EPITA SPE promo 2014
Programmation - Épreuve machine

Marwan Burelle*

Vendredi 7 janvier 2011

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 11 questions dans cet examen.

*marwan.burelle@lse.epita.fr

Récursions arithmétiques classiques

Question 1 (1)

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

```
val pgcd : int -> int -> int
```

`pgcd a b` calcule le pgcd des entiers (initialement) strictement positifs `a` et `b`.
On rappelle l'algorithme d'Euclide pour le calcul du pgcd :

$$pgcd(a,b) = \begin{cases} a & \text{si } b = 0 \\ pgcd(b, a \bmod b) & \text{sinon} \end{cases}$$

Exemple 1.1:

```
> ./question01 0 3
pgcd 598 414 = 46
  Verification:
  598 mod 46 = 0
  414 mod 46 = 0

pgcd 804 961 = 1
  Verification:
  804 mod 1 = 0
  961 mod 1 = 0

pgcd 389 338 = 1
  Verification:
  389 mod 1 = 0
  338 mod 1 = 0
```

Question 2 (1)

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

```
val intlog : int -> int -> int
```

`intlog base x` calcule le *logarithme entier* de `x` dans la base `base`.
On rappelle que le logarithme entier dans une base donnée correspond aux nombres de division possible par la base. Il vous suffit donc de diviser `x` par `base` jusqu'à ce que le résultat soit 0 et compter le nombre de division.

Exemple 2.1:

```
> ./question02 0 5
intlog 2 1 = 0 (should be 0)
intlog 3 1 = 0 (should be 0)

intlog 2 16 = 4 (should be 4)
```

```
intlog 3 81 = 4 (should be 4)

intlog 2 8 = 3 (should be 3)
intlog 3 27 = 3 (should be 3)

intlog 2 64 = 6 (should be 6)
intlog 3 729 = 6 (should be 6)

intlog 2 256 = 8 (should be 8)
intlog 3 6561 = 8 (should be 8)
```

Listes

Question 3 (2)

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

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

`fcount f l` compte le nombre d'élément de la liste `l` tel que `f` renvoie vrai. La fonction retourne un couple d'entier dont le premier membre indique le nombre d'élément ayant renvoyé vrai et le second membre indique le nombre d'élément ayant renvoyé faux *(on notera que la somme des éléments du couple doit être égal à la taille de la liste.)*

Exemple 3.1:

```
> ./question03 0 10
l = [ 27; 03; 21; 25; 05; 18; 04; 01; 22; 30; ]
even x = x mod 2 = 0
div3 x = x mod 3 = 0
fcount even l = (4,6) (somme = 10)
fcount div3 l = (5,5) (somme = 10)
```

Question 4 (2)

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

```
val average : int list -> float
```

`average l` renvoie la moyenne des éléments de la liste d'entier `l`. Le résultat doit être un floattant *(attention aux conversions.)*

Exemple 4.1:

```
> ./question04 0 10
l = [ 155; 131; 149; 249; 133; 82; 36; 193; 86; 158; ]
average l = 137.20
```

Question 5 (3)

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

```
val len : 'a llist -> int
val map : ('a -> 'b) -> 'a llist -> 'b llist
val tolist : 'a llist -> 'a list
```

`len l` renvoie la taille de la *lazy liste* `l`.
`map f l` renvoie la liste (structurellement identique à `l`) des résultats de `f` sur les éléments de `l` (comme à la fonction `List.map`.)
`tolist l` convertit la liste `l` (de type `llist`) en une liste OCaml.
Le type `llist` définit des listes similaires aux listes d'OCaml, sauf que la concaténation est un constructeur : au lieu de construire le résultat de la concaténation comme d'habitude, on se contente de regrouper les deux listes dans une paire. D'un point de vue *logique* les deux représentations sont identiques, seule la façon de les parcourir change. Le type des listes est le suivant :

```
type 'a llist =
  | Empty
  | Cons of 'a * 'a llist
  | Concat of 'a llist * 'a llist
```

