EPITA SPE promo 2016
Programmation - Épreuve machine

Marwan Burelle *

Vendredi 21 décembre 2012

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. Dans le répertoire* `sujet` *vous trouverez également deux scripts* `startemacs` *et* `startemacs-nw` *qui permettent de démarrer* `emacs` *avec le mode* `tuareg` *chargé.*

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 décrivants :

- *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

Question 1 (1)

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

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

`fact n` calcule $n!$. Pour n positif ou nul (aucune valeur négative ne sera passée à votre fonction.)

On rappelle la formule de factorielle :

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

Exemple 1.1:

```
un_shell> ./question01 0 5
fact 7 = 5040
fact 0 = 1
fact 12 = 479001600
fact 10 = 3628800
fact 8 = 40320
```

Question 2 (1)

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

```
val reduction : int -> int
```

`reduction n` calcule la réduction décimale (pour la preuve par 9) de n. La réduction décimale se calcule en effectuant la somme de tous les chiffres du nombre n, puis en recommencant avec le résultat obtenu, jusqu'à ce que le résultat n'est plus qu'un seul chiffre.

Exemple 2.1:

```
un_shell> ./question02 0 3
let proofby9 a b c =
  reduction (reduction a * reduction b) = reduction c

proofby9 1630 2480 4042400 = true (should be true)
proofby9 1630 2480 4042401 = false (should be false)

proofby9 2856 6098 17415888 = true (should be true)
proofby9 2856 6098 17415889 = false (should be false)

proofby9 4491 7514 33745374 = true (should be true)
proofby9 4491 7514 33745375 = false (should be false)
```

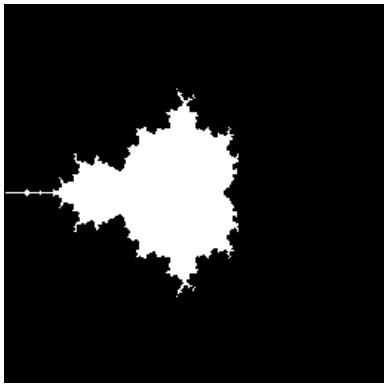

FIGURE 1 – Affichage pour la question 3

Question 3 (2)

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

```
val escape : float * float -> int -> bool
```

`escape (x0,y0) 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 :

$$z_0 = \texttt{x0} + i \times \texttt{y0}$$
$$z_n = z_{n-1}^2 + z_0$$

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 $z_{\texttt{borne}}$ tant qu'ils vérifient la propriété.

On rappelle :

$$(a + i \times b)^2 = a^2 - b^2 + i \times 2 \times a \times b$$
$$|a + i \times b|^2 = a^2 + b^2$$

Exemple 3.1:

```
un_shell> ./question03 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 appeller 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> ./question03 0 12 draw
```

Question 4 (1)

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

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

`rotate n l` fait *tourner* de *n* places la liste `l`. À chaque tour, l'élément en tête se retrouve en dernière position. On garantie que *n* est toujours strictement inférieur à la taille de la liste.

Exemple 4.1:

```
un_shell> ./question04 0 5
l = [ 62; 36; 54; 90; 52; ]
rotate 3 l = [ 90; 52; 62; 36; 54; ]
```

Question 5 (2)

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

```
val partition : 'a array -> int -> int -> int -> int
```

`partition tab left right pivotIndex` effectue une *partition* du tableau `tab` entre les bornes `left` et `right` vis à vis de la valeur se trouvant à la case `pivotIndex` et renvoie la nouvelle position du pivot. La partition consiste à regrouper l'ensemble des valeurs inférieures d'un côté et l'ensemble des valeurs supérieures de l'autre.

Vous devrez implémenter un algorithme sur le même modèle que celui-ci :

```
algorithme fonction partition : entier
  parametres locaux
    t_vect_entiers    tab
    entier            left, right, pivotIndex
  variables
    entier            i, pivot, storeIndex, tmp
debut
  pivot ← tab[pivotIndex]
  tab[pivotIndex] ← tab[right]
  tab[right] ← pivot
  storeIndex ← left
  pour i ← left jusqu'a (right - 1) faire
    si (tab[i] < pivot) alors
      tmp ← tab[i]
      tab[i] ← tab[storeIndex]
      tab[storeIndex] ← tmp
      storeIndex ← (storeIndex + 1)
    fin si
    tab[right] ← tab[storeIndex]
    tab[storeIndex] ← pivot
    retourne (storeIndex)
  fin pour
fin algorithme fonction partition
```

Cette fonction sera utilisée pour un algorithme de tri (code fourni.)

