EPITA SPE promo 2011
Programmation
Épreuve machine

Marwan Burelle*

18 janvier 2008

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 dûe au non respect des consignes explicites ne sera pas contestable, en particulier pour les problèmes de compilation de votre code.

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.)*

Le fichier `Makefile` *permet d'engendrer un petit programme de test pour chaque question. La méthode de compilation pour la correction sera celle décrite dans ce fichier, 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 8 questions dans cet examen, tout point au dessus 20 est considéré comme point bonus.

Programmation OCaml : Listes

Question 1 (2)

On se propose de coder la fonction suivante :

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

`phicount f l` renvoit le nombre d'élément de `l` pour lesquels `f` renvoit vrai.

Question 2 (2)

On se propose de coder la fonction suivante :

*Burelle@kh405.net

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

`dbllist [a1; ...; an]` renvoit `[a1; a1; ...; an; an]` *càd* elle double tous les éléments de la liste.

Question 3 (3)

On se propose de coder la fonction suivante :

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

`join l1 l2` renvoit la liste des couples `(a,c)` tels que le couple `(a,b)` appartient à `l1` et `(b,c)` appartient à `l2` (donc les couples de `l1` et `l2` qui correspondent sur le seconde et première composante.)

Exemple :

```
join [(1,2); (3,4); (5,6)] [(2,1); (2,2); (4,8)]
   = [(1,1); (1,2); (3,8)]
```

Question 4 (3)

Soit les définitions de types suivantes :

```
type ('a,'b) biList_a =
    Nil_a
  | B_a of 'a * ('a,'b) biList_b

and ('a,'b) biList_b =
    Nil_b
  | B_b of 'b * ('a,'b) biList_a

type ('a,'b) biList =
  | A of ('a,'b) biList_a
  | B of ('a,'b) biList_b
```

Un élément de type `('a,'b) biList` est une pseudo-liste dans laquelle s'alterne des éléments de type `'a` et `'b`.

Exemple :

```
A(B_a(1, B_b("1", B_a(2, Nil_b))));;
- : (int, string) biList = A (B_a (1, B_b ("1", B_a (2, Nil_b))))
B(B_b(1, B_a("1", B_b(2, Nil_a))));;
- : (string, int) biList = B (B_b (1, B_a ("1", B_b (2, Nil_a))))
```

On se propose de coder la fonction suivante :

```
val deinterlace : ('a, 'b) biList -> 'a list * 'b list
```

`deinterlace l` renvoie un couple de listes dont la première contient tous les éléments de type `'a` et la seconde tous les éléments de type `'b`.

Exemple :

```
deinterlace (A(B_a(1, B_b("1", B_a(2, Nil_b)))));;
- : int list * string list = ([1; 2], ["1"])
deinterlace (B(B_b(1, B_a("1", B_b(2, Nil_a)))));;
- : string list * int list = (["1"], [1; 2])
```

Programmation OCaml : Arbres

Question 5 (3)

Soit les définitions de types suivantes :

```
type ('a,'b) bloc = {
  key   : 'a;
  value : 'b;
}

type ('a,'b) arbre =
    Nil
  | Node of ('a,'b) arbre * ('a,'b) bloc * ('a,'b) arbre
```

Le type `('a,'b) arbre` définit des arbres binaires où chaque noeud est étiquetté par une clef et une valeur associée à cette clef. Les arbres sont triés par rapport aux clefs : toutes les clefs des fils à gauche sont inférieures et toutes les clefs des fils à droite sont supérieures.

On se proposer de coder la fonction suivante :

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

`find k t` cherche la clef `k` dans l'arbre `t` et renvoit la valeur associée. Si la clef n'est pas l'arbre la fonction doit lever l'exception prédéfinie `Not_found`.

Question 6 (3)

À partir des types de la question précédante, on se propose de coder la fonction suivante :

```
val add : 'a -> 'b -> ('a, 'b) arbre -> ('a, 'b) arbre
```

`add k v t` ajoute la clef `k` avec la valeur associée `v` dans l'arbre `t`. Si la clef est déjà présente dans l'arbre l'ancienne valeur est remplacée par `v`. L'ajout doit respecter l'ordre dans l'arbre et s'insérer en feuille (on remplace un noeud vide.) L'insertion est *fonctionnelle* : vous renvoyez un nouvel arbre.

