EPITA SPE promo 2013
Programmation
Épreuve machine

Marwan Burelle*

Lundi 9 novembre 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 23 points et 12 questions dans cet examen.

*marwan.burelle@lse.epita.fr

Récursion et arithmétique

Question 1 (1)

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

```
val inner_sum : int -> int
```

`inner_sum n` renvoie la somme des chiffres de `n`.

Exemple 1.1:

```
un_shell> ./question01 0 5
0414 ->  9
0598 -> 22
0961 -> 16
0804 -> 12
0338 -> 14
```

Question 2 (2)

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

```
val dec2hex : int -> string list
```

`dec2hex n` renvoie une chaîne de caractères contenant la représentation en hexadécimal (base 16) de `n`. La fonction `digit2hex c` (fournie) renvoie la chaîne de caractères contenant la représentation en hexadécimal du chiffre `c` (`c` doit être compris entre 0 et 15.)

décimale	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hexadécimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

TAB. 1 – Compter en hexadécimal

Exemple 2.1:

```
un_shell> ./question02 0 5
158 -> 9E (devrait etre 9E)
086 -> 56 (devrait etre 56)
193 -> C1 (devrait etre C1)
036 -> 24 (devrait etre 24)
082 -> 52 (devrait etre 52)
```

Question 3 (1)

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

```
val sum_fact : int -> int
```

`sum_fact n` renvoie la somme des valeurs de factorielle de 0 à `n`, *i.e.* : $\sum_{i=n}^{i=0} i!$

Exemple 3.1:
```
un_shell> ./question03 0 6
000 -> 1
004 -> 34
007 -> 5914
004 -> 34
008 -> 46234
001 -> 2
```

Listes

Question 4 (1)

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

```
val str_of_str_list : string list -> string
```

`str_of_str_list [s1; ...; sn]` renvoie la chaîne `(s1 ^ ... ^ sn)`

Exemple 4.1:
```
un_shell> ./question04 0 5
l = [ "8"; "6"; "3"; "4"; "0"; ]
str_of_str_list l = "86340"
```

Question 5 (2)

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

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

`select f l` renvoie la liste des `y` tel que `(x,y)` soit dans `l` et `f x` soit vrai.

Exemple 5.1:
```
un_shell> ./question05 0 5
f n = n mod 2 = 0

l = [ (51,95); (85,37); (05,78); (76,33); (54,90); ]

select f l = [ 33; 90; ]
```

Question 6 (3)

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

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

`rotation [x1; ... ; x(n-1); xn]` renvoie la liste `[xn; x1; ...; x(n-1)]`.

Exemple 6.1:

```
un_shell> ./question06 0 0
l = [ ]
rotation l = [ ]
un_shell> ./question06 0 1
l = [ 90; ]
rotation l = [ 90; ]
un_shell> ./question06 0 5
l = [ 78; 76; 33; 54; 90; ]
rotation l = [ 90; 78; 76; 33; 54; ]
```

Références

Question 7 (1)

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

```
val make_uniq_name : string -> unit -> string
```

`make_uniq_name prefix` renvoie une fonction qui renvoie une nouvelle chaîne de caractères commençant par `prefix` suivit d'un entier. À chaque appel, l'entier est incrémenté de 1, la valeur initiale étant 0.

Exemple 7.1:

```
un_shell> ./question07 0 5
let next = make_uniq_name "lrehyw"
next () = "lrehyw0"
next () = "lrehyw1"
next () = "lrehyw2"
next () = "lrehyw3"
next () = "lrehyw4"

let next = make_uniq_name "myndzd"
next () = "myndzd0"
next () = "myndzd1"
next () = "myndzd2"
next () = "myndzd3"
next () = "myndzd4"
```

Records et listes

Question 8 (2)

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

```
val len : 'a t_list -> int
```

`len l` renvoie la longueur de la liste `l`. Le type `t_list` est défini comme suit :

```
type 'a s_list =
    {
      value : 'a ;
      next  : 'a t_list;
    }
and 'a t_list = Nil | Cons of 'a s_list
```

Exemple 8.1:

```
un_shell> ./question08 0 5
l =
{ value= 78; next=
  { value= 76; next=
    { value= 33; next=
      { value= 54; next= { value= 90; next= {} } }
    }
  }
}
len l = 5
```

Question 9 (2)

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

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

`to_list l` renvoie la liste *OCaml* représentant la liste `l` (dans le même format que pour la question précédente.)

