cours8

November 21, 2023

1 Programmation fonctionnelle

1.1 Semaine 8 : Évaluation paresseuse

Etienne Lozes - Université Nice Sophia Antipolis - 2019

[1]:
```
let () = ()
```

1.2 Plan de la séance

Durant cette séance, nous allons étudier la programmation paresseuse: c'est le style de programmation fonctionnelle Haskell *par essence*.

Nous allons voir deux concepts clés d'Haskell.

1. L'appel par nécessité: comment éviter des calculs inutiles en ne calculant pas la valeur d'un argument dont on n'a pas besoin
2. Les flots, ou listes paresseuses: comment ne calculer le contenu d'une liste qu'au moment où on en a réellement besoin.

Bien sûr, nous restons en `OCaml`, et nous allons devoir "bricoler" un peu, en Haskell ce serait plus direct. Mais vous apprendrez ainsi à programmer en style Haskell dans n'importe quel langage!

1.3 Les glaçons: des calcul retardés

Un **glaçon** (en anglais *thunk*) est une clôture représentant une fonction anonyme sans paramètre .

[2]:
```
let glacon0 = fun () -> (1 + 1)
```

[2]:
```
val glacon0 : unit -> int = <fun>
```

Idée plûtot que de lancer immédiatement l'évaluation d'une expression `e`, on peut la *geler* en construisant un glaçon `fun () -> e`.

Un `'a glacon` est donc simplement une fonction sans argument qui calcule une valeur de type `'a`.

[3]:
```
type 'a glacon = unit -> 'a
```

[3]:
```
type 'a glacon = unit -> 'a
```

On pourra effectuer le calcul plus tard en dégelant le glaçon.

Pour dégeler le glaçon, il me suffit de lui passer l'argument `()` attendu.

[4]:
```
let degeler un_glacon = un_glacon ()
```

[4]:
```
val degeler : (unit -> 'a) -> 'a = <fun>
```

[5]:
```
degeler glacon0
```

[5]:
```
- : int = 2
```

[6]:
```
degeler glacon0
```

[6]:
```
- : int = 2
```

Question combien de fois a-t-on calculé `1+1`?

Autre exemple: un dé à 6 faces

[7]:
```
let glacon1 = fun () -> 1 + Random.int 6
```

[7]:
```
val glacon1 : unit -> int = <fun>
```

[8]:
```
degeler glacon1
```

[8]:
```
- : int = 1
```

[9]:
```
degeler glacon1
```

[9]:
```
- : int = 2
```

Conclusion lorsqu'on a dégelé deux fois le glaçon, on a refait le calcul qui était gelé! Ne peut-on pas espérer mieux?

Une **paresse**, c'est justement cela: un glaçon dont ne refait pas le calcul après le premier dégel.

1.4 Les paresses

Une paresse est donc un glaçon amélioré; le calcul contenu dans une paresse a deux états possibles:

- l'état suspendu : le calcul n'a pas encore été effectué
- l'état terminé : le calcul a été effectué, on se souvient du résultat pour un éventuel nouveau dégel.

Le dégel fait passer de l'état suspendu à l'état terminé: la paresse doit donc être une valeur *mutable*.

Question comment définir les types `'a calcul` et `'a paresse`?

[10]:
```
type 'a calcul =
  | Suspendu of (unit -> 'a)   (* un glaçon  *)
  | Termine of 'a              (* une valeur *)

type 'a paresse = 'a calcul ref
```

[10]:
```
type 'a calcul = Suspendu of (unit -> 'a) | Termine of 'a
```

[10]:
```
type 'a paresse = 'a calcul ref
```

Je peux donc définir un entier paresseux comme suit (j'ajoute le type pour plus de clarté).

[11]:
```
let entier_paresseux : int paresse =
  ref (Suspendu (fun () -> 1 + Random.int 6))
```

[11]:
```
val entier_paresseux : int paresse = {contents = Suspendu <fun>}
```

Comment dois-je définir la fonction `degeler : 'a paresse -> 'a`?

