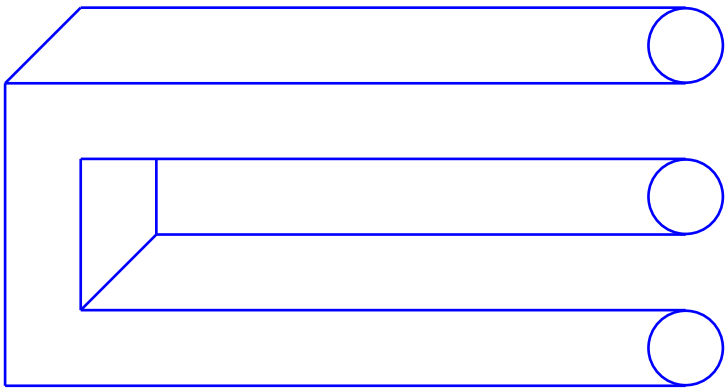
Polytech Grenoble

PF : Programmation Fonctionnelle

Livret de TP



Erwan Jahier, Jean-François Monin et Benjamin Wack

Avant-propos

Ce document va évoluer au cours du semestre. Il contiendra, au fur et à mesure, les énoncés des séances de TP proposés par l'équipe pédagogique de PF. Nous indiquons par des étoiles la difficulté des exercices proposés : plus il y a d'étoiles plus l'exercice est jugé difficile.

Plan : Nous commencerons par des exercices basiques permettant de s'approprier la syntaxe d'OCaml, puis nous approfondirons l'utilisation des listes et plus généralement des données récursives. Ensuite nous utiliserons l'analyse syntaxique pour lire un fichier, et nous verrons comment écrire un `Makefile` et utiliser le compilateur d'OCaml. Enfin nous utiliserons le système de modules et foncteurs d'OCaml. La fin du semestre sera consacrée à la réalisation d'un projet utilisant toutes ces compétences.

Préparation : Avant chaque séance de TP, des jeux de tests de vos différentes fonctions devront être réalisés et envoyés par email à votre enseignant de TP.

Pour certains TP, des fichiers sont à télécharger depuis l'URL :
`https://ltpf.gricad-pages.univ-grenoble-alpes.fr/PF/`

La documentation complète de la bibliothèque standard est accessible par l'URL :
`https://ocaml.org/api/Stdlib.html`

Table des matières

1 Introduction à OCaml

Objectifs : Il s'agit dans ce premier TP de se familiariser avec l'environnement de développement Emacs couplé au mode Tuareg et de s'interroger sur les différents paradigmes du langage OCaml à travers des exercices ń jouets ż.

Toutes les constructions syntaxiques qui vous seront nécessaires ne sont pas données ou décrites en détail [1]. Il est **fortement conseillé** de tester vos fonctions avec des entrées judicieusement choisies.

1.1 Environnement de travail

Objective Caml est la principale implémentation du langage OCaml. Pour travailler sur votre machine personnelle, veillez à ce que OCaml, Emacs, le mode Tuareg, Make et la bibliothèque LablGL soient installés sur votre système. Normalement, si vous travaillez avec l'image docker fournie, c'est déjà le cas !

Pour information, sur une distribution Debian (et donc également Ubuntu), il est aussi possible d'installer le méta-paquet `ocaml-core` (en suivant les recommandations et suggestions) et les paquets `make` (installé par défaut) et `liblablgl-ocaml-dev`. Cependant les versions proposées ne seront pas forcément compatibles avec les fichiers proposés pour cette UE.

1.2 Premiers pas avec OCaml

OCaml peut être utilisé en mode compilé ou en mode interactif.

- Le mode compilé permet de créer un fichier binaire exécutable. Nous en parlerons en deuxième partie de semestre.
- Le mode interactif permet d'évaluer des expressions OCaml de manière incrémentale. C'est celui qui nous intéresse dans le cadre de ce premier TP. Nous l'utiliserons par l'intermédiaire de l'éditeur Emacs, grâce au mode Tuareg.

Tests pour vérifier que Tuareg *fonctionne :*

Utiliser la coloration syntaxique :

1. Ouvrir Emacs.
2. Ouvrir/créer un nouveau fichier `test.ml` (l'extension `.ml` indique à Emacs de lancer Tuareg).
3. Taper dans ce nouveau tampon vierge l'expression `let a = 1+1`. Chaque terme de cette expression apparaît dans une couleur différente.

Évaluer une expression :

1. Taper `C-c C-e` qui permet d'évaluer une expression dans la boucle interactive d'OCaml.
2. À la première utilisation de cette commande, Emacs demande dans la ligne de commande si on veut utiliser la boucle interactive standard (`ocaml`) ; confirmer en tapant entrée.
3. Une nouvelle fenêtre doit s'ouvrir et afficher le résultat de l'évaluation. Les résultats suivants seront toujours affichés dans ce nouveau tampon nommé `*caml-toplevel*`.

Note : pour stopper la boucle interactive, taper `C-c C-k`.

1. Pour plus de précisions référez-vous à la documentation en ligne `http://caml.inria.fr` dans ń Ressources ż suivre ń Manuel d'OCaml ż puis ń The core language ż

Indenter une expression sur plusieurs lignes *(attention :* `C-M-\` *ne fonctionne pas sur toutes les machines, donc ne pas insister dans ce cas)* :

1. Insérer un retour à la ligne entre `let a =` et `1+1`.
2. Taper `C-x h` pour sélectionner toutes les lignes.
3. Taper `C-M-\` pour indenter correctement la sélection.

Remarques sur OCaml **:**

— OCaml est sensible à la casse, c'est à dire qu'il prend en compte les majuscules et les minuscules. En particulier, seuls les noms de constructeurs de types commencent par une majuscule alors que les autres noms commencent par une minuscule.
— Les commentaires sont placés entre `(*...*)`.

Quelques raccourcis Emacs **utiles** À utiliser sans modération. Il existe une abondante documentation sur internet [2].

— couper la région sélectionnée	`C-w`
— coller la région sélectionnée	`C-y`
— indenter une ligne	`Tab`
— indenter un paragraphe	`Esc q`
— ouvrir un fichier	`C-x C-f`
— sauvegarder le buffer courant	`C-x C-s`
— quitter Emacs	`C-x C-c`
— changer de buffer	`C-x b + tab`
— aller en fin de ligne	`C-e`
— aller en début de ligne	`C-a`
— supprimer le buffer courant (n'affecte pas le fichier)	`C-x k`
— évaluer l'expression OCaml sous le curseur	`C-c C-e`
— évaluer l'ensemble des expressions OCaml de la région sélectionnée	`C-c C-r`
— évaluer l'ensemble des expressions OCaml du buffer	`C-c C-b`
— fermer l'interpréteur OCaml	`C-C C-k`
— commenter la région sélectionnée	`M-x comment-region`
— décommenter la région sélectionnée	`C-u M-x comment-region`

*1.3 Découverte d'*OCaml

En OCaml, écrire un programme consiste à déclarer un ensemble d'expressions qui peuvent être des valeurs ou des fonctions. Comme le calcul correspond à l'évaluation des fonctions déclarées, on parle de programmation fonctionnelle.

Exercice 1.1 *Quel sera le type du résultat des expressions suivantes ?*

Essayer de prévoir ce résultat, puis les évaluer dans OCaml afin de vérifier vos prévisions.

