DS 1 du 15 mars 2024 - 2 heures

Calculatrice, téléphone et documents interdits

Les programmes non commentés ou dont le fonctionnement n'est pas expliqué ne seront pas relus. Une présentation raisonnablement aérée (avec une identation convenable des structures de choix ou de répétition), ainsi que des résultats soulignés ou encadrés, sont des éléments de l'évaluation... C'est vous qui voyez...

Exercice 1 La touche  du clavier est cassée... vous ne pouvez pas utiliser ce symbole dans cet exercice...

1. Écrire une fonction `produit` de type `int -> int -> int` qui reçoit 2 entiers **naturels**.
2. En déduire une fonction `puissanceN : int -> int -> int` qui reçoit deux entiers naturels non simultanément nuls
3. En déduire enfin une fonction `puissance : int -> int -> int` telle que `puiss a n` renvoie $\texttt{a}^{\texttt{n}}$ pour $(\texttt{a},\texttt{n}) \in \mathbb{Z} \times \mathbb{N}$ et $(\texttt{a},\texttt{n}) \neq (0,0)$.

Exercice 2 Le **FizzBuzz** est un test de compétences utilisé lors des entretiens d'embauche en informatique qui permet d'éliminer les candidats les plus faibles. Il s'agit d'écrire une fonction qui affiche les entiers compris entre 1 et `n` donné en paramètre (`n` > 1), mais où les multiples de 3 sont remplacés par `"Fizz"`, les multiples de 5 par `"Buzz"` et les multiples à la fois de 3 et de 5 par `"FizzBuzz"`. Par exemple :

```
# fizzbuzz 20;;
1 ; 2 ; Fizz ; 4 ; Buzz ; Fizz ; 7 ; 8 ; Fizz ; Buzz ; 11 ; Fizz ; 13 ; 14 ;
FizzBuzz ; 16 ; 17 ; Fizz ; 19 ; Buzz ; - : unit = ()
```

Exercice 3 [**Cadeau de la maison**]

```
let rec truc a = function
        | 1 -> a
        | b when b mod 2 = 0 -> 2 * truc a (b/2)
        | b -> a + truc a (b-1)
;;
```

1. Quel est le type (la signature) de la fonction `truc` ?
2. Que renvoie `truc 5 4` et `truc 10 15` ? (Détailler les étapes)
3. Que peut-on conjecturer pour `truc a b` ?
4. Démontrer ce résultat en effectuant une récurrence sur `b`.
5. Que se passe-t-il si on tape `truc 5 0` ? Corriger ce problème.

Exercice 4 On considère la fonction `f` définie en OCaml par :

```
let rec f = function
        | 0 -> 0
        | n when n*n mod 2 = 0 -> n + f (n-1)
        | n -> f (n-1)
;;
```

1. Quel est type de la fonction `f` ?
2. Calculer `f 7`, en détaillant les calculs
3. Que se passe-t-il si on souhaite calculer `f (-2)` ?
4. Que se passe-t-il si dans le $2^{\text{ème}}$ filtrage, on remplace `n*n mod 2 = 0` par `n mod 2 = 0` ?
5. En déduire l'écriture mathématique de $\texttt{f}(n)$ sous forme d'une somme pour $n \in \mathbb{N}$.
6. En déduire alors une déclaration non récursive de `f` qui n'utilise pas non plus de boucle.

Exercice 5 Sur les listes :

1. Quelques questions indépendantes :
   a. Écrire une fonction `pair : int -> bool` qui indique si un entier est pair ou non.
   b. Écrire une fonction `unite : int -> int` qui renvoie le chiffre des unités d'un entier naturel donné.
   c. En déduire une fonction `unichiffre : int -> bool` qui indique si un entier naturel n'est écrit qu'à l'aide d'un seul chiffre (par exemple 4 ; 11 ; 333 ; ....).
2. a. Écrire une fonction `tous : ('a -> bool) -> 'a list -> bool` qui, prenant comme arguments une fonction à valeurs booléennes f et une liste ℓ, renvoie `true` si $f(x)$ est vrai pour tout x de ℓ, et `false` sinon. Par exemple :

```
# tous pair [2;3;4;6];;
- : bool = false
# tous unichiffre [1;22;333;6666];;
- : bool = true
```

   b. En déduire une fonction `au_moins` avec la même signature, mais qui indique si au moins un élément x vérifie $f(x)$. Par exemple :

```
# au_moins unichiffre [23; 44; 56];;
- : bool = true
# au_moins pair [3;7;11;31];;
- : bool = false
```

Correction 1

1. On se base sur le fait que $a \times 0 = 0$ et $a \times b = a + a \times (b-1)$ si $b \in NN^*$.

```
let rec produit a = function
          0 -> 0
        | b -> a + (produit a (b-1))
;;
```

2. Même idée : $a^0 = 1$ et $a^n = a \times a^{n-1}$ si $n \in NN^*$.

```
let rec puissanceN a = function
          0 -> 1
        | n -> produit a (puissanceN a (n-1))
;;
```

3. Il faut faire attention aux boucles infinies et ne faire des puissance qu'avec des entiers naturels, une solution :

```
let puissance a n =
        if a > 0
        then puissanceN a n
        else if n mod 2 = 0
                  then puissanceN (-a) n
                  else -puissanceN (-a) n
;;
```

Une autre plus courte utilisant la valeur absolue :

```
let puissance a n =
        if a > 0 || n mod 2 = 0
        then puissanceN (abs a) n
        else -puissanceN (abs a) n
;;
```

Encore un peu plus court :

```
let puissance a n =
   let r = puissanceN (abs a) n in if a > 0 || n mod 2 = 0 then r else -r
;;
```