[12]:
```
let degeler (p: 'a paresse) = match !p with

  | Suspendu(glacon) ->      (* CAS 1 : je n'ai encore jamais fait le calcul :
 ↪    *)
    let res = glacon ()      (* 1. je fais le calcul et je mets le résultat dans
 ↪res *)
    in p := Termine res;     (* 2. je modifie l'état de ma paresse
 ↪    *)
    res                      (* 3. je renvoie le résultat
 ↪    *)

  | Termine(res) ->          (* CAS 2 : j'ai déjà fait le calcul :
 ↪    *)
    res                      (* je connais le résultat, je le renvoie
 ↪    *)
```

```
[12]: val degeler : 'a paresse -> 'a = <fun>
```

```
[13]: degeler entier_paresseux
```

```
[13]: - : int = 5
```

```
[14]: degeler entier_paresseux
```

```
[14]: - : int = 5
```

Coup de chance? pas vraiment...

```
[15]: List.init 30 (fun i -> degeler entier_paresseux)
```

```
[15]: - : int list =
      [5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5;
       5; 5; 5; 5; 5]
```

1.5 Comment rendre une expression paresseuse?

On peut définir une fonction qui transforme un glaçon en paresse

```
[16]: let paresse g = ref (Suspendu g)
```

```
[16]: val paresse : (unit -> 'a) -> 'a calcul ref = <fun>
```

Prenons un exemple: l'expression `Unix.time()` a pour valeur l'heure courante. Considérons maintenant le glaçon

```
[17]: let g = fun () -> Unix.time()    (* <==> let g = Unix.time *)
```

```
[17]: val g : unit -> float = <fun>
```

La valeur que j'obtiens en dégelant `g` est l'heure qu'il est au moment du dégel. Je vais donc pouvoir vérifier si le flottant paresseux `paresse g` est bien dégelé une seule fois.

```
[18]: let flottant_paresseux = paresse g
```

```
[18]: val flottant_paresseux : float calcul ref = {contents = Suspendu <fun>}
```

```
[19]: degeler flottant_paresseux  (* <- l'heure du dégel *)
```

[19]:
```
- : float = 1576069221.
```

(je reviens après avoir pris un café…)

[20]:
```
degeler flottant_paresseux  (* <- l'heure du dégel précédent *)
```

[20]:
```
- : float = 1576069221.
```

Ça marche!

Mais il reste un point un peu désagréable: pour créer une paresse pour le calcul de l'expression `e`, je dois écrire `paresse (fun () -> e)`, et non `paresse e`.

Question N'y a-t-il pas moyen de programmer `paresse` autrement pour pouvoir écrire `paresse e`?

Malheureusement, non. En effet, lorsque `OCaml` doit évaluer une expression de la forme `e1 e2`, il procède comme suit:

1. évaluer `e1` et en déduire une fonction `f = fun x -> e'`
2. évaluer `e2` et en déduire une valeur `v`
3. évaluer `e'` dans lequel `x` est remplacé par `v`.

Quoi que fasse la fonction `paresse`, elle prendra donc son argument sous la forme d'une valeur, et non d'une expression… donc trop tard pour la geler.

Cette stratégie d'évaluation est connue sous le nom d'**appel par valeur**, c'est la stratégie d'évaluation "classique".

1.6 L'appel par nécessité

Haskell est *le* langage qui fait exception. En Haskell, la stratégie d'évaluation est l'**appel par nécessité**: on n'évalue l'argument que si on en a besoin, au moment où on en a besoin, et seulement la première fois qu'on en a besoin.

Prenons un exemple: la fonction `produit` sur les entiers

[21]:
```
let produit a b = a * b
```

[21]:
```
val produit : int -> int -> int = <fun>
```

Je voudrais définir une fonction `produit_paresseux` qui ne calcule son deuxième argument que si le premier argument est non nul.