— `let r = let x = 7 in 6 * x`
— `let a = (r - 6) / 6 - 6`
— `let o = r * r - x * x - 51`
— `let u = let x = 9 in if (x < 9) then 9 / (x - x) else (x + x) / 9`

Exercice 1.2 *Évaluer chacune des expressions suivantes, observer les messages d'erreur et expliquer chacun d'entre eux.*

2. par exemple `http://www.ocamlpro.com/files/tuareg-mode.pdf`

Proposer une expression correcte (attention, l'erreur de programmation n'est pas toujours là où OCaml indique l'erreur de type !).

- `let pi_sur_4 = 3.14 / 4.`
- `let dans_l_ordre = 1 < 2 < 3`
- `let positif = let a = 42 in if a >= 0 then true`
- `let double_absolu = let x = -2.7 in (if (x < 0) then x else -x) *. 2`

1.4 Définitions de types

Exercice 1.3 *Échauffement.*

1. *Définir un type* `semaine` *dont les éléments permettent de représenter chaque jour de la semaine.*
2. *Définir un type* `point2D` *qui permet de représenter les points du plan.*
3. *Définir un identifiant de type* `point2D` *qui représente l'origine du repère.*
4. *Définir un type* `segment` *(à l'aide du précédent).*
5. *Définir un type somme* `figure` *pour représenter les figures géométriques carré, rectangle et cercle.*
6. *Dessiner (sur feuille) une figure pour chacun de ces cas, puis définir des identifiants correspondant à vos trois figures.*

Exercice 1.4 *Jeu de Uno.*

Le Uno est un jeu de cartes inspiré du 8 américain, mais qui utilise des cartes spécifiques.

Consulter la page `https://fr.wikipedia.org/wiki/Uno#Contenu_du_jeu` *et proposer une hiérarchie de types permettant de représenter les 108 cartes du jeu standard.*

On ne tiendra pas compte du fait que certaines cartes sont présentes en plusieurs exemplaires dans le paquet, par contre on s'arrangera pour qu'il soit facile de retrouver la couleur et la « valeur » de chaque carte.

1.5 Premières fonctions

Des fonctions Vous pouvez déclarer des fonctions de la manière suivante : `let f = fun a -> a - 1`. La fonction définie (`ń f ż`) est une fonction de `a` (`ń fun a ż`) qui renvoie (`ń -> ż`) la valeur de `a` moins un.

Exercice 1.5 *Petites fonctions*

Écrire des fonctions qui déterminent :

- *le cube d'un flottant.*
- *si un entier est positif.*
- *si un entier est pair.*
- *le signe d'un entier (c'est-à-dire qu'elle renverra -1, 0 ou +1 selon les cas).*

1.5.1 Types de base et fonctions

Fonctions à plusieurs arguments Pour définir une fonction somme, il nous faut deux arguments. On pourrait pour cela écrire quelque chose de similaire aux autres langages de programmation :

```
let somme = fun (x,y) -> x + y
```

mais ce faisant, on utilise en fait un *type couple* sans l'avoir vraiment voulu.

En Ocaml, on préférera écrire comme suit : `let somme = fun x -> fun y -> x + y`

Nous verrons plus tard que cela revient à considérer les fonctions à plusieurs arguments comme des fonctions à un seul argument renvoyant une fonction. Cette dernière forme est dite forme *curryfiée*.

Exercice 1.6 *Écrire sous forme curryfiée et tester une fonction* `f1` *qui calcule le produit de trois entiers et une fonction* `f2` *qui calcule leur somme.*

Exercice 1.7 *Prédicats.*
On appelle prédicat une fonction qui renvoie un booléen.

Écrire des fonctions qui déterminent :
- *si les trois paramètres entiers forment un triplet pythagoricien.*
- *si deux entiers sont de même signe.*

Du typage En OCaml l'évaluateur, comme le compilateur, calcule le type de chaque expression et vérifie que celui-ci existe. Puisqu'il s'agit d'un langage fonctionnel, un type fréquent est celui des fonctions, que l'on écrit à l'aide de la flèche $t_1 \to t_2$ et qui signifie qu'une fonction ayant ce type calcule un élément de type t_2 à partir d'un élément de type t_1. Les types t_i peuvent être des types de base ou des types de fonctions.

Exercice 1.8 *Manipulons les types (facultatif).*
- *Observer et expliquer les types des fonctions définies dans les deux exercices précédents.*
- *Donner une fonction ayant le type* `int` $\to$ `int` $\to$ `int`*, et une autre ayant le type* `int` $\to$ `int` $\to$ `bool`*.*

Exercice 1.9 *Minimum de deux entiers.*
Écrire une fonction `min2entiers` *qui calcule le minimum de deux entiers passés en paramètres.*

Exercice 1.10 *Min de trois entiers.*
Écrire une fonction `min3entiers` *qui calcule le minimum de trois entiers passés en paramètres.*

1.5.2 Types composés

Exercice 1.11 *Point 2D.*

À l'aide des types `point2D` *et* `segment`*, définir des fonctions :*
- *qui renvoie le milieu d'un segment.*
- *qui détermine si un point appartient ou non à un segment.*

Lorsqu'une fonction reçoit un argument d'un type somme, elle doit pouvoir distinguer les différents cas (constructeurs) de ce type, et proposer une réponse adaptée. Le *filtrage* (construction `match`) permet cette opération.

Par exemple, supposons défini `type nombre = Ent of int | Reel of float`

On peut alors écrire la fonction suivante :

```
let doubler = fun n ->
  match n with
  | Ent (i) -> Ent (2 * i)
  | Reel (x) -> Reel (2 *. x)
```

Exercice 1.12 *Jour de la semaine.*

À l'aide du type `semaine` *défini précédemment, écrire une fonction qui teste si un jour de la semaine est un jour du week-end.*

Exercice 1.13 *Aires.*
À l'aide du type `figure` *défini plus haut, écrire des fonctions qui calculent l'aire de la figure reçu en paramètre.*

Aviez-vous choisi des informations pertinentes dans le type `figure` *pour réaliser cette fonction ? Si non, n'hésitez pas à modifier votre type.*

1.6 Jeu de Uno (*)

Exercice 1.14 *Définir diverses fonctions se rapportant au jeu de Uno :*
- *calculer la valeur d'une carte (voir* `https://fr.wikipedia.org/wiki/Uno#Fin_d'une_manche`*)*
- *le joueur suivant (dans l'ordre normal du tour) pourra-t-il jouer ?*
- *une carte peut-elle être jouée sur une autre ?*
- *Définir un type pour un jeu de 54 cartes standard, et des fonctions de traduction entre les deux types de cartes lorsque c'est possible.*

1.7 Un peu de récursivité

Définissez un type récursif de listes d'entiers comme vu en cours :

```
type listent = Nil | Cons of int * listent
```

Une telle liste peut être parcourue à l'aide d'une fonction récursive et du filtrage :

```
let rec longueur = fun l -> match l with
  | Nil -> 0
  | Cons(x, s) -> 1 + longueur s
```

Exercice 1.15 *Écrire des fonctions qui calculent les valeurs suivantes, qui prennent toutes en argument une liste d'entiers (et si nécessaire des arguments supplémentaires) :*
- *la somme de tous les entiers de la liste*
- *un booléen exprimant si tous les entiers de la liste sont positifs ou non*
- *(*) l'ajout d'un élément supplémentaire en fin de liste*
- *(*) la concaténation de deux listes*

Réécrire ces fonctions avec la syntaxe OCaml des listes, utilisant `::` *et* `[]`*.*

Exercice 1.16 *Encore du Uno !*

Définissez une main de cartes *à l'aide d'une liste.*

Écrivez des fonctions pour :
- *calculer la valeur totale d'une main*
- *déterminer lesquelles des cartes d'une main peuvent être jouées sur une carte donnée*
- *déterminer si les cartes d'une main peuvent être toutes jouées à la suite, dans l'ordre de la main*
- *(***) déterminer s'il existe un ordre permettant de jouer toutes les cartes d'une main à la suite*

2 Arbres binaires

Il est demandé, pour ce TP et ceux qui vendront ensuite, de faire suivre chaque définition de fonction par un ou plusieurs tests significatifs.