Correction 2 Pas de difficulté particulière pour cet exercice, un filtrage et une fonction récursive feront l'affaire (attention à l'ordre des filtrages et des affichages !) :

```
let rec fizzbuzz = function
          0 -> ();
        | n when n mod 15 = 0 -> fizzbuzz (n-1); print_string "FizzBuzz␣;␣";
        | n when n mod 3 = 0 -> fizzbuzz (n-1); print_string "Fizz␣;␣";
        | n when n mod 5 = 0 -> fizzbuzz (n-1); print_string "Buzz␣;␣";
        | n -> fizzbuzz (n-1); print_int n; print_string "␣;␣";
;;
```

Correction 3 1. `truc : int -> int -> int`

2. `truc 5 4` renvoie 20, en effet : `truc` $5\ 4 = 2 \times$ `truc` $5\ 2 = 2 \times 2 \times$ `truc` $5\ 1 = 2 \times 2 \times 5 = 20$
`truc 15 10` renvoie 150, en effet :
$$\begin{aligned}\texttt{truc}\ 10\ 15 &= 10 + \texttt{truc}\ 10\ 14 = 10 + 2 \times \texttt{truc}\ 10\ 7 = 10 + 2 \times \big(10 + \texttt{truc}\ 10\ 6\big)\\ &= 10 + 2 \times \big(10 + 2 \times \texttt{truc}\ 10\ 3\big) = 10 + 2 \times \big(10 + 2 \times (10 + \texttt{truc}\ 10\ 2)\big)\\ &= 10 + 2 \times \big(10 + 2 \times (10 + 2 \times \texttt{truc}\ 10\ 1)\big) = 10 + 2 \times \big(10 + 2 \times (10 + 2 \times 10)\big)\\ &= 10 + 2 \times \big(10 + 2 \times 30\big) = 10 + 2 \times 70 = 150\end{aligned}$$

3. On peut donc conjecturer que `truc a b` renvoie `a` $\times$ `b`

4. Pour $n \in \mathbb{N}$, notons $\mathcal{P}_n$: « pour tout entier `a`, `truc a` n vaut `a`$\times n$ »
   - pour tout entier `a`, `truc a 1` renvoie `a`, donc $\mathcal{P}_1$ est vraie.
   - Supposons pour $n \geqslant 1$ fixé, $\mathcal{P}_1, ..., \mathcal{P}_n$ soient vraies.
     - ◇ Si $n+1$ est pair, posons $n+1 = 2p$, alors `truc a (n+1)`=2×`truc a` p. Or $p = \dfrac{n+1}{2} \leqslant n$, donc $\mathcal{P}_p$ est vraie. Ainsi : `truc a (n+1)`=2×`truc a` p=2 × (`a` × p) = $a \times (2p) = a \times (n+1)$ et $\mathcal{P}_{n+1}$ est vraie.
     - ◇ Si $n+1$ est impair, alors `truc a (n+1)`=`a` + `truc a` n= `a` + `a`×n=`a`×$(n+1)$ et $\mathcal{P}_{n+1}$ est vraie aussi.

   Dans tous les cas $\mathcal{P}_{n+1}$ est vraie. Et par le principe de récurrence forte, $\forall n \in \mathbb{N}^*,\ \mathcal{P}_n$ est vraie.

5. Si $b = 0$, on entre dans une boucle infinie, on peut modifier ainsi :

```
let rec truc a = function
        | 0 -> 0
        | 1 -> a
        | b when b mod 2 = 0 -> 2 * truc a (b/2)
        | b -> a + truc a (b-1)
;;
```

En fait, le filtrage dans le cas où le second paramètre vaut 1 n'est pas utile :

```
let rec truc a = function
        | 0 -> 0
        | b when b mod 2 = 0 -> 2 * truc a (b/2)
        | b -> a + truc a (b-1)
;;
```

Correction 4 1. La fonction `f` est de type `int -> int`.

2. $f(7) = f(6) = 6 + f(5) = 6 + f(4) = 6 + 4 + f(3) = 6 + 4 + f(2) = 6 + 4 + 2 + f(1) = 6 + 4 + 2 + f(0)$
$= 6 + 4 + 2 + 0 = 12$.

3. Si on tape `f(-2)`, on entre dans une boucle infinie.

4. Cela ne change rien, car : n^2 est pair $\iff n$ est pair.

5. Finalement pour $n \in \mathbb{N}$, `f`(n) calcule et renvoie donc la somme de tous les nombres pairs de 0 à n :
$$\texttt{f}(n) = \sum_{\substack{k=0\\ k \text{ pair}}}^{n} k = \sum_{k=0}^{\lfloor n/2 \rfloor} 2k$$

6. En continuant de simplifier, pour $n \in \mathbb{N}$,

$$\mathtt{f}(n) = \sum_{k=0}^{\lfloor n/2 \rfloor} 2k = 2 \sum_{k=0}^{\lfloor n/2 \rfloor} k = 2 \times \frac{\lfloor n/2 \rfloor \times \big(\lfloor n/2 \rfloor + 1\big)}{2} = \lfloor n/2 \rfloor \times \big(\lfloor n/2 \rfloor + 1\big)$$

D'où la nouvelle déclaration pour $n \in \mathbb{N}$:

```
let f n = let p = n / 2 in p*(p+1);;
```

Correction 5

1. Pas de difficulté pour ces 3 premières questions :

```
let pair n = n mod 2 = 0;;

let unite n = n mod 10;;

let rec unichiffre = function
        n when n < 10 -> true;
        | n -> let m = n /10 in (unite n = unite m) && unichiffre m
;;
```

2. Les 2 suivantes (pour la seconde, on a utilisé la négation) :

```
let rec tous f = function
          [] -> true
        | h::q -> (f h) && (tous f q)
;;

let au_moins f l = not( tous (function x -> not(f x) ) l);;
```