TP2 Caml : prise en pied

Sujet à finir pour le dimanche 30 novembre 2014.
À rendre par mail à aurelie.lagoutte@ens-lyon.fr

1 Communiquer via la console

Il est possible d'utiliser `oCaml` en mode compilé. Éditez un fichier `hello.ml`, contenant la commande `print_string "Hello World!"; print_newline();;`. Dans le terminal, compilez puis lancez le programme :

```
ocamlc -o hello hello.ml
./hello
```

Le programme `oCaml` compilé exécute toutes les commandes du code source. Testez le code source suivant :

```
print_string "What is your name? ";;
let s = read_line();;
print_string "Hello world!\n";;
print_string "Welcome to "^ s;;
print_newline();;
print_endline "Nice to meet you.";;
5+12;;
print_newline();;
let x = 5+12;;
print_int(x);;
print_newline();;
```

Question 0 Quelle est la différence entre `print_string` et `print_endline` ?

Exercice 1 *Utilisez le code de la suite de Fibonacci du TP1, pour obtenir un fichier exécutable qui lit un entier, et renvoie le n-ième terme de la suite de Fibonacci (voir exemple ci-dessous). On utilisera la fonction* `int_of_string: string -> int` *qui transforme une chaîne de caractère contenant un entier en l'entier lui-même (par exemple,* `int_of_string "34"` *renvoie* `-: int = 34`*). On peut decliner cette formule à volonté :* `float_of_int`, `string_of_int`, *...*

```
Bonjour, veuillez entrer un entier:
4
fibo(4)=3
```

2 Construction de types

Un nouveau type peut être défini à l'aide de la commande suivante :

```
type couleur =
 |Rouge
 |Jaune
 |Vert
 |Bleu ;;

Rouge ;; (* les constructeurs doivent commencer par une majuscule *)

let est_couleur_primaire=function
 |Vert->false
 |_->true;;
```

Les constructeurs peuvent être accompagnés d'arguments ou être récursifs :

```
type nombre =
 |Int of int
 |Real of float ;;

Int(3) ;;
Real(5) ;;

type liste =
 |Liste_vide
 |Cons of (int*liste) ;;
```

Exercice 2 (Pattern matching sur types construits) *En réutilisant les types définis ci-dessus, écrivez une fonction* `somme : nombre -> nombre -> nombre`. *Écrivez également une fonction* `longueur:liste -> int`.

3 Exceptions

Comme de nombreaux autres langages de programmation, Caml permet de soulever des exceptions grâce à la commande `raise` *mon_ exception*. L'expression *mon_ exception* doit être de type `exn`, le type de Caml pour les exceptions. Le constructeur le plus simple pour le type `exn` est `Failure of string`, comme sur l'exemple ci-dessous :

```
let diviser a b= match b with
   |0-> raise (Failure "Division par zéro")
   |_->a/b;;
```

Lorsqu'une exception est soulevée, le calcul est arrêté et l'exception est renvoyée. On peut la rattraper avec `try .... with ....` :

```
let rec moyenne_aux=
  |[]->(0,0)
  |t::q-> let (s,l)=(moyenne_aux q) in (s+t,l+1);;
```

Question Que fait la fonction `moyenne_aux` ?

```
let moyenne l=
  let (s,long)=moyenne_aux l in
    try diviser s long with
      |Failure "Division par zéro"->
              Failure "Calcul de la moyenne impossible sur une liste vide" ;;
```

La construction `try` *expr0* `with` *filtrage sur les expressions* veut dire "essaye de calculer expr0, et s'il n'y a pas d'exception soulevée, renvoie sa valeur. Sinon, filtre l'exception soulevée avec le filtrage et renvoie la valeur appropriée selon les cas".

Exercice 3 *Écrire une fonction* `nth: 'a list-> int -> 'a` *qui renvoie le n-ième élément d'une liste. Elle soulèvera l'exception* `Failure " Liste trop courte"` *ou l'exception* `Failure "n négatif ou nul"` *les cas échéants. Écrire ensuite une fonction* `nth_or_the_devil: int-> 'a list-> 'a` *qui renvoie le n-ième élément de la liste si celui-ci existe, qui renvoie* `666` *si n est trop grand, et* `-666` *si n est trop petit. Elle devra utiliser la fonction* `nth` *et rattraper les exceptions.*

Un point intéressant est que l'on peut rajouter des exceptions à loisir :

```
exception Failure_entier of int;;
```

Comparer ensuite les types des expressions suivantes :

```
Failure "Une erreur s'est produit";;
Failure_entier 3;;
```

Ceci permet de rajouter un nouveau constructeur `Failure_entier` au type `exn`, qui prend comme argument un entier.

