

EPITA SPE promo 2017

Programmation - Épreuve machine

Marwan Burelle *

Vendredi 8 novembre 2013

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 due au non respect des consignes explicites ne sera pas contestable, en particulier pour les problèmes de compilation de votre code ou l'existence de sous-répertoire dans votre rendu.

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

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/
```

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 questionXX.c ou questionXX.ml. Seuls les fichiers de question peuvent être modifiés, les autres seront remplacés lors de la correction par les originaux, toute modification sera perdue.

Le Makefile permet d'engendrer un petit programme de test pour chaque question. Le programme de test se charge des interactions avec l'utilisateur et appelle votre fonction avec les paramètres attendus. Pour obtenir le programme de test de chaque question il faut faire la commande : make questionXX.

La compilation lors de la correction utilisera ce Makefile, par conséquent vous n'aurez les points à la question XX que si la commande "make questionXX" réussit et bien sûr que le résultat est correct.

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

À la fin de ce document vous trouverez des annexes décrivant :

- Quelques consignes sur les programmes de tests*
- La liste des fichiers à rendre (ceux modifiables et ceux à ne pas toucher.)*

Il y a 25 points et 12 questions dans cet examen.

*marwan.burelle@lse.epita.fr

OCaml : bases

Question 1

(1)

Écrire la(les) fonction(s) suivante(s):

val fact : int -> int

fact n renvoie factorielle de n. La fonction devra lever l'exception Neg_param(n) si n est négatif.

Exemple 1.1:

```
un_shell> ./question01 0 5
```

Fixed values:

fact 0 = 1 (should be 1)

fact 1 = 1 (should be 1)

fact 5 = 120 (should be 120)

Random tests:

fact 12 = 479001600

fact 10 = 3628800

fact 14 = 87178291200

fact 16 = 20922789888000

fact 2 = 2

fact (-4) = Neg_param (-4)

Question 2

(2)

Écrire la(les) fonction(s) suivante(s):

val int_sqrt : int -> int

int_sqrt x calcule la valeur entière de \sqrt{x} . Votre résultat devra vérifier :

$(\text{int_sqrt } x) * (\text{int_sqrt } x) \leq x < (1 + \text{int_sqrt } x) * (1 + \text{int_sqrt } x)$

Pour le calcul de la racine de x on utilisera le principe suivant :

- soit y une approximation supérieure de la racine (on commencera avec $y = x$)
- On sait que $x/y \leq \sqrt{x} \leq y$. On obtient donc une meilleure approximation avec :

$$y' = \frac{(y + x/y)}{2}$$

- On recommence en utilisant y' à la place de y.
- L'algorithme s'arrête lorsque $y \leq y'$.

L'usage de la fonction sqrt (ainsi que tout passage par les float) est strictement interdit, son usage pourra être détecté pendant la correction et entrênera un 0

Exemple 2.1:

```
un_shell> ./question02 0 5
```

Power of 2:

int_sqrt 4 = 2 (should be 2)

```
int_sqrt 16 = 4 (should be 4)
int_sqrt 64 = 8 (should be 8)
int_sqrt 256 = 16 (should be 16)
int_sqrt 1024 = 32 (should be 32)
```

Random tests:

```
int_sqrt 985520 = 992
int_sqrt 370270 = 608
int_sqrt 710610 = 842
int_sqrt 158504 = 398
int_sqrt 335194 = 578
```

OCaml : listes

Question 3

(1)

Écrire la(les) fonction(s) suivante(s):

val digit : int -> int list

digit n construit la liste des *chiffres* (en décimal) représentant le nombre n. Le nombre 0 sera représenté par la liste vide.

Par exemple le nombre digit 123 renverra la liste [1; 2; 3].

Exemple 3.1:

```
un_shell> ./question03 0 5
digit 985520 = [ 9; 8; 5; 5; 2; 0; ]
digit 370270 = [ 3; 7; 0; 2; 7; 0; ]
digit 710610 = [ 7; 1; 0; 6; 1; 0; ]
digit 158504 = [ 1; 5; 8; 5; 0; 4; ]
digit 335194 = [ 3; 3; 5; 1; 9; 4; ]
digit 651659 = [ 6; 5; 1; 6; 5; 9; ]
```

Question 4

(1)

Écrire la(les) fonction(s) suivante(s):

val bin_digit : int -> int list

bin_digit n construit une liste des *chiffres binaires* de n. Le nombre 0 sera représenté par la liste vide.

Par exemple le nombre bin_digit 42 renverra la liste [1; 0; 1; 0; 1; 0] (42 correspond à 101010 en binaire.)

Exemple 4.1:

```
un_shell> ./question04 0 5
```

```

bin_digit 432 = [ 1; 1; 0; 1; 1; 0; 0; 0; 0; ]
bin_digit 606 = [ 1; 0; 0; 1; 0; 1; 1; 1; 1; 0; ]
bin_digit 978 = [ 1; 1; 1; 1; 0; 1; 0; 0; 1; 0; ]
bin_digit 808 = [ 1; 1; 0; 0; 1; 0; 1; 0; 0; 0; ]
bin_digit 346 = [ 1; 0; 1; 0; 1; 1; 0; 1; 0; ]
bin_digit 395 = [ 1; 1; 0; 0; 0; 1; 0; 1; 1; ]

```

Question 5

(3)

Écrire la(les) fonction(s) suivante(s):

val join : ('a * 'b) list -> ('b * 'c) list -> ('a * 'b * 'c) list

join l1 l2 réalise la *jointure* entre les deux listes : le résultat est une liste dont les éléments ont la forme (a,b,c) et tel qu'il existe dans l1 un couple (a,b) et dans l2 il existe un couple (b,c). L'ordre de la liste résultat n'a aucune importance (le programme de test fourni trie le résultat.)

En d'autre terme, la fonction joint toutes les paires avec un élément commun. **Attention**, il est important de comparer toutes les paires de la première liste avec toutes les paires de la seconde et de renvoyer tous les triplés correspondants.

Exemple 5.1:

```

un_shell> ./question05 0 10
l1 = [ (14,12); (03,24); (30,23); (23,06); (01,24);
      (17,10); (18,20); (11,26); (08,18); (30,16); ]
l2 = [ (03,04); (13,25); (28,00); (16,07); (31,06);
      (02,25); (01,14); (04,26); (09,26); (12,13); ]
join l1 l2 = [ (14,12,13); (30,16,07); ]

```

Question 6

(2)

Écrire la(les) fonction(s) suivante(s):

val product : 'a list -> 'b list -> ('a * 'b) list

product l1 l2 renvoie le produit cartésien des deux listes : $\{(a,b) | \forall a \in l1 \text{ et } \forall b \in l2\}$.

Exemple 6.1:

```

un_shell> ./question06 0 3
l1 = [ 0432; 0606; 0978; ]
l2 = [ 0346; 0395; 0808; ]
product l1 l2 =
  [ (0432,0346); (0432,0395); (0432,0808); (0606,0346);
    (0606,0395); (0606,0808); (0978,0346); (0978,0395);
    (0978,0808); ]

```

Question 7

(2)

Écrire la(les) fonction(s) suivante(s):

```
val pipe : 'a -> ('a -> 'a) list -> 'a
```

pipe x [f0; f1; ...; fn] renvoie (fn (... (f1 (f0 x))))

Exemple 7.1:

```
un_shell> ./question07 42 10
s = "rRGddY RsV"
pipe s [lowercase; capitalize] = "Rrgddy rsv"
pipe s [uppercase; uncapitalize] = "rRGDDY RSV"
pipe s [rot47] = "C#v55* #D'"
pipe s [rot47; rot47] = "rRGddY RsV"
```

OCaml : types avancés

Question 8

(2)

Écrire la(les) fonction(s) suivante(s):

```
val check : 'a mutable_list -> bool
```

check l vérifie la cohérence entre la taille stockée dans la liste l (dans le champ size et sa taille réelle (obtenue en parcourant la liste.)

Nous travaillerons avec un type de listes modifiables *maison* que voici :

```
type 'a inner = {
  mutable content : 'a;
  next           : 'a inner option;
}
```

```
type 'a mutable_list = {
  mutable size : int;
  mutable inner : 'a inner option;
}
```

Le type 'a option est défini (dans la *core library* d'OCaml) comme :

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

Exemple 8.1:

```
un_shell> ./question08 0 5
l = (size=5)[ 62; 36; 54; 90; 52; ]
check l = OK
l = (size=6)[ 62; 36; 54; 90; 52; ]
```