À cet effet, employez des appels sur le modèle `assert` *`expr = résultat_attendu`* comme dans l'exemple suivant.

```
let affine a b = fun x y -> a*x + b*y
let aff21 = affine 2 1
let _ = assert (aff21 3 4 = 10)
```

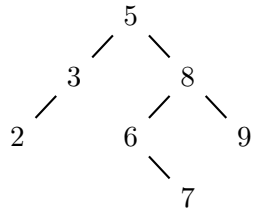
2.1 Fonctions simples sur un arbre

- Définir un type `abin` pour un arbre binaire d'entiers (les nœuds doivent être étiquetés, pas les feuilles).
- Écrire une fonction qui calcule le nombre de nœuds d'un arbre binaire.
- Écrire une fonction qui calcule la hauteur d'un arbre binaire.
- Écrire une fonction qui calcule le produit de tous les éléments d'un arbre binaire.
- Écrire une fonction qui calcule la somme de tous les éléments d'un arbre binaire.
- Écrire une fonction qui détermine si un élément appartient à un arbre binaire.
- Écrire une fonction qui calcule le maximum d'un arbre binaire.
- Écrire une fonction qui calcule le nombre de nœuds "vraiment binaires" d'un arbre binaire (autrement dit le nombre de nœuds ayant 2 fils non vides).

2.2 Arbres Binaires de Recherche

Un Arbre Binaire de Recherche (en abrégé ABR) est un arbre étiqueté tel que pour tout nœud N :

- les étiquettes de tous les nœuds du sous-arbre gauche de N sont strictement inférieures à l'étiquette de N ;
- et les étiquettes de tous les nœuds du sous-arbre droit de N sont strictement supérieures à l'étiquette de N.



Exemple d'ABR (où les constructeurs ont été gommés)

Pour définir un tel objet en OCaml, on peut se contenter du type d'arbre binaire ordinaire `abin` ci-dessus. En effet, ce n'est pas le type mais les *fonctions de manipulation* qui assureront que l'objet construit reste bien un ABR.

Exercice 2.1 *Construire (à la main) un (petit) exemple d'ABR en OCaml en utilisant ce type.*

2.2.1 Fonctions de manipulation

Exercice 2.2 *Écrire la fonction* `mem` *qui recherche si un entier donné appartient à un ABR donné.*

Il s'agit ici de profiter des caractéristiques de l'ABR pour ne pas effectuer une recherche exhaustive.

Exercice 2.3 *Écrire la fonction* `insert` *qui insère un entier donné dans un ABR donné, à une place appropriée pour conserver la propriété d'ABR.*

Là encore, les caractéristiques de cette structure doivent vous aider à trouver cette place facilement.

Exercice 2.4 *Écrire une fonction* `verif` *qui vérifie si un arbre donné est bien un arbre binaire de recherche.*

Différentes stratégies sont possibles.

2.2.2 Utilisation pour le tri

Exercice 2.5 *Écrire une fonction* `triABR` *qui prend en argument une liste, et renvoie la liste constituée des mêmes éléments en ordre croissant, à l'aide des étapes suivantes :*

- *insérer chacun des éléments de la liste dans un ABR ;*
- *parcourir l'ABR de façon à récupérer les éléments en ordre croissant.*

Vous êtes encouragés à décomposer cet exercice en plusieurs fonctions.

2.2.3 Arbres binaires de recherche et exceptions

Exercice 2.6 *Écrire une version efficace de la fonction* `verif`*, qui vérifie si un arbre donné est bien un arbre binaire de recherche, en écrivant préalablement une fonction qui*

- *renvoie le couple* $(minimum, maximum)$ *de l'arbre binaire en entrée si celui-ci est bien un ABR ;*
- *lève une exception si ce n'est pas un ABR.*

Exercice 2.7 *Écrire une fonction* `split` *qui prend en argument un entier* `x` *et un arbre binaire de recherche* `a`*, et renvoie un couple* `(a1, a2)` *d'arbres binaires de recherche :*

- `a1` *contient tous les éléments de* `a` *inférieurs à* `x` *;*
- `a2` *contient tous les éléments de* `a` *supérieurs à* `x` *;*
- *si* `x` *est absent de* `a` *la fonction lève une exception.*

Exercice 2.8 *Écrire une fonction* ***de complexité logarithmique*** `compare` *qui prend deux arbres binaires de recherche en arguments et vérifie s'ils contiennent les mêmes éléments (mais pas forcément avec la même structure). Pour cela, on devra utiliser la fonction* `split`*.*

Exercice 2.9 (facultatif) *(**) Écrire une fonction* `suppr` *qui, à partir d'un ABR a et d'un élément x présent dans a, renvoie un ABR comportant les mêmes éléments que a sauf x. Si x comporte plusieurs occurrences dans a, seule une de ces occurrences est supprimée.*
On pourra procéder par étapes :

- *écrire une fonction qui étant donné un ABR, renvoie un couple formé : d'une part de son élément maximal, d'autre part de l'ABR dans lequel cet élément est supprimé ;*
- *écrire une fonction qui « supprime la racine d'un ABR » a, ou plus exactement construit un ABR à partir des deux sous-arbres à la racine de a ;*
- *et enfin écrire la fonction* `suppr`*.*

3 Manipulation de listes

Objectifs : Mettre en application des algorithmes de manipulation de listes. Comparer l'efficacité de différentes solutions algorithmiques.

Échauffement sur les listes : terminer l'exercice 15 du TP1 si ce n'est pas encore fait.

3.1 Tri par sélection

Le principe du tri par sélection est de rendre la liste dont la tête est le minimum de la liste en entrée et dont la queue est la liste des éléments restants, triée de la même manière.

Remarque. Une formulation en style impératif serait la suivante : trouver le minimum de la liste, le mettre au début, puis recommencer avec les éléments restants. Mais attention, en programmation fonctionnelle on *ne modifie pas* une liste existante, on construit une nouvelle liste.