En Haskell, j'écrirais quelque chose comme

[22]:
```
let produit a b =   (* produit qui serait paresseux en Haskell *)
  if a = 0 then 0 else a * b
```

```
[22]: val produit : int -> int -> int = <fun>
```

En OCaml, à cause de l'appel par valeur, cela ne suffit pas: le calcul de b a lieu avant le test `a=0`.

Par exemple, le calcul ci-dessous prend un temps déraisonnable en OCaml.

```
[23]: let rec fibo n = if n < 2 then 1 else fibo (n-1) + fibo (n-2)
      let _ = produit 0 (fibo 42)  (* <- le calcul de fibo 42 est effectué, mais␣
       ↪inutile *)
```

```
[23]: val fibo : int -> int = <fun>
```

```
[23]: - : int = 0
```

En Haskell, on n'éffectuerait pas le calcul de `fibo 42`, et tout irait très vite!

Nous allons voir comment coder l'appel par nécessité dans un langage en appel par valeur comme `Ocaml` en utilisant une paresse.

Et tant qu'à faire, nous allons utiliser les paresses natives en OCaml.

1.7 Le module `Lazy` : la paresse native en `OCaml`

`Ocaml` dispose d'une *pseudo-fonction* `lazy`.

`lazy(e)` correspond, sur le principe, à `paresse (fun () -> e)`.

C'est la "fonction" `paresse` telle que je rêvais de l'écrire mais qui ne peut pas exister en tant que fonction à cause de l'appel par valeur.

```
[24]: let flottant_paresseux2 = lazy(Unix.time())
```

```
[24]: val flottant_paresseux2 : float lazy_t = <lazy>
```

Pour "dégeler" une paresse, il faut **forcer** le calcul.

```
[25]: Lazy.force flottant_paresseux2  (* <- calcul de Unix.time() *)
```

```
[25]: - : float = 1576069230.
```

```
[26]: Lazy.force flottant_paresseux2  (* <- pas de second calcul de Unix.time() *)
```

```
[26]: - : float = 1576069230.
```

Note Le type d'une expression paresseuse est `'a lazy_t`. Il est recommandé de ne pas l'utiliser pour construire un type de données. À la place, on peut utiliser le type `'a Lazy.t`.

[27]:
```
type flottant_paresseux = float Lazy.t
```

[27]:
```
type flottant_paresseux = float Lazy.t
```

Si je demande au toplevel d'afficher une paresse qui n'a pas été forcée, il m'indique une valeur "opaque" `<lazy>`.

[28]:
```
let x = lazy(Float.pi ** 2.)
```

[28]:
```
val x : float lazy_t = <lazy>
```

[29]:
```
x
```

[29]:
```
- : float lazy_t = <lazy>
```

En revanche, si la paresse a été forcée et que le résultat `v` est connu, le toplevel m'indique `lazy v`.

[30]:
```
Lazy.force x
```

[30]:
```
- : float = 9.86960440108935799
```

[31]:
```
x
```

[31]:
```
- : float lazy_t = lazy 9.86960440108935799
```

1.8 Simuler l'appel par nécessité en `OCaml`

Revenons maintenant sur le problème du produit paresseux. Plutôt que de multiplier des entiers, je vais multiplier des entiers paresseux.

Idée: je ne forcerai mes entiers paresseux qu'au moment où j'ai besoin de connaitre leur valeur.

[32]:
```
let produit a b =
  if Lazy.force a = 0
  then 0
  else Lazy.force a * Lazy.force b
```

[32]:
```
val produit : int Lazy.t -> int Lazy.t -> int = <fun>
```

[33]:
```
produit (lazy 0) (lazy (fibo 42))          (* <- réponse immédiate! *)
```

[33]:
```
- : int = 0
```

Question : Soit le programme suivant:

[34]:
```
let carre a = produit a a  (* a est un entier paresseux *)
let _ = carre (lazy (fibo 42))
```

[34]:
```
val carre : int Lazy.t -> int = <fun>
```

[34]:
```
- : int = 187917426909946969
```

Combien de fois vais-je calculer `fibo 42`?

