Programmation Comparée

Programmation fonctionnelle

De la prise du contrôle de l'état
. . . au coup d'état sur le contrôle

Yann **Régis-Gianas**
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

Plan

La programmation fonctionnelle

D'une certaine façon, la programmation fonctionnelle hérite de la programmation procédurale la **vision d'un programme comme une fonction** qui attend des données en entrée et produit des données en sortie. Ainsi, une spécification d'un programme est un couple formé d'un précondition sur les entrées et d'une postcondition sur les sorties.

Elle se différencie de la programmation procédurale à travers deux aspects :

- Une **restriction** : un programme ne modifient plus l'état de la machine pour transformer les entrées en des sorties valides mais engendre de nouvelles données disjointes des données d'entrées.
- Une **extension** : le domaine des données, l'objet des calculs, inclue désormais les fonctions, c'est-à-dire des objets d'ordre supérieur, contenant du code.

Pour donner du sens à cette extension, les langages fonctionnels s'appuient sur un **formalisme** développé en logique mathématique par Alonzo Church en 1930 : le λ-calcul.

Les fonctions de première classe

Fonction de première classe

Dire que les fonctions sont de **première classe** signifie que les calculs peuvent produire de nouvelles fonctions, les stocker dans des structures de données ou attendre des fonctions en entrée.

```
(* Une implémentation (inefficace) d'un dictionnaire *)
(* par une fonction. *)
let add f k v = fun k' → if k = k' then v else f k'
let find f = f
exception Not_found
let empty = fun k → raise Not_found
```

Fonction d'ordre supérieur

Une fonction est dite d'**ordre supérieur** quand elle attend une fonction en entrée.

Comment représenter des fonctions ?

L'architecture de Von Neumann intègre déjà l'idée que **le code est une donnée comme une autre**. Cependant, la représentation du code, dans un langage de programmation de haut-niveau, comme une fragment de code machine n'est pas très pratique. (En effet, cela demanderait au programmeur de comprendre précisément le processus de compilation, ce qui est en contradiction avec l'abstraction visée par un langage de haut-niveau.)

Le langage LISP, créé par McCarthy en 1958, apporte une première réponse : dans ce langage, les données et les programmes sont représentés d'une façon uniforme par des **S-expressions**, une syntaxe pour des listes. (Voir les transparents suivants.)

En 1964, Peter Landin montre qu'une fonction, dans un langage de programmation, peut-être efficacement représentée par une **fermeture**. Une fermeture est un fragment de code clos, c'est-à-dire qu'il contient tout ce qui est nécessaire à son exécution. Une façon particulière d'implémenter une fermeture consiste à capturer la valeur des variables dont dépend un fragment de code dans un environnement que l'on adjoint à un pointeur vers ce fragment.

Petite digression au sujet de LISP...

Les S-expressions

- La syntaxe des expressions de LISP :

$$\begin{array}{rcl} \mathit{Sexp} & ::= & \mathit{atom} \\ & | & (\mathit{Sexp}\ .\ \mathit{Sexp}) \end{array}$$

- Cette syntaxe est aussi celle des valeurs !

Type des expressions

- On classifie les expressions :
  - `cons` : une paire formée de deux expressions.
    - On appelle `car` la première composante ;
    - et `cdr` la seconde. [1]
  - `int` : un entier.
  - `sym` : un symbole.
  - `prim` : une fonction primitive.
  - `comp` : une fonction anonyme.
  - `spec` : une forme spéciale (une fonction qui n'évalue pas tout ses arguments).

1. `car` pour *Contents of Address Register* et `cdr` pour *Contents of Decrement Register*

Boucle d'interaction de LISP

`read-eval-print`

- La boucle d'interaction de LISP se décompose en trois fonctions :
  - `read` : transforme une chaîne de caractères en S-expression ;
  - `eval` : transforme une S-expression en sa forme évaluée, une S-expression ;
  - `print` : affiche une S-expression sous la forme d'une chaîne de caractères.

$\Rightarrow$ "`read`" et "`print`" sont particulièrement simples à réaliser, reste "`eval`".

Exemples

```
;; Cette expression s'évalue en 3. "+" est une primitive.
(+ 1 2)
;; Cette expression définit un symbole x valant 1.
(define x 1)
;; Une fonction anonyme à deux arguments.
(lambda (x y) (+ x y))
;; La fonction identité appliquée à la liste vide.
((lambda (x) x) ())
;; "if" est une forme spéciale : seule une branche est évaluée.
(if (= x 2) (1 + 2) (3 + 4))
;; "quote" permet de retarder l'évaluation d'une expression
((lambda (x) (car (cdr x))) ('(1 2 3 4)))
```

Environnements d'interprétation des S-expressions

- LISP utilise deux environnements :
  - un environnement lexical Γ qui associe des S-expressions à des symboles ;
  - un environnement global Ξ qui enregistre les définitions de fonctions.

Interprétation des S-expressions

- L'évaluation est définie par cas sur le type de S-expressions à évaluer :

$$\begin{array}{rl} \forall e \in \{\texttt{int}, \texttt{spec}, \texttt{comp}, \texttt{prim}\} & \text{eval}\,(e) = e \\ \forall e \in \{\texttt{sym}\} & \text{eval}\,(e) = \Gamma(e) \\ \forall e \in \{\texttt{cons}\} & \text{eval}\,(e) = \text{apply}(\text{eval}(\texttt{car}(e)), \texttt{cdr}(e)) \end{array}$$

où apply est aussi définie par cas sur le type de son premier argument :

$$\begin{array}{rl} \forall f \in \{\texttt{comp}, \texttt{prim}\} & \text{apply}(f, (e_1 \ldots e_n)) = \text{eval}(\text{body}(f)) \\ \text{dans} & \Gamma' \equiv ((x_1\ \text{eval}(e_1)) \ldots (x_n\ \text{eval}(e_n))\ \Gamma) \\ \text{où} & \text{args}(f) = x_1 \ldots x_n \\ \forall f \in \{\texttt{sym}\} & \text{apply}(f, (e_1 \ldots e_n)) = \text{apply}(\Xi(f), (e_1 \ldots e_n)) \\ \forall f \in \{\texttt{spec}\} & \text{apply}(f, (e_1 \ldots e_n)) = \text{eval}(\text{body}(f)) \\ \text{dans} & \Gamma' \equiv ((x_1\ e_1) \ldots (x_n\ e_n)\ \Gamma) \\ \text{où} & \text{args}(f) = x_1 \ldots x_n \end{array}$$

Interprète LISP en O'Caml

Exercice

Écrivez un évaluateur de LISP en O'Caml.

Interprète LISP en LISP (de Paul Graham)

```
(defun null. (x)
  (eq x '()))

(defun and. (x y)
  (cond (x (cond (y 't) ('t '())))
        ('t '())))

(defun not. (x)
  (cond (x '())
        ('t 't)))

(defun append. (x y)
  (cond ((null. x) y)
        ('t (cons (car x) (append. (cdr x) y)))))

(defun list. (x y)
  (cons x (cons y '())))
```

Interprète LISP en LISP (par Paul Graham)

```
(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list. (car x) (car y))
               (pair. (cdr x) (cdr y))))))

(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))
```

Interprète LISP en LISP (par Paul Graham)

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ;; Special forms
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom) (atom (eval. (cadr e) a)))
       ((eq (car e) 'eq) (eq (eval. (cadr e) a) (eval. (caddr e) a)))
       ((eq (car e) 'car) (car (eval. (cadr e) a)))
       ((eq (car e) 'cdr) (cdr (eval. (cadr e) a)))
       ((eq (car e) 'cons) (cons (eval. (cadr e) a) (eval. (caddr e) a)))
       ((eq (car e) 'cond) (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a) (cdr e)) a))))
    ;; Anonymous function
    ((eq (caar e) 'lambda)
     (eval. (caddar e) (append. (pair. (cadar e) (evlis. (cdr e) a)) a)))))
```

Interprète LISP en LISP (par Paul Graham)

```
(defun evcon. (c a)
  (cond ((eval. (caar c) a)
         (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval. (car m) a) (evlis. (cdr m) a)))))
```

Portée dynamique

```
> (define foo 1)
foo
> (define add-foo (lambda (x) (+ x foo)))
add-foo
> ((lambda (foo) (add-foo 3)) 5)
4
```