Exercice 3.1

- *Écrire une fonction* `trouve_min_i` *de type* `int list -> int * int list` *qui calcule le plus petit entier d'une liste non vide ainsi que la liste sans cet entier.*
- ***Généralisation.***
  Écrire une fonction `trouve_min` *de type* `('a -> 'a -> bool) -> 'a list -> 'a * 'a list` *qui calcule la plus petite valeur d'une liste non vide ainsi que la liste sans cette valeur ; le premier paramètre de* `trouve_min` *est une fonction de comparaison sur* `'a` *qui rend* `true` *si son premier argument est plus petit que le second.*
  Redéfinir `trouve_min_i` *à partir de* `trouve_min`*.*

Exercice 3.2

- *Écrire une fonction* `tri_selection` *de type* `('a -> 'a -> bool) -> 'a list -> 'a list` *qui trie la liste donnée en second argument suivant le critère de comparaison donné en premier argument.*
  En déduire une fonction `tri_selection_i` *de type* `int list -> int list` *qui trie une liste d'entiers en ordre croissant.*

3.2 Mesure de performances

Exercice 3.3 (Génération de tests) *Écrire une fonction* `liste_alea`*, qui prend en argument un entier n et génère une liste de taille n d'entiers tirés au hasard (peu importe la façon exacte de choisir ces entiers). Utiliser la librairie* `https://ocaml.org/api/Random.html`*.*
Attention : l'initialisation `Random.init un_entier` *s'effectue une seule fois, en dehors des fonctions qui fabriquent une liste aléatoire.*

Pour obtenir des mesures précises, utiliser la fonction `Sys.time()` en suivant le schéma suivant :

```
let deb = Sys.time() in
  let _ = <...code à chronométrer...>
   in Sys.time() -. deb;;
```

Exercice 3.4 *Générez des données aléatoirement et utilisez-les pour comparer les performances du tri par sélection, et du tri par ABR que vous avez écrit la semaine dernière.*

Il faudra créer d'autres fonctions de génération pour commenter ce qui se passe sur des listes déjà triées, ou triées en ordre décroissant.

3.3 Renversement de listes

3.3.1 Deux programmes

Exercice 3.5 (Renversement naïf) *Écrire une fonction* `renv` *qui prend en argument une liste, et renvoie la liste formée des mêmes éléments en ordre inverse.*

Cette première version devra opérer sur le principe suivant :
- *renverser la queue de la liste ;*
- *puis placer la tête de la liste en dernière position de la queue renversée (à l'aide d'une fonction auxiliaire).*

Exercice 3.6 (Renversement efficace) *Écrire une fonction* `renv_app` *qui prend en argument* deux listes `l1` *et* `l2`*, et renvoie une liste formée :*
- *des éléments de* `l1` *en ordre inverse ;*
- *puis des éléments de* `l2`

le tout sans utiliser la fonction de l'exercice précédent ni la concaténation.

Comment utilisez-vous `renv_app` *pour renverser une seule liste ?*

3.3.2 Mesure de performances

Exercice 3.7 (Performance de `renv`) *Comparer les temps d'exécution de vos deux fonctions de renversement, pour des listes d'entiers de tailles allant jusqu'à 10 000 éléments (ou plus si nécessaire pour constater une différence, selon votre machine).*

Faire plusieurs mesures pour évaluer le coût d'exécution de vos fonctions en fonction de la taille de la liste, déterminer si ce coût dépend des valeurs présentes dans la liste, etc.

Comment expliquer cette différence ?

Exercice 3.8 (Performance de `@`) *Comparer les temps d'exécution de* `l1@(l2@l3)` *et* `(l1@l2)@l3`*.*

Expliquez ce que vous observez.

3.4 Chameau et dromadaire

Exercice 3.9 *Écrire une fonction* `dromadaire` *qui prend une liste* `l` *et une fonction de comparaison* `comp` *en paramètre et retroune le plus grand élément de* `l` *selon l'ordre induit par* `comp`*.*

Exercice 3.10 *Écrire une fonction* `chameau` *similaire à* `dromadaire`*, mais qui rend les 2 plus grands éléments.*

3.5 Le magasin (facultatif)

Le rayon électronique d'un magasin contient des produits de différentes marques : il y a des lecteurs MP3, des appareils photo, des caméras numériques, des téléphones portables ainsi que des ordinateurs portables. Les marques vendues sont Alpel, Syno, Massung et Liphisp. Bien sûr, chaque appareil est caractérisé par un prix, en plus du nombre restant en stock.

Exercice 3.11 *Définir des types adaptés pour représenter la situation décrite ci-dessus.*

Si besoin, vous compléterez ces définitions au fur et à mesure des questions qui suivent.

Exercice 3.12 *Écrire une fonction* `est_en_stock` *qui indique si un élément est présent en stock (c'est-à-dire si le nombre d'articles disponibles est strictement positif). Elle prend en argument un produit, une marque et un prix (caractéristiques de l'élément) et la liste des articles répertoriés.*

Exercice 3.13 *Écrire une fonction* `ajoute_article` *qui ajoute un article dans la liste, en vérifiant qu'il n'y est pas déjà, et s'il y est déjà, modifie le nombre d'éléments en stock dans la liste en additionnant le nombre en stock de l'argument article. Elle prend en argument un article et la liste des articles répertoriés.*

Exercice 3.14 *Écrire une fonction* `enleve_article` *qui enlève un article de la liste. Elle prend en argument la liste des articles répertoriés et l'article à enlever.*

Aider un client

Dans les 5 exercices suivants, on ne tient pas compte du fait qu'un produit soit en stock ou non (c'est-à-dire qu'on peut effectuer des réponses comportant des produits non présents en stock).

Exercice 3.15 *Écrire une fonction* `ces_produits` *qui prend en argument un produit et une liste d'articles et renvoie la liste des articles qui conviennent dans la liste d'articles (ex :* `ces_produits`*(MP3,L) renvoie tous les MP3 présents dans L).*

Exercice 3.16 ***(Le choix le plus courant)*** *Le vendeur se rend compte que les clients choisissent généralement le deuxième produit le moins cher, i.e. si l'on classe tous les produits de la liste par ordre de prix croissant, le deuxième produit de cette liste. Sans classer tous les articles correspondant à un même produit par ordre croissant, écrivez une fonction* `deuxieme_moins_cher` *qui prend en argument une liste d'articles répertoriés, un produit, et renvoie le choix préféré des clients.*

Pouvez-vous écrire cette fonction de sorte qu'elle ne parcoure qu'une seule fois la liste ?

Exercice 3.17 *Écrire une fonction* `budget` *qui prend en argument deux entiers m, budget minimal, et M budget maximal, ainsi qu'une liste d'articles, et renvoie la liste des articles compris dans ce budget (i.e. dont le prix p est tel que $m \leq p \leq M$).*

Gestion des stocks

Exercice 3.18 *Écrire une fonction* `achete` *qui prend en argument une liste d'articles et les nom de produit, marque et prix pour un élément, et fait diminuer de 1 la quantité en stock d'un article.*

Exercice 3.19 *Écrire une fonction* `commande` *qui prend en argument la liste des articles et renvoie la liste des articles à commander au fournisseur (ceux dont le nombre en stock est nul).*

4 File et file à priorités

Objectifs : codage de files fonctionnel en temps amorti constant ; files à priorité.

4.1 Structure de file

Une file est une structure de données modélisant un contenant d'objets et dans lequel les objets sont retirés dans le même ordre que celui dans lequel on les a insérés (on parle de *FIFO* pour *First In, First Out*). On considère ici des files polymorphes, pouvant contenir des éléments d'un type `'a` arbitraire.

Exercice 4.1 *Définir un type* `file` *et les opérations et constantes suivantes :*

```
val file_vide: 'a file
val est_file_vide: 'a file -> bool
val enfile: 'a -> 'a file -> 'a file
val defile: 'a file -> ('a * 'a file)
```

Vous représenterez vos files à l'aide d'une liste OCaml, dans laquelle les éléments sont enfilés en tête de liste et « défilés » en fin de liste.

