1. Définir les constantes globales s_0 et c.
2. Pour pouvoir changer **en place** (comme en Python) le tableau quand il doit être agrandi, on utilisera **exceptionnellement** une *référence* sur un tableau pour implémenter un tableau extensible (dont le type sera donc `int array ref`). Écrire la fonction de création :

   ```
   val make : unit -> int array ref
   ```

   qui renvoie **exceptionnellement** une référence sur tableau de taille $s_0 + 1$ (pour pouvoir stocker le nombre d'éléments utilisés dans la première case du tableau) initialisé à 0.
3. Écrire la fonction d'accès au i^{e} élément d'un tableau extensible :

   ```
   val get : int array ref -> int -> int
   ```

   en utilisant la fonction :

   ```
   val invalid_arg : string -> 'a
   ```

pour lever une exception si on essaie d'accéder à un élément non encore utilisé.

4. Écrire la fonction d'ajout d'un élément en fin de tableau :

   ```
   val append : int array ref -> int -> unit
   ```

   L'élément sera ajouté après le dernier élément utilisé si le tableau n'est pas plein et le nombre d'éléments sera mis à jour. Sinon, il faudra créer un nouveau tableau et **mettre également à jour la référence passée en paramètre**.

5. Écrire une fonction pour mesurer le temps de calcul nécessaire à l'ajout d'un nouvel élément (fonction `append`) en fonction du nombre d'éléments utilisés :

   ```
   val time : int -> int -> unit
   ```

   On passera en paramètres à la fonction le nombre n d'éléments à ajouter (e.g. $n = 36000$) et le nombre k de tableaux – on effectue les ajouts sur un grand nombre de tableaux pour obtenir des temps de calculs significatifs, e.g. $k = 1000$. On pourra construire un tableau « normal » de k tableaux extensibles, puis mesurer le temps de calcul pour ajouter aux k tableaux le i^{e} élément et écrire le résultat sur la sortie standard sous la forme d'un couple de coordonnées afin de pouvoir visualiser le résultat avec `gnuplot` :

   ```
   ...
   19787 0.000319
   19788 0.000319
   19789 1.704347
   19790 0.000671
   ...
   ```

6. Écrire l'équivalent de la fonction `main` avec une liaison globale `let () = ...` qui prendra en ligne de commande n et k grâce au tableau `Sys.argv` et à la fonction :

   ```
   val int_of_string: string -> int
   ```

7. Pour visualiser la courbe, on redirigera la sortie standard du programme dans un fichier :

   ```
   ./extarray 36000 1000 > append.txt
   ```

   puis on lancera `gnuplot` et on tracera la courbe avec la commande suivante :

   ```
   plot "append.txt" w l
   ```

8. Afficher également le temps de calcul *amorti* à chaque étape, i.e. la somme des i premiers temps de calcul divisée par i :

   ```
   ...
   19788 0.000319 0.001009
   19789 1.704347 0.001095
   ...
   ```

   On pourra visualiser les deux courbes simultanément avec la commande suivante (on multiplie les temps amortis par 500 pour qu'ils ne soient pas « écrasés » par la première courbe) :

   ```
   plot "append.txt" u 1:2 w l, "append.txt" u 1:($3)*500 w l
   ```