```
> (define foo 1)
foo
> (define add-foo (lambda (x) (+ x foo)))
add-foo
> ((lambda (foo) (add-foo 3)) 5)
8
```

Exercice

Parmi ces deux comportements, lequel correspond à notre interprète ?

Fin de la digression...

Comment construire des fonctions ?

Par définition

La façon la plus simple d'écrire une fonction est bien sûr d'écrire son code, comme en programmation procédurale.

Par composition

Un langage fonctionnel permet facilement de **composer** deux fonctions afin d'en obtenir une nouvelle qui fait l'union de leurs comportements.

Par spécialisation

Le mécanisme de **currification** permet de transformer une fonction qui attend plusieurs arguments en une fonction qui attend un argument et produit une fonction avec un argument en moins.

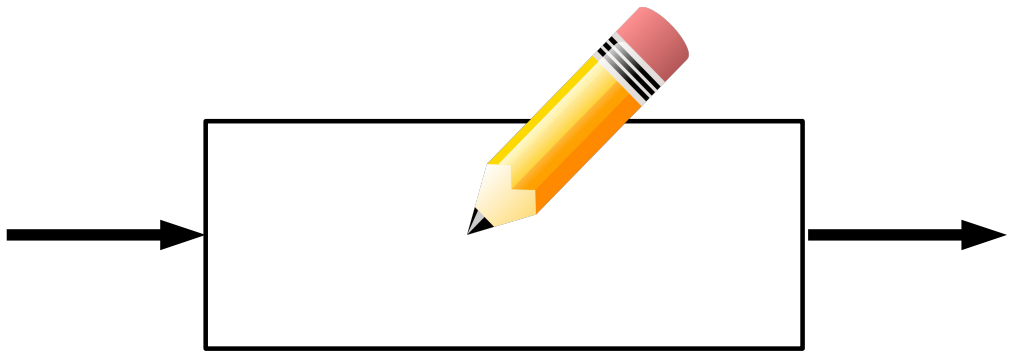
Par induction

Le style fonctionnel préconise la **définition de fonction par induction** sur son entrée : une telle fonction est un point fixe des fonctions définies pour chaque cas de l'induction.

Construire une fonction par définition

Par définition

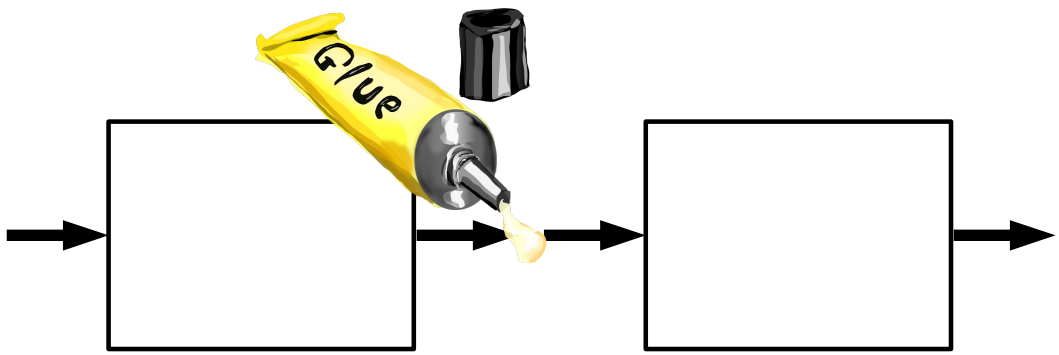
La façon la plus simple d'écrire une fonction est bien sûr d'écrire son code, comme en programmation procédurale.



Construire une fonction par composition

Par composition

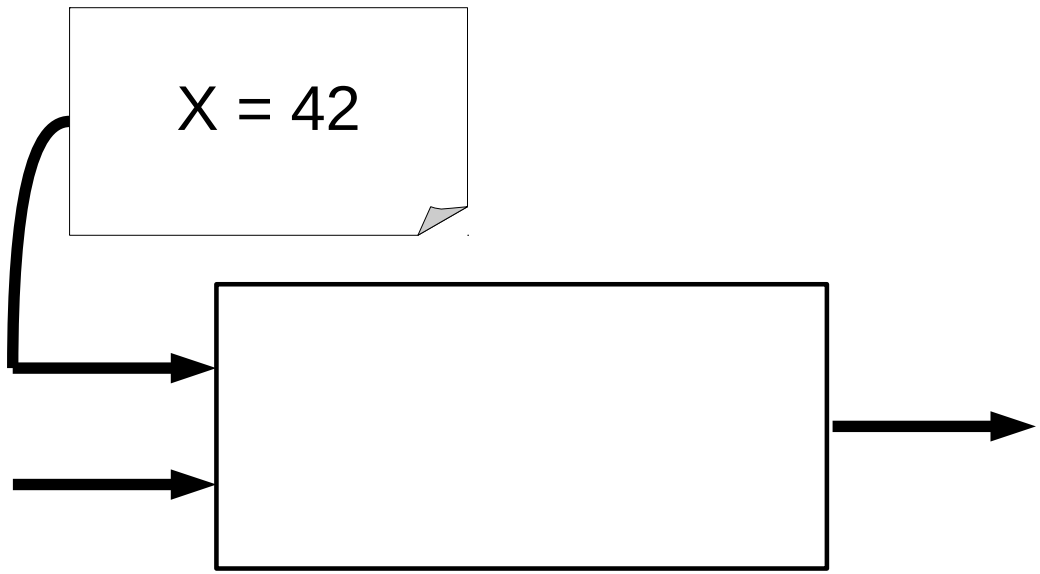
Un langage fonctionnel permet facilement de composer deux fonctions afin d'en obtenir une nouvelle qui fait l'union de leurs comportements.



Construire une fonction par spécialisation

Par spécialisation

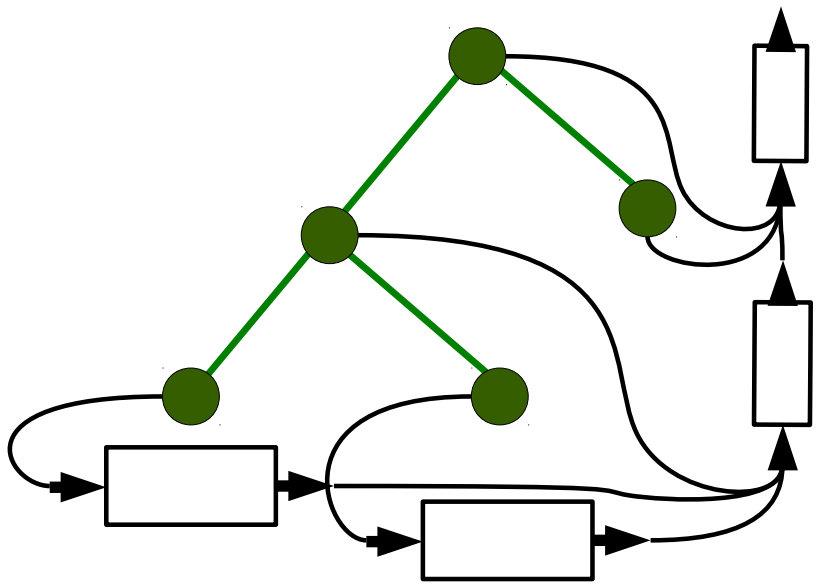
Le mécanisme de currification permet de transformer une fonction qui attend plusieurs arguments en une fonction qui attend un argument et produit une fonction avec un argument en moins.



Construire une fonction par induction

Par induction

Le style fonctionnel préconise la définition de fonction par induction sur son entrée : une telle fonction est un point fixe des fonctions définies pour chaque cas de l'induction.



Famille de langages fonctionnels

Il existe plusieurs familles de langages fonctionnels :

- Les **purs** (Haskell, Clean) n'autorisent pas les effets de bords (non contrôlés) tandis que les **impurs** (O'Caml, Scheme) les autorisent.
- Les **paresseux** (Haskell) ont une stratégie d'évaluation retardant les calculs jusqu'à ce que leurs résultats soient explicitement utilisés tandis que les **stricts** (O'Caml) suivent la stratégie habituelle qui calcule totalement les arguments des fonctions avant de les appliquer.
- Les **dynamiques** (Scheme) ne conditionnent pas leur exécution et leur compilation à la validation par une analyse statique à base de types tandis que les **statiques** (Haskell, O'Caml) le font.

Le problème des anagrames en fonctionnel

Reprenez le problème du calcul des anagrammes :

> *Soit un dictionnaire d, représenté par une liste de mots et un mot w. Calculer tous les mots de d qui sont des anagrammes de w.*

Pour le moment, n'utilisez pas de fonctions d'ordre supérieur et n'écrivez qu'une seule fonction non-récursive.

Version procédurale

```
let anagrams dict word =
  let canon word =
    string_of_array
      (mutate_with
           (Array.stable_sort compare) (array_of_string word))
  in
  let d = ref dict in
  let r = ref [] in
  while ( !d ≠ []) do
    if (canon (List.hd !d) = canon word) then
      r := (List.hd !d) :: !r;
    d := List.tl !d
  done;
  !r
```

Le problème des anagrames en fonctionnel

Écrivez maintenant une version utilisant des fonctions de la bibliothèque standard de O'Caml.

Version fonctionnelle, toujours assez itérative

```
let anagrams dict word =
  let canon word =
    string_of_array
      (mutate_with
          (Array.stable_sort compare) (array_of_string word))
  in
  let cword = canon word in
  List.filter (fun x → cword = canon x) dict
```

Cette code est encore assez itérative puisqu'elle itère une fonction de test sur une séquence de mots.

Le problème des anagrames en fonctionnel

Écrivez maintenant une version utilisant la currification pour que cette fonction soit plus efficace lorsque le dictionnaire est fixé pour plusieurs interrogations futures sur une liste de mots.

Version (peut-être plus) fonctionnelle

```
let anagrams dict =
  let module C = Map.Make (String) in
  let find d w = (try C.find (canon w) d with Not_found → []) in
  let union d w = C.add (canon w) (w :: find d w) d in
  let canonical_dict = List.fold_left union C.empty dict in
  find canonical_dict
```

La fonction `anagrams` appliquée sur un dictionnaire produit une fonction qui attend un mot et répond plus efficacement que la fonction de la version précédente grâce à un pré-calcul qui construit une structure de données intermédiaire plus efficace.

Version plus générale

La version du transparent précédent est dédiée aux dictionnaires de chaînes de caractères. Comment généraliser ce programme pour qu'il soit construit à l'aide de fonctions plus générales et donc plus réutilisables ?

Des concepts plus généraux

La notion importante de ce problème est celle de la gestion d'un ensemble de valeurs quotienté par une relation d'équivalence. On suppose de plus que l'on est capable de calculer un représentant canonique pour chaque classe d'équivalence.

Ce problème revient donc à calculer une fonction qui, à partir d'un tel ensemble, calcule la fonction qui associe à tout élément sa classe d'équivalence, si elle existe dans l'ensemble initiale.

Écrivez une nouvelle version de ce problème avec ce point de vue plus général !

Raisonnement sur un programme fonctionnel

Les fonctions écrites dans un style fonctionnel **ne dépendent que de leur entrée** : elles sont donc plus faciles à **tester unitairement** que les procédures qui, elles, dépendent d'une certaine configuration de l'état.

D'une manière générale, les langages fonctionnels sont des langages à expressions qui vérifient la propriété de **transparence référentielle** : on peut substituer toute sous-expression par son résultat sans modifier le sens de l'expression englobante. Ceci est dû au fait que des évaluations multiples d'un même expression conduisent au même résultat : **un programme fonctionnel ne dépend pas du temps**.

Du point de vue du raisonnement, cela signifie que le programmeur peut effectuer des **raisonnements équationnels** en substituant une expression par une expression équivalente pour comprendre ce que fait son programme. Ceci facilite le raffinement puisque l'on peut substituer une expression abstraite par une implémentation qui la raffine, et ce en tout point du programme. Enfin, les fonctions définies par induction sont naturellement **prouvables par un raisonnement inductif**.

Exemple de raisonnement sur un programme fonctionnel

Soit les fonctions O'CAML suivantes :

```
let sum = List.fold_left ( + ) 0
let double x = 2 × x
let double_list = List.map double
```

Montrons que `sum (double_list l) = 2 × sum l` par induction sur la liste `l` [2] .

Le cas de base : `sum (double_list []) = 0` et `2 × sum [] = 0`.

Le cas d'induction :

`sum (double_list (x :: xs)) = 2 × x + sum (double_list xs)`

Par l'hypothèse d'induction :

`sum (double_list (x :: xs)) = 2 × x + 2 × sum xs`

`sum (double_list (x :: xs)) = 2 × sum (x :: xs)`

2. Exemple tiré du cours de Didier Verna : http://www.lrde.epita.fr/~didier/lectures/afp_04_proof.pdf

Un autre exemple de raisonnement

Soit les fonctions O'CAML suivantes :