Exemple 5.1:

```
un_shell> ./question05 0 5
Before sort:
-------------------------------
| 752 | 190 | 154 | 936 | 162 |
-------------------------------
After sort:
-------------------------------
| 154 | 162 | 190 | 752 | 936 |
-------------------------------
```

Question 6 (2)

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

```
val find : int -> Node.vnode option -> int option
```

`find x t` cherche la clef *x* dans l'arbre binaire de recherche `t` et renvoie la valeur associée sous forme de type `option`.

On rappelle qu'un arbre binaire de recherche respecte une relation d'ordre, l'ensemble des clefs du sous-arbre gauche sont inférieures à la clef de la racine et l'ensemble des clefs du sous-arbre droit sont supérieures à la clef de la racine.

Les arbres sont implémentés en utilisant le type suivant :

```
class type node :
  object
    method get_key : int
    method get_value : int
    method left : vnode option
    method right : vnode option
  end
```

Il faut donc passer par les différentes méthodes fournies (`t#get_key` pour la obtenir la clef par exemple.)

Attention, on fourni à votre fonction une valeur de type option : si l'arbre est vide vous recevrez `None` (et `Some t` sinon.)

L'utilisation de la méthode `get_key` est compatibilisé, pour estimer le nombre de comparaisons effectuées.

Exemple 6.1:

```
n_shell> ./question06 0 3 10

Node: (1091,10)
  Node: (463,201)
```

```
    Node: (288,681)
      Leaf: (31,628)
      Leaf: (327,161)
    Node: (692,740)
      Leaf: (512,398)
      Leaf: (1027,195)
  Node: (1399,479)
    Node: (1231,535)
      Leaf: (1202,673)
      Leaf: (1344,657)
    Node: (1768,322)
      Leaf: (1498,603)
      Leaf: (2040,936)

key not found
key access: 4
un_shell> ./question06 0 3 1768

Node: (1091,10)
  Node: (463,201)
    Node: (288,681)
      Leaf: (31,628)
      Leaf: (327,161)
    Node: (692,740)
      Leaf: (512,398)
      Leaf: (1027,195)
  Node: (1399,479)
    Node: (1231,535)
      Leaf: (1202,673)
      Leaf: (1344,657)
    Node: (1768,322)
      Leaf: (1498,603)
      Leaf: (2040,936)

found: 322
key access: 3
```

Question 7 (2)

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

```
type peano = Zero | Succ of peano | Pred of peano
val add : peano -> peano -> peano
```

`add a b` calcule la somme de `a` et `b` dans l'arithmétique de Peano. Dans l'arithmétique de Peano, il n'existe qu'une constante : 0 et les opération `Succ` (successeur) et `Pred` (prédécesseur.)

On par exemple les correspondances suivantes entre les entiers de Peano et nos entiers habituels :

`Zero`	=	0
`Succ(Zero)`	=	1
`Succ(Pred(Succ(Zero)))`	=	1

Il vous faut donc implémenter l'addition sur ce modèle sans bien sûr repasser par les entiers.

Le résultat final de tous vos calculs sera affiché après avoir été *normalisé*, c'est à dire ramené à une forme ne contenant plus que des `Succ` ou des `Pred` afin que l'ordre de vos opérations dans l'algo ne perturbe pas le résultat.

Exemple 7.1:

```
un_shell> ./question07 0 5 2
a =
  Succ(Succ(Succ(Pred(Succ(Zero)))))
b =
  Pred(Succ(Pred(Succ(Zero))))
a + b =
  Succ(Succ(Succ(Zero)))

a =
  Pred(Succ(Succ(Succ(Succ(Zero)))))
b =
  Succ(Pred(Pred(Pred(Succ(Zero)))))
a + b =
  Succ(Succ(Zero))
```

Question 8 (4)

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

```
val eval : register -> instr array -> unit
```

`eval reg insts` évalue l'instruction courante dans le tableau d'instructions `insts`. Le numéro de l'instruction est déterminée par la valeur du registre `pc` (que l'on trouve dans `reg`.) Après l'évaluation, le registre `pc` contient le numéro de la prochaine instruction (la suivante sauf pour les branchements et la fin.)

Les instructions (de type `instr`) forme un *mini-assembleur* à registre pure. Nous disposons d'un ensemble de registre de données et de trois registres particuliers. Il n'y a ni mémoire, ni pile. Les registres sont représentés par un *record* de type `register` décrit par le type suivant :

```
type register = {
  mutable pc : int;
  mutable count : int;
  data : int array;
  mutable result : int;
}
```

Le champ `pc` est le compteur de programme (le numéro de l'instruction à exécuter), le champ `count` est un registre servant de compteur, le champ `result` sert à stocker le résultat à la fin de l'exécution et enfin le champ `data` est un tableau représentant les registres de données (à la case `i` on trouve le registre `i`.)