Réponse : une seule fois! Détaillons le calcul de `carre (lazy (fibo 42)`

1. `OCaml` évalue l'argument `lazy (fibo 42)`. Il crée en mémoire une paresse `p` et met le calcul suspendu de `fibo 42` à l'intérieur.
2. `OCaml` évalue `carre` et le remplace par `fun a b -> produit a b`
3. `OCaml` "passe les arguments" : l'expression à évaluer devient `carre p p`
4. Nouveau passage d'arguments: il faut évaluer

```
if Lazy.force p = 0 then 0 else Lazy.force p * Lazy.force p
```

5. `OCaml` évalue la condition, et pour cela force `p`. La paresse est modifiée et contient désormais le résultat du calcul de `fibo 42`, 187917426909946969.
6. `OCaml` évalue `Lazy.force p * Lazy.force p`. Les deux appels à `force` renvoient immédiatement l'entier 187917426909946969.

1.9 match lazy(...)

`OCaml` permet insérer `lazy` dans une définition par cas (c'est une pseudo-fonction, avec une vraie fonction ce n'est pas autorisé).

L'idée est que `match x with lazy(Cons y) -> ...`

est un raccourci pour

`match Lazy.force x with Cons y -> ...`

[35]:
```
let produit a b = match a, b with
| lazy(0), _ -> 0
| lazy(a'), lazy(b') -> a' * b'
```

[35]:
```
val produit : int lazy_t -> int lazy_t -> int = <fun>
```

1.10 Les flots, ou listes paresseuses

Terminologie >j'emploierai dans ce cours le mot flot comme un synonyme de liste paresseuse. Il s'agit d'un abus de langage, le module `Stream` d'`OCaml` propose des "flots" qui ne correspondent pas exactement à mes flots, en particulier les `'a stream` ne sont pas *persistants*, alors que mes flots, qui sont des listes paresseuses, le sont.

Soit à résoudre un problème ayant plusieurs solutions, peut-être une infinité.

Plutôt que de générer la liste de *toutes* les solutions, on génère un couple `(sol, p)` formé de la première solution et d'une paresse `p` qui n'est autre que la **promesse** de continuer le calcul si besoin.

On retarde ainsi le calcul des autres solutions: si la première est satisfaisante, pas besoin de les calculer.

Un **flot** est une "promesse": si la promesse est tenue, le flot me donne un couple `(sol, flot2)` constitué d'une solution et d'un nouveau flot pour les solutions restant à calculer.

J'appelerai le couple `(sol, flot)` un **item**.

[36]:
```
type 'a item = Item of 'a  * 'a flot    (* définition mutuellement récursive !␣
  ↪*)
and  'a flot = 'a item Lazy.t
```

[36]:
```
type 'a item = Item of 'a * 'a flot
and 'a flot = 'a item Lazy.t
```

1.11 Le flot vide

Un flot vide est une promesse qui ne sera pas tenue (on ne calculera pas un item).

[37]:
```
exception Flot_vide
let flot_vide : 'a flot = lazy(raise Flot_vide)
```

[37]:
```
exception Flot_vide
```

[37]:
```
val flot_vide : 'a flot = <lazy>
```

1.12 Ajouter une solution en début de flot

Pour ajouter une solution au début d'un flot, je dois la placer dans un item, et rendre cet item paresseux pour en faire un flot.

