Cours

- Récursivité, récursivité croisé. Arbre d'appels.
- Résoudre des équations sur la complexité
- Différence entre structure mutable et non mutable
- Structure de données abstraite = type muni d'opérations
- Polymorphisme (HP)
- Techniques avancées de fonctionnel (HP)

Je préviens (si tu commence par cette section) que les exercices ici (surtout vers la fin) sont particulièrement difficille. C'est des bon exos si un collé est trop fort ou pour une fin de DS, mais pas pour des gens qui débutent.

Mémoisation Automatique[1]

On s'intéresse à écrire une fonction dans un langage de programmation fonctionnel capable de prendre en entrée une fonction f et de renvoyer une version mémoïsée de f.

1. Écrire une fonction `val memo: ('a -> 'b) -> ('a -> 'b)` telle que si f est une fonction non récursive, alors `memo f` renvoie une version mémoïsée de f
2. Écrire une fonction `val memo_rec: ('a -> 'b) -> ('a -> 'b)` telle que si f est une fonction récursive, alors `memo f` renvoie une version mémoïsée de f. On suppose que les appels récursif sont tous de la forme `memo f`. Attention à préserver la structure des appels récursifs.
3. Écrire une fonction `val fibo: int -> int` qui calcule la fonction de Fibonacci avec une complexité en $O(n)$ en utilisant `memo_rec`.

[1] de Maxime BRIDOUX

Suspension et listes infinies

On se propose d'implémenter, en OCaml, une bibliothèque de calcul paresseux. Celle-ci expose un type paramétré de *suspensions* `'a susp` et des fonctions de types suivants:

- Une fonction `make: (unit -> 'a) -> 'a susp`
- Une fonction `force: 'a susp -> 'a`

Une suspension (de type `'a susp`) contient une fonction permettant de calculer une valeur de type `'a` quand cela sera nécessaire. Elle peut être construite facilement grâce à la fonction `make f`. Le calcul n'est effectué que lorsque l'utilisateur de la bibliothèque le demande via la fonction `force susp`. Cette dernière fonction vérifie si le calcul a déjà été effectué : si tel est le cas, elle en renvoie le résultat pré-calculé. Sinon, elle lance le calcul de `f ()`, stocke le résultat pour de futurs appels, et elle le renvoie.

2. Donner une implémentation possible de cette bibliothèque en OCaml. *On pourra utiliser des références.*

On définit le type des *listes paresseuses* par

```
type 'a slist_cell =
| SNil
| SCons of 'a * 'a slist
and 'a slist = 'a slist_cell susp
```

3. Comparer la notion de liste paresseuse avec la notion habituelle de liste.
4. Définir une fonction `scons: 'a -> 'a slist -> 'a slist` qui ajoute un élément en tête d'une liste paresseuse. Définir une valeur `snil` représentant la liste paresseuse vide.
5. Écrire deux fonctions `shd: 'a slist -> 'a` et `stl: 'a slist -> 'a slist` qui prennent une liste paresseuse non vide en paramètre, et qui renvoient respectivement son premier élément et la liste paresseuse des autres éléments
6. Écrire une fonction `sappend: 'a slist -> 'a slist -> 'a slist` qui concatène en $O(1)$ deux listes paresseuses.
7. Écrire une fonction `srev: 'a slist -> 'a slist` qui renverse une liste paresseuse de manière efficace. Quelle est la complexité des accès aux différents éléments de la liste renversée ?
8. Définir en OCaml une liste paresseuse qui contient *tous* les carrés parfaits.

Continuation

On se donne le type tree suivant pour représenter des arbres binaires:

```
type 'a tree = E | N of 'a * tree * tree
```

La hauteur d'un arbre peut être calculée par la fonction height suivante, de type `tree -> int` :

```
let rec height t =
  match t with
  | E -> 0
  | N (_, l, r) -> 1 + max (height l) (height r)
```

1. On regarde le code suivant:

   ```
   let rec left t n = if n = 0 then t else left (N ((), t, E)) (n - 1) in
   let t = left E 1000000
   let h = height t
   ```

   La première ligne est exécutée sans problème mais la seconde provoque l'erreur `Fatal error: exception Stack_overflow`. Expliquer l'erreur

Pour parvenir à calculer la hauteur en toute circonstance, une solution consiste à adopter un style de programmation dit par continuation. Plutôt que de calculer directement la hauteur $h(t)$ d'un arbre t, on va calculer $k(h(t))$ pour une fonction k quelconque en paramètre de la fonction `height`. La hauteur s'en déduira alors en prenant pour k la fonction identité.

