EPITA SPE promo 2014
Programmation - Épreuve machine

Marwan Burelle*

Vendredi 12 novembre 2010

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 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* **et** *sanctionnée (perte de points.)*

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. 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 méthode de compilation pour la correction sera celle décrite dans le `Makefile`*, par conséquent vous n'aurez les points à la question XX que si la commande "*`make questionXX`*" réussit* ***et que le résultat est correct****.*

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

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 fact : int -> int
```

`fact n` renvoie le résultat de factorielle de `n` si `n` est positif ou nul et `-1` sinon.

Exemple 1.1:

```
> ./question01 0 5
fact -3 = -1
fact 3 = 6
fact 8 = 40320
fact 11 = 39916800
fact 1 = 1
```

Question 2 (1)

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

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

`sigma f n` renvoie la somme $\sum_{i=1}^{n}(f\ i)$ (la somme des termes de `f` de 1 à `n`.) Le comportement de la fonction n'est pas défini si `n` est négatif. De plus, on notera que `sigma f 0` renvoie `0` quelque soit `f`.

Exemple 2.1:

```
> ./question02 0 5
carre x = x*x
uar (n) = 6 + uar (n-1)
uar (0) = 6
ugeom (n) = 4 * ugeom (n-1)
ugeom (0) = 1

sigma carre 5 = 55
sigma uar 5 = 120
sigma ugeom 5 = 1364
```

Listes

Question 3 (1)

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

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

`cond_map cond f l` applique la fonction `f` aux éléments de `l` pour lesquels la fonction `cond` renvoie vrai et renvoie la liste de ces résultats. La liste résultat est donc forcément plus petite (ou égale) à la liste `l`.

Exemple 3.1:

```
> ./question03 0 5
l = [ 18; 04; 01; 22; 30; ]
cond x = x est un carre parfait
f x = sqrt (float x)

cond_map cond f l= [ 2.0; 1.0; ]
```

Question 4 (2)

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

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

`filter_our cond l` renvoie la liste des éléments de `l` pour lesquels la fonction `cond` renvoie faux.

Exemple 4.1:

```
> ./question04 0 5
l = [ 18; 04; 01; 22; 30; ]
f1 x = x mod 2 = 0
f2 x = x mod 3 = 0
f3 x = x est un carre parfait
filter_out f1 l = [ 01; ]
filter_out f2 l = [ 04; 01; 22; ]
filter_out f3 l = [ 18; 22; 30; ]
```

Question 5 (3)

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

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

`join l1 l2` renvoie la liste des couples `(a,c)` tel qu'il existe dans `l1` un couple `(a,b)` et dans `l2` un couple `(b,c)`. L'ordre de la liste résultat n'a pas d'importance (la liste est triée avant affichage.)

Attention, pour chaque couple `(a,b)` dans `l1`, vous devez trouver tous les couples `(b,c)` dans `l2`.

Exemple 5.1:

```
> ./question05 42 6
l1 =
[ (10,11); (11,09); (13,04); (22,10); (31,08); (31,31); ]
l2 =
[ (08,23); (09,04); (11,22); (17,21); (21,00); (28,22); ]
join l1 l2= [ (10,22); (11,04); (31,23); ]
```

Structures avancées

Question 6 (2)

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

```
val find : 'a -> 'a tree -> bool
```

`find x t` cherche la clef `x` dans l'arbre général `t`.

Les arbres généraux sont représentés par le type (les nœuds sont composés d'une clef et de la liste des fils) :

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

Exemple 6.1:

```
> ./question06 0 3 0
t =
  (06(11(06(04)(12)(08))(07(09)(03)))
    (04(11(03)(05))(05(02))))

You didn't find 00 in t.
> ./question06 0 3 2
t =
  (06(11(06(04)(12)(08))(07(09)(03)))
    (04(11(03)(05))(05(02))))

You found 02 in t.
```

Remarque 1:

Pour visualiser vos arbres de manière plus graphique*, vous avez la possibilité d'utiliser le fichier* `.dot` *produit lors de l'éxecution de la question. Le plus simple est de suivre l'exemple suivant :*