```
let rev_aux accu = function
| [] → accu
| x :: xs → rev_aux (x :: accu) xs

let rev = rev_aux []
```

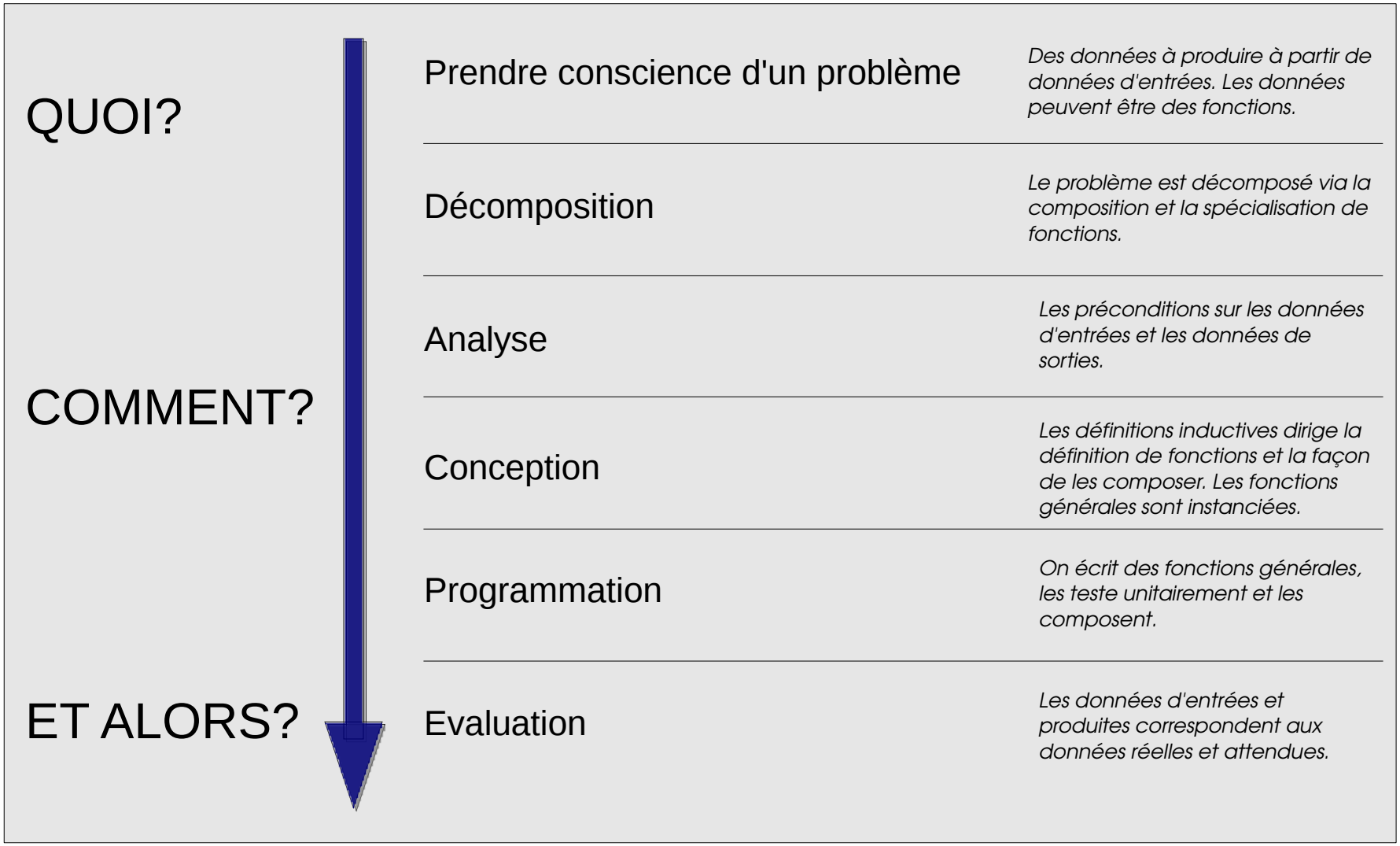
Montrez que `rev (rev l) = l` pour toute liste `l`.

Généraliser l'induction

Pour montrer l'égalité précédente, il faut généraliser l'induction en montrant pour toutes listes `l` et `l'` que :

- rev_{aux} `l' l = ` rev_{aux} `[] l @ l'`
- `rev (l @ l') = rev (l') @ rev (l)`

Le développement logiciel en programmation fonctionnelle



Avantages de la programmation fonctionnelle

Comme nous l'avons vu, la programmation fonctionnelle facilite le raisonnement.

Comme tout langage d'ordre supérieur, elle permet aussi d'écrire des composants plus **modulaires** qu'en programmation procédurale.
(Nous reviendrons sur ce point dans un prochain cours.)

La programmation fonctionnelle est **très expressive** : certains mécanismes calculatoires sont exprimables à l'aide de fonctions de première classe alors qu'ils ne le sont pas dans les langages procéduraux de la programmation structurée.
(Ce sera l'objet du prochain cours.)

Enfin, contrairement aux idées reçues, la programmation fonctionnelle permet d'écrire **des programmes qui s'exécutent efficacement**. Plusieurs raisons à cela : d'abord, nous allons voir que le contrôle des effets de bord rend possible de nombreuses optimisations *via* des **transformations de programmes**, ensuite, nous verrons dans un autre cours que les programmes fonctionnels se **parallélisent plus facilement** car ils n'utilisent pas d'état global. Enfin, de part leur haut niveau d'abstraction, les langages fonctionnels facilitent l'**implémentation d'algorithmes complexes** car le programmeur peut focaliser son attention sur les aspects importants en mettant de côté les détails de bas-niveau non pertinents.

Plan

Displiner l'état

Libérer le contrôle

- Révolution 1 : Écrire ses propres opérateurs de contrôle
- Révolution 2 : Explorer d'autres stratégies d'évaluation
- Révolution 3 : Créer et utiliser de nouveaux modèles de calcul

La fin de l'article de D. Knuth

À la fin de sa réflexion sur la programmation structurée, Donald Knuth fait remarquer que la question de mécanismes pour aider à la **structuration de l'état** est un problème ouvert et essentiel :

> *« Control structure is merely one simple issue, compared to questions of abstract data structure. I t will be a major problem to keep the total number of language features within tight limits. And we must especially look at problems of input/output and data formatting, in order to provide a viable alternative to CoBoL. »*
>
> *Donald Knuth – Structured Programming with Gotos*

Dans cette partie du cours, nous allons voir commencer les langages de programmation (fonctionnels) se sont abstraits progressivement de l'état et des gains en termes de déclarativité, de faciliter de raisonnement et d'opportunité d'optimisations que cette évolution a entrainés.

Un exemple récurrent

Soit une liste d'entiers naturels L, calculer la sous-liste P des entiers premiers de L.

On supposera que la liste des entiers est fournie dans un fichier dont le contenu peut croître irrégulièrement durant l'exécution du programme. Ce dernier doit s'arrêter de calculer la liste P seulement quand le fichier est fermé par son producteur.

On a de plus une garantie sur la valeur maximale M des entiers en fonction de la taille de la liste L : $M < |L|^2$.

État brut

L'état, dans sa forme la plus brute, peut être modélisé comme une mémoire de taille finie à laquelle on accède *via* des lectures et des écritures à des adresses. [3]

C'est sur un état de cette forme qu'un programme écrit en assembleur travaille (la plupart du temps) : **c'est au programmeur d'organiser lui-même l'agencement des données** dans un segment de la mémoire qui lui ai alloué statiquement.

De quelles opérations doit-il se munir pour résoudre le problème précédent ?

3. On met de côté les entrées/sorties pour le moment.

État brut

L'état, dans sa forme la plus brute, peut être modélisé comme une mémoire de taille finie à laquelle on accède *via* des lectures et des écritures à des adresses. [3]

C'est sur un état de cette forme qu'un programme écrit en assembleur travaille (la plupart du temps) : **c'est au programmeur d'organiser lui-même l'agencement des données** dans un segment de la mémoire qui lui ai alloué statiquement.

De quelles opérations doit-il se munir pour résoudre le problème précédent ?

Comme la taille des données n'est pas connue *a priori*, il faut des opérations d'allocation dynamique de l'espace mémoire pour stocker la liste des entiers restant à traiter, la liste P des entiers premiers calculées et un espace pour le crible d'Ératosthène. La liste de entiers restant à traiter grossit quand on lit de nouveau le contenu du fichier, tandis qu'elle décroît quand on traite les entiers qu'elle contient. Il faut donc aussi un procédé de désallocation dynamique.

3. On met de côté les entrées/sorties pour le moment.

Exercice

Dans le langage de votre choix, programmez le programme précédent dans un style procédural en ne travaillant que dans un espace mémoire linéaire alloué statiquement. (Par exemple, vous pouvez programmer en C dans un tableau de la forme `char memory[N]` où N est une constante.)

État abstrait

Les gestions manuelles de la mémoire ont rapidement montré leur limite quand les programmes sont devenues plus compliqués. C'est pour cela que l'on a introduit des modèles de plus en plus abstraits de cette mémoire, **fournissant des garanties et des primitives au programmeur** :

- ANSI C
- ANSI C + `malloc/free`
- LISP et les langages fonctionnels
- Les langages de flot de données

État abstrait

Les gestions manuelles de la mémoire ont rapidement montré leur limite quand les programmes sont devenues plus compliqués. C'est pour cela que l'on a introduit des modèles de plus en plus abstraits de cette mémoire, **fournissant des garanties et des primitives au programmeur** :

- ANSI C : Le langage C fournit une arithmétique de pointeurs qui permet d'effectuer des opérations de bas-niveau sur la mémoire. Cependant, ces opérations ne sont pas effectuées sur la mémoire vue comme un seul bloc mais sur des fragments "prédécoupés" de celle-ci. Ainsi, quand on prend l'adresse d'une variable, elle n'a *a priori* aucun lien avec l'adresse d'une autre variable.
- ANSI C + `malloc/free`
- LISP et les langages fonctionnels
- Les langages de flot de données

État abstrait

Les gestions manuelles de la mémoire ont rapidement montré leur limite quand les programmes sont devenues plus compliqués. C'est pour cela que l'on a introduit des modèles de plus en plus abstraits de cette mémoire, **fournissant des garanties et des primitives au programmeur** :

- ANSI C
- ANSI C + `malloc/free` : Certaines données ont une durée de vie inférieure au temps d'exécution total du programme. Les fonctions `malloc/free` de la bibliothèque standard de *C* offrent des opérations de gestion d'un **tas**, représentant un ensemble de données allouées dynamiquement.
- LISP et les langages fonctionnels
- Les langages de flot de données

État abstrait

Les gestions manuelles de la mémoire ont rapidement montré leur limite quand les programmes sont devenues plus compliqués. C'est pour cela que l'on a introduit des modèles de plus en plus abstraits de cette mémoire, **fournissant des garanties et des primitives au programmeur** :

- ANSI C
- ANSI C + `malloc/free`
- LISP et les langages fonctionnels : On peut prendre un parti plus extrême en ne présentant l'état du programme **qu'à travers des objets structurés allouées dynamiquement**, comme les listes. C'est le choix de LISP, qui a été le premier langage dans lequel les programmes sont des manipulateurs de données purement symboliques, représentées uniquement par des listes et dont la vie (l'allocation) et la mort (la désallocation) est gérée automatiquement.
- Les langages de flot de données

État abstrait

Les gestions manuelles de la mémoire ont rapidement montré leur limite quand les programmes sont devenues plus compliqués. C'est pour cela que l'on a introduit des modèles de plus en plus abstraits de cette mémoire, **fournissant des garanties et des primitives au programmeur** :

- ANSI C
- ANSI C + `malloc/free`
- LISP et les langages fonctionnels
- Les langages de flot de données : Les abstractions au-dessus de la mémoire peuvent prendre des formes variées. Les langages de flot de données fournissent une vue de l'état du programme dans laquelle une variable peut être en cours d'écriture et où la lecture peut bloquer. (Nous reviendrons sur cette notion quand nous parlerons de concurrence.)

État structuré *a priori*

Nous allons nous focaliser sur la structuration de l'état apportée par les langages fonctionnels. Deux outils fondamentaux pour cela sont :

- Un mécanisme calculatoire : les valeurs inductives. listes ou les arbres.
- Un mécanisme méta-linguistique : les types de données.

Les **types de données**, déjà utilisés par des langages procéduraux comme PASCAL, fournissent un langage de classification des données manipulées par le programme. Les types sont des informations **statiques** qui sont déterminées par l'une des premières passes des compilateurs, appelée typeur. Quand un programme est bien typé, cela signifie que le typeur a prouvé que toutes les opérations appliquées sur les valeurs sont licites.

Les types de données utilisés pour classifier les valeurs inductives comme les listes ou les arbres sont des **types algébriques**.

Les types algébriques

Type algébrique

Un **type algébrique** est défini par un ensemble de **constructeurs de données**. Chaque constructeur de données a une signature indiquant le type de ces arguments. La seule façon de construire une valeur d'un certain type algébrique est d'utiliser l'un de ses constructeurs de données.

En échange de cette restriction, on a la garantie que, quelque soit la valeur d'un certain type algébrique, on peut procéder à une **analyse par cas** sur la forme du constructeur qui a servi à la construire. On utilise pour cela une analyse de motifs.

Les types algébriques sont en général des **types récursifs** dont la définition peut-être résumée par une équation récursive de la forme :

$$T = C_1 : T_{11} \times \ldots T_{1N_1} + \ldots + C_M : T_{M1} \times \ldots T_{MN_M}$$

Par exemple, le type L des listes d'entiers est défini par l'équation :

$$L = Nil : Unit + Cons : Int \times L$$

Exercice

Réécrivez le programme de l'exercice précédent à l'aide des listes d'un langage fonctionnel de votre choix.

Les types abstraits

Type abstrait

Un **type abstrait** est un type dont on ne connaît que le nom et les opérations travaillant sur les valeurs de ce type en ignorant la façon dont les valeurs de ce type sont implémentées concrètement.

Les types abstraits sont le mécanisme fondamental pour obtenir la propriété d'**encapsulation** d'un composant logiciel. Nous reviendrons sur cette notion dans le cours sur la modularité.

Exercice

Donnez un type abstrait aux listes et aux flux des données issues d'un fichier.

État absent

Dans un langage fonctionnel pur, aucune notion d'état modifiable n'est accessible au programmeur (directement). Cette propriété garantit la transparence référentielle et rend possible le raisonnement équationnel.

Par ailleurs, comme le calcul effectué par le programme ne dépend plus de l'état, et donc de l'ordre dans lequel celui-ci est modifié, on gagne l'opportunité d'**évaluer le programme dans un ordre arbitraire** (en étant tout de même obligés d'évaluer les arguments d'une fonction quand le calcul décrit par celle-ci en a besoin). Le langage HASKELL tire parti de cela en proposant, par défaut, une évaluation paresseuse des expressions.

Nous allons voir les multiples intérêts de cette notion.

Exercice

Pouvez-vous réécrire entièrement votre programme pour le rendre fonctionnel pur ? Si non, pour quelle partie du programme vous semble-t-il impossible de le faire ?

Les données persistentes

Persistence

Une donnée est **persistente** si les opérations que l'on peut lui appliquer n'invalident pas (observationnellement) ses versions précédentes mais produisent au contraire de nouvelle version de cette donnée.

Ce n'est pas parce qu'une donnée est persistente qu'elle n'est pas modifiable.

Exercice :
Implémenter une version persistente des arbres de recherche nommés *splay trees* et des listes *skip-lists*.

Des structure de données rétroactives

Retroactivité

Une structure de donnée est **rétroactive** si on peut défaire efficacement une des opérations qui l'a produite.

Des valeurs intermédiaires gênantes

Supposons que nous étendions notre exemple en rajoutant un critère supplémentaire pour filtrer la liste des entrées, comme par exemple, que les entiers doivent être congrus à 3 modulo 7.

Dans un langage fonctionnel, il est naturel de composer les fonctions de filtrage ainsi :