Les instructions sont décrites par le type `instr` et le tableau qui suit explique leur sémantique :

```
type instr =
    LOAD of int * int
  | MOVE of int * int
  | SET of int
  | INCR
  | DECR
  | JMP of int
  | BCZR of int
  | BRZR of int * int
  | ADD of int * int * int
  | MIN of int * int * int
  | MUL of int * int * int
  | DIV of int * int * int
  | RET of int
```

Manipulations des registres	
`LOAD(i,r)`	charge la valeur immédiate `i` dans le registre `r`
`MOVE(r1,r2)`	copie la valeur du registre `r1` vers le registre `r2`
Manipulations du compteurs	
`SET(r)`	charge la valeur du registre `r` dans le compteur (`count`)
`INCR`	incrémente le compteur
`INCR`	décrémente le compteur
Sauts	
`JMP(a)`	saute à l'instruction numéro `a`
`BCZR(a)`	si le compteur est à zéro, saute à l'instruction numéro `a`
`BRZR(r,a)`	si le registre `r` est à zéro, saute à l'instruction numéro `a`
Instructions arithmétiques	
`ADD(r1,r2,r3)`	le registre `r3` reçoit `r1 + r2`
`MIN(r1,r2,r3)`	le registre `r3` reçoit `r1 - r2`
`MUL(r1,r2,r3)`	le registre `r3` reçoit `r1 * r2`
`DIV(r1,r2,r3)`	le registre `r3` reçoit `r1 / r2`
Sortie du programme	
`RET(r)`	met la valeur du registre `r` dans le registre `result` et passe le registre `pc` à `-1`

Au départ les registres sont tous initialisés à zéro et les instructions seront évaluées par la fonction :

```
let exec inst =
  let reg =
    {
      pc = 0;
      count = 0;
      data = Array.make 16 0;
      result = 0;
    }
  in
    while reg.pc >= 0 do
      eval reg inst;
    done;
    reg
```

Exemple 8.1:

```
> ./question08 0
START:
00 LOAD 3 d0;
01 BRZR d0 12;
02 LOAD 1 d1;
03 LOAD 1 d2;
```

```
04 MIN d0 d2 d0;
05 SET d0;
06 ADD d0 d2 d0;
07 BCZR 12;
08 MUL d0 d1 d1;
09 MIN d0 d2 d0;
10 DECR
11 JMP 7;
12 RET d1;
END
EXECUTION ... result = 6
```

Question 9 (2)

Compléter la définition du foncteur suivants:

```
module MaxBox (T:OrderedType) =
struct
  type content = T.t
  type t = {
    mutable content : content;
  }
  let create v = { content = v; }
  let update box v = (* FIX ME *)
  let get box = box.content
end
```

La fonction du foncteur `update box v` devra mettre à jour le contenu du `box`, uniquement si la valeur `v` est supérieure ou égale que la valeur actuellement contenue dans `box`. Attention, vous devrez utiliser la fonction de comparaison `T.compare a b` qui retourne 0 si `a` est égal à que `b`, un nombre positif si `a` est plus grand et un nombre négatif sinon.

Votre foncteur sera utilisé pour extraire le maximum d'une liste avec comparaison *non-standard* (en l'occurrence, pendant l'exam, une comparaison sur les floattants avec une marge de tolérance pour l'égalité.)

Exemple 9.1:

```
un_shell> ./question09 0 5
l = [ 12.1857; 1.73154; 6.49775; 9.4427; 12.7394; ]
max l = 12.7394
```

Question 10 (2)

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

```
val simplify : bw_img -> bw_img
```

Nous allons travailler sur des images représentées sous forme d'un arbres quaternaires : elles sont soit blanches, soit noires, soit composées de quatre images (sous forme d'arbre également.)

```
type bw_img =
  | White
  | Black
  | Composed of bw_img * bw_img * bw_img * bw_img
```

Comme nos images sont générées aléatoirement, elles peuvent parfois être de couleur unie (toute blanche ou toute noire) et pourtant être composées. C'est pourquoi vous devez écrire la fonction de simplification `simplify img` qui renvoie une nouvelle image telle que seules les sous-images qui ne sont pas de couleur unie soient construites par composition.

Exemple 10.1:

```
un_shell> ./question10 0 2
img =
  Composed(Composed(White, White, White, White),
    Composed(White, White, White, White),
    Composed(White, White, White, White),
    Composed(White, White, White, White)
  )
simplify img = White
un_shell> ./question10 0 3
img =
  Composed(
    Composed(Composed(White, White, White, White),
      Composed(Black, Black, Black, Black),
      Composed(White, White, White, White),
      Composed(Black, Black, Black, Black)
    ),
    Composed(Composed(White, White, White, White),
      Composed(White, White, White, White),
      Composed(Black, Black, Black, Black),
      Composed(White, White, White, White)
    ), Composed(White, White, White, White),
    Composed(White, White, White, White)
  )
simplify img =
  Composed(Composed(White, Black, White, Black),
    Composed(White, White, Black, White), White, White
  )
```

Question 11 (3)

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

```
class treneau :
  object
    method comet : float -> unit
    method dasher : float -> unit
    method iter : float -> int -> unit
    method rudolph : int -> unit
  end
```

Définir la manquante méthode `iter` (les autres sont obtenus par la classe `Noel.noel` (fournie.)

