Université Paris Diderot
L3 Info

Programmation Fonctionnelle
Année 2015-2016

Examen

lundi 20 juin 2016

Tout document papier est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints. Le temps à disposition est de **3 heures**.

Les exercices doivent être rédigés **en fonctionnel pur** : ni références, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements à champs mutables. Une fonction écrite en style impératif ne rapportera aucun point.

Les questions sont indépendantes mais il est parfois nécessaire (ou recommandé) d'utiliser les fonctions définies précédemment (même si elles sont non traitées) ou prédéfinies dans la bibliothèque standard (notamment dans le module sur les listes).

On recommande de *bien lire* l'énoncé d'un exercice avant de commencer à le résoudre. Cet énoncé a **4 pages**.

Exercice 1 (Expressions). Qu'affiche le toplevel Caml quand on entre l'une après l'autre les expressions suivantes ? Donner pour chaque expression ou définition, son type, sa valeur, et, le cas échéant, son effet de bord ou l'exception levée. Si l'expression est mal typée ou pas correcte, donner le message d'erreur. Justifier si nécessaire.

1. `let x = 3+4/2;;`
2. `if x = 5. then "true" else "false";;`
3. `let f x = if x = 5. then print_string "true" else print_string "false";;`
4. `f 3.;;`
5. `[1;2;"three"];;`
6. `type numbers = A of int | B of string;;`
7. `[A 1;A 2;B "three"];;`
8. `(1,2,"three");;`
9.
```
let rec mi g l = match l with
      [] -> []
    | h::t -> (g 0 h):: (mi (fun x -> g (x+1)) t);;
```
10. `mi (+) [2;3;1];;`
11.
```
let rec a = ref 0
and f () =
  let b = ref 0 in
  a := !a+1; b := !b+1; !a * !b ;;
```
12. `let x1 = f() in let x2 = f() in let x3 = f() in [x1;x2;x3];;`

Exercice 2. Écrire une version du programme suivant qui n'utilise ni boucles, ni données mutables (e.g. références, tableaux, etc...). La fonction doit avoir exactement le même type et le même comportement.

```
let mot m =
  let flag = ref true in
  while (!flag) do
    print_endline "Devine le mot" ;
    let guess = read_line()
    in if guess = m then flag := false
    else
      let common = ref "Caracters en commun : "
      in let f c = if (String.contains m c)
                   then common := !common ^ (String.make 1 c) ^" "
         in String.iter f guess ; print_endline (!common)
  done ;
  print_endline "Bravo !"
```

On rappelle que dans le module String contient les fonctions suivantes :

- `contains : string -> char -> bool` tel que `contains s c` vaut `true` si le caractère `c` se trouve dans la chaîne de caractères `s`, sinon `contains s c` vaut `false`
- `make : int -> char -> string` tel que `make n c` renvoie une chaine de caractères fait par `n` copies du caractère `c`
- `iter : (char -> unit) -> string -> unit` tel que `iter f s` applique la fonction `f` sur tous les caractères de la chaine `s`, c.à-d. `iter f s` est equivalent à : `(f s.[0]); (f s.[1]); ...; (f s.[String.length s - 1]); ()`.

Exercice 3. Transformer les fonctions suivantes en récurrence terminale. Attention : dans chaque cas, la fonction obtenue doit avoir exactement le même type et le même comportement.

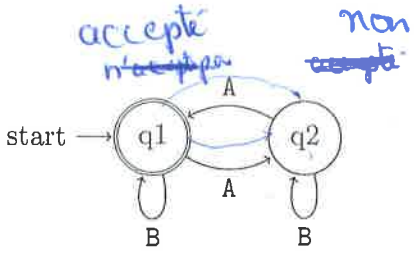
1.
```
let rec expo x =
    if x = 0 then 1
    else 2*(expo (x-1))
```

2.
```
let divisors x =
    let rec aux x c =
        if c = x then [ x ]
        else if (x mod c = 0) then (c::(aux x (c+1)))
        else aux x (c+1)
    in aux x 1
```

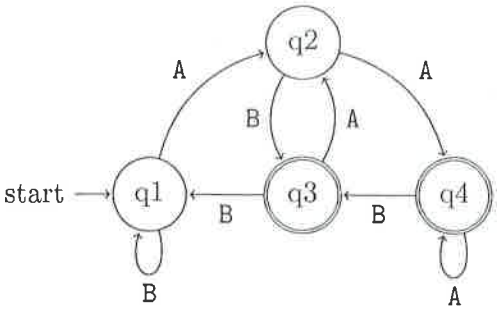
3.
```
type intTree = Nil | Node of int*intTree*intTree;;

let rec sum t = match t with
    Nil -> 0
    | Node (n,t1,t2) -> n + (sum t1) + (sum t2)
```

Exercice 4. Le but de cet exercice est de mettre en œuvre une interface modélisant les automates finis déterministes. On rappelle que un automate fini déterministe est défini par un n-uplet $(\Sigma, S, s_0, \delta, F)$, où :



(a) Un exemple d'automate, acceptant les mots avec un nombre pair de A.



(b) Un autre exemple d'automate.

FIGURE 1

— Σ est un ensemble fini, non vide, de lettres. C'est l'alphabet d'entrée sur lequel sont construits les mots à reconnaitre,
— S est un ensemble fini, non vide d'états,
— s_0 est l'état initial, élément de S
— δ est la fonction de transition : $\delta : S \times \Sigma \mapsto S$
— F est l'ensemble des états terminaux, qui est un sous-ensemble de S.