```
let filter1 = List.filter is_prime
let filter2 = List.filter is_congruent_to_3_mod7
let p = filter2 (filter1 l)
```

Cependant, ce programme souffre d'une certaine inefficacité en espace puisqu'une liste intermédiaire est créée puis immédiatement jetée. On aimerait que ce programme soit équivalent à :

```
let p =
List.filter (fun x → is_congruent_to_3_mod7 x ∧ is_prime) l
```

Sans avoir à écrire cette version spécialisée contraire au principe de modularité et de réutilisation.

La déforestation

La transformation précédente, appelée **déforestation**, est justifiée dans le cas général lorsque les effets de bords sont interdits car on a alors :

$$\texttt{List.filter p1 (List.filter p2 l)} = \texttt{List.filter (}\textbf{fun}\ \texttt{x} \rightarrow \texttt{p1} \wedge \texttt{p2)\ l}$$

En HASKELL, il est possible de déclarer ce genre d'égalité entre applications de fonction. (Attention, le compilateur ne les vérifie pas donc il faut les avoir vérifier scrupuleusement sur papier ou en Coq par exemple.) Le compilateur peut ensuite en tirer parti en **transformer systématiquement les instances de cette règle apparaissant dans le code source en utilisant cette égalité** et ainsi appliquer des optimisations de haut-niveau sur les programmes.

État caché localement

Les langages fonctionnels impurs, tout en préconisant une programmation avec le moins d'effet de bords que possible, ne les interdisent pas pour des raisons d'efficacité. En effet, en général, certains algorithmes nécessitent des opérations destructives pour offrir des complexités

Pour minimiser l'impact des effets de bords, on a coutume d'essayer de les **encapsuler de façon à ce que leur portée soit locale**. Pour cela, il existe plusieurs techniques. Une technique très simple consiste à utiliser l'environnement des fermetures lexicales pour y stocker l'état sur lequel travaille localement une fonction. En voici deux exemples d'application classiques de cette méthode :

```
let fresh =
  let c = ref 0 in
  fun () → incr c ; !c

let memo f =
  let h = Hashtbl.create 73 in
  fun x → try
    Hashtbl.find h x
  with Not_found → let v = f x in Hashtbl.add h x v ; v
```

Quelles différences faites-vous entre ces deux fonctions ?

État contrôlé

Pour terminer cette partie du cours consacré à la gestion de l'état des programmes, nous allons introduire la notion de **monade**. Une monade `M t` est un type abstrait paramétré qui **représente un calcul dans un modèle calculatoire particulier**. Ce type abstrait est fourni avec (au moins) les deux opérations suivantes :

- L"'unité" (`return`) de type `t → M t` qui plonge toute valeur de type `t` dans le modèle calculatoire de la monade.
- La "liaison" (`bind`, notée `>>=`) de type `M t → (t → M u) → M u` qui décrit la façon dont les calculs se composent.

Nous allons voir comment on peut embarquer un modèle calculatoire, comme celui de la programmation impérative, dans un programme fonctionnel pur à l'aide des monades.

Lois monadiques

Les opérations d'une monade doivent vérifier les égalités suivantes :

- `(return x) >>= f ≡ f x`
- `(m >>= return) ≡ m`
- `((m >>= f) >>= g) ≡ (m >>= (fun x → f x >>= g))`

La monade de listes

La **monade des listes** est une des monades les plus simples : elle décrit le modèle calculatoire des « résultats multiples ». On peut écrire un programme comme si chaque étape de son calcul ne produisait qu'un résultat et la monade s'occupe de combiner l'ensemble des combinaisons possibles. Voici la signature de ce type abstrait :

```
(* Monad type which represents a list of results. *)
type α m = α list

(* [bind x f] applies [f] to a list of results, *)
(* returning a list of results. *)
val bind : α m → (α → β m) → β m
val ( >>= ) : α m → (α → β m) → β m

(** [return x] is the left and right unit of [bind]. *)
val return : α → α m
```

Implémentez-la !

La monade de listes

```
type α m = α list

let return x = [ x ]

let bind l f = List.flatten (List.map f l)

let ( >>= ) l f = bind l f
```

Montrez que les lois monadiques sont respectées par cette implémentation.

La monade d'état

Soit un type `state` représentant un état. La **monade d'état** modélise un calcul qui transforme cet état. En voici la signature :

type α t
val return : $\alpha \to \alpha$ t
val bind : α t $\to (\alpha \to \beta$ t$) \to \beta$ t
val get : state t
val put : state $\to$ unit t
val run : α t $\to$ state $\to (\alpha \times$ state$)$

Implémentez cette monade et écrivez les équations vérifiées par ses opérations.

Exercice

Reprenez l'exemple de programme et essayez de l'écrire en fonctionnel pur. (Pour cela, vous pouvez utiliser le langage HASKELL si vous le connaissez ou à défaut O'CAML.)

Plan

Displiner l'état

Libérer le contrôle

- Révolution 1 : Écrire ses propres opérateurs de contrôle
- Révolution 2 : Explorer d'autres stratégies d'évaluation
- Révolution 3 : Créer et utiliser de nouveaux modèles de calcul

Les programmes du premier ordre sont “monolithiques”

Pour un programmeur habitué à la programmation dans un style procédural, les fonctions de première classe vont d'abord apparaître comme un procédé pour **abstraire** un fragment de code par rapport à un autre.

Par exemple, dans un langage procédural du premier ordre comme C, on ne peut pas “découper” facilement le programme suivant :

```
for (unsigned i = 0 ; i < N ; ++i) {
   t[i] -= sqrt (t[i])
}
```

en deux programmes :

```
for (unsigned i = 0 ; i < N ; ++i)
   ?
```

```
? {
   t[i] -= sqrt (t[i])
}
```

(mais d'où viennent ce `i` et ce `t` ?)

Quand les “blocs” sont des fonctions de première classe

Dans un style fonctionnel, on peut écrire les programmes `I` et `F` suivants et les **composer** par application : `I F`.

```
f =>
  for (unsigned i = 0 ; i < N ; ++i)
    f i
```

L'itérateur I

```
i => {
  t[i] -= sqrt (t[i])
}
```

L'itéré F

La fonction d'ordre supérieur `I` est un itérateur spécialisé sur le segment d'entiers $[0; N[$. Une fois définie, elle peut être vue comme un nouvel opérateur de contrôle du langage, capturant une abstraction plus précise que celle du `for` habituel.

L'itéré `F` est une fonction de première classe qui attend un entier `i` et applique une opération sur l'élément d'indice `i` du tableau `t`. Notez que la variable `t` n'est pas fournie en argument, elle fait référence à la valeur `v` de `t` au moment où la fonction `F` a été créée. Une fonction de première classe **capture** l'environnement lexical actif au moment de sa création. Cette opération est nécessaire pour permettre l'**exécution future** de cette fonction.

Exemple 1 : Variations autour de `while`

À l'aide des **définitions récursives de fonction** et des **fonctions d'ordre supérieur**, on peut programmer les opérateurs de contrôle de la programmation structurée. Par exemple, l'opérateur de contrôle `while` s'écrit en O'CAML :

```
let rec while_ cond body =
  if cond () then (body (); while_ cond body)
```

Exercice : Programmer les opérateurs de contrôle suivants.

- "`do something while (condition)`"
- "`coordinate produce consume`"
- "`until` *event*$_1$ `or ... or` *event*$_n$ `do something and then` *event*$_1$ `=> ... | ... |` *event*$_n$ `=> ...`"

“do something while (condition)”

```
let rec do_while cond body =
  body ();
  if cond () then do_while cond body
```

“coordinate produce consume”

Cet opérateur de contrôle de haut-niveau sert à réaliser une interaction asynchrone entre un producteur et un consommateur, dans laquelle c’est le producteur qui est responsable d’arrêter l’interaction.

Exercice : Écrivez cet opérateur, d’une façon qui vous semble raisonnable et appliquez-le au problème du dernier cours. (Le filtrage des nombres entiers écrits irrégulièrement dans un fichier.)

“coordinate produce consume”

Voici une version de cet opérateur utilisant une file pour établir une communication entre le producteur et le consommateur.

