Programmation fonctionnelle 2024-2025

Feuille 2 : Premières fonctions, fonctions récursives

Exercice 2.1 1. Écrire une fonction `test` qui prend en arguments trois entiers `x`, `y`, `z` et retourne `true` si `z` est la somme de `x` et `y`, et `false` sinon.

2. Utiliser la fonction `test` pour les valeurs 1, 2, 3, puis 2, 3, 4 des arguments `x`, `y`, `z`.

Exercice 2.2 Écrire une fonction `valeur_p4` qui prend en argument un entier x et retourne $3x^4 + 7x - 1$.

Exercice 2.3 Écrire une fonction `polynome3 a b c` qui prend en arguments trois entiers `a, b, c` et retourne la fonction polynôme $x \mapsto ax^2 + bx + c$. Donner des exemples d'appels de la fonction retournée par `polynome3`

Exercice 2.4 Soit la fonction `fact` définie comme suit :

```
let rec fact n =
if n = 0 then 1 else n * fact (n-1)
```

Dessiner la pile des appels pour `fact 5`.

Exercice 2.5 Soient n, p deux entiers. On rappelle que $\text{pgcd}(n, 0) = n$ et que $\text{pgcd}(n, p) = \text{pgcd}(p, n \mod p)$. Écrire une fonction `pgcd` calculant le pgcd de deux entiers.

Exercice 2.6 La fonction d'Ackermann est un exemple de fonction à croissance **très** rapide. Elle est définie par

$$ack(m, n) = \begin{cases} n+1 & \text{si } m = 0 \\ ack(m-1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ ack(m-1, ack(m, n-1)) & \text{sinon} \end{cases}$$

Écrire un programme `ack` calculant cette fonction. **Attention à ne pas la tester sur de trop grands entiers.**

Exercice 2.7 La suite de Fibonacci est définie par $u_0 = u_1 = 1$ et pour tout $n \geqslant 2$, $u_n = u_{n-1} + u_{n-2}$.

1. Écrire une fonction `fib` calculant le n-ème terme de cette suite.
2. Combien de sommes sont utilisées lors de l'appel `fib n` ?
3. En utilisant une fonction auxiliaire qui calcule le couple (u_n, u_{n+1}), améliorer `fib` pour que l'appel `fib n` n'utilise qu'un nombre linéaire de sommes.

Exercice 2.8 Supposément posé par les recruteurs d'**Amazon**.

- https://youtu.be/5o-kdjv7FD0
- Une personne monte un escalier à n marches.
- Elle monte à chaque pas soit une, soit deux, soit trois marches.
- De combien de façons peut-elle monter l'escalier ?

Écrire une fonction `stairs` qui prend en argument un entier n et calcule le nombre de façons de gravir un escalier de n marches sachant qu'on peut choisir de monter une, deux ou trois marches à chaque pas.

Exercice 2.9 1. Écrire une fonction `power` qui prend en argument deux entiers b et n avec $b \neq 0$, et qui calcule b^n.

2. Combien de multiplications sont effectuées lors de l'appel `power b n` ?

3. Peut-on en utiliser moins ?

Exercice 2.10 Écrire une fonction `iterate` qui prend en argument une fonction f et un entier k et qui renvoie la fonction $x \mapsto \underbrace{f(f(\cdots f}_{k \text{ fois}}(x)\cdots)$. Quel est le type de cette fonction ?

Exercice 2.11 1. Écrire la fonction `multiplicateur` qui prend en paramètre un entier `n` et retourne la fonction de type `int -> int` qui à un entier `i` associe `n * i`.

$$\begin{array}{rccc} \mathit{multiplicateur}: & \mathbb{N} & \to & \mathbb{N} \mapsto \mathbb{N} \\ & n & \mapsto & i \mapsto n * i \end{array}$$

Exemples :

```
utop[2]> multiplicateur;;
- : int -> int -> int = <fun>
utop[3]> multiplicateur 10;;
- : int -> int = <fun>
utop[4]> multiplicateur 10 2;;
- : int = 20
utop[5]> multiplicateur 100 3;;
- : int = 300
```

Exercice 2.12 On considère la suite récurrente d'ordre 2 suivante :

$$\begin{cases} u_0 & = & 0 \\ u_1 & = & 3 \\ u_n & = & -u_{n-1} + 2u_{n-2}, \forall n \geqslant 2 \end{cases}$$

1. Écrire une fonction récursive `seq_aux` de type `int -> int * int` qui à tout entier naturel n associe le couple (u_n, u_{n+1}).