Exemple 5.1:

```
> ./question05 0 5
l1 = [ 18; 04; 01; 22; 30; ]
l2 = [ 27; 03; 21; 25; 05; ]
l = append l1 l2 =
  [ [18; 04; 01; 22; 30; ] @ [27; 03; 21; 25; 05; ] ]
l' = append l l =
  [ [[18; 04; 01; 22; 30; ] @ [27; 03; 21; 25; 05; ] ]
    @ [[18; 04; 01; 22; 30; ] @ [27; 03; 21; 25; 05; ] ] ]

len l1 = 5
len l2 = 5
len l = 10
len l' = 20

f x = x * 2

map f l' =
[ [[36; 08; 02; 44; 60; ] @ [54; 06; 42; 50; 10; ] ]
  @ [[36; 08; 02; 44; 60; ] @ [54; 06; 42; 50; 10; ] ] ]

tolist l' =
  [ 18; 04; 01; 22; 30; 27; 03; 21; 25; 05; 18; 04; 01; 22;
    30; 27; 03; 21; 25; 05; ]
```

Structures avancées

Question 6 (2)

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

```
val find : 'a -> ('a, 'b) tree -> 'b
```

`find k t` cherche la clef `k` dans l'arbre de `t` et renvoie la valeur associée. `t` est un arbre binaire de recherche (l'ensemble des clefs du sous-arbre gauche est inférieure à l'ensemble des clefs du sous-arbre droit) où chaque nœud contient une clef et une valeur, l'arbre est trié par rapport à ses clefs. Votre fonction devra lever l'exception `Not_found` si la clef n'est pas dans l'arbre.

Le type des arbres binaires de recherche est :

```
type ('a, 'b) tree =
  | Empty
  | Node of ('a, 'b) tree * 'a * 'b * ('a,'b) tree
```

Exemple 6.1:

```
t =
  Node(
    Node(
      Node(Empty,2,1017,Empty),
      4, 389,
      Node(Empty,6,338,Empty)
      ),
    9, 804,
    Node(
      Node(Empty,11,961,Empty),
      13, 598,
      Node(Empty,15,414,Empty)
      )
    )
11 -> 961
```

Question 7 (3)

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

```
val bf : ('a -> 'b) -> (unit -> 'c) -> 'a tree -> unit
```

`bf print sep t` parcours l'arbre binaire de `t` en **largeur** et affiche ses clefs à l'aide de la fonction `print` passée en argument. À chaque **fin** de niveau, votre fonction devra appeler `sep ()` qui affichera *notre marqueur* de fin de niveau (dans les tests un saut de ligne.)

Les arbres binaires sont représentés par le type :

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Remarque 1:
Vous avez besoin du module `Queue` *(file classique) d'OCaml. Pour gérer les changements de niveau, pensez à utiliser le constructeur de l'arbre vide (*`Empty`*) puisque vous n'avez pas de raison d'enfiler les arbres vides.*

Exemple 7.1:

```
> ./question07 0 3
t =
  Node(
    Node(
      Node(Empty,37,Empty),
      05,
      Node(Empty,78,Empty)
      ),
    76,
    Node(
      Node(Empty,33,Empty),
      54,
      Node(Empty,90,Empty)
      )
    )

bf print sep =
   76;
   05; 54;
   37; 78; 33; 90;
```

OCaml avancée

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 (3)

Écrire la classe suivante:

```
exception Stack_full
exception Stack_empty
class ['a] stack : int -> 'a ->
object
  method is_empty : bool
  method pop : 'a
  method push : 'a -> unit
```

```
  method size : int
  method top : 'a
end
```

Les objets obtenus par `new stack max init` représentent des piles de taille maximale `max`. La valeur `init` sert à *pré-remplir* la pile.

Les méthodes à fournir sont les suivantes :

- `is_empty` renvoie vrai si la pile est vide
- `size` renvoie la taille de la pile (nombre d'éléments dans la pile.)
- `push x` empile `x` sur la pile. Lève l'exception `Stack_full` s'il n'y a plus de place sur la pile.
- `pop` dépile et renvoie l'élément au sommet de la pile. Lève l'exception `Stack_empty` si la pile est vide.
- `top` renvoie (sans le dépiler) l'élément au sommet de la pile. Lève l'exception `Stack_empty` si la pile est vide.