```
let coordinate produce consume init =
  let work_list = Queue.create () in
  let rec aux accu =
    match produce () with
      | 'End → Queue.fold consume accu work_list
      | 'Some x → Queue.add x work_list ; aux accu
      | 'Nothing →
        try
          aux (consume accu (Queue.take work_list))
        with Queue.Empty → aux accu
  in
  aux init
```

“coordinate produce consume”

```
let read_char =
  let buffer = String.create 1 in
  fun cin → try
    if Unix.read cin buffer 0 1 = 1 then ‘Some (buffer.[0]) else ‘Nothing
  with
    | Unix.Unix_error (Unix.EAGAIN, _, _) → ‘Nothing
    | Unix.Unix_error (err, _, _) →
      Printf.printf "Stopping because ‘%s’." (Unix.error_message err);
      ‘End

let producer cin =
  Unix.set_nonblock cin;
  let push, flush =
    let current_entry = Buffer.create 42 in
    (Buffer.add_char current_entry,
    (fun () →
      let x = int_of_string (Buffer.contents current_entry) in
      Buffer.clear current_entry ; x))
  in
  fun () →
    match read_char cin with
      | ‘Some c when c = ’,’ → ‘Some (flush ())
      | ‘Some c → push c; ‘Nothing
      | ‘End → ‘End
      | ‘Nothing → ‘Nothing
```

“coordinate produce consume”

```
let isqrt x = int_of_float (sqrt (float_of_int x))

let is_prime n =
  (n > 1) ∧
    (let rec f x =
      (x = 1)
    ∨ (n mod x ≠ 0 ∧ f (x - 1))
    in f (isqrt n))

let consumer accu x =
  if is_prime x then Printf.printf "%d %!" x
```

“`until` $event_1$ `or ... or` $event_n$ `do something and then` $event_1$ `=> ... | ... |` $event_n$ `=> ...`“

Cet opérateur de contrôle sert à traiter les sorties prématurées de boucle. Tant qu'un événement parmi une liste d'événements n'est pas advenu, on itère un calcul. Ensuite, on poursuit le calcul en fonction de l'événement qui a conduit à arrêter l'itération.

Exercice :
Écrivez une fonction d'ordre supérieur qui réalise cet opérateur.

“until $event_1$ or ... or $event_n$ do something and then $event_1$ => ... | ... | $event_n$ => ...“

```
let until events body init =
  let rec aux accu =
      match body accu with
        | ‘Continue accu → aux accu
        | ‘Stop event → event accu
  in
  aux init
```

“`until` $event_1$ `or ... or` $event_n$ `do something and then` $event_1$ `=> ... | ... |` $event_n$ `=> ...`“

```
let find y a =
  let rec do_ init =
    until [found ; not_found]
      (fun i →
        if i = 0 then ‘Stop not_found
        else if !a.(i) = y then ‘Stop found
        else ‘Continue (i - 1))
  and not_found i = a := Array.append !a [|y|] ; found i
  and found i = i
  in
  do_ (Array.length !a)
```

(Notez que la fonction `do_` n’est pas récursive. On utilise le mot-clé `rec` pour pouvoir définir les fonctions `found` et not_{found} après la définition de `do_`.)

L'abstraction du contrôle

Grâce aux fonctions d'ordre supérieur, le programmeur n'est plus limité aux seules boucles `while` (issue de la programmation structurée) pour décrire le flot de contrôle de son programme. Des "protocoles" arbitrairement complexes (mais déterministes) peuvent ainsi être définis pour décrire la façon dont les fonctions se "**transmettent le contrôle**".

En particulier, l'expressivité de l'ordre supérieur apparaît pleinement quand on prend conscience qu'**un appel d'une fonction n'est pas nécessairement obligé de rendre le contrôle à son appelant**. Il est possible de continuer à calculer `f x` *ad eternam* sans jamais rendre la main à son appelant.

Nous allons maintenant voir comment on peut écrire ce genre de programmes fonctionnels et à quoi ils peuvent servir.

Les co-routines

Les co-routines ont été introduites par Conway en 1963. Il s'agit d'une généralisation des sous-routines. Une sous-routine n'a qu'un point d'entrée tandis qu'une co-routine en a plusieurs.

De plus, les séquences d'appels des sous-routines suivent une discipline de pile : on sort d'une sous-routine appelante une fois que la sous-routine appelée a terminé son calcul.

Avec les co-routines, il se passe une **inversion du contrôle** : la co-routine appelée prend en main de contrôle et peut en faire ce qu'elle veut : soit redonner la main à la co-routine appelante (dans ce cas, on simule des sous-routines à l'aide de co-routines) ou bien donner la main à une autre co-routine.

Les opérateurs de contrôle des co-routines

Supposons donner les opérateurs de contrôle suivants :

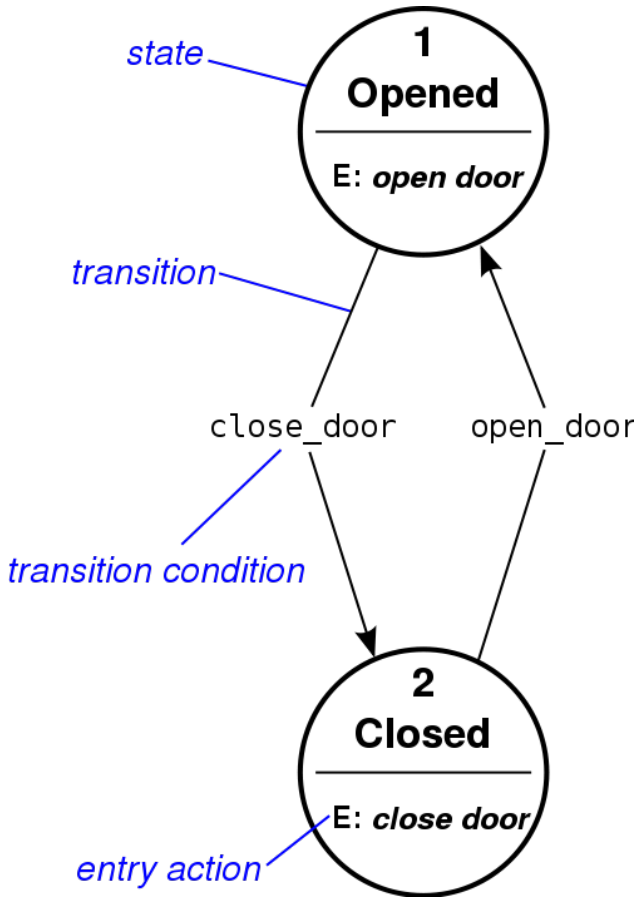
- `yield e` : retourne la valeur de `e` à l'appelant d'une co-routine et "se souvient" de l'endroit où l'on se trouve dans le code de cette co-routine appelée.
- `resume e` : redonne la main à une co-routine (l'exécution se poursuit à l'endroit où on l'avait laissé).

À l'aide de ces opérateurs, on peut par exemple implémenter des **diagrammes d'états-transitions** de façon modulaire.

Diagrammes d'états-transitions

Un diagramme d'états-transitions est un diagramme dont les états contiennent chacun un événement, une condition de garde et une liste d'actions et dont les transitions sont étiquetées par des conditions de sortie.

Exercice :
Modéliser notre exemple à l'aide d'un tel diagramme.



Co-routine en O'Caml

Contrairement à d'autres langages comme C#, LUA, F# ou Ruby, O'Caml ne propose de co-routines par défaut. Elles sont implémentées à l'aide d'une bibliothèque appelée Ocoro, basée sur une monade de co-routine :

```
(* Coroutines with input 'i and output 'o *)
type ('i,'o) coro
(* Coroutine monad (for use in coroutine bodies) *)
type ('i,'o,α) cm
(* Inject a value into the monad *)
val return : α → ('i,'o,α) cm
(* Monad bind *)
val (>>=) : ('i,'o,α) cm → (α → ('i,'o,β) cm) → ('i,'o,β) cm
(* Monad bind, discarding the value *)
val (») : ('i,'o,α) cm → ('i,'o,β) cm → ('i,'o,β) cm
(* create coroutine from body function *)
val create : ('i → ('i,'o,'o) cm) → ('i,'o) coro
(* Asymmetric coroutine operator: resume a suspended coroutine. *)
val resume: ('i,'o) coro → 'i → 'o
(* Complement to resume: suspend the current coroutine, return to caller *)
val yield : 'o → ('i,'o,'i) cm
```

Générateurs à l'aide de co-routines

```
(* s = init :: [ f x | x <- s ] *)
let from_fun init (frec : α → α) : (unit, α) coro =
  let rec routine accu () : (unit, α, α) cm =
    (yield accu) >>= (routine (frec accu))
  in
  create (routine init)

(* map f s = [ f x | x <- s ] *)
let map (f : α → β) : (unit, α) coro → (unit, β) coro =
  fun co →
    let rec aux () : (unit, α, β) cm =
      (yield (f (resume co ()))) >>= aux
    in
    create aux

(* naturals = 0 :: [ x + 1 | x <- naturals ] *)
let naturals = from_fun 0 succ
(* squares = [ x * x | x <- naturals ] *)
let squares = map (fun x → x × x) naturals
```

Générateurs à l'aide de co-routines

Exercice : Implémentez les fonctions suivantes :

- `zip : (unit, α) coro → (unit, β) coro → (unit, α × β) coro`
- `filter : (unit, α) coro → (α → bool) → (unit, α) coro`

Un moyenneur

issu de la distribution de OCORO

Une co-routine est plus générale qu'un générateur car on peut **communiquer** une valeur à une co-routine quand on la réactive :

```
let averager_f (k: int) : (int,string,string) cm =
    let rec avg n sumk : (int,string,string) cm =
      yield (string_of_int (sumk/n))
      >>= (fun k → avg (n+1) (sumk + k))
    in avg 1 k
 in let averager1_co = create averager_f
 in let averager2_co = create averager_f
 in let x1 = resume averager1_co 1
 in let x2 = resume averager1_co 3
 in let y1 = resume averager2_co 10
 in let x3 = resume averager1_co 5
 in let y2 = resume averager2_co 30
 in ...
```

Que valent `x1;x2;y1;x3` et `y2` ?

Filtrage de nombre premiers par co-routines

Exercice :
Écrivez une version à base de co-routines du programme de filtrage de nombres premiers.

Sur le même thème : les co-structures

L'idée des co-routines consistant à transférer le contrôle à une fonction peut être étendue aux structures de données.

Si on considère la fonction `List.map` en O'CAML :

```
let rec map f = function
| [] → []
| x :: xs → f x :: map f xs
```

Cette fonction **observe la liste** et décide en fonction de cette observation si le calcul doit se poursuivre ou s'arrêter.

Comment représenter différemment une liste de façon à ce que cette décision concernant la continuation du calcul soit internalisée par la structure de données ?

Une co-structure : le flux

Un flux est une co-structure de données de la forme `Stream (next, s)` où `s` est l'état courant du flux et `next` est la fonction qui décide de la suite du calcul :

```
let map f =
  fun (Stream (next0, s0)) →
    let next = fun s → match next0 s with
        | Done → Done
        | Yield (x, s') → Yield (f x, s')
        | Skip s' → Skip s'
    in
      Stream (next, s0)
```

Exercice : Quels sont les types des identificateurs de ce programme ?

Une co-structure : le flux

Un flux est une co-structure de données de la forme `Stream (next, s)` où `s` est l'état courant du flux et `next` est la fonction qui décide de la suite du calcul :

```
let map (f : α → β) : α stream → β stream =
  fun (Stream (next0 : ? → α step, s0 : ?)) →
    let next = fun s → match next0 s with
        | Done → Done
        | Yield (x, s') → Yield (f x, s')
        | Skip s' → Skip s'
    in
      Stream (next, s0)
```

Le type de l'état du flux `s` doit apparaître quelque part...

Une première proposition

```
type ('s, 'x) stream =
  | Stream of ('s → ('x, 's) step) × 's

and ('x, 's) step =
  | Done
  | Yield of 'x × 's
  | Skip of 's

let map f =
  fun (Stream (next0, s0)) →
    let next = fun s → match next0 s with
      | Done → Done
      | Yield (x, s') → Yield (f x, s')
      | Skip s' → Skip s'
    in
    Stream (next, s0)
```

Pourquoi cette solution n'est-elle pas satisfaisante ?

Cachez donc cet état que je ne saurais voir...

Quels sont les types des valeurs suivantes ?