Exercice 4.2 *Écrire des fonctions de conversion de* `'a list` *vers* `'a file` *et réciproquement. Il est demandé que ces fonctions manipulent les files uniquement avec les fonctions de l'interface ci-dessus et soient écrites avec un accumulateur. Par exemple la première conversion enfile les entiers de la liste en entrée dans une file initialement vide.*

Écrire une fonction `teste_file` *de type* `'a list -> bool` *qui combine les deux fonctions précédentes et vérifie que le résultat est le miroir de la liste initiale. On utilisera la fonction de renversement sur les listes de complexité linéaire.*

Exercice 4.3 *Construire une autre implémentation de votre type* `file` *de la façon suivante :*

- *Une file est représentée par* deux *listes, une d'entrée et une de sortie.*
- *Les éléments sont toujours enfilés en tête de la liste d'entrée et défilés depuis la tête de la liste de sortie.*
- *On transfère les éléments de la liste d'entrée vers la liste de sortie lorsqu'on essaye de défiler mais que cette dernière liste est vide,* ***et seulement dans ce cas****.*

4.2 File à Priorités

Une File à Priorités (FàP) est une structure de données similaire à la file, mais dans laquelle chaque élément est pourvu d'une *priorité* (par exemple, un entier), qui définit dans quel ordre les éléments seront extraits de la FàP. La priorité d'un élément est spécifiée lors de son insertion, et lors d'une extraction on choisit l'élément ayant la priorité la plus haute.

On considère ici des files à priorité polymorphes, pouvant contenir des éléments d'un type `'a` arbitraire.

Exercice 4.4 *Définir un type* `fap` *et les opérations et constantes suivantes :*

```
val fap_vide: 'a fap
val est_fap_vide: 'a fap -> bool
val insere: 'a -> int -> 'a fap -> 'a fap
val extrait: 'a fap -> ('a * 'a fap)
```

Vous représenterez vos FàP à l'aide d'une liste OCaml de couples (élément, priorité) *ordonnée par priorité décroissante.*

Exercice 4.5 *Écrire une fonction* `teste_fap` *de type* `'a list -> bool` *qui enfile les caractères de la liste dans une FàP initialement vide, en leur donnant des priorités croissantes, puis reconstruit une liste en défilant totalement la FàP et vérifie que le résultat correspond bien au miroir de la liste initiale.*

Exercice 4.6 *Construire une autre implémentation de votre type* `fap` *basée sur l'un des principes suivants au choix :*

- *les éléments sont rangés dans un Arbre Binaire de Recherche, ordonné selon les priorités*
- *les éléments sont rangés dans un Arbre partiellement Ordonné : chaque nœud porte un élément de priorité supérieure à celle de ses deux fils. Pour insérer un nouvel élément ou extraire l'élément de priorité maximale, on est amené à faire des échanges entre éléments le long d'un chemin entre la racine et une feuille*
- *les éléments sont rangés dans un Tas, représenté par un* `('a * int) array` *qui contient le parcours en largeur du tas.*

5 Analyse lexicale et syntaxique

On utilisera la technique d'analyse basée sur les types `analist` et `ranalist`, ou leurs variantes travaillant sur des listes paresseuses ou des flots. Commencer par récupérer le fichier définissant les primitives adéquates, qui sera au choix :

- `analist.ml`
- `anacomb.ml`
- `anacomblazy.ml`
- `anacombstream.ml`

Si l'on choisit `analist.ml`, le fichier OCaml débutera par :

```
#use "analist.ml" ;;
```

(Procéder de façon analogue pour `anacomb.ml`, etc.).

5.1 Suite de chiffres

Exercice 5.1 (Lecture d'un entier, préparé en TD)

Exercice 7.1 du poly de TD : lecture d'une suite de chiffres (caractères) ; calcul de la somme des entiers correspondants ; calcul de l'entier représenté par une suite de chiffres en base 10 par le schéma de Horner.

5.2 Expressions arithmétiques

Exercice 5.2 (Expressions arithmétiques, préparé en TD)

Exercice 7.3 du poly de TD : analyse d'expressions arithmétiques.

5.3 (*) Articulation entre passes d'analyse lexicale et syntaxique

On pourra se baser sur la version `analist.ml` ou sur la version `anacomb.ml` qui donne un code plus concis et proche de la grammaire.

Pour obtenir sous forme de `string` le contenu d'un fichier on pourra utiliser le code suivant :

```
let readfile : string -> string = fun nomfic ->
  let ic = open_in nomfic in
  really_input_string ic (in_channel_length ic)
```

Exercice 5.3 (Connect6 – facultatif)

Le jeu de Connect6 est un jeu de stratégie à deux joueurs (Noir et Blanc) se jouant sur un plateau quadrillé (de type Go) initialement vide :

1. *Les joueurs jouent l'un après l'autre en posant des pierres de leur couleur, à commencer par Noir.*
2. *Au premier tour, Noir pose une pierre, après quoi chaque joueur pose à son tour deux pierres sur des intersections libres.*
3. *Le premier joueur à réaliser un alignement de* 6 *pierres adjacentes de sa couleur gagne la partie. Un alignement peut être horizontal, vertical ou diagonal.*
4. *Si le plateau est entièrement rempli sans qu'un joueur ait gagné, la partie est déclarée nulle.*

Nous souhaitons parser un fichier qui contient une partie de Connect6 selon la grammaire suivante :

$$\begin{aligned} P &::= \textit{noir} \mid \textit{blanc} \\ C &::= (\ P\ \textit{int}\ \textit{int}\ ) \\ Cl &::= \varepsilon \mid C\ Cl \\ S &::= (\ \textit{int}\ \textit{int}\ )\ Cl \end{aligned}$$

Pour simplifier les choses la grammaire ci-dessus ne tient pas compte de l'alternance des joueurs dans les suites de coups.

Par exemple voici le contenu d'un fichier d'une partie de Connect6 :

```
(19 19) (noir 3 5) (blanc 5      8 )
    (blanc 5 9) (      noir 3 4)
(noir 3 6)
```

Dans cet exemple, la partie se joue sur un plateau de taille 19 *par* 19*. Noir pose sa première pierre en position* $(3, 5)$*, puis Blanc en pose deux, et ainsi de suite. On numérote les positions à partir de* 0*.*

1. *Définir un type* `coup` *permettant de représenter un coup d'une partie de Connect6, puis un type* `partie` *permettant de représenter une partie (dimensions du plateau incluses) telle que celle du fichier donné en exemple.*
2. ***Analyse lexicale**. Écrire un type* `token` *approprié, puis une fonction d'analyse lexicale pour chaque constructeur du type* `token`*, puis une fonction* `token_here` *qui rend le token en tête d'une liste de caractères. Écrire ensuite une fonction* `espaces` *qui aspire rend une liste de caractères privées des caractères d'espacement en préfixe de cette liste, puis une fonction* `next_token` *qui rend le premier token d'une liste de caractères commençant éventuellement par des espaces.*

   Enfin, écrire une fonction `tokens` *qui analyse une liste de caractères représentant une partie de Connect6 et qui retourne la liste de tokens correspondante.*

   On pourra, au choix, utiliser les combinateurs d'analyseurs.
3. ***Analyse syntaxique**. Écrire une fonction* `lit_partie` *qui analyse une liste de tokens représentant une partie de Connect6 et qui retourne la taille du plateau de jeu et la liste des coups de la partie.*

   On pourra ou non, au choix, utiliser les combinateurs d'analyseurs.
4. *(**) Définir un type* `plateau` *qui représente l'état d'un plateau de jeu de Connect6.*
5. *(**) Écrire une fonction* `joue_coup` *qui calcule un nouvel état à partir d'un état du plateau et d'un coup.*
6. *(**) Écrire une fonction* `resultat` *qui détermine l'issue de la partie pour un état du jeu passé en paramètre. L'issue pourra être une victoire (noire ou blanche), une partie nulle ou encore une partie non terminée.*

