UNIVERSITÉ GRENOBLE ALPES

UFR IM²AG

DÉPARTEMENT LICENCE SCIENCES ET TECHNOLOGIE

# LICENCE SCIENCES & TECHNOLOGIES, 1re ANNÉE

## UE INF201/231

## ALGORITHMIQUE ET PROGRAMMATION FONCTIONNELLE

2022-2023

[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

# EXERCICES DE TRAVAUX PRATIQUES

# Table des matières

```
OCaml version 4.14.1
Enter #help;; for help.

Findlib has been successfully loaded. Additional directives:
#require "package";;      to load a package
#list;;                   to list the available packages
#camlp4o;;                to load camlp4 (standard syntax)
#camlp4r;;                to load camlp4 (revised syntax)
#predicates "p,q,...";;   to set these predicates
Topfind.reset();;         to force that packages will be reloaded
#thread;;                 to enable threads
```

L'objectif principal des travaux pratiques proposés ici est de renforcer la compréhension des concepts et outils développés en cours et en travaux dirigés. Les savoir et savoir-faire exigibles sont :

- **Maîtriser les notations** concernant les types de base et le produit de types.
- Exploiter les messages de l'interprète concernant les types. En particulier, **identifier les erreurs typiques** liées à des incohérences de types.
- **Implémenter en OCaml** notations mathématiques, concepts et outils des cours / TD.

Un *interprète* est un logiciel capable d'évaluer des expressions et d'exécuter des programmes.

Toute expression OCaml soumise à évaluation doit être suivie de deux points-virgules (`;;`). L'interprète du langage attend la saisie d'une expression en affichant le symbole d'invite[1] `#`. L'utilisateur fournit l'expression à évaluer et appuie sur la touche Entrée[2]. Deux réactions sont alors possibles :

- Tout se passe bien : l'interprète répond en affichant un type et une valeur.
- L'expression fournie ne respecte pas les contraintes de typage : l'interprète répond alors par un message d'erreur et la partie de l'expression qui pose problème est soulignée.

[1] En Anglais : prompt.
[2] Enter sur certains claviers

## Première partie

# TYPES, EXPRESSIONS ET FONCTIONS

# Chapitre 1

# TP1 une séance d'expérimentation type

L'objectif de ce chapitre est de vous permettre de vous familiariser rapidement avec l'environnement de développement OCaml. Il explique comment lancer un éditeur et l'interpréteur OCaml, utiliser ces deux outils pour construire des programmes et les tester, construire le compte rendu de TP et terminer la session.

## 1.1 Déroulement d'une séance de TP

Pour débuter cette première séance, suivre les instructions de la section OCaml dans ton TP sur Caseine.

### 1.1.1 Édition du fichier de compte rendu

Commencez tout d'abord par insérer un entête de compte rendu, comme par exemple (n'oubliez pas de modifier les noms ...) :

```
(* ------------------------------------------------------------------------
   inf201_Puitg_Basset_Couillet_TP1.ml : cr exercices TP no1

   François Puitg <francois.puitg@univ-grenoble-alpes.fr> \
   Nicolas Basset <bassetni@univ-grenoble-alpes.fr>        >  Groupe boss1
   Romain Couillet <Romain.Couillet@grenoble-inp.fr>      /
   ------------------------------------------------------------------------
*)
```

Avant de continuer, sauvegarder le fichier. Au DLST, pensez à sauvegarder régulièrement votre compte-rendu au cours de la séance, il arrive qu'il y ait des coupures de courant.

### 1.1.2 Utilisation de l'interpréteur OCaml

L'interpréteur OCaml `utop` affiche l'invite (prompt) suivi du curseur : `utop [...]:`
Lorsque vous taperez une expression terminée par deux points virgules (`;;`), celle-ci sera évaluée par l'interpréteur, qui en retour affichera un résultat. Tapez par exemple l'expression suivante dans la fenêtre de l'interpréteur OCaml :

```
3 + 12 ;;
```

Après avoir appuyé sur la touche Entrée vous obtiendrez le résultat suivant :

```
- : int = 15
#
```

L'interprèteur a évalué l'expression et a affiché comme résultat la valeur correspondante (`15`) ainsi que le type de l'expression (`int` pour «integer»). Après chaque évaluation, l'interprèteur affiche de nouveau l'invite et attend une nouvelle expression.