Observer ce code:

```
let rec aux1 t k =
  match t with
  | E -> k 0
  | N (_, l, r) ->
    aux1 l ((*1*) fun hl ->
      aux1 r ((*2*) fun hr ->
      k (1 + max hl hr)))
let height1 t = aux1 t ((*3*) fun h -> h)
```

2. Donner le type de aux et montrer que `height1 t` calcule la hauteur de `t`
3. On définit la taille d'un arbre $|t|$ par son nombre de noeuds. Donner une version par continuation de `size t k` qui calcule `k t'` pour t' la taille de t et k une fonction
4. Donner une version par continuation de `get_prefixe: 'a tree -> 'a list` qui calcule la liste du parcours préfixe de l'arbre donné en argument. On fera attention à avoir une bonne complexité.

On peut modifier le code par continuation pour qu'il n'utilise plus de fonctions anonymes, mais des valeurs d'un type somme OCaml qui représente les différentes fonctions qui rentrent en jeu. On appelle cela la *défonctionnalisation*. On reprend le code d'exemple pour la fonction `height1`

5. Compléter le code à trou suivant. K1, K2, K3 sont les fonctions anonymes marquées par les commentaires `(*1*)`, `(*2*)` et `(*3*)` respectivement. La composition d'une suite de fonctions anonymes est alors représentée par une liste desdites fonctions qui la composent, aka le type `cont`. `apply k v` simule l'application de la fonction anonyme représentée par `k` avec `v`. La fonction `aux2` est l'analogue de la fonction `aux1`

```
type cont =
  | K1 of tree * cont
  | K2 of int * cont
  | K3
let rec aux2 t k =
  match t with
  | E -> apply ...
  | N (l, r) -> aux2 ...
and apply k v =
  match k with
  | K1 (r, k) -> aux2 ...
  | K2 (h, k) -> apply ...
  | K3 -> ...
let height2 t = aux2 ...
```

6. Montrer que cette version défonctionalisé est "efficace" et permet de simuler des fonctions anonyme dans du code n'en permettant pas de base comme C.
7. Défonctionaliser le code obtenu question 4.

Binary random acces lists

On étudie ici un type inductif de liste qui stocke dans sa structure de type sa longueur, et qui permet d'avoir des opérations en $O(\log n)$ avec n sa longueur.

On ce done le type suivant:

```
type 'a seq =
  | NIL
  | ZERO of ('a * 'a) seq
  | ONE of 'a * ('a * 'a) seq
```

Dans cet exercice, quand on écrit un entier n en binaire, on l'écrit avec le bit de poids fort à la fin (à l'envers par rapport à l'écriture standard). On remarquera alors qu'un tel nombre fini toujours par 1, et peut commencer par 0. Par exemple, 6 s'écrit 011. Dans ce cas, la liste de longueur n est représenté par une suite de `ZERO` et de `ONE` qui correspon à cette écriture.

1. Donner en OCaml des valeurs de type `int seq` représentant respectivement les listes $[1, 2]$, $[4, 2, 3]$ et $[9, 1, 2, 3, 4, 5]$

Petit encart intéressant mais non obligatoire pour comprendre: En OCaml, normalement, un appel récursif doit forcément être sur le même type que l'entrée. Cela peut poser quelques problèmes. Considérer le code suivant:

```
let rec always_true a b =
  if a = b then true
  else always_true 27 27
```

Ici OCaml infère le type `int -> int -> bool`, car il y a un appel récursif de la forme `always_true 27 27`. Mais le type `'a -> 'a -> bool` marche aussi ! Pour préciser à OCaml qu'il doit généraliser un paramètre de type, on doit écrire le type de la fonction avant, en précisant qu'une variable doit marcher "pour tout type"

```
let rec always_true: 'a. 'a -> 'a -> bool = fun a b ->
  if a = b then true
  else always_true 27 27
```

Pour les questions suivantes, on ignorera ce genre de subtilitées (aka on donne le type de la fonction).

2. Écrire une fonction `val cons: 'a. 'a -> 'a seq -> 'a seq` tel que `cons x xs` ajoute `x` au début de la liste `xs`
3. Écrire une fonction `val uncons: 'a. 'a seq -> 'a * 'a seq` tel que `uncons xs` pour `xs` une liste non vide, renvoie le couple du premier élément et du reste de la liste.
4. Quels sont les complexités de `cons` et `uncons` ?
5. Écrire une fonction `val get: 'a. 'a seq -> int -> 'a` tel que `get xs i` renvoie le $(i+1)$-ème élément de `xs`.
6. Écrire une fonction `val set: 'a. 'a seq -> int -> 'a -> 'a seq` tel que `set xs i x` remplace le $(i+1)$-ème élément de `xs` par `x`. On pourra s'aider des autres fonctions. Une complexité de $O(\log^2 n)$ est attendu.

