0 Mise en route

Lancer le programme WinCaml. Il propose deux fenêtres : celle de *gauche* vous permet de saisir vos programmes, celle de droite vous permet de visualiser vos résultats.
Les retours à la ligne servent juste à présenter le texte de façon plus lisible. Pour exécuter le calcul d'une expression il faut presser sur `[Ctrl]+[Entrée]` (et il faut bien sûr que cette expression se termine par `;;`
Dans la fenêtre de gauche, taper le petit programme suivant :

```
let a = 3 in
  a+5
;;
```

Ensuite presser sur `[Ctrl]+[Entrée]` et observez le résultat dans la fenêtre de droite. Normalement vous devez voir apparaître :

```
# let a = 3 in a+5;;
- : int = 8
```

Vérifiez que l'interpréteur est bien Ocaml et pas Camllight (sinon passez en Ocaml en allant dans le menu CamlTop).

1 Le langage Ocaml

1) Chacune des syntaxes suivantes est incorrecte. La corriger (à chaque fois, la valeur de l'expression corrigée est donnée). Bien tester vos expressions.

   a)
   ```
   let mot1 = "Bonjour " in
   let mot2 = "et au revoir" in
   let concat = mot1+mot2 in
   print_string concat
   ;;
   ```
   On doit trouver une expresion de type `unit` dont l'évaluation provoquera l'affichage de "Bonjour et au revoir".

   b)
   ```
   let a = 1515 in # Annee de Marignan
   let b = a**2 in
   b - 1;;
   ```
   On doit trouver 2 295 224 de type `int`.

   c) Le but est de calculer les termes d'une suite (A_n) définie par $A_0 = 1$ et $\forall n \in \mathbb{N}, A_{n+1} = \sin(A_n)$.

   ```
   def A (n) :
       if n=0 then 1
       else sin(A(n-1))
   ;;
   ```

Test : on doit trouver une fonction de type `int` → `float` et l'appel A_5 vaut (environ) 0.58718

2) a) Définir la fonction `f` : `int` → `int` définie par $\forall n \in \mathbb{Z}, f(n) = n^8$. On pourra éventuellement utiliser l'opérateur d'exponentiation `**` (attention : on rappelle que cet opérateur agit sur des `float`).
Vérifier que l'appel `f 2` renvoie bien 256.

b) Si on pose $a = n * n$ et $b = a * a$ alors $f(n) = b * b$: en utilisant ce principe et des définitions locales, définir `f` de façon à ce qu'elle soit non récursive, et que son calcul ne nécessite que 3 multiplications, uniquement entre entiers.

2 Récursivité

3) Suite récurrente d'ordre 1

On définit la suite u par $u_0 = 1$ et $\forall n \in \mathbb{N}, u_{n+1} = 3u_n - 1$.

Définir une fonction récursive `u` : `int` → `int` qui permet de calculer u_n.

Test : $u_{15} = 7\,174\,454$

4) Somme finie, type `float`

Écrire une fonction récursive `h` : `int` → `float` qui permet de calculer $h_n = \sum_{k=1}^{n} \frac{1}{k}$.

Indication : quelle formule de récurrence simple vérifie la suite (h_n) ?

Test : $h_{15} \simeq 3.3182$

5) Calcul des coefficients binomiaux

On peut calculer pour tous $k, n \in \mathbb{N}$ le coefficient binomial $\binom{n}{k}$ avec la formule :

$$\binom{n}{k} = \begin{cases} 0 & \text{si } k > n \\ 1 & \text{si } k = 0 \\ \frac{n}{k}\binom{n-1}{k-1} & \text{sinon} \end{cases}$$

Écrire une fonction `b` : `int` → `int` → `int` qui permet de calculer des coefficients binomiaux avec ce principe. Test : $\binom{15}{8} = 6\,435$.

6) Un exemple d'explosion combinatoire

On définit la suite de Fibonacci par $f_0 = 0$, $f_1 = 1$ et $\forall n \geqslant 2, f_n = f_{n-1} + f_{n-2}$.

