EPITA SPE promo 2011
Programmation
Épreuve machine

Marwan Burelle*

Instructions :

Vous devez lire l'intégralité du sujet, ainsi que l'ensemble de ces instructions. Tout ce qui est explicitement demandé dans ce sujet est obligatoire. La perte de points dûe au non respect des consignes explicites ne sera pas contestable, en particulier pour les problèmes de compilation de votre code.

Dans le répertoire `sujet` *vous trouverez un sous-répertoire* `Skel` *vous devez copier le* **contenu** *de ce répertoire dans votre répertoire de rendu.*

Dans ce répertoire vous trouverez : un fichier `Makefile` *permettant de compiler votre code, un certain nombre de fichiers annexes, un fichier pour chaque question de la forme* `questionX.c` *ou* `questionX.ml`*. Seuls les fichiers de question peuvent être modifiés, les autres seront remplacés lors de la correction par les originaux, toute modification sera perdue* **et** *sanctionnée (perte de point.)*

Le fichier `Makefile` *permet d'engendrer un petit programme de test pour chaque question. La méthode de compilation pour la correction sera celle décrite dans ce fichier, par conséquent vous n'aurez les points à la question X que si la commande "*`make questionX`*" réussit et que le résultat est correct.*

Pour chaque question le nombre de points est indiqué en face du numéro sur la droite de la page et entre parenthèse. Ce barème est donné à titre indicatif et peut être sujet à changement.

Il y a 23 points et 10 questions dans cet examen, tout point au dessus 20 est considéré comme point bonus.

Programmation OCaml : Récurrences arithmétiques

Question 1 (2)

On se propose de coder la fonction suivante :

```
val suite : int -> int -> int -> int
```

`suite u0 r n` renvoie le $n-$ième terme de la suite définit par récurrence :

$$\begin{cases} u_0 = \texttt{u0} \\ u_n = u_{n-1} + r \end{cases}$$

Attention : si `n` est négatif votre fonction devra échouer avec l'exception `Failure("erreur")` (que l'on obtient grâce `failwith "erreur"`.)

*Burelle@kh405.net

Question 2 (2)

On se propose de coder la fonction suivante :

```
val serie : (int -> int) -> int -> int
```

`serie f n` calcule la somme suivante :

$$\sum_{i=0}^{n} (\texttt{f}\ \texttt{n})$$

Attention : si `n` est négatif votre fonction devra échouer avec l'exception `Failure("erreur")` (que l'on obtient grâce `failwith "erreur"`.)

Programmation OCaml : Les listes

Question 3 (2)

On se propose de coder la fonction suivante :

```
val mapList : ('a -> 'b list) -> 'a list -> 'b list
```

`mapList f [a1; ...; an]` renvoie `(f a1)@ ... @(f an)`.

Question 4 (2)

On se propose de coder la fonction suivante :

```
val subset : ('a -> bool) -> 'a list -> 'a list
```

`subset f l` renvoie la liste des éléments `x` de `l` tel que `f x` renvoie vrai. **Attention l'ordre doit être préserver.**

Exemple : `subset (fun x -> x mod 2 = 0) [1;2;3;4] = [2;4]`

Question 5 (2)

On se propose de coder la fonction suivante :

```
val partie : ('a -> bool) -> 'a list -> 'a list * 'a list
```

`partie f l` renvoie un couple de liste `(l1,l2)` telles que pour tout élément `x` de `l` si `f x` renvoie alors `x` est dans `l1` sinon `x` est dans `l2`. **Attention l'ordre doit être préserver.**

Exemple : `partie (fun x -> x mod 2 = 0) [1;2;3;4] = ([2;4],[1;3])`

Programmation OCaml : Les records

Question 6 (3)

On définit les deux types suivants :

```
type listrec = { a : int; b : int; }
type reclist = { c : int list; d : int list; }
```

On se propose de coder la fonction suivante :

```
val listrec2reclist : listrec list -> reclist
```

`listrec2reclist l` renvoie un record de type `reclist` dont le champ `c` contient la liste des valeurs pour le champs `a` des records dans la liste `l` et le champ `d` contient la liste des valeurs pour le champs `b` des records dans la liste `l`.

Exemple : `listrec2reclist [{a=0;b=2};{a=1;b=3}] = {c=[0;1];d=[2;3]}`

Question 7 (2)

On définit le type suivant :

```
type r = { mutable n : int; mutable fact : int; }
```

On utilisera ce type pour *sauver* la valeur de factorielle d'un entier.

Exemple : `{n=4 ;fact=24}`

En utilisant le type `r` écrire une fonction qui renvoie à chaque appel la valeur suivante de factorielle en commencant par factorielle de 0. Votre fonction ne devra pas utiliser de variable globale, par contre elle pourra (devra) utiliser une partie *statique*.

```
val next_fact : unit -> int
```

Exemple :

```
next_fact();;
-: int = 1
next_fact();;
-: int = 1
next_fact();;
-: int = 2
next_fact();;
-: int = 6
next_fact();;
-: int = 24
```

Programmation OCaml : Les tableaux

Question 8 (2)

On se propose de coder la fonction suivante :

```
val somme_tab : int array -> int -> int -> int
```

`somme_tab t i j` renvoie la somme des cases du tableau `t` de `i` jusqu'à `j` (inclus)

`Exemple : somme_tab [| 0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 |] 2 4 = 9`

Attention : si les bornes sont mauvaises (`i` négatif ou plus grand que `j` ou `j` plus grand que la taille du tableau) alors votre fonction devra sortir avec l'exception :
`Invalid_argument("Mauvaises bornes.")` (respectez bien la chaîne de caractères en paramètre de l'exception.)