```
let naturals = Stream ((fun s → Yield (s, s + 1)), 0)
let null = Stream ((fun () → Yield (0, ())), ())
```

D'un côté, `naturals` a le type `(int, int) stream` tandis que `null` a le type `(unit, int) stream`. Ce sont deux types incompatibles ce qui interdit des interactions légitimes entre ces deux flux.

Le problème vient de **l'exposition du type de l'état** des flux. On voudrait au contraire **cacher ces détails d'implémentation**.

Une idée ?

Les types existentiels

Une façon de résoudre le problème précédent consiste à donner à tous les flux produisant des valeurs de type α, le type : $\exists\beta.\ (\beta,\ \alpha)\ \texttt{stream}$. Les valeurs de ce type ont été construites avec une certaine implémentation de type β dont **on veut faire oublier l'exacte définition concrète au typeur**.

De cette façon, le programmeur s'interdit lui-même de pouvoir tirer parti de l'implémentation particulière de certains flux mais, en contrepartie, gagne l'autorisation de traiter tous les flux de façon homogène.

Malheureusement, il n'existe pas de syntaxe pour les types existentiels en O'Caml (contrairement à Haskell). Cependent, on peut les coder facilement à l'aide des modules et des champs d'enregistrement polymorphes...

Types existentiels en O'Caml

```
module Make(X : sig type (α, 'x) t end) : sig
  type 'x t
  val pack : (α, 'x) X.t → 'x t
  type (α, 'x) user = { f : β. (β, 'x) X.t → α }
  val use : (α, 'x) user → 'x t → α
end = struct
  type (α, 'x) user = { f : β. (β, 'x) X.t → α }
  type 'x t = { pack : α. (α, 'x) user → α }
  let pack impl = { pack = fun user → user.f impl }
  let use f p = p.pack f
end

type ('s, 'x) raw_stream =
  | Stream of ('s → ('x, 's) step) × 's

and ('x, 's) step =
  | Done
  | Yield of 'x × 's
  | Skip of 's

module Stream = Make (struct type (α, 'x) t = (α, 'x) raw_stream end)
open Stream
type 'x stream = 'x Stream.t
```

Types existentiels en O'Caml

```
module Make(X : sig
  (* Le module X fournit le type dont on veut cacher *)
  (* une composante. *)
  type (α, 'x) t
end) : sig
  (* Le foncteur Make va produire 'x t = ∃ 'a. ('a, 'x) t *)
  type 'x t
  (* pack transforme une valeur du type concret *)
  (* en une valeur du type existentiel.*)
  val pack : (α, 'x) X.t → 'x t
  type (α, 'x) user = { f : β. (β, 'x) X.t → α }
  (* Pour utilise une valeur du type existentiel, *)
  (* on fournit une fonction polymorphe vis-à-vis de ce *)
  (* qui est caché. *)
  val use : (α, 'x) user → 'x t → α
end = struct
  type (α, 'x) user = { f : β. (β, 'x) X.t → α }
  type 'x t = { pack : α. (α, 'x) user → α }
  let pack impl = { pack = fun user → user.f impl }
  let use f p = p.pack f
end
```

Nouvelle définition des flux

```
type ('s, 'x) raw_stream =
  | Stream of ('s → ('x, 's) step) × 's

and ('x, 's) step =
  | Done
  | Yield of 'x × 's
  | Skip of 's

module Stream = Make (struct type (α, 'x) t = (α, 'x) raw_stream end)
open Stream
type 'x stream = 'x Stream.t
```

Nouvelle définition de `map`

```
let map f s =
  use { f =
        fun (Stream (next0, s0)) →
          let next = fun s → match next0 s with
            | Done → Done
            | Yield (x, s') → Yield (f x, s')
            | Skip s' → Skip s'
          in
          pack (Stream (next, s0))
        } s
```

Quels sont les types des identificateurs de ce programme ?

Un autre exemple

```
let from_fun init f =
  let next accu = Yield (accu, f accu) in
  pack (Stream (next, init))

let naturals = from_fun 0 succ
let squares = map (fun x → x × x) naturals
```

Quels sont les avantages de flux sur les listes ?

Exercice

Programmez les fonctions suivantes :

val of_{list} : α list $\to \alpha$ Stream.t
val to_{list} : α Stream.t $\to \alpha$ list
val $from_{fun}$: $\alpha \to (\alpha \to \alpha) \to \alpha$ Stream.t
val take : int $\to \alpha$ Stream.t $\to \alpha$ list $\times \alpha$ Stream.t
val zip : α Stream.t $\to \beta$ Stream.t $\to (\alpha \times \beta)$ Stream.t
val append : α Stream.t $\to \alpha$ Stream.t $\to \alpha$ Stream.t
val foldl : $(\alpha \to \beta \to \alpha) \to \alpha \to \beta$ Stream.t $\to \alpha$
val foldr : $(\alpha \to \beta \to \beta) \to \beta \to \alpha$ Stream.t $\to \beta$
val $concat_{map}$: $(\alpha \to \beta$ Stream.t$) \to \alpha$ Stream.t $\to \beta$ Stream.t

Retour sur les co-routines

Comment implémenter les co-routines en O'CAML ?

```
(* Coroutines with input 'i and output 'o *)
type ('i,'o) coroutine
type ('i,'o) coroutine_body
(* create coroutine from body function *)
val create : ('i, 'o) coroutine_body → ('i,'o) coroutine
(* Asymmetric coroutine operator: *)
(* resume a suspended coroutine. *)
val resume: ('i,'o) coroutine → 'i → 'o
(* Complement to resume: *)
(* suspend the current coroutine, return to caller *)
val yield : ('i, 'o) coroutine → 'o → 'i
```

Retour sur les co-routines

Comment implémenter les co-routines en O'CAML ?

```
(* Coroutines with input 'i and output 'o *)
type ('i,'o) coroutine
type ('i,'o) coroutine_body
(* create coroutine from body function *)
val create : ('i, 'o) coroutine_body → ('i,'o) coroutine
(* Asymmetric coroutine operator: *)
(* resume a suspended coroutine. *)
val resume: ('i,'o) coroutine → 'i → 'o
(* Complement to resume: *)
(* suspend the current coroutine, return to caller *)
val yield : ('i, 'o) coroutine → 'o → 'i
```

Pour suspendre et reprendre un calcul, il faut se donner un mécanisme pour expliciter "la suite de ce calcul" comme un objet de première classe.

La réification du contrôle

Soit l'expression O'CAML suivante :

$$(\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } 2) \bullet \times 100$$

Imaginons que l'on souhaite **représenter** la suite du calcul à partir du symbole $\bullet$ (juste après avoir évaluer l'expression conditionnelle mais avant de faire la multiplication finale).

La fonction suivante peut très bien faire l'affaire :

$$\textbf{let } cont = \textbf{fun } (k : int) \rightarrow k \times 100$$

En effet, l'expression suivante est équivalente à l'expression initiale :

$$cont\ (\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } 2)$$

Les continuations du calcul

La fonction `cont` est une représentation fonctionnelle d'une **continuation de première classe**. En écrivant des programmes manipulant des continuations de première classe, le programmeur peut écrire des opérateurs de contrôle très fins.

Par exemple, si on se donner une fonction `abort` servant à sortir de façon prématurée d'une itération, on peut se contraindre à écrire les fonctions récursives de son programme de la façon suivante :

```
let while_
      (condition :  unit → bool)
      (f :  abort: (unit → unit) → continue: (unit → unit) → (unit → unit)) =
   let rec aux () =
      f
         ~abort: (fun () → ())
         ~continue: (fun () → if condition () then aux ())
         ()
   in
   aux ();;

let c = ref 10 in
while_ (fun () → !c > 0) (fun ~abort ~continue →
   Printf.eprintf "%d\n%!" !c; decr c;
   if !c < 5 then abort else continue)
```

Le style de programmation par continuations

Toute expression e du $\lambda-$calcul peut être transformée en une expression $[\![e]\!]$ explicitant la continuation du calcul.

Les programmes en style CPS sont des fonctions qui attendent la suite du calcul sous la forme de leur argument formel `k`.

Par exemple :

- "`fun x → x + x`" donne en CPS "`fun k x → k (x + x)`"
- "`(fun x → x + 1) (40 + 1)`" donne en CPS :

```
fun k →
    (fun k' → k' (40 + 1))
      (fun k'' x → k (x + 1))
```

Mise en forme CPS

Mettez le programme suivant en forme CPS :

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

Mise en forme CPS

Mettez le programme suivant en forme CPS :

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

On obtient :

```
let rec cps_fact k n =
  if n = 0 then k 1 else cps_fact (fun x → k (n × x)) (n - 1)