```
> ./question06 0 3 2
t =
  (06(11(06(04)(12)(08))(07(09)(03)))
    (04(11(03)(05))(05(02))))

You found 02 in t.
> make show06
dot -Tpng q06_tree.dot -O
display q06_tree.dot.png || echo done
```

Cette commande vous affichera l'arbre généré pour la question.

Question 7 (2)

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

```
val to_list : 'a list -> 'a tree -> 'a list
```

`to_list [] t` renvoie la liste des clefs de `t` dans l'ordre inverse d'un parcours en profondeur (plus exactement, la fonction doit ajouter les clefs en tête de liste lors du traitement préfixe, donc avant les appels récursifs.)

Exemple 7.1:

```
> ./question07 0 3
t =
  (06(27(22(20)(28)(24))(23(09)(03)))
    (20(27(03)(21))(05(18))))


to_list [] t =
  [ 18; 05; 21; 03; 27; 20; 03; 09; 23; 24; 28; 20; 22; 27;
    06; ]
```

Remarque 2:
Le type des arbres généraux est le même que pour la question précédante et vous pouvez également les visualiser avec la cible `show07` *du* `Makefile` *(comme pour la question 6.)*

Question 8 (4)

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

```
val create : unit -> t_list
val is_empty : t_list -> bool
val size : t_list -> int
val add : int -> t_list -> unit
val drop_head : t_list -> unit
val get_head : t_list -> int
val iter : (int -> 'a) -> t_list -> unit
```

Vous devez écrire les opérations de bases sur un type de listes *impératives* (avec modifications en place.)

- `create ()` renvoie une nouvelle liste vide ;
- `is_empty l` renvoie vrai si la la liste `l` est vide ;
- `add x l` ajoute `x` en tête de la liste `l` (la liste est modifiée) ;
- `drop_head l` supprime la tête de la liste `l` (le comportement sur une liste vide n'est pas spécifié) ;
- `get_head l` renvoie la valeur en tête de la liste `l` sans modifier la liste (la liste est supposée non vide) ;
- `iter f l` applique la fonction `f` à chaque élément de la liste `l` (comme `List.iter`.)

Les listes sont représentées à l'aide des types :

```
type s_list =
```

```
{
  mutable value : int;
  mutable next  : s_list option;
}

type t_list =
{
  mutable head : s_list option;
  mutable size : int;
}
```

Le *record* `t_list` représente la *sentinelle* en début de liste. Il contient la taille (`size`) de la liste (supposée correcte) et le champ `head` *pointe* sur la tête de la liste. Si la liste est vide se champ contient `None` sinon, il contient `Some v` où `v` est un *record* de type `s_list` contenant la valeur et le *pointeur* sur suivant.

On rappelle que le type `'a option` est défini comme :

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

Donc `None` correspond au *pointeur* `NULL` et `Some v` correspond à une sorte de *pointeur* sur le *record* `v` (ce n'est pas une référence, les champ sont tous *mutable*.)

Exemple 8.1:

```
> ./question08 0 5
Test create:
size l = 0
Test add:
Test iter (et résultat add):
l =
  [
   18; 36; 01; 22; 30;
  ]
size l = 5

Test is_empty, get_head et drop_head:

head = 18; drop_head l (size = 4)
head = 36; drop_head l (size = 3)
head = 01; drop_head l (size = 2)
head = 22; drop_head l (size = 1)
head = 30; drop_head l (size = 0)
```

Le programme teste successivement toutes les opérations sur les listes : création, accès à la taille, ajout d'éléments, itérations (via `iter`), test du vide, accès à la tête et suppression de la tête . . .

Question 9 (3)

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

```
val fibo : int -> Big_int.big_int
```

`fibo n` calcule le rand `n` de la suite de Fibonacci. Le résultat est de type `Big_int.big_int` correspondant aux entiers à précision infinie. On rappelle la définition récursive de la suite :

$$\texttt{fibo}(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \texttt{fibo}(n-1) + \texttt{fibo}(n-2) & \text{sinon} \end{cases}$$

L'utilisation des `Big_int.big_int` permet de calculer des rangs de Fibonacci beaucoup plus gros qu'avec des entiers simples. Par conséquent, votre version de Fibonacci devra être performante ! La meilleure solution consiste à faire une version *mémoizée* de la fonction.

L'usage des `Big_int.big_int` est simplifié à l'aide du module `BI` fourni dans le fichier de la question :