On va essayer de faire `set` en $O(\log n)$. Pour cela, on chercher à définir `val transform: 'a. 'a seq -> int -> ('a -> 'a) -> 'a seq` tel que `transform xs i f` applique `f` au $(i+1)$-ème élément de la liste `xs` et renvoie la nouvelle liste.

7. Coder `transform`, de manière à ce que la complexité soit en $O(\log n + C_f)$ avec C_f la complexité de f
8. Donner une nouvelle version de `set` à l'aide de `transform` qui est en $O(\log n)$
9. En appliquant une technique de défonctionalisation vu dans l'exercice sur les continuations, proposer un type enum `type trans = | ...` et une implémentation de `transform` et `set` de telle sorte à ce que le type soit maintenant `val transform2: 'a. 'a seq -> int -> trans -> 'a seq`. A la fin, on ne devrai plus voir de fonction anonymes.

   Petite rappel de l'intuition sur la défonctionalisation: on simule la stack avec une liste.

Trop cool, on a obtenu un `set` en $O(\log n)$ qui n'utilise pas de fonctions anonymes !

Monades

On considère les modules de signature qui suit:

```
module type Monad = sig
  type 'a m
  val return: 'a -> 'a m
  val bind: 'a m -> ('a -> 'b m) -> 'b m
end
```

On définit une *monade* comme étant un module de signature `Monad` qui respecte 3 égalitées:

- `return` est neutre à gauche: `bind (return x) f = f x`
- `return` est neutre à droite: `bind m return = m`
- La monade est associative: `bind (bind m f) g = bind m (fun x => bind (f x) g)`

Question 1 On se donne le module suivant:

```
module OptionMonad = struct
  type 'a m = | None | Some of 'a;
  let return x = Some x;
  let bind x f = match x with
    | None -> None
    | Some x -> f x
end
```

Montrer que `OptionMonad` est une monade

Question 2 Proposer une implémentation de `ListMonade` avec `type 'a m = 'a list`

Soit M une monade, on définit alors les fonctions suivantes:

```
let map f x = bind x (fun y -> return (f y))
let join x = bind x (fun y -> y)
```

Question 3 Donner le type de `map` et `join`. Pour les monades de la question 1 et 2, à quoi correspond `map` et `join` ? *`join` est parfois appellé `flatten`*

Question 4 Montrer que `map` et `join` satisfait `join (map join x) = join (join x)`.

On admettra que `map` et `join` satisfont aussi

- `join (map return x) = join (return x)`
- `join (map (map f) x) = map f (join x)`

Je ne suis pas sur de cette question et de la question 5

Question 5 Montrer que si un module possède `join` et `map` avec les 3 équations précédente alors on peut le munir d'une structure de monade. On donnera la définition de `bind` et la preuve que la loi d'associativié est respectées (on admettera les 2 autres).

Question 6 On s'intéresse maintenant à la monade suivante:

```
module ContIntMonad = struct
  type 'a m = ('a -> int) -> int
  let return x k = k x
  let bind c f k = c (fun t -> f t k)
end
```

Montrer que c'est une monade.

Question 7 Que retourne le code suivant ?

```
open ContIntMonad
let rec mystere n =
  if n <= 1 then return n
  else bind (mystere (n-2)) (fun v -> return (v+1))
in mystere 45 (fun i -> i)
```

Question 8 Proposer une implémentation récursive naïve de la fonction de Fibonacci (sans se soucier de l'explosion des appels récursifs) en utilisant la monade `ContIntMonad`

Parsing avec des monades[2]

Attention: Ce sujet nécessite de comprendre les `lazy` (voir sujet "Suspensions et files persistantes"), et sera bien plus facile avec la compréhension des monades.

On cherche à définir un parser en OCaml en utilisant des combinateurs que l'on pourra imbriquer les uns aux autres. Un parser sera une fonction de `char list -> ('a * char list) seq`, avec `seq` les séquences (potentiellement infinies) suspendues

```
type 'a seq = | Nil | Cons of 'a * 'a node
and 'a node = 'a next
type 'a parser = char list -> ('a * char list) seq
```

L'idée est qu'un objet de type `'a parser` est une fonction qui prend une liste de caractère et qui renvoie une séquence (potentiellement infinie) de résultats de match. Chaque résultat est modélisé par un couple `(res,reste)` tel que `res` soit le résultat du parsing correct d'un préfixe et `reste` soit la suite de liste de caractère pas encore lu. En particulier, si la séquence est vide, c'est que le parsing a échoué. On dit qu'un caractère est consommé par le parser s'il est lu.

