Lycée Chateaubriand
Informatique MP 2002/03
Mathieu Giraud

TP Caml n° 6 – Corrigé
Petits exercices et systèmes monétaires

1 Quatre exercices

1.1 Suites

a.

```
let rec calcul u0 f = function
  | 0 -> u0
  | n -> f (calcul u0 f (n-1)) ;;
```

On pourrait aussi écrire `calcul (f u0) f (n-1)`. Dans ce dernier cas, lors de l'appel au rang $n-1$ de `calcul`, il n'est plus utile de se "souvenir" de l'appel au rang n puisque l'appel récursif est la dernière instruction de `calcul`. On dit alors que la récurrence est *terminale*, et elle est équivalente à un algorithme itératif.

b. L'utilisation de `calcul` pour p appels successifs sur les entiers $n_1, n_2, ...n_p$ prend un temps de calcul en $\mathcal{O}(\sum_i n_i)$. En utilisant un tableau pour mémoriser une fois pour toutes les différents éléments de la suite, cette complexité baisse à $\mathcal{O}(\max n_i)$.

```
let précalcul u0 f nmax =
  let tab = make_vect nmax u0
  in for i = 1 to nmax do
        tab.(i) <- f (tab.(i-1)) ;
     done ;
     tab ;;
```

1.2 Arithmétique

c. L'algorithme d'Euclide utilise le fait que $a \wedge b = a \wedge (b \bmod a)$ tant que $b \bmod a$ n'est pas nul. L'opération $b \bmod a$ n'est "efficace" que si $a > b$ (dans le cas contraire, $b \bmod a = b$). Pour éviter un test de comparaison, on échange les arguments à chaque appel de `pgcd`.

```
let rec pgcd a b =
    if a = 0 then b
             else pgcd (b mod a) a ;;

let ppcm a b = (a * b) / (pgcd a b) ;;
```

d. La recherche de l'inverse d'un élément peut se faire par recherche brutale sur tous les éléments de $\mathbb{Z}/n\mathbb{Z}$.

```
let inverse n x =
   let res = ref 0
   in for i = 1 to (n-1) do
         if ((i * x) mod n = 1)
         then res := i ;
      done ;
      if (!res = 0)
      then failwith "Elément non inversible"
      else !res ;;
```

1.3 Analyse

e. Nous avons vu en TP que le choix d'un `epsilon` trop petit peut conduire à des résultats totalement faux. En effet l'évaluation de $f(x+\epsilon) - f(x)$ demande au minimum que $x + \epsilon$ soit différent que x, ce qui n'est pas le cas en machine si ϵ est trop petit.

```
let epsilon = 1e-6 ;;

let dérivée f x = ((f (x +. epsilon)) -. (f x)) /. epsilon ;;
```

L'évaluation de l'intégrale d'une fonction se fait par les sommes de Riemann

$$\int_a^b f(x)\,dx \quad \sim \quad \frac{b-a}{n} \sum_{i=1}^{n} f\left(a + i\,\frac{b-a}{n}\right)$$

Pour éviter des calculs inutiles, on calcule une fois pour toute la quantité $\epsilon = \frac{b-a}{n}$. De plus, au lieu d'effectuer n multiplications $i \cdot \epsilon$, on peut effectuer n additions $a_{i+1} = a_i + \epsilon$.

```
let n_riemann = 100000 ;;

let intègre f a b =
  let epsilon = (b -. a) /. (float_of_int n_riemann)
  and aire = ref 0.
  and ai = ref a in
     for i = 0 to n_riemann do
       aire := !aire +. (f !ai) ;
       ai := !ai +. epsilon ;
     done ;
     !aire *. epsilon ;;

intègre (fun x -> x) 6. 8. ;;
- : float = 14.00014


let primitive f = intègre f 0. ;;
```

1.4 Tours de Hanoi

f.

```
let affiche m n =
   print_string  (m ^ " -> " ^ n) ;
   print_newline () ;;
```

g. La solution des tours de Hanoi se contente d'effectuer des appels récursifs à elle-même en positionnant correctement à chaque fois les tours à utiliser.

```
let rec hanoi n départ intermédiaire arrivée =
    if (n>0) then (
        hanoi (n-1) départ arrivée intermédiaire ;
        affiche départ arrivée ;
        hanoi (n-1) intermédiaire départ arrivée
     );;

hanoi 3 "1" "2" "3" ;;

1 -> 3
1 -> 2
3 -> 2
1 -> 3
2 -> 1
2 -> 3
1 -> 3
- : unit = ()
#
```