```
module BI :
sig
  val zero : Big_int.big_int
  val one : Big_int.big_int
  val add : Big_int.big_int -> Big_int.big_int -> Big_int.big_int
end
```

- `BI.zero` correspond au 0
- `BI.one` correspond au 1
- `BI.add x y` calcule l'addition de `x` et `y` en `Big_int.big_int`.

On rappelle que la mémoization consiste à conserver les résultats précédants dans une structure pratique (une table de hash par exemple) et de ne calculer que les rangs qui ne sont pas déjà dans la structure. Schématiquement, votre fonction devrait ressembler à :

```
fibo(n) =
  si n est dans la table -> renvoyer sa valeur
  sinon
    calculer fibo(n-1) et fibo(n-2)
    sauver la somme dans la table
    renvoyer le résultat.
```

On rappelle que la table de hash pour être réutilisée à tous les appels, doit être une *variable statique* : elle définit à l'extérieur de la fonction. À titre d'exemple, voici une fonction (non récursive) mémoizée :

```
let memo_f =
  let h = Hashtbl.create 101 in
  let f x =
    try Hashtbl.find h x with
        Not_found ->
          begin
            let r = ref 0 in
```

```
            for i=1 to x do
              r := !r + i*x
            done;
            !r
        end
in f
```

Exemple 9.1:

```
> ./question09 0 5 10
fibo 0 = 0
fibo 4 = 3
fibo 3 = 2
fibo 6 = 8
fibo 8 = 21
> ./question09 0 5 100
fibo 90 = 2880067194370816120
fibo 54 = 86267571272
fibo 33 = 3524578
fibo 76 = 3416454622906707
fibo 78 = 8944394323791464
```

`./question09 seed size max` calcule `size` fois Fibonacci en tirant une valeur aléatoire (la séquence aléatoire est initialisée avec la graine `seed`) majorée par `max`.

Question 10 (4)

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

```
val eval :
  (string,(int -> int -> int)*(float -> float -> float)) Hashtbl.t
  -> value Env.t -> expr -> value
```

`eval openv env e` évalue l'expression `e` dans l'environnement de variables locales `env` et avec l'environnement d'opérateurs `openv`.

Les expressions sont décrites par les types suivants :

