Devoir sur table

Novembre 2021

Documents autorisés: poly et notes de cours, notes de TD

L'épreuve durera 1 heure. Le sujet vaut 25 points plus 2 points de bonus. Si vous obtenez `n` points, votre note sera égale à `(min n 20)`.

Exercice I : Problèmes de croissance

On suppose dans tout cet exercice que l'on ne travaille qu'avec des entiers positifs ou nuls.

Q1 – (2pts) Une population double sa taille tous les ans.

Définir la fonction récursive de signature

```
taille (t:int) (n:int) : int
```

qui donne la taille de la population au bout de `n` années si la taille initiale de la population est `t`.

Exemples: `(taille 7 5)` vaut 224 | `(taille 5 1)` vaut 10
`(taille 13 0)` vaut 13

Q2 – (2pts) On considère encore une population dont la taille double chaque année.

Définir la fonction récursive

```
duree (t:int) (m:int) : int
```

qui donne le nombre d'années qu'il faut à une population de taille initiale `t` pour dépasser la taille `m`. On suppose ici que `t` est strictement positif.

Exemple: `(duree 7 224)` vaut 5 | `(duree 34 8)` vaut 0

1pt de bonus pour une définition récursive terminale.

Q3 – (3pts) On considère maintenant un modèle de croissance plus complexe. Chaque année la population croît de la moitié: $t_{n+1} = t_n + \frac{t_n}{2}$. Mais lorsque la population dépasse (strictement) une taille limite alors elle est divisée par 2. On fixe cette taille limite à **128**.

Si on considère une taille initiale de 42, l'évolution de la population est donnée par la suite $\begin{cases} t_0 = 42 \\ t_{n+1} = \frac{t_n}{2} & \text{si } t_n + \frac{t_n}{2} > 128 \\ t_{n+1} = t_n + \frac{t_n}{2} & \text{sinon} \end{cases}$

Pour simplifier, **on ne considère que des opérations sur les entiers**.

Définir la fonction récursive

```
taille2 (n:int) : int
```

qui donne le n-ième terme de cette suite.

Exemples: `(taille2 0)` vaut 42 | `(taille2 1)` vaut 63
`(taille2 2)` vaut 94 | `(taille2 3)` vaut 47
`(taille2 4)` vaut 70

Indication: en utilisant des définitions locales (avec `let in`) vous pouvez éviter de calculer plusieurs fois la même chose.

Solution Exercice I

Q1

```
let rec taille (t:int) (n:int) : int =
  if (n > 0) then (taille (2*t) (n-1))
  else t
```

Q2

```
let rec duree (t:int) (m:int) : int =
  if (t < m) then 1+(duree (2*t) m)
  else 0
```

Version récursive terminale

```
let duree_term (t:int) (m:int) : int =
  let rec loop (t:int) (res : int) : int =
    if (t < m) then loop (2*t) (res+1) else res
  in loop t 0
```

Q3

```
let rec taille2 (n:int) : int =
  if (n=0) then  42
  else let m = (taille2 (n-1)) in
     if (m+m/2 > 128) then m/2
     else m+m/2
```

Fin Solution

Exercice II : Listes bien balisées

Dans beaucoup de langages informatiques on utilise des symboles «ouvrants» et «fermants» pour structurer du texte. On utilise les parenthèses dans les expressions en Ocaml; on utilise des *balises* dans les langages XML ou HTML (par exemple les balises notées `<p>` et `</p>` délimitent un paragraphe).

Pour simplifier, on utilise les caractères `'<'` et `'>'` pour indiquer les balises (ou parenthèses) ouvrantes et fermantes. Pour simplifier encore, on considère des listes de caractères plutôt que des textes. Et pour simplifier complètement, on ne considère que des **listes qui ne contiennent que les caractères `'<'` et `'>'`**.

Une liste ℓ ne contenant que les caractères `'<'` (balise *ouvrante*) et `'>'` (balise *fermante*) est dite *bien balisée* si et seulement si elle vérifie les deux conditions suivantes :

1. ℓ contient exactement autant de balises ouvrantes que de balises fermantes
2. tout préfixe de ℓ contient un nombre de balises ouvrantes supérieur ou égal au nombre de balises fermantes

Rappel: Une liste `l1` est un préfixe d'une liste `l2` s'il existe une liste `l3` telle que `l1 @ l3 = l2`. Par exemple la liste `['b';'a']` est un préfixe de la liste `['b';'a';'t';'e';'a';'u']`. La liste vide est un préfixe de toutes les listes. Une liste est préfixe d'elle même.

Exemples de bons et mauvais balisages :

- `['<';'<';'>';'<';'>';'>';'<';'>']` est bien balisée;
- `['<';'>';'<';'>';'>';'<']` est mal balisée : la liste contient le même nombre de balises ouvrantes que de balises fermantes mais le préfixe `['<';'>';'<';'>';'>']` de cette liste contient plus de balises fermantes que de balises ouvrantes;
- `['<';'>';'<';'>';'<']` est mal balisée : tout préfixe de cette liste contient bien un nombre de balises ouvrantes supérieur ou égal au nombre de balises fermantes, mais la liste ne contient pas autant de balises ouvrantes que de balises fermantes.

À la fin de cet exercice, nous allons définir une fonction qui teste le bon balisage d'une liste.