   Attention : certaines entrées sont invalides, par exemple un état dans lequel Noir comme Blanc a réalisé un alignement de 6 *pierres. Vous êtes libres de renvoyer ce que vous voulez sur une entrée invalide.*

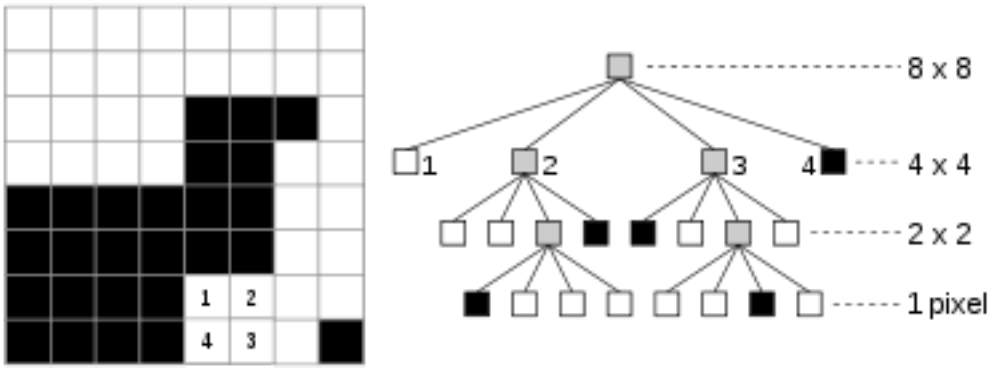
6 Quadtrees

Objectifs : Utiliser des exceptions pour programmer efficacement.

Manipuler des arbres représentant des images en niveau de gris.

Dans une image il est fréquent que de larges portions contiguës aient la même couleur, et il est souhaitable d'en tirer parti pour représenter ces images de façon compressée. Nous allons étudier une représentation de ce type, appelée *quadtree*.

Pour simplifier, on suppose les images carrées, de côté 2^n, et en niveaux de gris. L'idée est la suivante : une image unie se représente par sa couleur, tandis qu'une image composite se divise naturellement en quatre images carrées.



Nous considérons le type suivant :

```
type quadtree = Feuille of int
              | Noeud of quadtree * quadtree * quadtree * quadtree
```

Par convention l'ordre des sous-images dans le quadruplet sera le suivant : Nord-Ouest, Nord-Est, Sud-Est, Sud-Ouest.

6.1 Travail préliminaire : apprivoiser les quadtrees

1. Dessiner (sur papier !) une image simple en noir et blanc de 8x8 pixels, par exemple un smiley ou une icône.
2. Construire, toujours sur papier, le quadtree correspondant.
3. Représenter (dans un fichier .ml) ce quadtree dans le type OCaml proposé dans cet énoncé.
4. Récupérer les fichiers `qtparser` proposés à l'adresse :
   `https://ltpf.gricad-pages.univ-grenoble-alpes.fr/PF/fichiers-quadtrees/`
   puis essayez de les utiliser pour visualiser votre image, et éventuellement la modifier puis reconstruire le quadtree correspondant.

Cet exemple pourra vous servir de premier cas de test pour vos fonctions. Que ne permet-il pas de tester correctement ?

6.2 Fonctions de transformation des quadtrees

Exercice 6.1 *Écrire une fonction* `rot_pos` *de type* `quadtree -> quadtree` *qui effectue une rotation de l'image dans le sens inverse des aiguilles d'une montre (c'est-à-dire dans le sens positif).*

Exercice 6.2 *Écrire une fonction* `rot_neg` *de type* `quadtree -> quadtree` *qui effectue une rotation de l'image dans le sens des aiguilles d'une montre (c'est-à-dire dans le sens négatif).*

Exercice 6.3 *Écrire une fonction* `miroir_hori` *de type* `quadtree -> quadtree` *qui effectue une symétrie selon l'axe horizontal.*

Exercice 6.4 *Écrire une fonction* `miroir_vert` *de type* `quadtree -> quadtree` *qui effectue une symétrie selon l'axe vertical.*

Exercice 6.5 *Écrire une fonction* `inversion_video` *de type* `quadtree -> int -> quadtree` *qui modifie une image en inversant les niveaux de gris : le niveau* 0 *devient le niveau maximal (passé en argument) et inversement.*

Exercice 6.6 *Écrire une fonction* `max_gris` *de type* `quadtree -> int` *qui parcourt une image et retourne la valeur du niveau de gris maximal présent dans l'image.*

Exercice 6.7 *Le codage d'une image par un quadtree n'est pas unique : par exemple* `Feuille(0)` *et* `Noeud (0,0,0,0)` *représentent la même image.*

Écrire une fonction `compare : quadtree -> quadtree -> bool` *qui détermine si les deux quadtrees passés en arguments représentent une image identique.*

Il est conseillé pour cette fonction de faire bon usage des exceptions, afin de stopper la comparaison dès qu'on trouve un élément distinct dans les deux images.

Exercice 6.8 (facultatif) *Modifier la fonction précédente et les exceptions utilisées afin d'être capable d'exhiber une zone qui diffère dans les deux quadtrees.*

Exercice 6.9 *Écrire une fonction* `min_quad` *de type* `quadtree -> quadtree` *qui minimise le* `quadtree` *passé en entrée, l'idée étant que lorsque les quatre fils d'un nud ont la même couleur, on peut les remplacer par une unique feuille de cette couleur.*

6.3 Images rectangulaires (facultatif)

Afin de pouvoir représenter des images qui ne sont pas forcément des carrés, et dont les dimensions ne sont pas des puissances de 2, on introduit les types suivants :

```
type mixtree = Q of quadtree * int | T of recttree
 and recttree = NoeudT of mixtree * mixtree * mixtree * mixtree
```

Pour découper une image de taille $n \times m$, on commence par chercher le plus grand carré de taille une puissance de 2 qui rentre dans l'image. On le code par un `quadtree` (dont on mémorise la taille), et on recommence récursivement avec les trois autres rectangles restants. Les quatre rectangles, dont un au moins est un carré, sont les quatre fils du `recttree` codant l'image entière.

Exercice 6.10 *Écrire une fonction* `taille_rect` *de type* `recttree -> int * int` *qui donne les dimensions d'une image codée par un* `recttree`*.*

Exercice 6.11 *Écrire une fonction* `rect_valide` *de type* `recttree -> bool` *qui vérifie si un* `recttree` *vérifie la contrainte de toujours contenir un* `quadtree` *de dimensions maximales dans son coin Nord-Ouest.*

7 Un mini-interpréteur pour While

7.1 Définition et analyse d'un langage de programmation simple (à traiter en TD)

Comme en While, on considère qu'un programme est :
- soit ne rien faire,
- soit une affectation (d'une expression à une variable),
- soit deux programmes mis bout-à-bout (séquence),
- soit une instruction conditionnelle (constituée d'une expression, d'un programme à exécuter si l'expression vaut 1, et d'un second programme à exécuter si l'expression vaut 0),
- soit une boucle while (constitué d'une expression et d'un corps ; la condition d'arrêt étant que l'expression vaut 0).

De plus, par souci de simplification, on considérera ici :
- que toutes les variables sont booléennes (et valent 0 ou 1)
- que la condition d'un if ou d'un while est toujours constituée d'une variable seulement
- que le membre droit d'une affectation peut être : soit 0, soit, 1, soit une autre variable.
- Enfin on se contentera de 4 variables booléennes a, b, c et d.

