EPITA SPE promo 2012
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.)*

Pour information, la copie des fichiers du répertoire d'origine vers votre répertoire de rendu, ce fait à l'aide des commandes :

```
> cd
> cd sujet
> cp Skel/* ~/rendu/
```

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.

*marwan.burelle@lse.epita.fr

Programmation OCaml : Récurrences arithmétiques

Question 1 (2)

On se propose de coder la fonction suivante :

```
val fact : int -> int
```

`fact n` renvoie factorielle de `n` et `-1` si `n` est négatif.

Question 2 (2)

On se propose de coder la fonction suivante :

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

`serie f n` calcule la somme suivante :

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

Attention : si `n` est négatif le comportement de votre fonction n'est pas défini et ne sera donc pas testé.

Programmation OCaml : Les listes

Question 3 (2)

On se propose de coder la fonction suivante :

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

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

Question 4 (2)

On se propose de coder la fonction suivante :

```
val existsn : ('a -> bool) -> int -> 'a list -> bool
```

`existsn f n l` renvoit vrai si pour au moins `n` éléments de `l`, la fonction `f` renvoit vrai.

Question 5 (2)

On se propose de coder la fonction suivante :

```
val valid2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
```

`valid2 f [a1 ; ... ; an] [b1 ; ... ; bn]` renvoit vraie si les deux liste ont la même taille et que pour tous i de 1 à n, `f ai bi` renvoit vrai.

Question 6 (3)

On se propose de coder la fonction suivante :

```
val inject : 'a list -> int -> 'a list -> 'a list
```

`inject [a1 ; ... ;an] k [b1 ; ... ; bj ; bk ; bm]` renvoit la deuxième dans laquelle la première a été insérée à la position `k`, *c.à.d.* `[b1 ; ... ; bj ; a1 ; ... ; an ; bk ; ... ; bm]`. Si `k` est plus grand que la taille de la seconde liste, l'injection se fait en fin. Les positions commencent à 0.

Exemples :

```
inject [42] 0 [1;2];;
-: int list = [42;1;2]
inject [1;2;3] 3 [4;5;6;7;8];;
-: int list = [4;5;6;1;2;3;7;8]
inject [42] 5 [1;2];;
-: int list = [1;2;42]
```

Programmation OCaml : Les références

Question 7 (2)

On se propose de coder la fonction suivante :

```
val next_odd : unit -> int
```

`next_odd ()` renvoit le prochain nombre pair en commencant par 2. À chaque appel, la fonction renvoit le nombre suivant de celui renvoyer la fois d'avant.

Exemples :

```
next_odd ();;
-: int = 2
next_odd ();;
-: int = 4
next_odd ();;
-: int = 6
next_odd ();;
-: int = 8
```

Question 8 (2)

On se propose de coder la fonction suivante :

```
val pre_incr : int ref -> int
```

`pre_incr i` incrémente de 1 la référence `i` et renvoit la valeur **après** incrémentation.

Question 9 (2)

On se propose de coder la fonction suivante :

```
val post_incr : int ref -> int
```

`post_incr i` incrémente de 1 la référence `i` et renvoit la valeur **avant** incrémentation.

Programmation OCaml : Les tableaux et records

Question 10 (4)

Dans cet exercice on va travailler avec des vecteurs (ou liste statique) construits sur le même modèle qu'en TD d'algo : un tableau et un champ contenant le nombre de cases vraiment utilisées dans le tableau. On donne les définitions de type et la fonction de création d'un vecteur vide :

```
type vecteur = {
  mutable nbElem : int;
  tab : int array;
}
```

```
let make_vecteur max init = {
  nbElem = 0;
  tab = Array.make max init;
}
```

On se propose d'écrire une fonction quit convertit une liste OCaml en vecteur. Cette fonction travaillera *en place*, *c.à.d.* elle prend un vecteur supposé vide et suffisament grand et le remplit. Elle ne renvoit rien, mais modifie le vecteur. Voici sa spécification :

```
val list2vect : vecteur -> int list -> unit
```

`list2vect v l` insère les éléments de la liste `l` dans le vecteur `v` en préservant l'ordre d'origine bien sûr.

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
question10.mli

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