Programmation OCaml : Évaluation d'expressions

Question 7 (5)

On définit un mini-langage de type assembleur à pile avec deux registres à l'aide du type suivant :

```
type asm =
    Nop
  | PUSH of int
  | POPA
  | POPB
  | PUSHA
  | PUSHB
  | ADD
  | SUB
  | MUL
  | JMP of string
  | BEQ of int * string
```

```
  | BNZ of string
  | LAB of string
  | OUT
  | STOP
```

Les instructions ont la sémantiques suivantes :

`Nop`	Opération vide (ne fait rien)
`PUSH i`	Empile l'entier `i`
`POPA`	Dépile la valeur de sommet de pile dans le registre `A`
`POPB`	Dépile la valeur de sommet de pile dans le registre `B`
`POPA`	Empile la valeur du registre `A`
`POPB`	Empile la valeur du registre `B`
`ADD`	Dépile les deux premiers éléments de la pile et empile leur somme
`SUB`	Dépile les deux premiers éléments de la pile et empile leur différence (le sommet de pile moint le suivant.)
`MUL`	Dépile les deux premiers éléments de la pile et empile leur produit
`JMP l`	Saute au label `l`
`BEQ(i,l)`	Saute au label `l` si le sommet de pile est égal à `i` (la pile n'est pas modifiée.)
`BNZ l`	Saute au label `l` si le sommet de pile est différent de `0` (la pile n'est pas modifiée.)
`LAB l`	Ajoute label
`OUT`	Affiche l'entier au sommet de la pile
`STOP`	Fin du programme (obligatoire.)

Un programme est un tableau contenant les instructions. L'évaluation d'un programme consiste à parcourir avec un compteur indiquant la position dans le tableau. Un saut à label revient à changer la valeur du compteur pour qu'il pointe sur l'instruction suivant la définition du label. Si le programme atteint la fin du tableau sans avoir rencontrer l'instruction `STOP`, l'exception levée par OCaml pour le dépassement des bornes d'un tableau sera utilisée (en gros, vous ne vous souciez pas des bornes du tableau, OCaml le fait pour vous.)

Afin de coder l'évaluation, vous avez besoin : d'une pile, d'un envirronement pour associer les labels et leur position, des deux registres `A` et `B` et du compteur de programme. Vous pouvez (sans obligation) vous inspirez de la structure suivante :

```
type register = {
  mutable a : int;
  mutable b : int;
}

let eval instTab =
  let pc = ref 0 in
  let env = Hashtbl.create 17 in
  let stack = Stack.create () in
  let run = ref true in
  let reg = { a = 0; b = 0; } in
  let next () = pc := !pc + 1 in
    (* Boucle d'évaluation des instructions *)
```

Le module `Stack` fournit une implémentation de pile standard avec les fonctions classiques `Stack.push, Stack.pop, Stack.top` (référez vous à la doc fournit.)

La gestion des erreurs (pile vide, label inexistant . . .) est laissé à OCaml : si vous n'utilisez pas les modules `Stack` et `Hashtbl`, votre programme devra lever les mêmes exceptions que celles de ces modules.

L'instruction `OUT` devra afficher l'entier sur la pile (en le dépilant) suivit **d'un saut de ligne**.

Le programme suivant calcule et affiche factorielle de 5 :

```
let prg = [|
  PUSH 5;
  PUSH 1;
  POPB;
  LAB "boucle";
  POPA;
  PUSHA;
  PUSHB;
  MUL;
  POPB;
  PUSH 1;
  PUSHA;
  SUB;
  BNZ "boucle";
  PUSHB;
  OUT;
  STOP
|]
```

Coder la fonction suivante :

```
val eval : asm array -> unit
```

Programmation C

Question 8 (2)

On se propose de coder la fonction suivante :

```
size_t mystrlen(char *s);
```

`mystrlen(s)` renvoit la longueur de la chaîne `s`.

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

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

```
Prog TP: question 4
question4 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` font tous référence à l'initialisation d'un générateur de nombre aléatoire. Dans le cas présent la commande `./question4 X Y` va initialiser le générateur de nombre aléatoire avec X et l'utiliser pour engendrer une liste de taille Y. Pour les mêmes valeurs de X et de Y on obtiendra la même liste (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 jetter un oeil sur le fichier contenant les tests de la question X (`test-qX.ml`.)

Listes des fichiers à ne pas modifier

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 original :

Makefile
question1.ml
question2.mli
question3.mli
question4.mli
question5.mli
question6.mli
question7.mli
question8.h
test-q1.ml
test-q2.ml
test-q3.ml
test-q4.ml
test-q5.ml
test-q6.ml
test-q7.ml
test-q8.c

Listes des fichiers modifiables

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.)

question1.ml
question2.ml
question3.ml
question4.ml
question5.ml
question6.ml
question7.ml
question8.c