[38]:
```
let scons x flot = lazy(Item(x, flot))
```

```
[38]: val scons : 'a -> 'a flot -> 'a item lazy_t = <fun>
```

```
[39]: let flot_42 = scons 42 flot_vide     (* un flot qui contient uniquement 42 *)
```

```
[39]: val flot_42 : int item lazy_t = <lazy>
```

1.13 Construire un flot de solutions

Problème soit à trouver les éléments d'une liste `l:'a list` vérifiant un prédicat `p:'a -> bool`. La fonction `solutions` ci-dessous renvoie le flot des solutions, autrement dit la promesse d'une première solution et d'un flot d'autres solutions.

```
[40]: let rec solutions p l = match l with
        | [] -> flot_vide

        | x::tl when p x ->
           lazy(Item(x, solutions p tl)) (* <- bloquage de la récursion*)

        | _::tl ->
           solutions p tl
```

```
[40]: val solutions : ('a -> bool) -> 'a list -> 'a flot = <fun>
```

```
[41]: let flot0 = solutions Float.is_integer [1.0; 2.17;  2.0; 0.1]
```

```
[41]: val flot0 : float flot = <lazy>
```

Le flot `flot0` promet les deux "solutions" `1.0` et `2.0`.

1.14 Premier item d'un flot

Comment récupérer le premier couple `(sol, flot)` d'un flot?

Facile: 1. Je force le flot pour récupérer un item 2. Je déconstruis cet item pour récupérer le couple

```
[42]: let get_item (flot : 'a flot) =
        match Lazy.force flot with    (* je force le flot et je déconstruit l'item␣
       ↪obtenu *)
        | Item(solution, flot2) -> (solution, flot2)
```

```
[42]: val get_item : 'a flot -> 'a * 'a flot = <fun>
```

[43]:
```
(* ou encore *)
let get_item = function lazy(Item(sol, flot)) -> (sol, flot)
```

[43]:
```
val get_item : 'a item lazy_t -> 'a * 'a flot = <fun>
```

Testons ma fonction `get_item` sur le flot `flot0`= 1.,2. calculé précédemment

[44]:
```
let (sol1, flot1) = get_item flot0 in
let (sol2, flot2) = get_item flot1 in
(sol1, sol2)
```

[44]:
```
- : float * float = (1., 2.)
```

[45]:
```
let (sol1, flot1) = get_item flot0 in
let (sol2, flot2) = get_item flot1 in
get_item flot2   (* <- flot2 est le flot vide : il n'y a pas d'autres solutions␣
 ↪*)
```

```
Exception: Flot_vide.
Raised at file "[37]", line 2, characters 37-46
Called from file "camlinternalLazy.ml", line 29, characters 17-27
Re-raised at file "camlinternalLazy.ml", line 36, characters 10-11
Called from unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

1.15 N-ième solution d'un flot

Je vais définir un analogue de `List.nth` pour les flots.

Pour cela, je dois itérer `get_item` pour calculer le flot restant n fois, puis renvoyer la première solution de ce flot.

[46]:
```
let rec n_ieme n flot =
   if n<0
   then invalid_arg "n_ieme n flot: n >= 0 attendu"
   else begin
     let (x,flot2) = get_item flot in
     if n = 0 then x else n_ieme (n-1) flot2
   end
```

[46]:
```
val n_ieme : int -> 'a flot -> 'a = <fun>
```

[47]:
```
n_ieme 0 flot0
```

[47]:
```
- : float = 1.
```

[48]:
```
n_ieme 1 flot0
```

[48]:
```
- : float = 2.
```

[49]:
```
n_ieme 2 flot0
```

```
Exception: Flot_vide.
Raised at file "camlinternalLazy.ml", line 35, characters 62-63
Called from file "camlinternalLazy.ml", line 29, characters 17-27
Re-raised at file "camlinternalLazy.ml", line 36, characters 10-11
Called from unknown location
Called from file "[46]", line 5, characters 21-34
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

1.16 Utiliser un flot pour résoudre un problème

Problème Soit à calculer le 42-ième nombre premier d'un gros intervalle, par exemple $\{1, 2, 3, 4, 5, 6 \dots, 500\,000\}$.