On pourrait ainsi écrire un programme comme :

```
a := 1 ;
b := 1 ;
c := 1 ;
while(a) {
    if(c) {
       c := 0 ;
       a := b
    } else {
       b := 0 ;
       c := a
    }
}
```

Exercice 7.1 *Définir une hiérarchie de types OCaml permettant de représenter tous les programmes admis par la description ci-dessus.*

Pour éviter une analyse lexicale qui nous détournerait du cœur du sujet, on écrit les mots-clés du langage sur un seul caractère, on délimite le corps des if et des while par des accolades et on se dispense du mot-clé `else` (quitte à laisser un programme vide pour le second bloc du `if`).

Ainsi, notre programme exemple du début de l'énoncé s'écrit :

```
a:=1;
b:=1;
c:=1;
w(a){
  i(c){
    c:=0;
    a:=b
  }{
    b:=0;
    c:=a
```

```
  }
}
```

On a ici conservé les tabulations et les retours à la ligne pour que le programme reste lisible, mais on devrait même s'en dispenser également, et donc notre programme s'écrit finalement

```
a:=1;b:=1;c:=1;w(a){i(c){c:=0;a:=b}{b:=0;c:=a}}
```

Exercice 7.2 *Donner une grammaire décrivant ce langage.*

Exercice 7.3 *La grammaire que vous avez écrite est très probablement récursive gauche dans le cas de la séquence de programmes. Modifiez-la pour remédier à ce problème.*

Exercice 7.4 *Étendre votre grammaire et vos types pour traiter les affectations de la forme* `V:=#` *où* `V` *est une variable, et qui signifiera « remplacer la valeur de la variable par sa négation ».*

7.2 SOS

Sémantique Opérationnelle Structurelle (SOS), à traiter en TD

La SOS est une méthode permettant de donner un sens à un programme dans un certain langage ou plus précisément donner une signification à chaque instruction, c'est-à-dire la manière dont elle s'exécute et ses effets.

L'exécution d'un programme est donnée par des transitions entre configurations, notées *config* $\longrightarrow$ *config_suivante*, une configuration étant soit un couple noté $\Gamma \vdash (P)$, où Γ est un état mémoire (partie *data*), parfois appelé *environnement* et P est un programme à exécuter à partir de cet état, soit un état final. Une transition est donc soit de la forme $\Gamma \vdash (P) \longrightarrow \Gamma' \vdash (P')$, soit de la forme $\Gamma \vdash (P) \longrightarrow \Gamma' \vdash ()$. Pour chaque forme possible de P, on indique que sera, après **une** étape d'exécution, le nouvel état Γ' et quelle est la suite éventuelle du programme P'.

La première règle de transition s'applique au cas où P est une affectation :

$$\frac{}{\Gamma \vdash (i := expr) \longrightarrow \Gamma \,/\, i = [\![expr]\!]_\Gamma \vdash ()}$$

Ici, $[\![expr]\!]_\Gamma$ représente la valeur renvoyée par l'évaluation de *expr* dans l'environnement Γ, et la notation $\Gamma \,/\, i = v$ représente l'environnement obtenu à partir de Γ dans lequel toutes les associations sont conservées, sauf pour la variable i qui est associée à la valeur v. Une fois la modification faite dans l'environnement, il ne reste plus rien à exécuter et on aboutit donc à une configuration « finale ».

Les règles suivantes s'appliquent aux programmes qui sont des séquences, de la forme $P\,;Q$. Il faut d'abord exécuter le premier pas de P, puis considérer deux cas, suivant que la continuation de P est vide ou non. S'il ne reste rien à faire dans P après en avoir exécuté un pas, le programme devient le second membre de la séquence (règle de gauche). Sinon la règle de droite s'applique.

$$\frac{\Gamma \vdash (P) \longrightarrow \Gamma' \vdash ()}{\Gamma \vdash (P\,;Q) \longrightarrow \Gamma' \vdash (Q)} \qquad \frac{\Gamma \vdash (P) \longrightarrow \Gamma' \vdash (P')}{\Gamma \vdash (P\,;Q) \longrightarrow \Gamma' \vdash (P'\,;Q)}$$

Enfin, cette dernière règle s'applique aux programmes qui sont des boucles conditionnelles : on « déplie » une itération de la boucle pour la remplacer par une condition, qui permettra de décider si on exécute le corps et recommence, ou s'il ne reste rien à faire.

$$\frac{}{\Gamma \vdash (\texttt{while}\ \textit{expr}\ P) \quad \longrightarrow \quad \Gamma\ \vdash\ (\texttt{if}\ \textit{expr}\ \texttt{then}\ (P\ ;\ \texttt{while}\ \textit{expr}\ P)\ \texttt{else}\ \texttt{Skip})}$$

Exercice 7.5 *Écrire la (ou plutôt les) règles de transition de l'instruction* `if`*.*

Exercice 7.6 (facultatif) *Améliorer votre grammaire et vos types pour pouvoir stocker des entiers dans les variables, et écrire des expressions arithmétiques simples dans les membres droits des affectations, ainsi que dans les conditions des if et des while.*

7.3 Implémentation de l'analyseur

Exercice 7.7 *Implémenter un analyseur syntaxique en OCaml pour la grammaire obtenue à la fin du TD (dérécursivée).*

Exercice 7.8 *Écrire quelques programmes While pour tester votre analyseur. (Vous pouvez augmenter le nombre de variables disponibles si vous en ressentez le besoin.)*

Exercice 7.9 (facultatif) *Améliorer l'analyseur pour qu'il accepte des programmes avec des blancs arbitraires : espaces, indentations et retours à la ligne.*

7.4 Mécanique d'environnement et interpréteur

Le principe d'un interpréteur est d'exécuter pas à pas les instructions d'un programme dans un certain langage. Il s'agit ici d'une donnée arborescente Ocaml, il faut donc parcourir l'arbre représentant le programme en exécutant au fur et à mesure les instructions rencontrées ce qui revient à traduire les règles de transition en Ocaml.

Il nous faudra pour cela une modélisation de l'état contenant les valeurs courantes des variables.

On représente l'état par un `bool array` de taille 4, dans lequel la cellule d'indice 0 contient la valeur de a, celle d'indice 1 la valeur de b, et ainsi de suite.

Exercice 7.10 *Écrire des fonctions permettant respectivement :*
- *d'initialiser cet état (avec toutes les variables à 0) ;*
- *de lire la valeur d'une variable ;*
- *de modifier la valeur d'une variable ;*
- *d'exécuter une instruction d'affectation.*

Exercice 7.11 *Écrire une fonction* `faire_un_pas_p` *qui rend le nouveau programme après exécution d'une transition, ainsi qu'une fonction* `faire_un_pas_e` *qui rend le nouvel état après exécution d'une transition.*

```
faire_un_pas_p : programme → etat → programme option
```

```
faire_un_pas_e : programme → etat → etat
```

Exercice 7.12 *Écrire une fonction* `executer` *qui exécute un programme jusqu'à ce qu'il soit terminé.*

```
executer : programme → etat
```

Indication. Attention, si le programme à exécuter boucle l'interpréteur bouclera en l'exécutant. D'autre part, on pourra utiliser une fonction auxiliaire comportant un état comme argument supplémentaire et qui sera appelée avec l'état initial.