```
check l = K0
l = (size=1)[ ]
check l = K0
```

Question 9

(2)

Écrire la(les) fonction(s) suivante(s):

val update : ('a -> 'a) -> 'a mutable_list -> unit

update f l remplace tous les éléments de la liste l par leur image via la fonction f. On utilisera le même type que pour la question précédente.

Exemple 9.1:

```
un_shell> ./question09 0 5
l = (size=5)[ 02; 06; 04; 00; 02; ]
f x = x*x
update f l
l = (size=5)[ 04; 36; 16; 00; 04; ]
```

Question 10

(2)

Écrire la(les) fonction(s) suivante(s):

val bin_search : 'a -> 'a array -> bool

bin_search x tab cherche x dans le tableau trié tab.

Remarque 1:

Le tableau étant trié, on attend de votre fonction de bonnes performances (au moins pour les tests avancés.) Il est donc fortement conseillé d'utiliser une technique de type dichotomie. La correction utilisera des tableaux beaucoup plus grands.

Exemple 10.1:

```
un_shell> ./question10 0 10 90
tab = [| 03; 22; 36; 52; 54; 57; 62; 68; 90; 94; |]
bin_search 90 tab = found
```

Question 11

(3)

Écrire la(les) fonction(s) suivante(s):

val escape : Complex.t -> int -> bool

escape c borne renvoie vrai si la suite complexe de Mandelbrot reste convergente en n itération et faux sinon. La suite complexe de Mandelbrot est définie de la manière suivante :

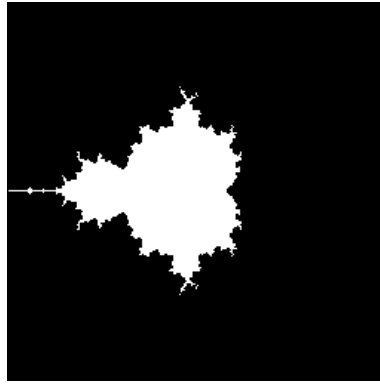


FIGURE 1 – Affichage pour la question 11

$$z_0 = c \quad \text{avec } c \text{ le paramètre de la fonction}$$

$$z_n = z_{n-1}^2 + z_0$$

On sait que la suite ne diverge pas tant que $|z_n| \leq 2$ (le module du terme z_n reste inférieur à 2, en pratique on vérifie plutôt $|z_n|^2 \leq 4$ pour éviter un calcul de racine carrée.) Par conséquent, on peut faire une approximation de la convergence de la suite en calculant les termes z_n jusqu'à z_{borne} tant qu'ils vérifient la propriété.

On utilisera le module Complex d'OCaml. Dont voici un extrait qui pourrait vous servir :

```
type t = { re: float; im: float }
(** The type of complex numbers. [re] is the real part and
    [im] the imaginary part. *)

val zero: t
(** The complex number [0]. *)

val add: t -> t -> t
(** Addition *)

val mul: t -> t -> t
(** Multiplication *)

val norm2: t -> float
(** Norm squared: given [x + i.y], returns [x^2 + y^2]. *)

val norm: t -> float
(** Norm: given [x + i.y], returns [sqrt(x^2 + y^2)]. *)
```

Exemple 11.1:

```
un_shell> ./question11 0 5
(-0.53125,0.75) : converge
```

```
(-1.375,1.28125) : diverge
(0.171875,-0.59375) : converge
(0.28125,1.8125) : diverge
(0.265625,0.15625) : converge
```

On peut aussi appeler la fonction avec un troisième paramètre (peut importe sa valeur), dans ce cas le programme de test passe en mode graphique, i.e. il va afficher en noir tous les points qui renvoie faux. Par exemple, la commande suivante donnera l'affichage de la figure 1 :

```
un_shell> ./question11 0 12 draw
```

Question 12

(4)

Écrire la(les) fonction(s) suivante(s):

```
val normalize : peano -> peano
val add : peano -> peano -> peano
```

`normalize p` normalise un entier de *Peano* : un entier normalisé est : soit 0, soit composé uniquement du même constructeur (successeur ou prédécesseur.)

`add p1 p2` renvoie l'addition des deux entiers de Peano `p1` et `p2`. **Attention**, vous devrez respecter les équations de récursion fournies plus loin (sinon, vous n'aurez pas les bons résultats.)

Un entier de Peano est décrit comme étant soit 0, soit le successeur d'un entier de Peano, soit le prédécesseur d'un entier de Peano. Le type OCaml correspondant est :