Écrire le programme suivant, qui calcule récursivement f_n :

```
let rec f n =
    if n < 2 then n
    else (f (n-1)) + (f (n-2))
;;
```

Tester le calcul de `f 30` puis celui de `f 40`. Que constate–t–on ? Le problème, c'est que chaque appel de la fonction `f` génère deux appels récursifs, d'où une progression géométrique du nombre d'appels récursifs. Nous étudierons en TD plus en détail ce problème.

7) Exponentiation rapide (algorithme à retenir...)

a) Soient $a \in \mathbb{Z}$ et $n \in \mathbb{N}$. Il est possible de calculer a^n avec un nombre restreint de multiplications en utilisant la remarque suivante :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ (a^2)^{n/2} & \text{si } n \text{ pair} \\ a.(a^2)^{(n-1)/2} & \text{si } n \text{ impair} \end{cases}$$

Par exemple, $a^{11} = a.(a^2)^5 = a.(a^2.(a^2)^2)$ ce qui peut se calculer avec seulement 4 multiplications (le calcul de $a^2 = a * a$ puis $(a^2)^2$ puis $a^2(a^2)^2$ puis $a.(a^2.(a^2)^2)$).

Exploiter cette propriété, écrire une fonction (récursive) `expo` : `int` $\to$ `int` $\to$ `int` qui calcule rapidement a^n.

Test : $3^5 = 243$

b) Si on exploite la propriété $a^n = \begin{cases} 1 & \text{si } n = 0 \\ (a^{n/2})^2 & \text{si } n \text{ pair} \\ a.(a^{(n-1)/2})^2 & \text{si } n \text{ impair} \end{cases}$ on obtient une autre fonction (qui cependant calcule la même chose). Écrire et tester cette nouvelle fonction.

8) Composée itérée

a) Taper la définition suivante (elle permet, si f et g sont des fonctions, de calculer $f \circ g$) :

```
let compose f g =
let h x = f (g x) in h
;;
```

Observer le type de `compose` (c'est `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`) Comment expliquez–vous ceci ?

Indication : `'a`, `'b`, `'c`... désignent des types quelconques.

b) Pour toute fonction `f` d'un ensemble vers lui–même on définit sa *composée itérée* $\texttt{f}^{\texttt{n}}$ par $\texttt{f}^{0} = \texttt{Id}$ et $\forall n \in \mathbb{N}, \texttt{f}^{\texttt{n+1}} = \texttt{f} \circ \texttt{f}^{\texttt{n}}$.

Écrire une fonction `compiter:('a -> 'a) -> int -> 'a -> 'a` telle que pour toute fonction `f:'a->'a` et pour tout `n` $\in \mathbb{N}$, l'appel `compiter f n` renvoie $\texttt{f}^{\texttt{n}}$.

Test : rentrez l'expression ci–dessous.

```
let a x = 2*x-1 in
let b = compiter a 3 in
b 2;;
```

Avec cette définition, $b : x \mapsto 2(2(2x-1)-1)-1 = 8x-7$, on doit trouver $b(2) = 9$.

9) Somme des termes d'une suite

On considère la suite u vue ci–dessus définie par $u_0 = 1$ et $\forall n \in \mathbb{N}, u_{n+1} = 3u_n - 1$.

Écrire une fonction qui calcule $s_n = \sum_{k=1}^{n} u_k$.

Indication : si on veut une complexité linéaire, on pourra utiliser une fonction récursive auxiliaire de type `a` : `int` → `int` → `int` telle que pour tous $0 \leqslant k \leqslant n$, l'appel :

$$\texttt{a}\ u_{n-k}\ k$$

calcule s_n. Le cas de base est obtenu en prenant $k = 0$; et l'appel initial se fait en prenant $k = n$ (c'est–à–dire que l'appel initial est `a 1` n).

Test : $s_{15} = 10\,761\,688$.