Exercice 7.13 (facultatif) *Instrumenter votre interpréteur pour qu'il compte et affiche le nombre de pas nécessaires pour interpréter le programme.*

Exercice 7.14 (facultatif) *Implémenter la sémantique naturelle sur le même principe et vérifier que les deux sémantiques correspondent sur vos programmes de test.*

7.5 Analyse en 2 temps

Ce TP prolonge le précédent, en proposant d'analyser un "vrai" programme While. Pour cela, on procèdera en deux phases :

1. l'analyse lexicale, qui a été ignorée pour l'instant et fait donc l'objet de ce TP
2. puis l'analyse syntaxique, qui a déjà été réalisée au TP précédent et doit pouvoir être réutilisée, à quelques ajustements mineurs près.

Exercice 7.15 *Écrire un analyseur lexical pour le mini-langage While proposé au TP précédent :*
- *On écrira intégralement les mots-clés du langage : `if`, `then`, `else`, `while`.*
- *Les valeurs prises par les variables pourront être des entiers arbitraires, pas seulement 0 et 1.*
- *Dans un premier temps on considère que `if` et `while` prennent toujours une unique variable en argument, et qu'ils la comparent à zéro.*

Il sera utile de définir un type `lexeme` pour produire un flot de lexèmes à la suite de cette analyse lexicale.

Exercice 7.16 *Adapter l'analyseur syntaxique du TP précédent pour qu'il prenne en entrée non pas directement le code source du programme While, mais le flot de lexèmes produit par votre analyseur lexical.*

Vérifier que l'analyse en 2 temps fonctionne correctement, et si vous avez implémenté l'interpréteur, qu'il vous est toujours possible de l'utiliser sur les nouveaux programmes considérés (un ajustement peut s'avérer nécessaire, par exemple pour la comparaison à zéro).

Exercice 7.17 (facultatif) *Étendre le langage (et donc les analyseurs et l'interpréteur) pour traiter différentes constructions, par exemple :*
- *des expressions arithmétiques simples (addition, opposé ou soustraction) à droite de `:=`*
- *des comparaisons simples (égalité ou inégalité entre deux variables, entre une variable et une valeur entière) dans le `if` et dans le `while`*
- *la possibilité d'écrire un `if`/`then` sans `else`*

8 Modules et foncteur

Objectifs : Définir différents modules respectant une signature donnée, puis construire et instancier un foncteur à l'aide de ces modules.

Contexte On cherche ici à modéliser une mémoire cache comme celle présente dans tous les ordinateurs modernes. Pour rappel, le principe en est le suivant :

- L'ordinateur dispose d'une grande quantité de mémoire, peu coûteuse à fabriquer mais relativement lente d'accès.
- On ajoute donc une mémoire *cache*, beaucoup plus rapide mais d'une capacité très limitée en raison de son coût.
- Les informations utilisées par un processus sont recopiées depuis la mémoire dans le cache, de sorte que son coût d'accès soit diminué.

Mais, en raison de la taille limitée du cache, il faut constamment y « faire de la place » pour que ce celui-ci contienne des informations pertinentes pour les processus en cours d'exécution. Plus concrètement, lorsqu'un processus demande d'accéder à un bloc mémoire i :

1. Si le cache contient déjà ce bloc, tout se passe bien, le processus peut y accéder directement.
2. Sinon (on parle alors de *défaut de cache*), il faut éliminer un des blocs du cache pour le remplacer par le bloc i.

Reste à choisir le bloc à éliminer... ce qu'on essaye de faire de sorte que les données qui restent dans le cache soient les plus utiles dans le futur.

Aucune politique de choix n'est optimale, mais nous allons ici implémenter trois des plus répandues.

8.1 Une première politique

On représente les blocs de mémoire par un entier, ici compris entre 0 et 9. Une politique de choix est alors constituée des éléments de la signature suivante :

```
module type POLITIQUE =
sig
  type t
    (* recense les numéros des blocs en cache *)
  val vide : t
    (* état initial du cache *)
  val est_plein : t -> bool
    (* pour ce TP on considère que le cache contient 3 blocs au maximum *)
  val inserer : int -> t -> t
    (* charge un nouveau bloc dans un cache non plein *)
  val extraire : t -> (int * t)
    (* renvoie le bloc à supprimer et le reste du cache *)
  val print : t -> unit
    (* affiche le contenu du cache à l'écran (pour les tests) *)
end
```

Exercice 8.1 (Cache FIFO) *À l'aide d'une file, implémenter une première politique consistant simplement à supprimer du cache son plus ancien élément (donc le premier qui y est entré).*

Vous pouvez définir toutes les fonctions auxiliaires et exceptions que vous jugerez utiles.

8.2 Notre cache

Nous pouvons maintenant construire notre cache, sous forme d'un foncteur `Cache` paramétré par une politique de signature `POLITIQUE`, et dont la signature sera :

```
functor (P : POLITIQUE) ->
  sig
    val en_cache : bool array
    val init : unit -> P.t
    val charger : int -> P.t -> P.t
  end
```

Quelques précisions :

- Pour savoir si un accès mémoire provoque ou non un *défaut de cache*, il faut déterminer efficacement si un bloc est présent ou non dans le cache. C'est le rôle rempli par le tableau `en_cache`, indexé par les numéros de blocs (donc pour nous de 0 à 9).
- La fonction `init` permet de remettre le cache dans son état initial, lorsqu'aucun bloc mémoire n'a encore été chargé.
- La fonction `charger` effectue ce qu'on attend du cache, à savoir :
  - vérifier si le bloc demandé est déjà présent ;
  - le cas échéant, le charger en cache ;
  - si nécessaire, éliminer (auparavant) un bloc du cache en suivant la poltique `P` ;
  - et bien sûr mettre à jour et renvoyer le nouvel état du cache.

Exercice 8.2 *Implémenter ce foncteur, puis l'instancier avec la politique FIFO.*

Exercice 8.3 *Ajouter à votre foncteur une fonction* `test_cache : int list -> unit` *qui simule le comportement du cache lorsqu'on accède successivement aux blocs de la liste donnée en argument, en affichant le contenu de ce cache après chaque accès mémoire.*

Observer par exemple le résultat de `test_cache [5;0;1;2;0;3;2;1;5;4]`.

8.3 Deux autres politiques

La politique FIFO a pour atout sa simplicité, en revanche elle ne permet pas de conserver en cache de façon durable des données utilisées régulièrement par le processeur. Les algorithmes les plus courants sont les deux suivants.

Exercice 8.4 (Cache LRU (Least Recently Used)) *Dans cette politique, on supprime le bloc mémoire utilisé le moins récemment. Ainsi, un bloc déjà présent dans le cache mais accédé à nouveau regagnera en « fraîcheur » et sera conservé plus longtemps.*

Pour cela, il nous faut une fonction `maj : int -> t -> t` *capable de réordonner les blocs si besoin. Déclarez cette fonction dans la signature* `POLITIQUE`*, définissez-la dans vos modules et utilisez-la dans votre foncteur* `Cache`.

Implémentez ensuite la politique LRU et testez le résultat sur votre cache.

Exercice 8.5 (Cache LFU (Least Frequently Used)) *Cette autre politique consiste à supprimer le bloc le moins utilisé. Il faut donc conserver, pour chaque bloc du cache, une information supplémentaire à savoir le nombre d'accès à ce bloc.*

À fréquence égale, on supprimera bien entendu le bloc le plus ancien.

Implémenter également cette politique et tester le résultat sur votre cache.