[50]:
```
(* on se donne pour la suite une fonction est_premier,
 * peu importe comment elle est écrite pour le propos
 *)

let est_premier n =
   let rec teste k = (k*k > n) || ((n mod k != 0) && teste (k+1))
   in n >=2 && teste 2
```

[50]:
```
val est_premier : int -> bool = <fun>
```

Méthode naïve Je vais construire la liste des entiers de 1 à 500 000, utiliser `List.filter` pour ne retenir que les nombres premiers, puis utiliser `List.nth` pour trouver le 42-ième nombre premier.

[51]:
```
let entiers_jusqu'a_500_000 = List.init 500_000 (fun i->i+1)
```

[51]:
```
val entiers_jusqu'a_500_000 : int list =
  [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20; 21;
```

```
22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34; 35; 36; 37; 38; 39;
40; 41; 42; 43; 44; 45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57;
58; 59; 60; 61; 62; 63; 64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 75;
76; 77; 78; 79; 80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92; 93;
94; 95; 96; 97; 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108; 109;
110; 111; 112; 113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123; 124;
125; 126; 127; 128; 129; 130; 131; 132; 133; 134; 135; 136; 137; 138; 139;
140; 141; 142; 143; 144; 145; 146; 147; 148; 149; 150; 151; 152; 153; 154;
155; 156; 157; 158; 159; 160; 161; 162; 163; 164; 165; 166; 167; 168; 169;
170; 171; 172; 173; 174; 175; 176; 177; 178; 179; 180; 181; 182; 183; 184;
185; 186; 187; 188; 189; 190; 191; 192; 193; 194; 195; 196; 197; 198; 199;
200; 201; 202; 203; 204; 205; 206; 207; 208; 209; 210; 211; 212; 213; 214;
215; 216; 217; 218; 219; 220; 221; 222; 223; 224; 225; 226; 227; 228; 229;
230; 231; 232; 233; 234; 235; 236; 237; 238; 239; 240; 241; 242; 243; 244;
245; 246; 247; 248; 249; 250; 251; 252; 253; 254; 255; 256; 257; 258; 259;
260; 261; 262; 263; 264; 265; 266; 267; 268; 269; 270; 271; 272; 273; 274;
275; 276; 277; 278; 279; 280; 281; 282; 283; 284; 285; 286; 287; 288; 289;
290; 291; 292; 293; 294; 295; 296; 297; 298; 299; …]
```

[52]:
```
let premiers_jusqu'a_500_000 =
   List.filter est_premier entiers_jusqu'a_500_000
```

[52]:
```
val premiers_jusqu'a_500_000 : int list =
  [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67;
   71; 73; 79; 83; 89; 97; 101; 103; 107; 109; 113; 127; 131; 137; 139; 149;
   151; 157; 163; 167; 173; 179; 181; 191; 193; 197; 199; 211; 223; 227; 229;
   233; 239; 241; 251; 257; 263; 269; 271; 277; 281; 283; 293; 307; 311; 313;
   317; 331; 337; 347; 349; 353; 359; 367; 373; 379; 383; 389; 397; 401; 409;
   419; 421; 431; 433; 439; 443; 449; 457; 461; 463; 467; 479; 487; 491; 499;
   503; 509; 521; 523; 541; 547; 557; 563; 569; 571; 577; 587; 593; 599; 601;
   607; 613; 617; 619; 631; 641; 643; 647; 653; 659; 661; 673; 677; 683; 691;
   701; 709; 719; 727; 733; 739; 743; 751; 757; 761; 769; 773; 787; 797; 809;
   811; 821; 823; 827; 829; 839; 853; 857; 859; 863; 877; 881; 883; 887; 907;
   911; 919; 929; 937; 941; 947; 953; 967; 971; 977; 983; 991; 997; 1009;
   1013; 1019; 1021; 1031; 1033; 1039; 1049; 1051; 1061; 1063; 1069; 1087;
   1091; 1093; 1097; 1103; 1109; 1117; 1123; 1129; 1151; 1153; 1163; 1171;
   1181; 1187; 1193; 1201; 1213; 1217; 1223; 1229; 1231; 1237; 1249; 1259;
   1277; 1279; 1283; 1289; 1291; 1297; 1301; 1303; 1307; 1319; 1321; 1327;
   1361; 1367; 1373; 1381; 1399; 1409; 1423; 1427; 1429; 1433; 1439; 1447;
   1451; 1453; 1459; 1471; 1481; 1483; 1487; 1489; 1493; 1499; 1511; 1523;
   1531; 1543; 1549; 1553; 1559; 1567; 1571; 1579; 1583; 1597; 1601; 1607;
   1609; 1613; 1619; 1621; 1627; 1637; 1657; 1663; 1667; 1669; 1693; 1697;
   1699; 1709; 1721; 1723; 1733; 1741; 1747; 1753; 1759; 1777; 1783; 1787;
   1789; 1801; 1811; 1823; 1831; 1847; 1861; 1867; 1871; 1873; 1877; 1879;
   1889; 1901; 1907; 1913; 1931; 1933; 1949; 1951; 1973; 1979; …]
```