En interne, la pile est représentée par tableau (d'où la nécessité d'une valeur d'initialisation) dans l'attribut `tab` et la prochaine case disponible est indiquée par `sp` (donc, le sommet se trouve en `sp-1`.)

Exemple 9.1:

```
> ./question09 0 5
Test de is_empty: empty
Test de size: 0
Test de push:
  OK pushed 90
  OK pushed 54
  OK pushed 33
  OK pushed 76
  OK pushed 78
  OK stack full.
Test de is_empty: not empty
Test de size: 5
Test de top: 78
Test de pop:
  OK pop 78
  OK pop 76
  OK pop 33
  OK pop 54
  OK pop 90
  OK stack empty.
Test de is_empty: empty
Test de size: 0
```

Question 10 (3)

Compléter la définition du foncteur suivants:

```
module type S =
  sig
    type elt
    type t
    val empty : unit -> t
    val is_empty : t -> bool
    val size : t -> int
    val push : elt -> t -> unit
    val pop : t -> elt
  end
module PriorityQueue :
  functor (T : OrderedType) -> S with type elt = T.t
```

Le foncteur `PriorityQueue` produit un module contenant la définition d'une file de priorité. Une file de priorité est une file où l'ordre n'est pas l'ordre d'arrivée, mais l'ordre des éléments ajoutés à la file défini par la fonction `T.compare`. Les opérations à fournir sont les suivantes :

- `empty ()` renvoie une nouvelle file vide
- `is_empty q` renvoie vrai si la file `q` est vide
- `size q` renvoie la taille de la file
- `push x q` ajoute l'élément `x` à la file `q` (les doublons sont possibles et conservés, ce n'est pas un ensemble !)
- `pop q` défile et renvoie le plus petit élément de la file.

Une file est un *record* contenant la taille (`s`) de la file et une liste (`q`) représentant la file. Le type est le suivant :

```
type elt = T.t
type t =
    {
      mutable q : elt list;
      mutable s : int;
    }
```

Exemple 10.1:

```
> ./question10 0 5
Test de is_empty: empty
Test de size: 0
Test de push:
  OK pushed gkz
  OK pushed sgf
  OK pushed vnb
  OK pushed tmt
  OK pushed twd
Test de is_empty: not empty
Test de size: 5
```

```
Test de pop:
  OK poped gkz
  OK poped sgf
  OK poped tmt
  OK poped twd
  OK poped vnb
  OK queue is empty.
Test de is_empty: empty
Test de size: 0
```

Question 11 (1)

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

```
val transtring : string -> unit
```

`transtring s` transforme (en place) la chaîne `s` en effectuant un *rot13* sur les lettres minuscules.

rot13 est une rotation des lettres de 13 positions dans l'alphabet, seules les lettres minuscules (de `'a'` à `'z'`) sont modifiées, les autres restent inchangées. On notera que *rot13* est symétrique : `s = transtring (transtring s)`. Pour calculer le décalage, il faut utiliser le code ASCII du caractère à décaler (obtenue par `Char.code`) et celui de la lettre `'a'`. Vous aurez également besoin de la conversion dans l'autre sens : `Char.chr` qui renvoie le caractère correspondant à son code (entre autre on a `c = Char.chr (Char.code c)`.

Exemple 11.1:

```
> ./question11 0 5
transtring "gkzsgfvnbt" =
  "txmftsiaog"
transtring "txmftsiaog" =
  "gkzsgfvnbt"

transtring "mttwdtpdlw" =
  "zggjqgcqyj"
transtring "zggjqgcqyj" =
  "mttwdtpdlw"

transtring "ayudsadnrp" =
  "nlhqfnqaec"
transtring "nlhqfnqaec" =
  "ayudsadnrp"

transtring "xjwvsruzyy" =
  "kwjifehmll"
transtring "kwjifehmll" =
```

```
  "xjwvsruzyy"

transtring "jtwafljiim" =
  "wgjnsywvvz"
transtring "wgjnsywvvz" =
  "jtwafljiim"

transtring "obaar naarr !" =
(* vous aurez la réponse à celui-là si votre code marche *)
```

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.)

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
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

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