L'objectif de cette méthode est d'aider *Rudolph* à guider le traîneau du père Noël. L'idée est simple : on donne à la méthode `iter` une distance à parcourir (en floattant) et un nombre de détours à faire :

- S'il n'y a pas de détours (`t#iter f 0`) il faut appeller le message `comet` avec la distance à parcourir `f` (*Comet* est chargé des lignes droites.)
- S'il y a `n` détours (`t#iter f n`), il faut faire un tier du trajet avec un détour de moins (`f/.3.` et `n-1`), puis demander un virage simple positif (`1.`) avec le message `dasher`, effectuer un tier du trajet avec un détours de moins, puis un virage double négatif (`-2.`) avec `dasher`, effectuer un autre tier du trajet avec un détours de moins, puis un virage simple positif (`1.`) avec `dasher` et enfin effectuer encore un tier du trajet (`f/.3.`) avec un détours de moins (`n-1`).

Au final, pour chaque détour, il faudra faire deux virages simples positifs, un virage double négatif et quatre appels récursifs avec chacun un tier de la distance et un détour de moins.

L'affichage renverra la distance au totale.

Question 12 (3)

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

```
val rot47 : string -> unit
```

On a retrouvé un vieux manuscrit Maya, malheureusement, celui est chiffré à l'aide d'un algorithme de connu sous le nom de `rot47`. Le principe est simple, chaque caractère est représenté par un nombre de 0 à 127, parmi ces caractères, seules les 94 caractères de 33 à 126 sont affichables. L'algorithme consiste à effectuer une rotation de 47 places du numéro du caractère si celui-ci est affichable. Cette rotation est directement réversible puisqu'il suffit de refaire une rotation pour obtenir la valeur originale.

À titre d'exemple `'A'` (de code ASCII 65) se transforme en `'p'` (de code ASCII 112.) Tandis que `'a'` (97) devient `'2'` (50, attention, il s'agit bien d'une rotation !) Pour effectuer ces calculs vous avez besoin des fonctions du module `Char` :

```
val Char.code: char -> int
val Char.chr:  int -> char
```

Votre fonction devra effectuer la rotation en place (et donc modifier la chaîne en paramètre.)

Si votre algo marche, vous obtiendrez la formule Maya !

Exemple 12.1:

```
un_shell> ./question12 0 5
Basic test:
  s = "A Simple Test ..."
  rot47 s
  s = "p $:>A=6 %6DE ]]]"
  rot47 s
  s = "A Simple Test ..."

Random test:
  s = "Mu/9;"
  rot47 s
  s = "|F^hj"
  rot47 s
  s = "Mu/9;"
```

Si les tests sont concluants, le programme affichera également le message maya original !

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) appellée `rendu` qui déclenche un rendu comme si vous quittiez (demande de mot de passe et envoie du rendu.) **Il est fortement recommandé d'utiliser cette commande avant la fin de l'épreuve pour être sûr d'avoir un vrai rendu !**

Que faire si le mot de passe ne marche pas ?

- S'assurer que la fenêtre a bien le *focus* (mettre le pointeur de la souris dessus.)
- Il n'y a pas d'écho du mot passe lorsque vous le tapez et c'est normal : taper le correctement, puis appuyiez sur *return*. Si vous avez des doutes sur ce que vous avez taper, vous pouvez utiliser la touche *backspace*.
- Enfin, si vous avez un message d'erreur, si votre mot de passe n'est pas accepté, si . . . vous pouvez redemarrez la machine, vous connectez et de nouveau reprendre la procédure de rendu au départ (quitter, si le temps n'est pas fini, et rendre.)

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 problamenent 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
```

Pour la partie C, le principe est le même. En plus, les paramètres n'étant pas utilisés dans le code il y a quelques lignes *pour faire comme-ci.* Un petit exemple vaut mieux qu'un long discourts :

Exemple 2:
Si l'on vous demande d'écrire la fonction C suivante :

```
void *identity(void *x);
```

`identity(x)` renvoie le pointeur `x`.
Vous trouverez dans le fichier de question correspondant le code :

```
void *identity(void *x)
{
  x = x;
  /* FIX ME */
  abort();
  return NULL;
}
```

Que vous devrez remplacer par :

```
void *identity(void *x)
{
  return 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 3:
Le binaire produit pour la question 3 fournit l'aide suivante :

```
Question 3:
./question03 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`.)

Bien évidement, les même remarques sont valables pour les questions en C.

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
node.ml
noel.ml
alumrof

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