val fact : int → int
val cps_fact : (int → α) → int → α
```

Remarquez que `fact n = cps_fact n (fun x → x)`

La monade des continuations

Implémentez la monade de continuation.

La continuation courante : Call/CC

Mettre un programme en forme CPS à la main est peu commode. Pour utiliser l'expressivité des continuations tout en restant en style direct, un certain nombre d'opérateurs ont été conçus.

Les opérateurs `callcc` et `throw` servent à manipuler des continuations de première classe. `callcc` attend une fonction en argument et l'applique à la continuation courante. `throw` évalue une continuation en lui passant un argument.

En O'Caml, il existe une bibliothèque `ocaml-callc` dont la signature est :

type α cont
val callcc : $(\alpha \text{ cont} \to \alpha) \to \alpha$
val throw : $\alpha \text{ cont} \to \alpha \to \beta$

Exemple

let f x = (callcc (**fun** k → 1 + throw k x)) × 6

Que vaut `f 6` ?

Exercice : Implémentation des co-routines

Implémentez la signature des co-routines à l'aide de `callcc` :

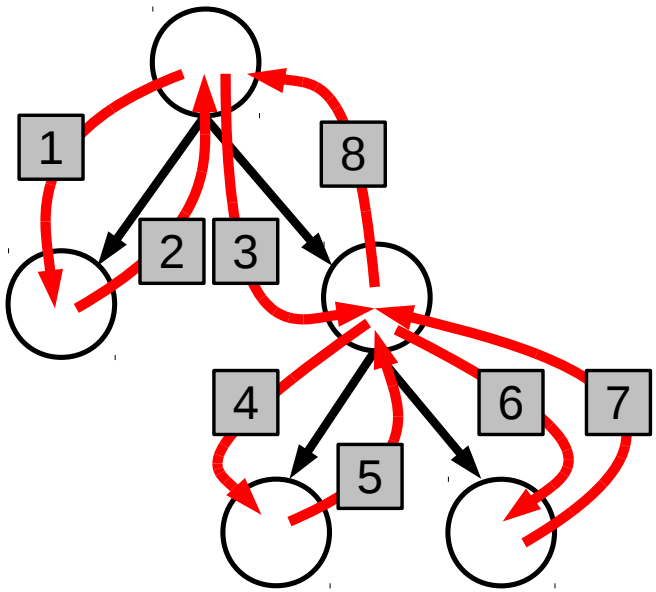
```
(* Coroutines with input 'i and output 'o *)
type ('i,'o) coroutine
type ('i,'o) coroutine_body
(* create coroutine from body function *)
val create : ('i, 'o) coroutine_body → ('i,'o) coroutine
(* Asymmetric coroutine operator: *)
(* resume a suspended coroutine. *)
val resume: ('i,'o) coroutine → 'i → 'o
(* Complement to resume: *)
(* suspend the current coroutine, return to caller *)
val yield : ('i, 'o) coroutine → 'o → 'i
```

La pile d'appels : le centre névralgique du contrôle

Le mécanisme d'appel de procédures s'appuie une **pile** : avant de déplacer le contrôle vers le code d'une procédure, l'appelant pousse l'adresse de l'instruction – typiquement celle suivant l'appel – qu'il souhaite exécutée lorsque la procédure lui redonnera la main. La pile est donc une représentation interne de la continuation du calcul. Dans un langage procédural, la discipline de pile est suivie à la lettre et impose que le contrôle suive un parcours infixe dans un arbre dont les nœuds sont les procédures et les arêtes les appels.

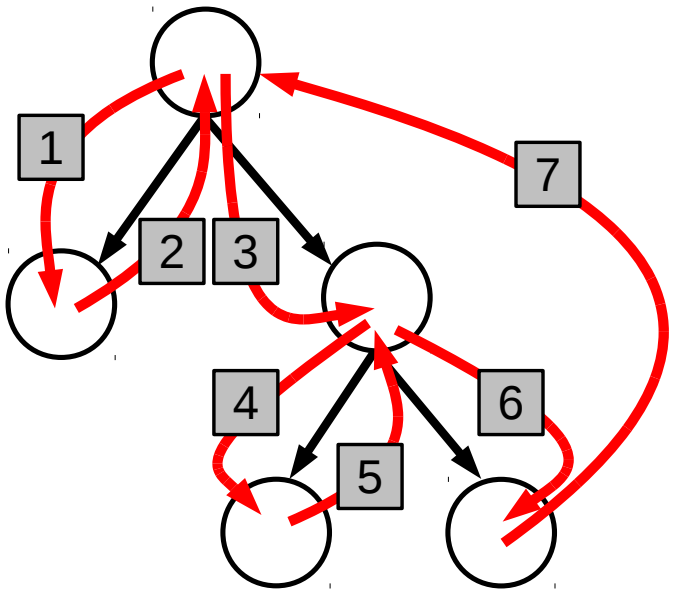
Comme nous l'avons vu avec les coroutines par exemple, cette discipline est parfois trop rigide : le contrôle devrait pouvoir suivre un parcours de graphe quelconque.

Exemples de flot du contrôle



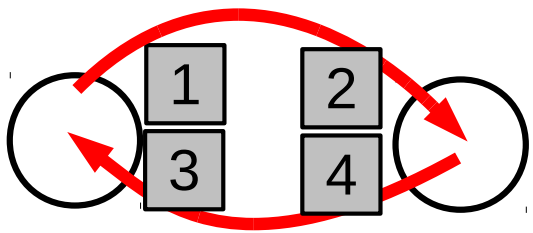
La discipline de pile utilisée dans les appels à des routines.

Exemples de flot du contrôle



Les continuations permettent par exemple de reprendre un calcul à une étape précédente sans nécessairement repasser par l'appelant.

Exemples de flot du contrôle



Pour implémenter les co-routines, il faut donner une liberté totale sur la façon dont le contrôle est transféré d'une co-routine à l'autre. Nous avons utilisé des continuations pour rendre possible ce transfert du contrôle.

Mais alors, cela signifie que l'on revient au `goto` ?

Du point de vue de l'expressivité, il est vrai que les opérateurs qui travaillent sur des continuations apportent une flexibilité comparable à celle de `goto`. Cependant, souvenons-nous de la critique du `goto` faite par Dijsktra : lorsque l'on raisonne sur un programme écrit à l'aide de `goto`, on ne peut pas dire facilement quels sont les points du programme qui mènent à une certaine étiquette sans observer l'ensemble du programme.

Dans le cas de la programmation fonctionnelle, les étiquettes sont moralement représentées par les continuations. La situation est différente puisque les **continuations sont des valeurs du langage de programmation**. Il est donc naturel que le programmeur sache donner un sens à ces valeurs, tout comme il donne un sens à toutes les valeurs que son programme produit.

Par exemple, dans le cas des co-routines, le programmeur maintient l'invariant que la continuation sauvegardée associée à chaque co-routine représente l'endroit où cette co-routine a été suspendue. Le type abstrait des co-routines garantit que l'unique façon d'exécuter le code de cette continuation est d'appeler la fonction `resume`.

En somme, grâce à ces opérateurs de contrôle : on est totalement libre de faire ce que l'on veut (même des gotos !) mais on peut aussi construire des abstractions qui restreignent le contrôle selon le besoin et offrent des garanties facilitant le raisonnement sur les programmes.

Retour sur Call/CC

```
let f x k = (if x = 0 then throw k 0 else x) + 1
(* p1 = fun x -> x + 1 *)
let g y = callcc (f y) × 2
(* p2 = fun x -> x * 2 *)
let h x = g (x + 1) + 1
(* p3 = fun x -> x + 1 *)
```

Moralement, l'opérateur `callcc` **recopie** la pile d'appels formée de p_1 à la base et p_2 au sommet dans la continuation `k`. L'opérateur `throw` **écrase** la pile d'appels courante par la pile contenue dans `k`.

La continuation `k` permet donc d'effectuer une sorte de "voyage dans le temps".

Voyage dans le temps avec `callcc`

```
let now () =
  let s = ref None in
  callcc (fun k → s := Some k);
  s

let past_travel_to date =
  match !date with
    | None → assert false
    | Some date → throw date ()

let eat () =
  Printf.printf "I am eating the cake! Miamee!\n"

let _ =
  let t = now () in
  Printf.printf "I am in front of the cake\n.";
  eat ();
  past_travel_to t
```

Une récursion sans utiliser `let rec` !

Voyage dans le temps avec `callcc`

```
let now () =
  let s = ref None in
  callcc (fun k → s := Some k);
  s

let past_travel_to date =
  match !date with
    | None → assert false
    | Some date → throw date ()

let cake = ref 'Cake
let eat () =
  assert (!cake = 'Cake);
  Printf.printf "I am eating the cake! Miamee!\n"
  cake := 'NoMoreCake

let _ =
  let t = now () in
  Printf.printf "I am in front of the cake\n.";
  eat ();
  past_travel_to t
```

Bien sûr, ce n'est pas un véritable voyage dans le temps puisque les effets de bords ne sont pas "défaits".

Des continuations sans limites

Revenons sur le programme suivant :

```
let f x k = (if x = 0 then throw k 0 else x) + 1
(* p1 = fun x -> x + 1 *)
let g y = callcc (f y) × 2
(* p2 = fun x -> x * 2 *)
let h x = g (x + 1) + 1
(* p3 = fun x -> x + 1 *)
```

L'opérateur `callcc` capture la continuation du calcul à partir du point où il est évalué **jusqu'à la fin du calcul**. Autrement dit, la pile d'appels est recopiée **entièrement**. Il existe d'autres opérateurs similaires à `callcc` qui permettent de ne capturer qu'une **partie délimitée de la pile des appels** dans la continuation et de limiter ainsi le style par continuation à un fragment du programme clairement défini.

Des continuations délimitées

Le même programme via une continuation délimitée :

let p = new_{prompt} ()
let f x = shift p (**fun** k $\rightarrow$ **if** x = 0 **then** 0 **else** k x) + 1
`(* p1 = fun x -> x + 1 *)`
let g y = prompt p (f y) $\times$ 2
`(* p2 = fun x -> x * 2 *)`
let h x = g (x + 1) + 1
`(* p3 = fun x -> x + 1 *)`

L'opérateur `prompt` pose une marque sur la pile (la valeur `p` est une marque créée par la fonction new_{prompt}). L'opérateur `shift` réifie la continuation du calcul en recopiant la pile **jusqu'à cette marque**. Cette continuation est la fonction `k` passée à l'argument fonctionnel de `shift` : dans le cas où on applique cette continuation le calcul va se poursuivre normalement mais si on ne l'applique pas le calcul se poursuit après avoir dépilé tout ce qu'il y a sur la pile d'appel jusqu'à la marque.

Continuation délimitée pour le non-déterminisme

Tiré de http://okmij.org/ftp/continuations/green-fork.html

> *"U2 has a concert that starts in 17 minutes and they must all cross a bridge to get there. All four men begin on the same side of the bridge. It is night. There is one flashlight. A maximum of two people can cross at one time. Any party who crosses, either 1 or 2 people, must have the flashlight with them. The flashlight must be walked back and forth, it cannot be thrown, etc.. Each band member walks at a different speed. A pair must walk together at the rate of the slower man's pace :"*

- Bono : 1 minute to cross
- Edge : 2 minutes to cross
- Adam : 5 minutes to cross
- Larry : 10 minutes to cross

Exercice

Écrivez un programme pour résoudre ce problème.

Solution

```
type u2 = Bono | Edge | Adam | Larry
type side = u2 list

let rec loop trace forward time_left = function
  | ([], _) when forward →
      print_trace (List.rev trace)
  | (_, []) when not forward →
      print_trace (List.rev trace)
  | (side_from, side_to) →
      let party = select_party side_from in
      let elapsed = elapsed_time party in
      let _ = if elapsed > time_left then fail () in
      let side_from' = without party side_from in
      let side_to' = side_to @ party in
      loop ((party,forward) :: trace) (not forward) (time_left - elapsed)
        (side_to',side_from')
```

Solution

```
module type SimpleNonDet = sig
      val choose : α list → α
      val run : (unit → unit) → unit
end

let select_party side =
    let p1 = choose side in
    let p2 = choose side in
    match compare p1 p2 with
    | 0 → [p1]
    | n when n < 0 → [p1 ;p2]
    | _ → fail ()

let fail () = choose []

run (fun () → loop [] true 17 ([Bono ;Edge ;Adam ;Larry],[])) ; ;
```

Comment implémenter `SimpleNonDet` ?

```
let rec choose = function
| [] → exit 666
| [x] → x
| (h :: t) →
    let pid = fork () in
    if pid = 0 then h else wait (); choose t

let run m = match fork () with
  | 0 → m (); printf "Solution found"; exit 0
  | _ →
  begin
      try while true do waitpid [] 0 done
      with | Unix_error (ECHILD,_,_) →
        Printf.printf "Done"
        | e →
       Printf.printf "Problem : %s\n" (Printexc.to_string e)
    end
```

Solution avec `Delimcc`

