TP OCaml

Type définis par l'utilisateur

Exercice 1. On définit le type pile d'objets quelconques :

```
type 'a stack = {mutable len : int; mutable content : 'a list};;
```

Q.1 Écrire la fonction `create : unit -> 'a stack` qui crée une pile vide.

Q.2 Définir l'exception `Empty` qui a vocation à être soulevée quand on veut retirer le sommet d'une pile vide.

Q.3 Écrire la fonction `push : 'a -> 'a stack -> unit` qui ajoute un élément à une pile. La fonction met également à jour la longueur de la pile.

Q.4 Écrire la fonction `pop : 'a stack -> 'a` qui retire et renvoie le sommet d'une pile. Une exception `Empty` est soulevée en cas d'impossibilité. Sinon, la fonction met également à jour la longueur de pile.

Q.5 Se documenter ici sur les paramètres optionnels en OCaml.
Écrire la fonction `affiche_stack : ?fin:string -> int stack -> unit` qui affiche le contenu d'une pile.
La fonction prend un paramètre optionnel `fin` qui vaut par défaut la chaîne de passage à la ligne (on implémente l'attribut `end` du `print` de Python).

```
# let s = create () in
push 2 s; push 1 s;
affiche_stack s;
try
  while true do
    let n = s.len in
    Printf.printf "len : %d, sommet : %d,  \n" n (pop s);
  done;
with Empty -> Printf.printf "la pile est vide \n";;

{len:2; content:[1;2;]}
len : 2, sommet : 1,
len : 1, sommet : 2,
la pile est vide
- : unit = ()
```

Exercice 2. D'après Wikipedia.

Les *tours de Hanoï* (originellement, la tour d'Hanoïa) sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de « départ » A à une tour d'« arrivée » C en passant par une tour « intermédiaire » B, et ceci en un minimum de coups, tout en respectant les règles suivantes :

— on ne peut déplacer plus d'un disque à la fois ;

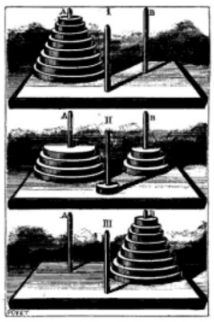

FIGURE 1 – La tour de Hanoï (original de l'œuvre de Lucas)

— on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ A contient la pile, B, C sont vides.

Q.1 Donner un algorithme de type « Programmation dynamique » pour résoudre le problème.

Q.2 Montrer la terminaison et la correction de cet algorithme.

Q.3 Calculer la complexité en nombre de déplacement de disques.

Q.4 Les tours de Hanoï sont représentés par des piles du type `stack` (voir exercice précédent). Par exemple :

```
{len = 8; content = [1;2;3;4;5;6;7;8]}
```

représente la configuration initiale de la figure 1.
Écrire la fonction `hanoi : 'a stack -> 'a stack -> 'a stack -> unit` qui déplace les disques de la première pile à la dernière en utilisant la seconde comme pile intermédiaire.

```
# let a = create () in
push 3 a; push 2 a; push 1 a;
let b= create () and c = create () in
hanoi a b c;
affiche_stack a; affiche_stack b; affiche_stack c;;

{len:0; content:[]}
{len:0; content:[]}
{len:3; content:[1;2;3;]}
- : unit = ()
```

Q.5 La fonction précédente est peu satisfaisante car on a l'impression (trompeuse) qu'il ne se passe grand chose pendant l'exécution.
Écrire le fonction `affiche_hanoi : int stack -> int stack -> int stack -> unit` telle que `affiche_hanoi a b c` affiche le contenu des 3 piles *initiales* au commencement puis entre les deux appels internes.

```
# let a = create () in
push 3 a; push 2 a; push 1 a;
let b= create () and c = create () in
affiche_hanoi a b c;;

n=-1 [{len:3; content:[1;2;3;]},{len:0; content:[]},{len:0; content:[]}]
```

```
n=1 [{len:2; content:[2;3;]},{len:0; content:[]},{len:1; content:[1;]}]
n=2 [{len:1; content:[3;]},{len:1; content:[2;]},{len:1; content:[1;]}]
n=1 [{len:1; content:[3;]},{len:2; content:[1;2;]},{len:0; content:[]}]
n=3 [{len:0; content:[]},{len:2; content:[1;2;]},{len:1; content:[3;]}]
n=1 [{len:1; content:[1;]},{len:1; content:[2;]},{len:1; content:[3;]}]
n=2 [{len:1; content:[1;]},{len:0; content:[]},{len:2; content:[2;3;]}]
n=1 [{len:0; content:[]},{len:0; content:[]},{len:3; content:[1;2;3;]}]
- : unit = ()
```

Ici, $n = -1$ représente l'état des piles au commencement.

Exercice 3. On définit le type *listes imbriquées* suivant :

```
# type 'a node =
    | One of 'a
    | Many of 'a node list;;
```

Un nœud d'une liste imbriquée est soit un élément soit une liste de nœuds.

Q.1 Écrire la fonction `flatten node` qui aplatit la liste imbriquée `node` :

```
# let node = Many [One "a"; Many [One "b"; Many [One "c" ;One "d"]; One "e"]]
    in flatten node;;
- : string list = ["a"; "b"; "c"; "d"; "e"]
```

Q.2 Ecrire la fonction `count a n` qui compte le nombre d'occurences de l'étiquettes `a` dans `n`

```
# let n = Many [One "a"; Many [One "a"; Many [One "c" ;One "d"]; One "e"];
    Many([Many([One "a"])])] in count "a" n;;
- : int = 3
```

Q.3 Écrire la fonction `subs (n:'a node) (x: 'a) (y: 'a) : 'a node` qui remplace toutes les occurrences de `x` par `y` :

```
let n = Many [One "a"; Many [One "b"; Many [One "c" ;One "d"]; One "e"];One "
    d"] in
subs n "d" "z";;
```

Q.4 La *hauteur* d'une liste de listes est le nombre maximum de `Many` imbriqués. Ecrire la fonction `hauteur n` qui retourne la hauteur de la liste de listes `n`.

```
 # let n = Many [One "a"; Many [One "b"; Many [One "c" ;One "d"]; One "e"]]
     in hauteur n;;
- : int = 3
```

Q.5 Ecrire la fonction `profondeur_max` qui prend en paramètre une liste de listes et retourne les étiquettes de plus grande profoncdeur :

```
# let n = Many [One "a"; Many [One "b"; Many [One "c" ;One "d"]; One "e"];
    Many([Many([One "f"])])] in profondeur_max n;;
- : string list = ["f"; "d"; "c"]
```

Q.6 Écrire la fonction `depth_and_labels (n:'a node) : (int * 'a list) list` qui renvoie niveau de profondeur par niveau de profondeur, une liste des contenus des singletons (comme `One 1`).

```
# let n = Many [One "a"; Many [One "b"; Many [One "c" ;One "d"]; One "e"];
    Many([Many([One "f"])]);One "g"] in
depth_and_labels n;;
  - : (int * string list) list =
[(0, []); (1, ["a"; "g"]); (2, ["b"; "e"]); (3, ["c"; "d"; "f"])]
```