Tapez maintenant :

```
x * 3
```

puis appuyez immédiatement sur Entrée sans taper les caractères ;; . Le curseur passe à la ligne, mais aucun résultat n'est affiché : vous pourrez ainsi entrer des expressions complexes en utilisant plusieurs lignes. Tant que vous ne taperez pas les deux points virgules ( ;; ), l'interprète ne fera rien, si ce n'est attendre la fin de l'expression à évaluer. Les deux points virgules permettent en fait de délimiter les expressions à évaluer.

Tapez les deux points virgules puis Entrée et observez la réponse de l'interprète :

```
Error: Unbound value x
```

ce qui signifie que la variable `x` n'est pas liée (`unbound`) : elle est absente du contexte d'évaluation courant. Dans cet exemple, l'interpréteur a répondu par un message d'erreur et souligne l'endroit présumé de l'erreur dans la ligne. Après avoir appuyé sur Entrée à la fin d'une ligne, vous pouvez la rappeler en appuyant sur la touche ↑.

**Par ailleurs, *lorsque vous quitterez l'interpréteur OCaml, tout ce que vous avez tapé sera perdu.* Il est donc nécessaire d'utiliser à la fois l'interpréteur OCaml et l'éditeur.**

### 1.1.3 Utilisation conjointe de l'éditeur et de l'interpréteur OCaml

Au cours des différents TP, lorsque vous devrez taper des expressions ou des fonctions complexes, utiliser la méthode suivante :

1. Taper le texte souhaité dans la fenêtre de l'éditeur. Vous pourrez ainsi utiliser toutes les fonctions d'édition disponibles (couper, coller, complétion, ...).

2. Transférer le texte dans la fenêtre de l'interpréteur afin de l'évaluer grâce à la combinaison de touche shift Entrée.

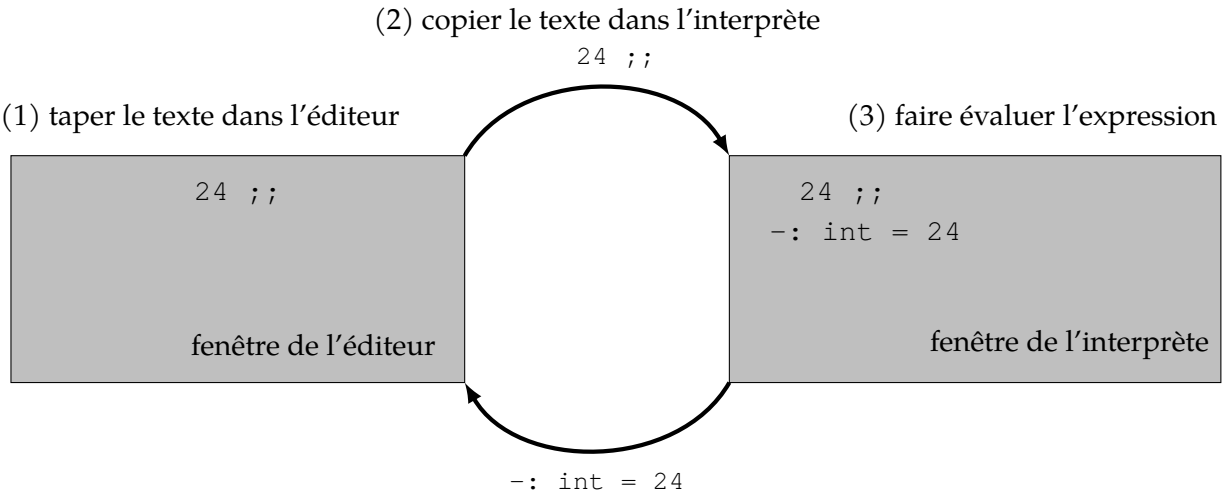
3. L'interpréteur OCaml donnera sa réponse.

4. Si le résultat est celui attendu, vous pourrez le copier-coller dans le fichier du compte rendu, sous la forme d'un commentaire OCaml : `(* ... *)` (utiliser les raccourcis clavier ctrl : et alt shit A pour générer les commentaires lignes ou bloc automatiquement).