2. En déduire une fonction `seq` qui à tout entier naturel n associe u_n.

3. Quel est le type de `seq` ?

4. Quelle est la complexité de `seq` en fonction de son paramètre n ?

5. Rechercher sur internet (ou dans le cours d'algo des arbres) comment on peut obtenir une complexité logarithmique pour ce format de suite (à la Fibonacci) en utilisant des matrices et la multiplication "à la Grecque" pour multiplier les matrices.

Exemples :

```
# seq_aux;;
- : int -> int * int = <fun>
# seq_aux 0;;
- : int * int = (0, 3)
# seq_aux 1;;
```

```
- : int * int = (3, -3)
# seq_aux 2;;
- : int * int = (-3, 9)
# seq 3;;
- : int = 9
```

Exercice 2.13 Soit f une fonction de $\mathbb{R}$ dans $\mathbb{R}$.

Si f est dérivable, la fonction dérivée de f, f' peut être définie par

$$\begin{array}{rccc} f': & \mathbb{R} & \to & \mathbb{R} \\ & x & \mapsto & \lim_{h\to 0} \tau(f,h,x) \end{array}$$

où

$$\tau(f,h,x) = \frac{f(x+h) - f(x-h)}{2h}$$

En prenant un h petit (proche de 0), on obtient la fonction f'_h, dérivée approchée de f, définie par

$$\begin{array}{rccc} f'_h: & \mathbb{R} & \to & \mathbb{R} \\ & x & \mapsto & \tau(f,h,x) \end{array}$$

Écrire la fonction `derivee` qui prend en paramètre f, h et retourne f'_h. Exemples :

```
# epsilon;;
- : float = 1e-06
# derivee;;
- : (float -> float) -> float -> float -> float = <fun>
# derivee (fun x -> 4. *. x +. 3.) epsilon 10.;;
- : float = 3.99999999700639819
# derivee (fun x -> x *. x *. x +. 5.) epsilon 2.;;
- : float = 11.9999999999009788
```

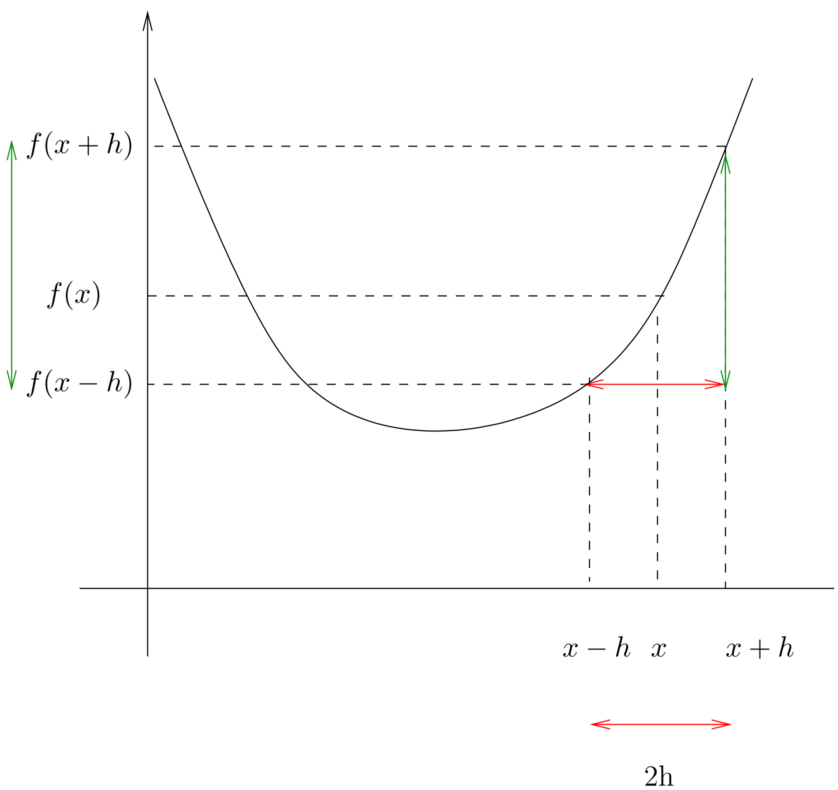
On s'intéresse maintenant à la dérivée $n$ème $f^{(n)}$ d'une fonction f qui peut être définie par

$$\left\{ \begin{array}{l} f^{(0)} = f \\ f^{(n)} = (f')^{(n-1)} \end{array} \right.$$

Écrire la fonction `derivee_n` qui prend en paramètre un entier n, f et h et retourne la fonction dérivée $n$ème (approchée) de f.

Exemples :

```
# derivee_n;;
- : int -> (float -> float) -> float -> float -> float = <fun>
# derivee_n 2 (fun x -> x *. x *. x +. 5.) epsilon 2.;;
- : float = 12.0015108961979422
```



$f(x+h)$
$f(x)$
$f(x-h)$
$x-h$
x
$x+h$
2h