Exemple 9.1:

```
un_shell> ./question09 0 5
l =
{ value= 78; next=
  { value= 76; next=
    { value= 33; next=
      { value= 54; next= { value= 90; next= {} } }
    }
  }
}
to_list l = [ 78; 76; 33; 54; 90; ]
```

Question 10 (2)

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

```
val from_list : 'a list -> 'a t_list
```

`from_list l` renvoie la liste (du même type que pour les questions précédentes) représentant la liste *OCaml* `l`.

Exemple 10.1:

```
un_shell> ./question10 0 5
l = [ 78; 76; 33; 54; 90; ]
from_list l =
{ value= 78; next=
  { value= 76; next=
    { value= 33; next=
      { value= 54; next= { value= 90; next= {} } }
    }
  }
}
```

Tableaux

Question 11 (3)

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

```
val fill_from_list : 'a array -> int -> 'a list -> unit
```

`fill_from_list a pos l` remplit le tableaux `a` à partir de la position `pos` avec les éléments de la liste `l`. Le tableaux est supposé comme étant de taille suffisante (c'est à dire `pos + List.length l`).

Exemple 11.1:

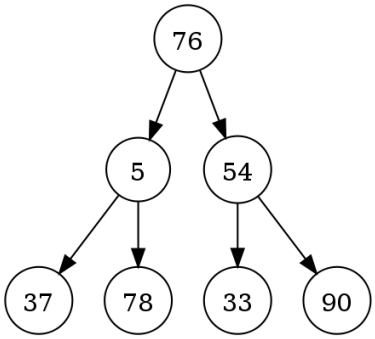
```
un_shell> ./question11 0 5
l = [ 78; 76; 33; 54; 90; ]

fill_from_list a 2 l
-------------------------
| 78 | 76 | 33 | 54 | 90 |
-------------------------

l = [ 51; 95; 85; 37; 5; ]

fill_from_list a 2 l
-----------------------------------
| -1 | -1 | 51 | 95 | 85 | 37 | 05 |
-----------------------------------
```

Question 12 (3)

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

```
val to_parstring : ('a -> string) -> 'a tree -> string
```

`to_parstring to_s t` renvoie la chaîne de caractères représentant l'arbre `t`. La fonction `to_s` est utilisée pour convertir les clefs de l'arbre en chaîne de caractères.



`(76(5(37()())(78()()))(54(33()())(90()())))`

FIG. 1 – Un arbre

La chaîne à renvoyer correspondra au format : `(<clef><fils-gauche><fils-droit>)` où `<clef>` est à remplacer par la chaîne représentant la clef et `<fils-gauche>` par la chaîne représentant le sous-arbre gauche et `<fils-droit>` celle représentant le sous-arbre droit. Un arbre vide est représenté par la chaîne `"()"`.

Les arbres sont décrit à l'aide du type suivant :

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

Remarque 1:
Le programme de test génère à chaque exécution un fichier `tree.dot` *au format* Graphviz. *Vous pouvez à l'aide de ce fichier visualiser votre arbre de manière graphique à l'aide la commande* `dot` *(ou* `twopi, neato, fdp` *...) Le* `Makefile` *fournit une cible* `tree.png` *qui engendre une image avec la commande* `dot`. *Sur de très gros arbres la commande* `twopi` *donnera probablement des résultats plus lisibles.*
Une fois l'image générée, vous pouvez la visualiser avec la commande `display`.

Exemple 12.1:

```
un_shell> ./question12 0 0
t = Node(Empty, 90, Empty)
to_parstring t = "(90()())"

un_shell> ./question12 0 2
t =
Node(
 Node(Node(Empty, 37, Empty), 05, Node(Empty, 78, Empty)),
 76,
```

```
 Node(Node(Empty, 33, Empty), 54, Node(Empty, 90, Empty))
)
to_parstring t =
  "(76(5(37()())(78()()))(54(33()())(90()())))"
```

On notera qu'ici le second paramètre correspond à la hauteur de l'arbre à tester (attention aux arbres trop grands.)

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 :

Makefile
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