EPITA SPE promo 2013
Programmation
Épreuve machine

Marwan Burelle*

Mercredi 23 décembre 2009

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* `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. 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 questionX`.

La méthode de compilation pour la correction sera celle décrite dans le `Makefile`*, 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 24 points et 12 questions dans cet examen.

*marwan.burelle@lse.epita.fr

Listes

Question 1 (1)

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

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

`funcount f l` renvoie le nombre d'élément `e` de `l` tel que `f e` renvoie vrai.

Exemple 1.1:

```
> ./question01 0 5
let f x = x mod 2 = 0
l = [ 338; 804; 961; 598; 414; ]
funcount f l = 4
```

Question 2 (1)

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

```
val insert : 'a -> 'a list -> 'a list
```

`insert x l` insert `x` dans la liste triée en ordre **décroissant** sans doublon `l`. La liste doit rester triée et sans doublon (`x` n'est pas insérée s'il est déjà dans la liste.)

Exemple 2.1:

```
> ./question02 0 5
l = [ 961; 804; 598; 414; 338; ]
insert 389 l = [ 961; 804; 598; 414; 389; 338; ]
```

Question 3 (2)

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

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

`phijoin f l1 l2` renvoie l'ensemble des couples `(a,b)` tel que `a` appartient à `l1`, `b` appartient à `l2` et tel que `f a b` renvoie vrai.

Exemple 3.1:

```
> ./question03 0 5
l1 = [ 82; 36; 65; 86; 30; ]
l2 = [ 27; 3; 21; 121; 5; ]
let f x y = (x + y) mod 3 = 0 && (x - y) mod 7 = 0
phijoin f l1 l2 = [ (121,65); (121,86); (05,82); ]
```

Structures avancées

Question 4 (3)

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

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

val xtransform : ('a -> 'a option) -> 'a tree -> 'a tree
```

`xtransform f t` renvoie une copie de l'arbre `t` où les nœuds de clefs `k` pour lesquels `f k` renvoie `Some k'` ont été remplacés par la feuille `Node(Nil,k',Nil)`. Si `f k` renvoie `None`, le nœud est inchangé et `xtransform` continue sur récursivement sur ses fils.

Exemple 4.1:

```
> ./question04 0 3
t =
  Node (
    Node ( Node ( Nil, 012, Nil) , 022,
      Node ( Nil, 060, Nil) )
    , 079,
    Node ( Node ( Nil, 040, Nil) , 044,
      Node ( Nil, 020, Nil) )
    )

let f x =
  if x mod 5 = 0 then Some (x/(-5))
  else None

xtransform f t =
  Node (
    Node ( Node ( Nil, 012, Nil) , 022,
      Node ( Nil, -12, Nil) )
    , 079,
    Node ( Node ( Nil, -08, Nil) , 044,
      Node ( Nil, -04, Nil) )
    )
```

Question 5 (2)

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

```
val mirror : 'a tree -> 'a tree
```

`mirror t` renvoie le mirroir de l'arbre binaire `t` (défini par le même type que pour la question précédante.)

On définit le mirroir d'un arbre de la manière suivante :

– Le mirroir de l'arbre vide (`Nil`) est l'arbre vide.
– Le mirroir d'un nœud `Node(ls,k,rs)` est le nœud `rs',k,ls'` où `rs'` est le mirroir du fils `rs` et `ls'` est le mirroir du fils `ls`.

Exemple 5.1:

```
> ./question05 0 3
t =
  Node (
    Node ( Node ( Nil, 012, Nil) , 022,
      Node ( Nil, 060, Nil) )
    , 079,
    Node ( Node ( Nil, 040, Nil) , 044,
      Node ( Nil, 020, Nil) )
    )

mirror t =
  Node (
    Node ( Node ( Nil, 020, Nil) , 044,
      Node ( Nil, 040, Nil) )
    , 079,
    Node ( Node ( Nil, 060, Nil) , 022,
      Node ( Nil, 012, Nil) )
    )
```

Question 6 (1)

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

```
type 'a vector =
{
  mutable size : int;
  tab : 'a array;
}

val reverse : 'a vector -> unit
```

`reverse v` inverse (en place) les valeurs du vecteur `v`.

Exemple 6.1:

```
> ./question06 0 5
v =
{
   size = 5;
   tab = [| 20; 44; 40; 79; 60; |];
}
```

```
reverse v
v =
{
   size = 5;
   tab = [| 60; 79; 40; 44; 20; |];
}
```

Question 7 (2)

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

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

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

`breadth_first_iter f t` applique la fonction `f` à tous les nœuds de l'arbre général `t` en respectant un parcours **largeur**. Les fils d'un nœud seront rencontrés dans l'ordre de la liste de fils de ce nœud.

Pour votre parcours largeur, vous pourrez utiliser le module `Queue` (file *fifo*) d'OCaml dont voici un extrait de la signature :

```
type 'a t
(** The type of queues containing elements of type ['a]. *)

exception Empty
(** Raised when {!Queue.take} or {!Queue.peek} is applied to an
    empty queue. *)

val create : unit -> 'a t
(** Return a new queue, initially empty. *)

val add : 'a -> 'a t -> unit
(** [add x q] adds the element [x] at the end of the queue [q]. *)

val push : 'a -> 'a t -> unit
(** [push] is a synonym for [add]. *)

val take : 'a t -> 'a
(** [take q] removes and returns the first element in queue [q],
   or raises [Empty] if the queue is empty. *)

val is_empty : 'a t -> bool
(** Return [true] if the given queue is empty, [false] otherwise. *)
```

Exemple 7.1:

```
> ./question07 0 3
```

```
t =
  Node(92, [
    Node(95, [ Node(22, [ Node(79, [ ]) Node(60, [ ]) ])
      Node(49, [ Node(15, [ ]) ])
      Node(12, [ Node(88, [ ]) ]) ])
    Node(77, [ Node(68, [ Node(76, [ ]) Node(23, [ ]) ])
      Node(68, [ Node(35, [ ]) ]) ])
    Node(90, [ Node(29, [ Node(70, [ ]) ])
      Node(59, [ Node(71, [ ]) Node(51, [ ]) Node(07, [ ])
        ])
      ])
    ])

breadth_first_iter = 92 95 77 90 22 49 12 68 68 29 59 79 60
  15 88 76 23 35 70 71 51 07
```

Modules, foncteurs et objets

Question 8 (4)

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

```
module type S =
  sig
    type t
    val zero : t
    val succ : t -> t
    val pred : t -> t
    val add : t -> t -> t
    val canon : t -> t
    val equal : t -> t -> bool
    val to_string : t -> string
  end
module Peano : S
```

Compléter la définition du module `Peano`. Ce module décrit les entiers de *Peano* : un entier est soit *0*, soit le successeur d'un entier, soit le prédécesseur d'un entier.

La définition partielle suivante est fournie :

```
module Peano : S =
struct
  type t = Zero | Succ of t | Pred of t
  let zero = Zero
  let succ i = Succ i
  let pred i = Pred i

  let rec add _ _ = (* FIX ME *) assert false
```

```
  let rec canon _ = (* FIX ME *) assert false

  let equal _ _   = (* FIX ME *) assert false

  let rec to_string = function
      Zero -> "0"
    | Succ n -> "Succ(" ^ (to_string n) ^ ")"
    | Pred n -> "Pred(" ^ (to_string n) ^ ")"
end
```

La fonction `add p1 p2` réalise l'addition entre les deux entiers `p1` et `p2` en représentation de *Peano* (*indice* : utiliser la définition récursive de l'addition.)

La fonction `canon p` renvoie la représentation canonique de l'entier `p` : l'entier est soit *0*, soit n'est constitué que de successeur de *0*, soit que de prédécesseur de *0*. Attention, vous ne pouvez pas passé par les entiers d'OCaml, le but est de construire des entiers possiblement plus grands que les entiers natifs. *Indice :* il peut être intéressant de séparer les marqueurs successeurs et prédécesseurs en deux ensembles.

`equal p1 p2` renvoie vrai si et seulement si les deux entiers en représentation de *Peano* sont égaux. Attention, deux entiers de *Peano* peuvent avoir une forme différente et pourtant être égaux (*indice* : passer par la forme canonique.)

Exemple 8.1:

```
> ./question08 0 5
p1 = Succ(Succ(Pred(Succ(Succ(0)))))
p1' = canon p1 = Succ(Succ(Succ(0)))
p2 = Pred(Pred(Pred(Pred(Pred(0)))))
p2' = canon p2 = Pred(Pred(Pred(Pred(Pred(0)))))
add p1 p2 = Pred(Pred(Pred(Pred(Pred(Succ(Succ(Pred(Succ(Succ(0)))))))))
add p1' p2' = Pred(Pred(Pred(Pred(Pred(Succ(Succ(Succ(0)))))))))
add p1 p2 = add p1' p2'
```

Question 9 (2)

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

```
module type OrderedType =
sig
  type t
  val compare : t -> t -> int
  val to_string : t -> string
end
module type S =
  sig
    type elm
    type t
    val compare : t -> t -> int
    val make : elm -> elm -> t
```

```
    val to_string : t -> string
  end
module ComparablePair :
  functor (T : OrderedType) -> S with type elm = T.t
```

Compléter la définition du foncteur `ComparablePair` qui engendre des pairs d'éléments *comparable*.

On donne la définition partielle du foncteur `ComparablePair` :

```
module ComparablePair (T:OrderedType) =
struct

  type elm = T.t
  type t = elm * elm

  let compare _ _ = (* FIX ME *) assert false

  let make a b = (a,b)

  let to_string (x,y) =
     Printf.sprintf "(%s,%s)" (T.to_string x) (T.to_string y)
end
```

Vous devez définir la fonction de comparaison `compare` qui compare les pairs en ordre *lexicographique* (premier membre, puis second membre) en utilisant la comparaison définie dans le module paramètre `T`.

Exemple 9.1:

```
> ./question09 0 5
{
   ((n,v),(f,g));
   ((s,z),(k,g));
   ((t,d),(w,t));
   ((t,m),(t,b));
   ((w,l),(d,p));
}
```

Question 10 (2)

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

```
exception Vector_full
class virtual ['a] iterable :
  object
    method iter : ('a -> unit) -> unit
    method virtual next : 'a option
    method virtual reset : unit
  end
```

```
class ['a] iterable_vector :
  int ->
  'a ->
  object
    val vect : 'a array
    method add : 'a -> unit
    method iter : ('a -> unit) -> unit
    method next : 'a option
    method reset : unit
  end
```

Compléter la définition de la classe `iterable_vector`. Vous compléterez la définition partielle fournie :

```
class ['a] iterable_vector max a =
object (s)

  inherit ['a] iterable

  val vect : 'a array = Array.make max a

  method add : 'a -> unit = (* FIX ME *) assert false

  method next = (* FIX ME *) assert false

  method reset = (* FIX ME *) assert false
end
```

`v#add x` ajoute `x` à la fin du vecteur `v`. Lève l'exception `Vector_full` si la taille maximale a été atteinte.

`v#next` renvoie le *prochain* élément du vecteur. La première fois il renvoie l'élément le plus ancien du vecteur, puis le suivant ... La méthode renvoie `Some x` en cours de parcours du vecteur (`x` l'élément correspondant) ou `None` lorsque l'on a atteint la fin du vecteur.

`v#reset` repositionne le prochain élément renvoyer par `v#next` sur le premier élément du vecteur.

Indice : vous devez maintenir la taille actuelle du vecteur (la position de prochaine ajout) et un indicateur de position pour les méthodes `next` et `reset`.

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`), 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 (1)

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

```
module MyInt :
  sig
    type t
    val zero : t
    val one : t
    val mul : int -> t -> t
    val to_string : t -> string
  end
val fact : MyInt.t -> int -> MyInt.t
```

`fact MyInt.one n` renvoie la factorielle de `n` en utilisant le premier paramètre comme accumulateur. Vous devez utiliser le module `MyInt` fourni (qui fournit des grands entiers.)

Exemple 12.1:

```
> ./question12 0 5
30! -> 265252859812191058636308480000000
22! -> 1124000727777607680000
01! -> 1
04! -> 24
18! -> 6402373705728000
```

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.

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 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 succincte s'il sont appelés avec l'option `-help`.

Exemple 2:
Le binaire produit pour la question 4 fournit l'aide suivante :

```
Question 4:
./question04 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 `./question04 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_qX.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 :

.depend
Makefile
noel.ml
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