Feuille d'exercices n°2

Cette feuille d'exercice porte sur les concepts de programmation et les éléments de langage du début de la section 2 des notes de cours («Programmation récursive», pp 18-20).

Travaux dirigés

Exercice I : Factorielle

La factorielle d'un nombre entier est définie comme :

$$\begin{cases} 0! & = & 1 \\ n! & = & n \times (n-1)! \quad \text{si } n > 0 \end{cases}$$

Q1 – Définir une fonction récursive `fact : int -> int` telle `(fact n)` donne la valeur de `n`!.

Exercice II : Sommes

Q1 – Définir la fonction de signature `sum_ (n:int) : int` qui calcule la somme des n permiers nombres entiers strictement positifs.

Par exemple: `(sum_n 5)` $= 5+4+3+2+1$ et `(sum_n 0)` $= 0$.

Q2 – En utilisant une fonction locale, redéfinir la fonction `sum_n` de manière à ce qu'elle déclenche l'exception `Invalid_argument "sum_n"` si `n` est négatif.

Q3 – Définir la fonction de signature `sum_p (n:int) : int` qui calcule la somme des n premiers nombre pairs positifs.

Par exemple: `(sum_p 4)` $= 8+6+4+2$.

Q4 – **Un peu plus difficile:** définir la fonction de signature `sum_f (f:int -> int) (n:int) : int` qui calcule la somme `(f n) + (f (n-1)) + ... + f(0)`.

Utiliser cette fonction pour redéfinir `sum_p`. Vérifiez c'est bien la *même fonction* que vous avez défini.

Exercice III : Termes d'une suite

Soit u_n définie par: $\begin{cases} u_0 & = & 42 \\ u_{n+1} & = & 3u_n + 4 \end{cases}$

Q1 – Donnez la définition de la fonction `u : int -> int` telle que `(u n)` donne $u_{\mathtt{n}}$. Quelle hypothèse faut-il faire sur `n` ?

Q2 – En utilisant la fonction `u`, donnez la définition de la fonction `sum_u` qui calcule $\Sigma_{i=0}^{n} u_n$, c'est-à-dire, la somme des $n+1$ premiers termes de la sute u_n.

Q3 – **Plus difficile:** définir `sum_u` sans utiliser la fonction `u`.

1. <u>Sans utiliser la fonction u</u>, définir une fonction de signature `loop (n:int) (t:int) : int` qui calcule la somme `t + u(t) + u`^{2}`(t) + ... + u`^{n}`(t)`.
   Indication: l'argument `t` est modifié à chaque itération : `3 * t + 4`.
2. En déduire une nouvelle définition de `sum_u`.
3. Faire de `loop` une définition locale pour définir `sum_u`.

Exercice IV : Récurrence sur un intervalle

Q1 – Donnez une définition de la fonction de signature `sum_inter (a:int) (b:int) : int` qui donne la somme des entiers compris dans l'intervalle $[\mathtt{a}, \mathtt{b}]$. Faut-il poser des hypothèses sur les arguments ? Élaborez un jeu de tests pour cette fonction.

Q2 – Définir la fonction de signature `sum1_inter (k:int) (a:int) (b:int) : int` qui donne la somme `(k+a) + (k+(a+1)) + .. + (k+b)`.

Q3 – Donnez la définition de la fonction de signature `sum2_inter (a:int) (b:int) : int` qui donne la somme de tous les couples d'entiers de l'intervalle $[\mathtt{a}, \mathtt{b}]$.

Par exemple: `(sum2_inter 2 4)` $= (2+2)+(2+3)+(2+4)+(3+2)+(3+3)+(3+4)+(4+2)+(4+3)+(4+4)$

Indication: on peut utiliser `sum1_inter`.

Travaux sur machines

Exercice V : Nombres premiers

Rappels: un nombre entier d est diviseur d'un nombre entier n si et seulement si il existe un nombre entier k tel que $n = d \times k$. On en déduit que $n \mod d = 0$.

Q1 – Donnez la définition de la fonction `less_divider : int -> int -> int` telle que `(less_divider i n)` donne le plus petit diviseur de `n` compris entre `i` (inclus) et `n` (exclu), s'il existe et 0 sinon. Quelles hypothèses faut-il poser pour `i` et `n` ?

Rappel: un nombre entier positif est dit *premier* si et seulement si il n'a pas de diviseur autre que 1 et lui-même. Par convention, on ne considère pas 1 comme un nombre premier.

Q2 – Déduire de la question précédente la définition de la fonction `prime : int -> bool` telle que `(prime n)` vaut `true` si et seulement si `n` est premier.

Q3 – Donnez la définition de la fonction `next_prime : int -> int` telle que `(next_prime n)` donne le plus petit nombre premier supérieur ou égal à `n`.

Cette fonction s'arrêtera-t-elle toujours ?

Q4 – On numérote les nombres premiers de cette manière: 2 a le numéro 0, 3 a le numéro 1, 5 a le numéro 2, 7 a le numéro 3, etc.

Déduire de la question précédente la définition de la fonction `nth_prime: int -> int` telle que `(nth_prime n)` donne le nombre premier de numéro `n`. On suppose `n` positif.

Exercice VI : Approximation de la racine carrée

On peut obtenir une valeur approchée de la racine carrée d'un nombre a en utilisant les termes de la suite

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Le choix de x_0 est arbitraire. par exemple, on peut prendre 1.

Q1 – Définir la fonction `f` telle que $f(x) = \frac{1}{2}\left(x + \frac{a}{x}\right)$. Quelle est le type de `f` ?

Q2 – Une première manière d'utiliser cette suite pour obtenir une valeur approchée de la racine carrée est de calculer le n-ième terme de la suite x_n.

Définir la fonction `sqrt_n (n: int) (a: float) (x0: float) : float` qui calcule le n-ième terme de la suite x_n en prenant `x0` comme valeur de x_0.

Utilisez cette fonction avec des valeurs croissantes de `n` et observez.

Point fixe

Le point fixe d'une fonction f et un x tel que $f(x) = x$. Cela donne une autre manière d'utiliser la suite des x_n pour déterminer la racine carrée: itérer le calcul de $x_0, x_1, x_2, etc.$ jusqu'à trouver un x_n tel que $x_n = x_{n-1}$. C'est un point fixe puisque qu'on aura que $x_n = (\texttt{f}\ x_{n-1}) = x_{n-1}$. On a alors atteint un *point fixe* et l'on obtiendra pas de meilleure approximation.

Attention: les calculs sur les flottants ne sont pas exacts. On testera donc l'égalité *"à epsilon près"*.

Q3 – Définissez la fonction `eq_eps (e: float) (x: float) (y: float) : bool` qui donne `true` si et seulement si la distance entre `x` et `y` est inférieure (strictement) à `e`.

La fonction de la bibliothèque standard `abs_float : float -> float` donne la valeur absolue de son argument.

Q4 – Définissez la fonction `sqrt_x (e: float) (a: float) (x0: float) : float` qui utilise la méthode du point fixe pour trouver une valeur approchée de la racine carrée de `a`. Le premier terme de la suite sera `x0`. On teste l'égalité *"à* `e` *près"*.

Utilisez cette fonction avec des valeurs décroissantes de `e` et observez.