Par exemple, si `parser_int: int parser` parse des nombres, on peut imaginer que `parser_int ['0','4','5','3','v','7']` renvoie la séquence contenant $[(0, 453\text{v}7), (4, 53\text{v}7), (45, 3\text{v}7), (453, \text{v}7)]$. Dans ce cas, les caractères v et 7 ne sont jamais consommé.

Question 1 Définir les fonctions suivantes:

- Une fonction `val fail: 'a parser` qui est le parser qui échoue tout le temps.
- Une fonction `val return: 'a -> 'a parser` tel que `return x` est un parser qui pour toute entrée l renvoie $[(x, l)]$.
- Une fonction `val satisfy: (char -> bool) -> char parser` telle que `satisfy predicate` renvoie un parser qui consomme qu'un caractère c s'il satisfait $\text{predicate}(c)$, ou qui échoue sinon.
- Une fonction `val char: char -> char parser` telle que `char c` renvoie le parser qui ne consomme que le premier caractère seulement s'il est égal à `c`.

Question 2 Une fonction `val bind: 'a parser -> ('a -> 'b parser) -> 'b parser` tel que `bind p f` renvoie un parser qui pour une entrée l donné, applique $p(l) = [(x_1, l_1), ..., (x_n, l_n), ...]$ et renvoie la concaténation des $(f(x_i)(l_i))_{i\in\mathbb{N}}$

Question optionelle Montrer que `parser` avec `bind` et `return` forment une monade.

Question 3 Définir la fonction `let (||) p1 p2 = ...` telle que `(||) p1 p2` renvoie le parser qui effectue la concaténation des résultats des deux parser. On rappelle que la syntaxe `(||)` en OCaml permet de redéfinir la fonction `||` et d'écrire `p1 || p2` à la place de `(||) p1 p2`.

Question 4 (dur) Définir la fonction `val fix: ('a parser -> 'a parser) -> 'a parser` qui prend en argument une fonction `f` et qui renvoie un parser `p` tel que `p` agit comme `f p`. On fera bien attention à ce que `fix` termine quand `f` termine.

Question 5 Définir la fonction `val map: ('a -> 'b) -> 'a parser -> 'b parser` tel que `map f p` renvoie le parser qui applique une fonction `f` au résultat du parsing de `p`

On définit alors le combinateur suivant:

```
let (<*>) (p1: ('a -> 'b) parser) (p2: 'a parser): 'a parser = bind p1 (fun f -
> map f p2)
```

Il prend en argument deux parser `p1` et `p2`, parse l'entrée d'abord avec p_1, puis ensuite avec p_2, et applique la fonction résultat du parsing de p_1 au résultat du parsing de p_2, et ceci pour chaque entrée. Comme pour `(||)`, on pourra écrire `p1 <*> p2` au lieu de `(<*>) p1 p2`.

Question 6 Quel est le type et que fait la fonction suivante ?

```
let parens p = ((map (fun _ x _ -> x) (char '(') <*> p) <*> char ')')
```

Question 7 Quel est le type et que fais `parse_d` définit si-dessous ?

```
let parse_d = fix (fun parse_d ->
    ((map (fun b -> max (1+b)) (parens parse_d)) <*> parse_d)
  || return 0)
```

Question 8 Définir un combinateur `val repeat: 'a parser -> int -> 'a list parser` tel que `repeat p n` parse `n` séquences concaténées reconnues par `p` et renvoie la liste des séquences

Question 9 Définir un combinateur `val list: 'a parser -> 'a list parser` qui reconnaît une suite arbitraire de concaténation de séquences reconnues par `p`

Question 10 Écrire un parser qui reconnaît exactement le plus grand préfixe de $\{0, 1\}$ et qui renvoie sa longueur. Le parser renverra une séquence d'exactement un élément $[(x, l)]$ avec x la longueur et l le reste de l'entrée. On n'utilisera que les combinateurs précédemment utilisés

Question 11 Écrire un parser qui reconnaît exactement les entrées (et pas leur préfixes) dans le langage $\{a^n b^n c^n : n \in \mathbb{N}\}$ et qui retourne sa longueur.

---

[2] MPRI - FUN Exam 2026