Un automate peut être aussi spécifié graphiquement. Par example, la Figure 1a définit un automate sur l'alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}\}$, avec deux états $S = \{q1, q2\}$, $q1$ étant en même temps état initial et final, et avec transition δ définit par :

$$(q1, \mathtt{A}) \mapsto q2 \qquad (q1, \mathtt{B}) \mapsto q1 \qquad (q2, \mathtt{A}) \mapsto q1 \qquad (q2, \mathtt{B}) \mapsto q2$$

Cet automate reconnait le langage constitué des mots avec un nombre pair de A.

Dans cet exercice on fixe un alphabet Σ de deux lettres :

```
type alphabet = A | B
```

Un automate sera alors défini par un enregistrement :

```
type 's automaton =
  {
    initial : 's ;
    final : 's -> bool;
    transition : 's -> alphabet -> 's;
  }
```

où la variable de type 's réfère au type des états de l'automate. Par exemple, l'automate de la Figure 1a est défini par :

```
type state = Q1 | Q2

let a_even = {
    initial = Q1 ;
    final = (fun q -> match q with
                        Q1 -> true
                      | Q2 -> false );
    transition =
      (fun a s -> match a,s with
```

```
                Q1,A -> Q2
              | Q2,A -> Q1
              | s,B -> s
    )
}
```

1. Écrire en OCaml un automate qui reconnait le langage constitué des mots avec un nombre impair de lettres `A`.
2. Écrire en OCaml l'automate définit par la Figure 1b. (*Facultatif :* quel est le langage reconnu par cet automate ?).
3. Écrire en OCaml un automate qui reconnait le langage constitué des mots qui alternent `A` et `B` en commençant par `A`. C'est à dire l'ensemble $\{\epsilon, \texttt{AB}, \texttt{ABAB}, \texttt{ABABAB}, \ldots\}$.
4. Écrire la fonction `exec : 's automaton -> alphabet list -> bool` qui calcule l'exécution d'un automate `a` sur un mot `w` (vu comme une liste des lettres de `alphabet`), c.-à-d. `exec a w` évalue à `true` si l'exécution de l'automate `a` sur le mot `w` termine sur un état final, et sinon `exec a w` évalue à `false`.
5. Écrire la fonction `neg : 'a automaton -> 'a automaton` qui prend en entrée un automate `a` et renvoie en sortie un automate qui décide le langage complementaire du langage décidé par `a`. C'est à dire : `exec (neg a) w` évalue à `true` si `exec a w` évalue à `false`, et réciproquement `exec (neg a) w` évalue à `false` si `exec a w` évalue à `true`.
6. Écrire la fonction `union : 'a automaton -> 'a automaton -> 'a automaton` qui prend en entrée deux automates `a1` et `a2` et renvoie un automate qui décide l'union des deux langages décidés respectivement par `a1` et `a2`. C.-à-d. `exec (union a1 a2) w` donne `true` si `exec a1 w` ou `exec a2 w` donne `true`, et `exec (union a1 a2) w` donne `false` si `exec a1 w` et `exec a2 w` donnent les deux `false`.
   (*Suggestion. On rappelle que l'union des deux automates* $\mathcal{A}_1 = (\Sigma, S^1, s_0^1, \delta^1, F^1)$ *et* $\mathcal{A}_2 = (\Sigma, S^1, s_0^2, \delta^2, F^2)$ *sur le même alphabet* Σ *peut être défini comme l'automate dont l'ensemble des états est le produit cartésien* $S^1 \times S^2$ *des ensembles des états de* $\mathcal{A}_1$ *et* $\mathcal{A}_2$*, l'état initial étant la pair* (s_0^1, s_0^2)*, un état* (s^1, s^2) *etant final si et seulement si* s^1 *est finale pour* $\mathcal{A}_1$ *ou* s^2 *est finale pour* $\mathcal{A}_2$*. Enfin, la transition envoie un état* (s^1, s^2) *et un lettre* a *dans l'état* $(\delta^1\ s^1\ a, \delta^2\ s^2\ a)$.)
7. Écrire la fonction `intersection : 'a automaton -> 'a automaton -> 'a automaton` qui prend en entrée deux automates `a1` et `a2` et renvoie un automate qui décide l'intersection des deux langages décidés respectivement par `a1` et `a2`. C'est à dire : `exec (intersection a1 a2) w` évalue à `true` si `exec a1 w` et `exec a2 w` évaluent à `true`, et `exec (intersection a1 a2) w` évalue à `false` si `exec a1 w` et `exec a2 w` évaluent à `false`.
8. (Difficile) Définir le type `'a nd_automaton` des automates finis *non-déterministes* sur l'alphabet de deux lettres `{A,B}`. On rappelle que un automate non-déterministe est "comme" un automate déterministe où la fonction de transition δ associe à un état et une lettre un *ensemble* finis des états et pas simplement un état.
   Écrire aussi la fonction `exec : 's nd_automaton -> alphabet list -> bool` qui calcule si un automate non-déterministe `a` accepte ou pas un mot `w`. C'est à dire `exec a w` évalue à `true` si une possible exécution de l'automate `a` sur le mot `w` termine sur un état final, et sinon `exec a w` évalue à `false`.