On rappelle qu'il existe la fonction :

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

qui lève l'exception `Invalid_argument` avec la chaîne passées en paramètre.

On rappelle qu'il existe la fonction :

```
val Array.length : 'a array -> int
```

qui renvoie la taille du tableau passé en argument.

Programmation OCaml : Les arbres

Question 9 (2)

Soit le type des arbres binaires définit à l'aide de record :

```
type 'a enr_arbre = { cle : 'a; fg : 'a arbre_bin; fd : 'a arbre_bin; }
and 'a arbre_bin = Vide | Noeud of 'a enr_arbre
```

Notez au passage la définition récursive du type utilisant le mot clef `and`.

On se propose de coder la fonction suivante :

```
val hauteur : 'a arbre_bin -> int
```

`hauteur a` calcule la hauteur de l'arbre fournit en argument. On rappelle que la hauteur est le plus chemin de la racine vers une feuille de l'arbre (ou le *nombre d'étages* de l'arbre.)

Question 10 (4)

Soit le type suivant représentant les arbres généraux avec la méthode *premier fils-frère droit* :

```
type 'a arbre_fils_frere =
    Nil
  | Node of 'a * 'a arbre_fils_frere * 'a arbre_fils_frere
```

On rappelle que la méthode *premier fils-frère droit* consiste à représenter un noeud dans un arbre général en lui associant son premier fils et son frère droit.

Exemple : L'arbre de la figure 1 sera représenté par la valeur :
`Node(1,Node(2,Nil,Node(3,Node(5,Nil,Nil),Node(4,Nil,Nil))),Nil)`

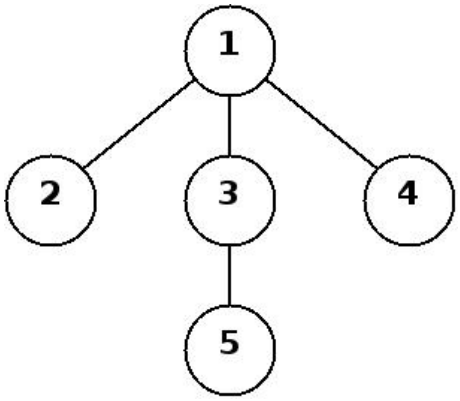

FIG. 1 – Un exemple d'arbre général

On définit une forme de représentation linéaire des arbres généraux : un arbre est représenté par un une chaîne entre parenthèse commencant parl a clef (la valeur du noeud) suivie des fils des cette arbre. L'arbre de la figure 1 sera ainsi représenté par la chaîne : `"(1(2(3(5)(4))))"`

On se propose de coder la fonction suivante :

```
val to_string : ('a -> string) -> 'a arbre_fils_frere -> string
```

`to_string ts t` renvoie la représentation linéaire (en chaîne de caractères) de l'arbre `t`. La fonction `ts` permet d'obtenir la chaîne de caractères représentant la clef de chaque noeud.

Exemple :

```
(* L'arbre de la figure 1 *)
let t =
  Node(1,Node(2,Nil,Node(3,Node(5,Nil,Nil),Node(4,Nil,Nil))),Nil);;
val t : int arbre_fils_frere = ...

to_string string_of_int t;;
-: string = "(1(2(3(5)(4))))"
```

Remarques sur les programmes de test

La commande `make questionX` produira un binaire `questionX`. Ce binaire peut être utilisé pour tester votre réponse à la question X du sujet. Tous les binaires attendent des paramètres et affichent une aide succinte s'il sont appelés avec l'option `-help`.

Exemple : le binaire produit pour la question 4 fournit l'aide suivante :

```
Prog TP: question 4
question4 graine taille
  -help   Display this list of options
  --help  Display this list of options
```

Les deux paramètres sont donc `graine` et `taille`. Les paramètres nommés `graine` font tous référence à l'initialisation d'un générateur de nombre aléatoire. Dans le cas présent la commande `./question4 X Y` va initialiser le générateur de nombre aléatoire avec X et l'utiliser pour engendrer une liste de taille Y. Pour les mêmes valeurs de X et de Y on obtiendra la même liste (la séquence engendrée par une graine donnée est toujours la même.)

Le reste du temps les noms des paramètres donnés dans l'aide sont explicites (par rapport au sujet.) Si vous avez une doute sur l'utilisation vous pouvez jetter un oeil sur fichier contenant les tests de la question X (`test-qX.ml`.)

Listes des fichiers à ne pas modifier

Ces fichiers **doivent être** dans votre répertoire de rendu (directement sans autre répertoire), mais ne doivent pas avoir été modifié par rapport à leur version original :

Makefile
test-q1.ml
test-q10.ml
test-q2.ml
test-q3.ml
test-q4.ml
test-q5.ml
test-q6.ml
test-q7.ml
test-q8.ml
test-q9.ml

Listes des fichiers modifiables

Ces fichiers **doivent être** dans votre répertoire de rendu (directement sans autre répertoire) et peuvent être modifiés (si vous répondez à la question bien sûr.)

question1.ml
question10.ml
question2.ml
question3.ml
question4.ml
question5.ml
question6.ml
question7.ml
question8.ml
question9.ml