2 Problème : systèmes monétaires et porte-monnaie

h.

```
let rec valeur = function
  | [] -> 0
  | t::q -> t + (valeur q) ;;
```

2.1 Payer gloutonnement le compte exact

i.

```
exception Echec_Glouton ;;

let rec glouton sys p = match (sys, p) with
  |    _, 0 -> []
  |   [], _ -> raise Echec_Glouton
  | t::q, _ -> if (t <= p)
               then t::(glouton sys (p-t))
               else     glouton  q    p   ;;
```

Par hypothèse, la dernière dénomination d_n vaut 1. À sa dernière étape, l'algorithme glouton pourra toujours utiliser p fois cette dernière espèce pour payer le prix p. Donc la stratégie gloutonne réussit toujours (et l'exception `Echec_Glouton` ne peut pas arriver).

j. La fonction `paye_glouton` ne diffère de la précédente que par le fait qu'on ne s'autorise pas à réutiliser l'espèce en cours.

```
let rec paye_glouton pf p = match (pf, p) with
  |    _, 0 -> []
  |   [], _ -> raise Echec_Glouton
  | t::q, _ -> if (t <= p)
               then t::(paye_glouton q (p-t))
               else     paye_glouton q   p  ;;
```

Cette fois-ci, il peut y avoir échec de la stratégie bien que l'acheteur dispose de suffisament d'argent pour payer le prix exact (comme avec le portefeuille $\langle 50, 20, 20, 20\rangle$ pour payer 60 euros).

2.2 Payer le compte exact de toutes les manières possibles

k.

```
let rec compte_paiements pf p = match (pf, p) with
  |    _, 0 -> 1
  |   [], _ -> 0
  | t::q, _ -> let z = compte_paiements q p
               in if (t <= p)
                  then z + compte_paiements q (p-t)
                  else z  ;;
```

2.3 Payer le compte exact et optimal

l. Dans lne système $\langle 240, 60, 30, 24, 12, 6, 3, 1\rangle$, le prix 48 a pour représentant glouton $\langle 30, 12, 6\rangle$ et pour représentant optimal $\langle 24, 24\rangle$.

m. La méthode proposée se formalise par

$$T(i+1) \;=\; \{\, c = a + b \mid a \in T(i) \,\wedge\, b \in P \backslash M_i(a) \,\} \;\;\setminus\;\; \cup_{0 \le j \le i}\, T(j)$$

Pour écrire le code de `étape`, on suppose que les fonction `ajoute` et `diff` qui gèrent l'union et la différence de listes triées sont programées.

```
let rec teste_et_ajoute portefeuille pmax a b0 ti = match b0 with
  | [] -> ti
  | b::q -> let z = teste_et_ajoute portefeuille pmax a q ti
            and c = a + b
            in if (c <= pmax) && (tab.(c) == []) then begin
                 tab.(c) <- ajoute tab.(a) b  ;
                 c::z ;
               end
               else z ;;

let rec étape portefeuille pmax t = match t with
  | [] -> []
  | a::q -> teste_et_ajoute portefeuille pmax a
              (diff portefeuille tab.(a)) (étape portefeuille pmax q) ;;
```

n. La fonction `précalcul_optimal` se contente d'initialiser le tableau `tab` puis d'appeler autant de fois que nécessaire la fonction `étape`.

```
let précalcul_optimal portefeuille pmax =
  for i = 0 to (vect_length tab) - 1
     do
       tab.(i) <- [] ;
     done ;
  let ti = ref [0]
  in for i = 1 to pmax
     do
       ti := étape portefeuille pmax !ti ;
     done ;;

let paye_optimal p = tab.(p) ;;
précalcul_optimal [ 10; 5; 2; 2; 2 ] 20 ;;
tab ;;
- : int list vect =
 [|[]; []; [2]; []; [2; 2]; [5];
   [2; 2; 2]; [5; 2]; []; [5; 2; 2]; [10];
   [5; 2; 2; 2]; [10; 2]; []; [10; 2; 2]; [10; 5];
   [10; 2; 2; 2]; [10; 5; 2];  []; [10; 5; 2; 2]; []|]
```

La suite

- Prochaine quinzaine → stratégies min/max