```
type peano =
  | Zero
  | Succ of peano
  | Pred of peano
```

L'addition est définie de la manière suivante :

$$\begin{aligned}
 0 + p &= p \\
 p + 0 &= p \\
 Succ(p) + Succ(p') &= p + Succ(Succ(p')) \\
 Pred(p) + Pred(p') &= p + Pred(Pred(p')) \\
 Succ(p) + Pred(p') &= p + p' \\
 Pred(p) + Succ(p') &= p + p'
 \end{aligned}$$

Exemple 12.1:

```
un_shell> ./question12 0 7
```

```
p =
  Pred ( Succ ( Succ ( Pred ( Pred ( Succ ( Pred ( Zero))))))
    )
)
```



```
normalize p = Pred ( Zero)
p1 = Pred ( Succ ( Pred ( Zero)))
p2 = Pred ( Succ ( Pred ( Zero)))
p1 + p2 = Pred ( Pred ( Succ ( Pred ( Zero))))
```

Remarques sur la session exam

À la fin de l'épreuve, vous devez quitter votre session en fermant l'horloge (et uniquement en fermant celle-ci.) Dans tous les cas, votre session sera également fermée à la fin du temps imparti.

Lorsque l'horloge est fermée (volontairement, ou en fin de session), la console qui vous a demandé le token vous demande votre mot de passe. **Vous devez rentrer votre mot de passe UNIX pour que le rendu ai bien lieu.**

Vous pouvez aussi rendre sans quitter : il y a une commande (dans le shell) appelée `rendu` qui déclenche un rendu comme si vous quittiez (demande de mot de passe et envoi du rendu.)

Après la fin du test (dans les minutes qui suivent) vous pouvez redémarrer votre machine et vous connecter pour re-rendre (si par exemple il y a eu une erreur pendant votre premier rendu.)

Remarques sur les squelettes de questions

Pour chaque question, un embryon de code vous est fourni. Ce code correspond au strict minimum pour que la question compile et que l'appel au programme de test échoue avec une erreur identifiable. Par conséquent, vous devez supprimer du corps des fonctions à écrire, le code provoquant l'erreur, sous peine de voir votre programme échouer lors des tests (ne laissez surtout pas la ligne `assert false`, même si elle n'est plus dans la fonction, elle sera probablement exécutée quand même.)

Exemple 1:

À titre d'exemple, si l'on vous demande la fonction OCaml suivante :

```
val identity: 'a -> 'a
```

identity x renvoi x.

Vous trouverez dans le fichier de question correspondant le code :

```
let identity _ =  
  (* FIX ME *)  
  assert false
```

Que vous devrez remplacer par :

```
let identity x = x
```

Remarques sur les programmes de test

La commande `make questionXX` produira un binaire `questionXX`. 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 succincte s'il sont appelés avec l'option `-help`.

Exemple 2:

Le binaire produit pour la question 1 (il s'agit d'un exemple qui ne correspond pas forcément au sujet)

Question 1:

```
./question01 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** (présent pour chaque question, ou presque) font tous référence à l'initialisation d'un générateur de nombre aléatoire. Dans le cas présent la commande `./question03 X Y` va initialiser le générateur de nombre aléatoire avec X (ici, Y servant de taille à la liste générée.) Pour les mêmes valeurs de X on obtiendra les mêmes données (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 jeter un oeil sur fichier contenant les tests de la question X (`test_qXX.ml`.)

Listes des fichiers à rendre

À rendre sans modifications

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 originale :

Makefile
question01.mli
question02.mli
question03.mli
question04.mli
question05.mli
question06.mli
question07.mli
question08.mli
question09.mli
question10.mli
question11.mli
question12.mli
test_frame.ml
test_q01.ml
test_q02.ml
test_q03.ml
test_q04.ml
test_q05.ml
test_q06.ml
test_q07.ml
test_q08.ml
test_q09.ml
test_q10.ml
test_q11.ml
test_q12.ml

Fichiers de réponses

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.)

question01.ml
question02.ml
question03.ml
question04.ml
question05.ml
question06.ml
question07.ml
question08.ml
question09.ml
question10.ml
question11.ml
question12.ml