3 Pour aller plus loin...

10) L'échelle de Montgomery

a) Il est possible de calculer le nombre de chiffres (en base 2) d'un entier naturel n grâce au programme Python suivant :

```
def nb_chiffres (n) :
    p = 0
    while n > 0 :
        p += 1
        n //= 2
    return p
```

Écrire une fonction OCaml `nb_chiffres` : `int` → `int` qui calcule la même chose (bien sûr, on n'a pas le droit à des boucles ou des modifications de variables : on doit utiliser la récursivité).

b) Pour tous $i, n \in \mathbb{N}$ soit $d_i(n) \in \{0, 1\}$ le i–ième chiffre de la décomposition de n en base 2 (ainsi $n = \sum_{i \geqslant 0} d_i(n) 2^i$, cette somme étant finie car la suite $(d_i(n))_{i \in \mathbb{N}}$ est stationnaire à 0).

Écrire une fonction OCaml `chiffre` : `int` → `int` → `int` tel que l'appel `chiffre n i` renvoie $d_i(n)$.

c) On considère le programme en Python suivant :

```
def montgomery (a, n) :
    a1 = a
    a2 = a*a
    p = nb_chiffres (n)
    for i in range (p-2, -1, -1) :
        if chiffre (n, i) == 0 :
            a2 = a1*a2
```

```
            a1 = a1*a1
        else :
            a1 = a1*a2
            a2 = a2*a2
    return (a1)
```

On pourrait montrer (on ne demande pas de le justifier) que cette fonction calcule $\texttt{a}^{\texttt{n}}$.

Écrire une fonction OCaml : `int` $\to$ `int` qui effectue le calcul de $\texttt{a}^{\texttt{n}}$ en utilisant le même principe (bien sûr, on n'a pas le droit à des boucles ou des modifications de variables : on doit utiliser la récursivité).

d) Prouver que la fonction `montgomery` que vous venez d'écrire permet bien de calculer $\texttt{a}^{\texttt{n}}$ (pour faire une preuve avec une fonction récursive, on n'utilise pas d'invariant de boucle, mais on peut quand même prouver des invariants vrais d'un appel récursif à l'autre).

11) Un entier est un palindrome si son écriture en base 10 est la même qu'on la lise dans un sens ou dans l'autre. Par exemple 17471 est un palindrome.

Écrire une fonction récursive `palindrome:int->bool` permettant de tester si un entier positif est un palindrome ou non.

12) Le problème du sac à dos

On dispose d'un nombre n d'objets numérotés de 0 à $n-1$, et d'un sac à dos qui ne craquera pas tant que ce qu'on met dedans ne dépasse pas une masse critique p_max. Les valeurs de n et p_max sont stockées dans des variables globales.

À chaque objet est associé une valeur. On souhaite mettre dans le sac à dos des objets de façon à maximiser la somme de leur valeur, sans que le sac ne craque. Les poids et les valeurs de chaque objet sont disponibles grâce à des fonctions `poids` : `int` $\to$ `int` et `valeur` : `int` $\to$ `int`. Les valeurs et les poids sont tous strictement positifs.

Par exemple on pourra utiliser les définition suivantes :

```
let n=5;;

let poids_max = 500;;

let poids k =
  if k=0 then 150
  else if k=1 then 223
  else if k=2 then 312
  else if k=3 then 175
  else 360
;;

let valeur k =
  if k=0 then 47
```

```
  else if k=1 then 62
  else if k=2 then 75
  else if k=3 then 51
  else 81
;;
```

On renverra le résultat sous forme d'un couple formé d'une entier correspondant au poids des objets transportés et d'un entier correspondant à la valeur des objets emportés. Avec l'exemple ci–dessus on doit trouver le couple $(487, 126)$ (avec les objets 2 et 3). Bien sûr votre programme doit pouvoir traiter des cas différents de celui de l'énoncé, avec des entiers `n` et `poids_max` quelconques et des fonctions `poids` et `valeur` quelconques. Le "jeu", c'est d'écrire ça sans utiliser plus de notions que celles qu'on a déjà vues (en particulier sans utiliser de listes ou de tableaux, que nous verrons dans des chapitres ultérieurs...)