Projet. Session 2. Documents autorisés. Durée 4h.

1 Flux et séries de Taylor

On va représenter et manipuler des séries de Taylor en 0 de fonctions à un paramètre. On n'étudiera pas les problèmes éventuels de convergence. À titre de rappel, voici deux séries classiques :

$$\frac{1}{1-x} = \sum_{i \in \mathbb{N}} x^i \qquad \log(1-x) = \sum_{i \in \mathbb{N}^*} \frac{x^i}{i}$$

On représentera une série comme le flux **infini** des coefficients associés aux monômes x^i. On donne ci-dessous une interface Iter du type abstrait Flux.

```
module type Iter =
sig
  type 'a t = 'a flux_t
  val cons : 'a -> 'a t -> 'a t
  val uncons : 'a t -> 'a * 'a t
  val apply : ('a -> 'b) t -> ('a t -> 'b t)
  val unfold : ('b -> 'a * 'b) -> ('b -> 'a t)
  val constant : 'a -> 'a t
  val filter : ('a -> bool) -> 'a t -> 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
  val print : (Format.formatter -> 'a -> unit) -> int -> Format.formatter -> 'a t -> unit
end
```

Les deux séries données en exemple seront respectivement représentées par les flux suivants :

```
cons 1. (cons 1. (cons 1. (...
cons 0. (cons 1. (cons 0.5 (cons 0.3333... (cons 0.25 (cons 0.2 (...
```

⚠ Rappel : on peut construire directement des flux récursifs, ou bien des fonctions récursives produisant un flux, comme montré dans le fichier `flux.ml` fourni. Pour éviter les boucles infinies, il faut penser à protéger le corps de chaque définition récursive par Tick (**lazy** (...)).

La fonction print affiche un certain nombre d'éléments successifs d'un flux. Ainsi l'expression suivante affiche les 10 premiers éléments d'un flux de nombres flottants sur la sortie standard :

```
Flux.print Format.pp_print_float 10 Format.std_formatter flux
```

Les fonctions pourront également être testées au travers d'une primitive de dessin :

```
trace_fonction : (float * float) -> (float -> float) -> unit
```

définie dans le module Dessin fourni. Le premier paramètre est l'intervalle $(xmin, xmax)$ dans lequel on trace la fonction passée en second paramètre. Un exemple d'utilisation des différentes fonctions demandées ainsi que l'affichage graphique est proposé à la fin du fichier `projet.ml` fourni.

⚠ Vous prendrez soin de résoudre les exercices dans le fichier `projet.ml` et de rendre ce fichier.

Exercice 1 (Flux élémentaires)

1. Définir la fonction constant : float -> float Flux.t renvoyant un flux infini représentant la série de Taylor d'une simple constante.
2. Définir le flux nats : float Flux.t des entiers successifs en partant de 1.
3. Définir var : float Flux.t, le flux infini représentant la série de Taylor du monôme x.

Exercice 2 (Évaluation)

1. Définir la fonction evalue d'évaluation d'une série en un point. La variable prend en paramètre un point `x : float` ainsi qu'un ordre `n : int` au-delà duquel les termes de la série sont ignorés.

Exercice 3 (Opérations linéaires) Définir les quatre fonctions suivantes :

1. somme et mult_const, respectivement addition de deux séries et multiplication d'une série par une constante.
2. derive et integre, respectivement dérivée et primitive (nulle en 0) d'une série donnée, en utilisant le flux nats.

On définit l'opération "shift" suivante (correspondant à `tail f`) : $\mathbf{S}(f) \triangleq \sum_{i\in\mathbb{N}} f_{i+1} * x^i$
La multiplication de $f = \sum_{i\in\mathbb{N}} f_i * x^i$ par g, notée $f \times g$, peut alors être décomposée ainsi :

$$f \times g = (f_0 * g) + x * \mathbf{S}(f) \times g$$

Exercice 4 (Multiplication)

1. Définir l'opération mult_var de produit d'une série par le monôme x.
2. À l'aide des fonctions précédentes, définir l'opération produit de multiplication entre deux flux, en suivant le schéma proposé.

La composition de $f(x) = \sum_{i\in\mathbb{N}} f_i * x^i$ par $g(x) = \sum_{i\in\mathbb{N}} g_{i+1} * x^{i+1}$, i.e. telle que $g(0) = 0$, notée $f \circ g$, peut être décomposée selon l'équation différentielle suivante :

$$\begin{aligned}(f \circ g)(x) &= (f \circ g)(0) + \int_x^x (f \circ g)'(x)dx \\ &= f(g(0)) + \int_x^x g'(x) \times (f' \circ g)(x)dx \\ &= f(0) + \int_x^x g'(x) \times (f' \circ g)(x)dx\end{aligned}$$

Ainsi, $f \circ g$ dépend de (l'intégrale de) $f' \circ g$, qui dépend de $f'' \circ g$, etc.

Exercice 5 (Composition)

1. Définir récursivement l'opération compose de composition entre deux séries f et g, tel que $g(0) = 0$, en suivant le schéma proposé.
2. En utilisant compose et la propriété $\exp(a+b) = \exp(a) * \exp(b)$, définir l'exponentielle d'une série quelconque (non nécessairement nulle en 0) [1].

Pour finir, la résolution d'équations différentielles, i.e. le calcul de la série de Taylor en 0 associée à la solution, est possible à travers la définition récursive de séries. Ainsi on peut définir la fonction $\exp(x)$ comme solution de : $y(x) = 1 + \int_x^x y(x)$, soit en OCAML :

```
let rec exp =
  Tick (lazy (Flux.uncons (
  somme (constant 1.0) (integre exp ))));;
```

⚠ Les équations différentielles ainsi définies récursivement n'ont de sens que si la variable définie apparaît toujours sous la portée d'un nombre d'intégrations strictement supérieur au nombre de dérivations.

L'utilisation de l'expression `Tick (lazy (Flux.uncons ...))` étant lourde, on propose de remplacer l'utilisation de la récursivité directe par la fonction fixpoint, telle que `fixpoint f` construit une série `s = (f s)`. Pour reprendre l'exemple ci-dessus, on aurait alors :

```
let exp = fixpoint (fun exp -> somme (constant 1.0) (integre exp));;
```

Exercice 6 (Équation différentielle ordinaire)

1. Définir l'opération fixpoint de définition d'un équation récursive.

1. On rappelle la série entière $\exp(x) = \sum_{i\in\mathbb{N}} \frac{x^i}{i!}$