Exercice 4 *Créer un constructeur d'exception* `Stop` *qui ne prend pas d'argument, et un constructeur d'exception* `Failure_bool` *qui prend comme argument un booléen.*

Exercice 5 *Vous pourrez créer d'autres exceptions avec des noms appropriés.*
- *Écrivez une fonction* `occurence` *qui, étant donné un élément x et un tableau tab, renvoie le nombre d'occurences de x dans tab. Écrivez ensuite une fonction qui parcourt un tableau avec une boucle* `for`*, qui s'interrompt et renvoie x si elle a trouvé un élément x qui apparaît plus de $n/2$ fois dans le tableau (où n est la longueur du tableau).*
- *Écrivez une fonction qui prend en argument une liste d'entiers, et renvoie le produit des éléments de la liste, sauf si la liste contient 0, et dans ce cas elle renverra la somme des éléments.*

4 Le type option

4.1 Rappel sur les tableaux

On rappelle la syntaxe des opérations de base sur les tableaux :

```
let tab=Array.create 10 0;; (* création d'un tableau*)
tab;;

tab.(1)<- 1;;
tab;;
```

```
for i= 2 to 9 do
  tab.(i)<- i; (* affectation d'une valeur à une case du tableau*)
done;;
tab;;

let trois=tab.(4);;
```

4.2 Qu'est-ce que le type option ?

Imaginons que l'on veuille écrire une fonction qui, étant donné un tableau (ou une liste) et un élément x, cherche l'index éventuel auquel l'élement x apparaît dans le tableau/la liste. Un problème se pose lorsque x n'est pas dans le tableau/la liste, dans ce cas on aimerait ne pas renvoyer d'indice (parfois, on utilise un indice "stupide", comme -1 par exemple). Caml dispose d'un type `option` pour gérer ce genre de cas. Il a été défini ainsi :

```
type 'a option=
  |None
  |Some of 'a
```

Exercice 6 (Type option) – *Déclarez une variable* `a` *de type* `int option` *et qui contient 3.*
- *Déclarez une variable* `b` *de type* `int option` *et qui ne contient rien. Quel est son type ? Comment faire pour avoir réellement le type souhaité ?*
  Indice : essayer la syntaxe `(variable:type)`*.*
- *Déclarez une varialbe* `tab` *contenant un tableau de longueur 10 rempli de valeur* `None`*. Quel est son type ? Regardez attentivement...*
- *Rangez la valeur* `a` *dans la première case du tableau.*
- *Quel est maintenant le type de* `tab` *?*
- *Remettez la valeur* `None` *à la place de* `a`*.*
- *Quel est maintenant le type de* `tab` *? Expliquez alors la différence entre un type* `'a truc` *et* `'_a truc`*.*
- *On suppose que l'on a une fonction f :' $a->'$ b qui déclenche l'exception* `Failure "invalide"` *pour certaines valeurs de son argument. Réalisez une fonction* `optionize` *qui prend en argument f et qui renvoie une fonction de type* `'a -> 'b option` *dont le résultat est* `None` *quand la fonction f déclenche l'exception, et* `Some valeur` *dans le cas où f renvoit* `valeur`*. Pour effectuer vos tests, vous écrirez une telle fonction f.*

Exercice 7 *Écrire la fonction* `recherche: 'a -> 'a list -> int option` *qui était notre but initial :* `recherche x l` *recherche l'index éventuel i de* `x` *dans* `l`*.*

5 Arbres binaires

Un *arbre binaire de recherche* est un arbre dont les noeuds sont étiquetés par des entiers, et tel que pour chaque noeud x, les étiquettes apparaissant dans le sous-arbre gauche de x sont inférieures à l'étiquette de x, et les étiquettes apparaissant dans le sous-arbre droit de x sont supérieures à l'étiquette de x. Il est dit *équilibré* si pour chaque noeud, les sous-arbres droit et gauche ont la même taille (plus ou moins 1).