```
type value = Int of int | Float of float
type expr =
    Val of value
  | Var of string
  | Op of expr * string * expr
  | Let of string * expr * expr
```

Les valeurs sont soient des entiers, soit des floattants. Le résultat de l'évaluation devra être de type `value` et donc soit entier, soit floattant.

Les opérateurs sont décrits dans un environnement particulier (une table de hash) qui contient (associé à la chaîne de l'opérateur) un couple de fonction, la première correspond à l'opérateur appliqué aux entiers et la seconde appliqué aux floattants.

Pour appliquer les opérateurs, vous devrez calculer les deux opérandes (les deux sous-expressions), déterminer leur *type* (entier ou floattant) et appliquer la fonction correspondante. Si les deux opérandes sont de type différents, il faut transformer l'entier en floattant et appliquer l'opération sur les floattants.
La sémantique est résumé par les règles suivantes :

$$\frac{}{\Gamma \vdash \texttt{Val}\ \ v \to v} \qquad \frac{(x, v) \in \Gamma}{\Gamma \vdash \texttt{Var}\ \ x \to v}$$

$$\frac{\Gamma \vdash e_1 \to v_1 \qquad \Gamma \cup (x, v_1) \vdash e_2 \to v}{\Gamma \vdash \mathit{Let}(x, e_1, e_2) \to v}$$

$$\frac{\Gamma \vdash e_1 \to \texttt{Int}\ \ v_1 \quad \Gamma \vdash e_2 \to \texttt{Int}\ \ v_2 \quad (\texttt{op}, (\texttt{fi}, \texttt{ff})) \in \texttt{openv} \quad \texttt{fi}\ \ v_1\ \ v_2 \to v}{\Gamma \vdash \mathit{Op}(op, e_1, e_2) \to \texttt{Int}\ \ v}$$

$$\frac{\Gamma \vdash e_1 \to \texttt{Float}\ \ v_1 \quad \Gamma \vdash e_2 \to \texttt{Float}\ \ v_2 \quad (\texttt{op}, (\texttt{fi}, \texttt{ff})) \in \texttt{openv} \quad \texttt{ff}\ \ v_1\ \ v_2 \to v}{\Gamma \vdash \mathit{Op}(op, e_1, e_2) \to \texttt{Float}\ \ v}$$

$$\frac{\Gamma \vdash e_1 \to \texttt{Float}\ \ v_1 \quad \Gamma \vdash e_2 \to \texttt{Int}\ \ v_2 \quad (\texttt{op}, (\texttt{fi}, \texttt{ff})) \in \texttt{openv} \quad \texttt{ff}\ \ v_1\ \ (\texttt{float}\ \ v_2) \to v}{\Gamma \vdash \mathit{Op}(op, e_1, e_2) \to \texttt{Float}\ \ v}$$

$$\frac{\Gamma \vdash e_1 \to \texttt{Int}\ \ v_1 \quad \Gamma \vdash e_2 \to \texttt{Float}\ \ v_2 \quad (\texttt{op}, (\texttt{fi}, \texttt{ff})) \in \texttt{openv} \quad \texttt{ff}\ \ (\texttt{float}\ \ v_1)\ \ v_2 \to v}{\Gamma \vdash \mathit{Op}(op, e_1, e_2) \to \texttt{Float}\ \ v}$$

Ces règles se lisent de la manière suivantes : la partie *haute* de la règle décrit les *prémisses* et la partie *basse* est un *jugement* qui se lit : $\Gamma \vdash e \to v$ dans l'environnement de variables Γ, l'expression e s'évalue en la valeur v.
Les environnements de variables locales sont des *map* fonctionnelles contenant la valeur associées au nom de la variable. Le module les décrivant est :

```
module Env :
sig
  type key = String.t
  type 'a t = 'a Map.Make(String).t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val add : key -> 'a -> 'a t -> 'a t
  val find : key -> 'a t -> 'a
  val remove : key -> 'a t -> 'a t
  val mem : key -> 'a t -> bool
  val iter : (key -> 'a -> unit) -> 'a t -> unit
  val map : ('a -> 'b) -> 'a t -> 'b t
  val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t
  val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  val compare : ('a -> 'a -> int) -> 'a t -> 'a t -> int
  val equal : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
end
```

Exemple 10.1:

```
> ./question10 expr01_q10
e =
  let x = 6 in (x * 7.000000)
e = 42.000000
```

Le programme prend en entier le d'un nom de fichier contenant une expression avec une syntaxe relativement proche de celle d'OCaml. Quelques fichiers d'exemple sont fournis, leur nom a la forme `exprXY_q10` *(avec* `XY` *le numéro de l'exemple.)*

Question 11 (2)

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

```
val dfiter : (int -> 'a) -> (int -> unit) -> int array -> unit
```

`dfiter pref suff t` parcours en profondeur l'arbre binaire `t` en appliquant en *prefixe* (avant de traiter les fils) la fonction `pref` sur la clef du nœud et en *suffixe* (après avoir traiter les fils) la fonction. `suff`.

Les arbres sont représentés en version *statique* : les clefs des nœuds sont stockées dans un tableau. La case `0` contient juste le nombre de clef de l'arbre et la racine est à la case `1` et le fils gauche d'un nœud dont la clef est à la case `i` se trouve à la case `2*i` et le fils droit à la case `1+2*i`.

Exemple 11.1:

```
> ./question11 0 15
t(15) =
  [| 90; 54; 33; 76; 78; 05; 37; 85; 95; 51; 32; 73; 15;
      42; 67;|]
Teste de dfiter:

(90
   (54(76(85)(95))
      (78(51)(32))
   )
   (33(05(73)(15))
      (37(42)(67))
   )
)
```

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 :

```c
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 :

.depend
Makefile
lexer.mll
parser.mly
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