[53]:
```
let _42_ieme_premier = List.nth premiers_jusqu'a_500_000 42
```

[53]:
```
val _42_ieme_premier : int = 191
```

Je fais évidemment beaucoup de calculs inutiles!

Seconde approche Je vais utiliser des flots à la place des listes!

Commençons par définir le flot des entiers de a à b.

[54]:
```
let rec entiers_entre a b =
  if a <= b
  then lazy(Item(a, entiers_entre (a+1) b))    (* <- blocage de la récursion *)
  else flot_vide
```

[54]:
```
val entiers_entre : int -> int -> int flot = <fun>
```

Je vais maintenant construire le flot des nombres premiers compris entre a et b.

Pour cela, je vais filtrer le flot précédent à l'aide d'une fonction

`filtre_flot : ('a -> bool) -> 'a flot -> 'a flot`

qui fait l'analogue de `List.filter` mais avec les flots.

[55]:
```
let rec filtre_flot condition flot =
  lazy begin                (* <- blocage de la récursion *)
    let (x,flot2) = get_item flot in
    if condition x
    then Item(x, filtre_flot condition flot2)
    else Lazy.force (filtre_flot condition flot2)
  end
```

[55]:
```
val filtre_flot : ('a -> bool) -> 'a flot -> 'a flot = <fun>
```

[56]:
```
let premiers_jusqu'a_500_000 =
   filtre_flot est_premier (entiers_entre 1 500_000)
```

[56]:
```
val premiers_jusqu'a_500_000 : int flot = <lazy>
```

Le résultat est immédiat!

Il ne me reste plus qu'à parcourir le flot jusqu'au 42-ième élément.

Pendant le parcours, je vais (enfin) calculer les 42 premiers nombres premiers.

[57]:
```
let _42_ieme_nombre_premier = n_ieme 42 premiers_jusqu'a_500_000
```

```
[57]: val _42_ieme_nombre_premier : int = 191
```

Si je demande le 41-ième nombre premier avec `n_ieme 41 premiers_jusqu'a_500_000`, le résultat sera immédiat: il a déjà été calculé.

1.17 Flots infini

Un flot n'a aucune raison d'être fini!

Le flot des entiers plus grands que n se définit par récurrence… mais il s'agit d'une récurrence un peu spéciale, sans cas de base.

```
[58]: let rec entiers_depuis n =
        lazy(Item(n, entiers_depuis (n+1)))  (* <- récurrence bloquée! *)
```

```
[58]: val entiers_depuis : int -> int flot = <fun>
```

C'est parce que la récurrence est bloquée que l'on peut appeler sans risque la fonction `entiers_depuis`.

```
[59]: let entiers = entiers_depuis 0  (* <- termine immédiatement! *)
```

```
[59]: val entiers : int flot = <lazy>
```