Préfixes

Q1 – (2pts) Définir une fonction de signature

```
map_cons (e:'a) (l:('a list) list) : ('a list list)
```

qui étant donnés un elément `e` et une liste de listes `l` construit la liste de listes obtenue en ajoutant en tête de chaque liste de `l` l'élément `e`.

```
# (map_cons 3 []);;
- : int list list = []
# (map_cons 'x' [ [] ]);;
- : char list list = [ ['x'] ]
# (map_cons true [ [true;false;false]; [false;true]; [] ]);;
- : bool list list = [ [true; true; false; false]; [true; false; true]; [true] ]
```

Q2 – (3pts) Définir une fonction de signature

```
prefixes (l:'a list) : (('a list) list)
```

qui étant donnée une liste `l` construit la liste de toutes les listes correspondant à un préfixe de `l`.

```
# (prefixes []);;
- : 'a list list = [ [] ]
# (prefixes ['x']);;
```

```
- : 'a list list = [ []; ['x'] ]
# (prefixes ['h';'o';'u';'x']);;
- : char list list = [ []; ['h']; ['h'; 'o']; ['h'; 'o'; 'u']; ['h'; 'o'; 'u'; 'x'] ]
# (prefixes ['c';'h';'o';'u';'x']);;
- : char list list = [ []; ['c']; ['c'; 'h']; ['c'; 'h'; 'o']; ['c'; 'h'; 'o'; 'u']; ['c'; 'h';
    'o'; 'u'; 'x'] ]
```

Vous utiliserez la fonction `map_cons`. Notez que le résultat de `prefixes` n'est jamais une liste vide et que le résultat de `prefixes` a toujours pour premier élément une liste vide.

Nombre de balises

Q3 – (2pts) Définir les deux fonctions

```
add_fst (c:(int*int)) -> (int * int)
add_snd (c:(int*int)) -> (int * int)
```

telles que `(add_fst (n1,n2))` = `(n1+1, n2)` et `(add_snd (n1,n2))` = `(n1, n2+1)`.

Q4 – (4pts) Définir une fonction de signature

```
nb_of (l:char list) : int*int
```

qui calcule le couple d'entiers (n_1, n_2) tel que n_1 est le nombre de balises ouvrantes présentes dans la liste `l` et n_2 est le nombre de balises fermantes présentes dans la liste `l`. Cette fonction ne doit effectuer qu'un unique parcours de la liste `l` pour construire le couple. En revanche, vous pouvez utiliser les fonctions `add_fst` et `add_snd`, mais ce n'est pas obligatoire.

```
# (nb_of []);;
- : int * int = (0, 0)
# (nb_of ['<';'<';'>';'<';'>';'>';'<']);;
- : int * int = (4, 3)
```

1 pt de bonus pour une définition récursive terminale.

Bon balisage

Q5 – (2pts) Utilisez `nb_of` pour définir une fonction de signature

```
o_sup_f (l:char list) : bool
```

qui détermine si le nombre de balises ouvrantes présentes dans la liste `l` est supérieur ou égal au nombre de balises fermantes présentes dans la liste `l`.

```
# (o_sup_f ['<';'<';'>';'<';'>';'>';'<']);;
- : bool = true
# (o_sup_f ['<';'<';'>';'>']);;
- : bool = true
#  (o_sup_f ['<';'>';'<';'>';'>']);;
- : bool = false
```

Q6 – (3pts) Définir une fonction de signature

```
all_o_sup_f (l:(char list) list) : bool
```

qui détermine si toutes les listes d'une liste de listes `l` contiennent un nombre de balises ouvrantes supérieur ou égal au nombre de balises fermantes.

```
# (all_o_sup_f []);;
- : bool = true
# (all_o_sup_f [['<';'<';'>';'>']; ['<';'<';'>';'<';'>';'>';'<']]);;
- : bool = true
```

```
# (all_o_sup_f [['<';'<';'>';'>']; ['<';'>';'<';'>';'>']; ['<';'<';'>';'<';'>';'>';'<']]);;
- : bool = false
```

Q7 – (2pts) Déduire de ce qui précède une fonction de signature

```
dyck (l:char list) : bool
```

qui détermine si une liste ne contenant que des caractères `'<'` et `'>'` est bien balisée

```
# (dyck ['<';'<';'>';'<';'>';'>';'<';'>']);;
- : bool = true
# (dyck ['<';'>';'<';'>';'>';'<']);;
- : bool = false
# (dyck ['<';'>';'<';'>';'<']);;
- : bool = false
```

Solution Exercice II

Q1

```
let rec map_cons (e:'a) (l:'a list list) : ('a list list) =
   match l with
      [] -> []
    | h::t -> (e::h) :: (map_cons e t);;
```

Q2

```
let rec  prefixes (l:'a list) : (('a list) list) =
    match l with
      [] -> [[]]
    | h::t -> ([] :: (map_cons h (prefixes t)));
```

Q3

```
let add_fst (c:int*int) : (int * int) =
  match c with
      (a,b) -> (a+1,b)

let add_snd (c:int*int) : (int * int) = let (n1,n2)=c in (n1,n2+1)
```

Q4

```
let rec nb_of (l : char list) : int * int =
  match l with
  [] -> (0,0)
  | x::xxs -> if (x = '<') then (add_fst (nb_of xxs)) else (add_snd (nb_of xxs))
```

Version récursive terminale

```
let nb_of_term (l : char list) : int * int =
  let rec loop (l : char list) (res : int * int) : int * int =
    match l with
    [] -> res
    | x::xxs -> if (x='<') then (loop xxs (add_fst res)) else (loop xxs (add_snd res))
  in loop l (0,0)
```

Q5

```
 let o_sup_f (l : char list) : bool =
  let (bo,bf) = (nb_of l) in bo >= bf
```

Q6

```
let rec all_o_sup_f (l:(char list) list) : bool =
  match l with
      [] -> true
    | h::t -> (o_sup_f h) && (all_o_sup_f t);;
```

Q7

```
let dyck (l:char list) : bool =
 let (no,nf) = (nb_of l) in
     (no=nf) && (all_o_sup_f (prefixes l));;
```

Fin Solution