Exercice 8 (Arbres binaires) *On reprend le type* `'a arbre` *défini au TP1.*

1. *Écrivez une fonction renvoyant la liste des nœuds d'un arbre binaire, lus en ordre infixe (gauche-nœud-droite). On pourra utiliser la fonction* `List.append`.
2. *À l'aide de la fonction tri-fusion du TP1, écrivez une fonction prenant un arbre binaire d'entiers et renvoyant un arbre binaire équilibré trié contenant les mêmes entiers : on pourra décomposer en trois étapes : obtenir la liste des étiquettes, la trier, puis construire un arbre binaire de recherche équilibré à partir de cette liste.*
3. *Écrivez une fonction de recherche d'un élément dans un arbre trié, donnant la profondeur à laquelle se trouve l'élément le cas échéant (*`None` *s'il n'y est pas).*
4. *Écrivez une fonction qui renvoie un booléen indiquant si l'arbre est équilibré ou non.*

6 Fonctions sur les entiers

Exercice 9 (pgcd) *Écrivez une fonction* `pgcd: int -> int -> int` *qui renvoie le pgcd de deux entiers. Pour réduire le nombre de cas dans le filtrage, on pourra filtrer sur* `(min a b, max a b)`.

Exercice 10 (Composition de fonctions) – *Écrivez une fonction* `composition_repetee: ('a -> 'a) -> int -> 'a -> 'a` *telle que* `composition f n a` *renvoie* `f ( f ( ... f(a)))` *où* `f` *est appliquée n fois.*
– *Que renvoie l'instruction* `map (fun n-> composition_repetee (fun x->2*x) n 1) [1;2;3;4;5];;` *et pourquoi ?*

Caml permet de créer ses propres opérateurs binaires infixes (c'est-à-dire que le symbole se place entre les deux arguments, comme dans `x+ y`). Le nom ne doit pas comporter de caractère alphanumérique, et doit être mis entre parenthèses lors de sa définition.

Exercice 11 (Opérateurs infixe) – *Écrivez l'opérateur infixe* `|..|: int-> int-> int list` *tel que* `3|..|10` *renvoie la liste* `[3; 4; 5; 6; 7; 8; 9; 10]`. *Elle soulèvera une exception si le premier entier est plus grand que le deuxième.*
– *Écrivez l'opérateur* `|&|` *du* ou exclusif .
– *Écrivez l'opérateur* `><` *telle que* `l >< x` *supprime de la liste* `l` *toutes les occurences éventuelles de l'élément* `x`.

7 Tris

Exercice 12 (Tri par insertion) *Écrivez une fonction* `insere: 'a -> 'a list -> 'a list` *telle que* `insere a liste_triee` *insère l'élément* `a` *au bon endroit dans la liste* `liste_triee` *que l'on suppose déjà triée. En déduire une fonction récursive* `tri_insertion: 'a list -> 'a list` *qui trie une liste en insérant les éléments les uns après les autres dans la partie déjà triée de la liste (la récursivité nous permettra d'avoir seulement le premier élément à insérer).*

Exercice 13 (Tri par sélection) *Écrivez une fonction* `select_min: 'a list -> 'a * 'a list` *telle que* `select_min l` *renvoie le couple* `(a,reste)` *où* `a` *est l'élément minimum de* `l`, *et* `reste` *est une liste contenant tous les éléments de* `l` *sauf* `a`. *Elle devra fonctionner en un seul parcours de la liste. En déduire une fonction récursive* `tri_selection: 'a list -> 'a list` *qui trie une liste en sélectionnant le minimum de la liste, en le plaçant au début, puis en recommençant avec le reste.*

Exercice 14 (Tri rapide) *Écrivez une fonction* `coupe_pivot: 'a -> 'a list -> 'a list * 'a list` *telle que* `coupe_pivot a l` *renvoie le couple* `(l1,l2)` *où* `l1` *contient tous les éléments de* `l` *qui sont plus petits que* `a`*, et* `l2` *contient tous les éléments de* `l` *qui sont plus grands que* `a`*. Elle devra fonctionner en un seul parcours de la liste. En déduire une fonction récursive* `tri_rapide: 'a list -> 'a list` *qui trie une liste en choisissant le premier élément comme pivot, qui divise la liste en deux morceaux (les plus petits que* `a` *et les plus grands que* `a`*), qui place* `a` *au milieu et qui recommence récursivement.*

Exercice 15 (Tri à bulle) *Écrivez une fonction* `remonte_max: 'a list -> 'a list` *qui fait remonter le maximum de la liste à la fin de celle-ci, en n'échangeant que des éléments successifs. On remarquer que sur une liste triér, cette fonction fait remonter le premier élément de la liste à sa bonne place dans la liste. En déduire une fonction récursive* `tri_bulle: 'a list -> 'a list` *qui trie une liste en faisant remonter chaque élément "dans sa bulle".*