```
open Delimcc
let p = new_prompt ()
let choose xs = shift p (fun k → List.iter k xs)
let run m = push_prompt p m
```

Co-routines via continuations délimitées

Exercice

Implémentez les opérateurs `resume` et `yield` des co-routines à l'aide de continuations délimitées. Comment se comparent vos deux implémentations en termes d'expressivité et de performance ?

Les continuations délimitées

Les continuations délimitées sont implémentées dans de nombreux langages (Haskell, O'Caml, Scala, Scheme, ...) et fournissent des primitives efficaces et très expressives pour définir ses propres opérateurs de contrôle.

Pour se souvenir de la différence entre les continuations délimitées et non délimitées, Olivier Danvy utilise la formule : avec les continuations non délimitées, on donne le contrôle et on ne le reverra jamais (*Leave now and never come back*) ; avec les continuations délimitées, la continuation représente seulement une partie du reste du calcul, donc l'appliquer est similaire à un appel de fonction (*I'll be back*).

Appel par valeur, par nom

Il existe deux façons d'évaluer une application `e1 e2` dans un langage fonctionnel :

- Appel par valeur : On évalue l'argument `e2` en une valeur `v`, on évalue `e1` en une fermeture `fun x → e [E]` et on évalue `e` dans l'environnement `E` augmenté par l'association `E + [x → v]`
- Appel par nom : On évalue `e1` en une fermeture `fun x → e [E]` et on évalue `e` dans l'environnement `E` augmenté par l'association `E + [x → e2']`. Où `e2'` représente le calcul de `e2` **retardé**.

Ces stratégies d'**évaluation** diffèrent dans leur traitement de la non-terminaison : le programme `f (loop ())` ne termine pas dans un langage utilisant la stratégie d'appel par valeur tandis qu'il termine dans un langage utilisant la stratégie d'appel par nom.

Implémentation de l'évaluation paresseuse

L'appel par nom peut dupliquer des calculs car une variable peut avoir plusieurs occurrences dans une expression. L'expression `(fun x → (x, x)) (6 × 7)` calculera deux fois `6 × 7` en appel par nom et une seule fois en appel par valeur.

Pour cette raison, l'appel par nom est en général raffiné en une évaluation paresseuse. Le principe de l'évaluation paresseuse est de suspendre un calcul à l'aide d'un type de donnée spécial, le **glaçon** (*thunk*) qui contient une fermeture représentant le calcul retardée. Lorsque l'on force l'évaluation d'un glaçon, ce calcul est effectué et la valeur `v` résultante est **stockée à la place de la fermeture dans le glaçon**. De cette façon, à la prochaine évaluation forcée du glaçon, on peut renvoyer cette valeur plutôt que de réévaluer la fermeture.

Exercice

Implémenter un type α `thunk` en O'Caml.

C'est de façon similaire qu'est implémenté le type `lazy` de O'Caml.

Exemple d'évaluation paresseuse

Exercice

Reprenez l'exemple du générateur de nombre premiers et réécrivez-le à l'aide de l'évaluation paresseuse.

Les forces de l'évaluation paresseuse

L'évaluation paresseuse permet de manipuler des valeurs de taille infinie, comme des ensembles infinis résultant de calcul sur des structures non bornées.

Elle permet aussi de minimiser "automatiquement" les calculs inutiles. On peut écrire des opérateurs très généraux et hériter d'une certaine forme de spécialisation automatique induite par seule une partie du calcul sera effectuée.

Tout ceci permet d'augmenter la **déclarativité** et la **généralité** de la programmation et donc de faciliter le raisonnement sur les programmes.

Les faiblesses de l'évaluation paresseuse

Manipuler des valeurs potentiellement infinies peut être la source de certains problèmes de divergence : par exemple, si une valeur a le type α `list`, il peut s'agir d'une liste potentiellement infinie et donc, il faut mieux y réfléchir à deux fois avant de se lancer dans le calcul de sa longueur ![4]

Les calculs en suspension dans des glaçons peuvent retenus très longtemps en mémoire sans que le programmeur en aie conscience. Ceci peut provoquer des **fuites de mémoire**.

Il est aussi assez difficile de raisonner sur la **complexité** des programmes. En effet, nous avons vu que la complexité en espace pouvait être minimisée en retardant l'évaluation de façon à ne pas créer de résultats intermédiaires. De même, on peut aussi réduire la complexité d'un programme lorsque l'on utilise l'évaluation paresseuse plutôt que la stratégie d'appel par nom mais la justification est plus difficile à fournir que pour des programmes en appel par valeur. (Voir le transparent suivant).

4. Notez que ce problème est aussi présent en O'Caml, par exemple sur des listes cycliques créées par des définitions de la forme `let rec x = 1 :: x`. Le problème est peut être plus rare car ce n'est pas un idiome très courant en O'Caml.

Fibonacci

```
let fib n =
  let rec aux n prev pprev =
    if n = 0 then pprev
    else if n = 1 then prev
    else aux (n - 1) (prev + pprev) prev
  in
  aux n 1 1
```

En appel par valeur, il est facile de montrer que cette fonction effectue un nombre d'additions proportionnel à `n`.

En appel par nom, le nombre d'additions est proportionnel à `n` $\times$ `n`

En présence d'évaluation paresseuse, on peut remarquer que le nombre d'additions suspendues est proportionnel à `n`, ce qui permet de se ramener à une complexité linéaire.

Une autre fonction récursive

```
let g n =
  let rec aux n prev pprev =
    if n = 0 then pprev
    else if n = 1 then prev
    else if n mod 2 = 0 then aux (n - 1) (pprev + pprev) pprev
    else aux (n - 1) (prev + pprev) pprev
  in
  aux n 1 1
```

En appel par valeur, il est facile de montrer que cette fonction effectue un nombre d'additions proportionnel à `n`.

En présence d'évaluation paresseuse, on peut remarquer que le nombre d'additions suspendues est proportionnel à `n`, mais que seules `n/2` seront réellement évaluées puisqu'une fois sur deux, l'addition suspendue dans `prev` est ignorée.

Exercice

Reprogrammez l'exemple de l'algorithme `minmax` en SCALA ou en O'CAMLtel qu'il est présenté dans cet article.

La programmation impérative en Haskell

In fine, un programme doit **communiquer** avec son environnement d'exécution, ne serait-ce que pour connaître les entrées choisies par l'utilisateur et émettre les sorties qu'il a calculées.

Les langages de programmation fonctionnels tels que O'Caml, sont aussi impératifs : ils fournissent des primitives impurs, généralement sous la forme de constantes fonctionnelles, que l'on peut appeler pour réaliser des effets de bord.

Ainsi, le programme O'Caml suivant :

```
let x = length (List.tl ([ print_int "42" ; print_int "42" ]))
```

affiche deux fois 42 sur le terminal.

Fournir de telles primitives n'aurait pas beaucoup de sens dans un langage fonctionnel utilisant une stratégie d'appel par nom ou d'évaluation paresseuse : Comment le programmeur pourrait-il raisonner localement sur les effets de bord qu'un fragment de code pourrait produire ? Il serait nécessaire d'effectuer des raisonnements globaux et complexes servant à expliciter certaines propriétés sur l'ordre des calculs (en suivant les dépendances), une tâche irréaliste.

Les solutions pré-monadiques

Une solution consiste à voir un programme fonctionnel pur comme une fonction de type `string → string`. Il suffit alors d'écrire un programme externe qui implémente la communication avec l'environnement à travers les chaînes de caractères d'entrée et de sortie.

En généralisant cette méthode, on peut imaginer une spécification du programme fonctionnel pur de la forme `response list → request list`. Dans ce modèle, le programme externe pourrait itérer les appels à cette fonction pour produire plusieurs aller-retour de communication entre le programme fonctionnel et son environnement. En définissant `response` et `request` à l'aide de types algébriques, on peut donner une structure plaisant aux messages échangés entre le programme et son environnement. Par exemple :

```
type request =
  | LoadFile of string
  | SaveFile of string × content

type response =
  | FileLoaded of content
  | FileNotLoaded of error
```

C'est ainsi que fonctionnait les premières versions de HASKELL

L'ère monadique

Le point de vue monadique sur la programmation impérative consiste à **se donner un langage pour décrire des commandes** et de **l'embarquer à l'intérieur du langage fonctionnel**.

Cette idée soulève deux questions :

1. Quelle spécification donner aux opérations travaillant sur les programmes impératifs embarqués à l'intérieur du langage fonctionnel ?
2. Comment représenter concrètement un programme à l'intérieur d'un autre ?

Un type abstrait pour les programmes embarqués

Le point essentiel à comprendre est que les programmes embarqués à l'intérieur du langage fonctionnel doivent pouvoir communiquer avec les programmes du langage hôte. Cette communication a lieu dans deux directions.

Du programme embarqué vers le programme hôte : Une valeur d'un type α issue du calcul effectué dans le programme embarqué doit pouvoir être communiquée au programme hôte. Le type des programmes embarqués doit donc être paramétré par ce type. On le note α `m`.

Du programme hôte vers le programme embarqué : Une valeur d'un type α provenant du programme hôte doit pouvoir être plongée dans le langage embarqué. On se donne donc une opération `return` : $\alpha \to \alpha$ `m`, qui construit le programme du langage embarqué dont l'évaluation mènera la valeur de type α fournie en argument.

Une implémentation *via* arbres de syntaxe abstraite

Une première idée consiste à se donner un type pour représenter la syntaxe abstraite du langage embarqué. Par exemple :

```
type _ t =
| LoadFile : string t → content t
| SaveFile : string t → content t → unit t
| Seq : forall α β. α t → β t → β t
| Value : forall α. α → α t
```

et une fonction d'interprétation :

```
val eval : α t → α
```

Inconvénients des arbres de syntaxe

Malheureusement, cette approche à plusieurs défauts :

- Le type `t` est un GADT, une forme généralisée des types algébriques qui n'est pas disponible dans de nombreux langages fonctionnels. [5]
- La fonction `eval` introduit une couche d'interprétation qui peut être dommageable pour les performances.

Avec quelques techniques, on arrive cependant à passer outre ces difficultés. Nous reviendrons sur cette méthode dans le cours sur les langages spécifiques.

5. À l'exception notable de Haskell, Scala et, bientôt peut-être O'Caml

Représentation via des fonctions

Une idée des monades est de représenter les programmes embarqués à l'aide de fonctions du langage hôte. Ainsi, nous avons déjà vu que la monade d'état permettait de représenter un calcul qui véhicule un état unique d'une commande à une autre :

```
type α m = world → (α × world)

let return x = fun s → (x, s)

let bind l f =
  fun s →
    let x, s = l s in
    f x s
```

Le compilateur pourra, avec un schéma de compilation spécialisé, **compiler ces fonctions en des fonctions réalisant réellement des effets de bord**.

Tous les ingrédients recherchés pour représenter un programme embarqué sont presque réunis : le calcul est retardé, on peut communiquer une valeur du langage hôte vers le langage embarqué.

`unsafePerformIO`

Reste à savoir comment communiquer une valeur depuis le langage embarqué vers le langage hôte.

Pour cela, il est naturel de définir une fonction de type $\alpha\ \mathtt{m} \rightarrow \alpha$.

Cependant, dans un langage paresseux, on ne peut pas prédire le moment où le résultat de cette fonction sera réalisée et donc à quel moment les effets de bord auront réellement lieu !

On a donc coutume de mettre en garde contre l'utilisation de cette fonction, nommée `unsafePerformIO`.

L'utilisation standard de la monade de effets de bord est d'y définir la fonction `main`, le point d'entrée du programme. Dans ce cas, on s'assure que les effets de bord auront bien lieu suite au calcul de la sortie du programme, déclenché par le lancement de l'exécutable.

Reinterprétation d'un programme dans une monade

Il existe une transformation systématique qui permet de réinterpréter un programme fonctionnel en appel par valeur dans une monade. C'est la transformation *Call-by-value* de Plotkin :

- `[ x ]` est `unit x`
- `[ fun (x : T) → e ]` est `unit (fun (x : [ T ]) → [ e ])`
- `[ let x = a in b ]` est `bind [ a ] (fun (x : [ T ]) → [ b ])`
- `[ t u ]` est `bind [ t ] (bind [ u ])`
- `[ if b then a else b ]` est
  `bind [ b ] (fun (x : bool) → [ a ] else [ b ])`
- `[` α `]` est α
- `T1 → T2` est `[ T1 ] → [ T2 ] m`

Conclusion : La programmation fonctionnelle

L'approche fonctionnelle de la programmation consiste à construire des **abstractions de l'état et du contrôle** pour se doter d'outils généraux les plus adaptés possibles , tant du point de vue calculatoire que du point de vue linguistique et méta-linguiste, à une **certaine classe de problèmes bien identifiés**.

Elle reprend dans l'esprit les idées de la programmation structurée : **contraindre l'expressivité des mécanismes calculatoires** utilisés par les programmes est une bonne idée **pour obtenir des garanties et ainsi raisonner plus facilement**. Elle diffère de la programmation structurée sur un point essentiel : c'est au programmeur et non au concepteur du langage de programmation de placer le curseur entre ce qui est autorisé et ce qui est interdit (tout en restant dans la limite de ce qui est "sûr" du point de vue du calcul grâce au typage).