On peut désormais suivre une troisième approche, plus épurée, pour calculer le 42 ième nombre premier.

```
[60]: let nombres_premiers = filtre_flot est_premier entiers
```

```
[60]: val nombres_premiers : int flot = <lazy>
```

```
[61]: n_ieme 42 nombres_premiers
```

```
[61]: - : int = 191
```

1.18 Des flots au listes

Pour transformer un flot en une liste, il va me falloir fixer une borne `k` sur la taille de la liste que j'obtiendrai: si le flot est plus long que `k`, je tronquerai.

```
[62]: let rec list_of_flot k flot =
        try begin
          match (k, get_item flot) with
```

```
      | (0, _) -> []
      | (k, (a,flot2)) -> a :: list_of_flot (k-1) flot2
  end with
    | Flot_vide -> []
```

[62]:
```
val list_of_flot : int -> 'a flot -> 'a list = <fun>
```

[63]:
```
list_of_flot 10 entiers
```

[63]:
```
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

[64]:
```
list_of_flot 100 (entiers_entre 1 10)
```

[64]:
```
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

1.19 Somme terme à terme de deux flots

Soient $\mathbf{a} = (a_0, a_1, \dots)$ et $\mathbf{b} = (b_0, b_1, \dots)$ deux flots d'entiers de longueurs quelconques.

Problème construire le flot $\mathbf{a} + \mathbf{b} = (a_0 + b_0, a_1 + b_1, \dots)$.

[65]:
```
let rec somme_flots flot_a flot_b  =
  lazy(begin
    match get_item flot_a, get_item flot_b with
      | (a0, flot_a'), (b0, flot_b') ->
          Item( a0 + b0, somme_flots flot_a' flot_b' )
  end)
```

[65]:
```
val somme_flots : int flot -> int flot -> int flot = <fun>
```

[66]:
```
let _2N = somme_flots entiers entiers
let _ = list_of_flot 10 _2N
```

[66]:
```
val _2N : int flot = <lazy>
```

[66]:
```
- : int list = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18]
```

1.20 Les flots définis par une équation de point fixe

Considérons le flot constant à 1

[67]:
```
let rec un = lazy(Item(1, un))
```

[67]:
```
val un : int flot = <lazy>
```

[68]:
```
list_of_flot 10 un
```

[68]:
```
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1]
```

Considérons maintenant le flot suivant

[69]:
```
let (++) = somme_flots
let rec flot_e = lazy(Item(0, un ++ flot_e ))
```

[69]:
```
val ( ++ ) : int flot -> int flot -> int flot = <fun>
```

[69]:
```
val flot_e : int flot = <lazy>
```

Question : quelle suite d'entiers correspond à `flot_e`?

Réponse `flot_e = entiers`!

[70]:
```
list_of_flot 10 flot_e
```

[70]:
```
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

	0	1	2	3	4	5	6	...	entiers
+	1	1	1	1	1	1	1	...	un
0	1	2	3	4	5	6	7	...	entiers

On a donné une définition "implicite" du flot d'entiers. Cette définition est associée à l'équation de point fixe

`x = 0 :: (un + x)`

Cette équation n'admet qu'une seule solution: `x=entiers`.

En exploitant de telles équations, vous pourrez calculer le développement en séries entières de l'exponentielle, des fonctions trigonométriques, etc… cf TP?

$$e^x = \sum_{n \geq 0} \frac{x^n}{n!}$$

`list_of_flot 5 exponentielle` $\rightarrow$ [1, 1, $\frac{1}{2}$, $\frac{1}{6}$, $\frac{1}{24}$]

1.21 Résumons

- un **glaçon** (*thunk*) `fun () -> e` est un calcul suspendu de l'expression `e`
- une **paresse** est un glaçon amélioré: on peut demander le résultats du calcul plusieurs fois sans le recalculer à chaque fois
- un **flot** est une "liste paresseuse" qui peut être infinie

L'évaluation paresseuse est celle par défaut en Haskell: - les appels de fonctions se font par nécessité - les listes sont paresseuses