La figure ci-dessous montre les différentes étapes de ce processus :



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg




Après ces opérations, votre compte-rendu doit ressembler à cela :

```
(* Expressions          Réponses du système *)          1
24 ;;                   (* - : int = 24 *)             2
3+4 ;;                  (* ... *)                      3
```

Noter les caractères `(*` et `*)` qui encadrent un texte destiné à l'utilisateur humain : il ne sera pas évalué par OCaml lors d'une lecture ultérieure du fichier.

Pour effacer la fenêtre de l'interpréteur OCaml, on peut appeler la commande `clear` de l'interpréteur de commandes UNIX sous-jacent. On utilise pour cela la fonction OCaml `command` de la librairie `Sys` :

```
# Sys.command "clear" ;;
```

### 1.1.4 Construction du compte-rendu de TP

**Objectifs**

Dans tous les cas de figure, construire le compte rendu est essentiel :

- il doit vous permettre de garder la trace des expérimentations effectuées et donc de vous préparer aux examens ;
- il doit permettre à vos enseignants d'évaluer le travail effectué pendant chaque séance et de vous donner des conseils pour les prochaines séances.

Le compte rendu est un fichier OCaml (dont le nom se termine par `.ml`) contenant les définitions des objets que vous avez élaborées au cours d'une séance **ainsi que les tests effectués**.

Lorsque vous sortirez de l'interpréteur OCaml, toutes les définitions seront perdues, seul le contenu du fichier de compte rendu (que vous aurez sauvegardé) sera conservé.

**Pour charger à nouveau les définitions de fonction lors des séances suivantes, tapez la commande :** `#use "inf201_Puitg_Basset_Couillet_TP1.ml" ;;`

`#use` est une *directive* (adressée à l'interpréteur).

Cette directive permet aussi de vérifier qu'il n'y a pas d'erreur de syntaxe dans le fichier. Elle peut être utilisée juste avant la fin d'une séance de TP pour laisser le fichier dans un état stable, et/ou au début de chaque séance pour vérifier que l'on repart sur des bases saines.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

**Structure du compte rendu**

Élaborer un compte rendu est un travail aussi important qu'utile. Comme le fichier résultant doit pouvoir être interprété par OCaml (pour pouvoir le relire lors des séances suivantes), toutes les informations qui ne correspondent pas à des expressions ou des fonctions OCaml valides doivent être mises entre commentaires (c'est-à-dire entre `(*` et `*)` en OCaml).

Dans le compte rendu, pour chaque fonction vous devrez fournir :

1. *La spécification de la fonction*. Il est impératif de donner la spécification *AVANT* de débuter la réalisation.
2. *Une réalisation en OCaml* avec des commentaires significatifs. La présentation des fonctions OCaml doit être soignée. En particulier, il est impératif que la mise en page fasse apparaître la structure (indentation).
3. *Les jeux de tests* validant la correction de la fonction réalisée. Chaque jeu d'essais doit être accompagné d'un commentaire justifiant la pertinence des données choisies.
4. Dans certains cas *la trace de l'évaluation de la fonction* vous sera demandée.

**Exemple de compte-rendu**

```
(* ------------------------------------------------------------------------
   inf201_Puitg_Basset_Couillet_TP1.ml : cr exercices TP no1

   François Puitg <francois.puitg@univ-grenoble-alpes.fr> \
   Nicolas Basset <bassetni@univ-grenoble-alpes.fr>        >  Groupe boss1
   Romain Couillet <Romain.Couillet@grenoble-inp.fr>      /
   ------------------------------------------------------------------------

EXERCICE 1.2
  somme3 : int -> int -> int -> int
  Sémantique : somme de trois entiers
  Algorithme : utilisation de +
*)
let somme3 (a:int) (b:int) (c:int) : int =
  a + b + c

(* Tests : *)
let _ = assert ((somme3 1 2 3) = 6)           (* cas général *)
let _ = assert ((somme3 (-1) 2 (-1)) = 0)     (* entiers négatifs *)
let _ = assert ((somme3 0 0 0) = 0)           (* cas zéro *)
```

La construction `assert expr`, où `expr` est une expression booléenne, évalue `expr` et :

- ne renvoie rien si l'expression est vraie (`true` en OCaml),
- lève une exception si elle est fausse (`false` en OCaml), ce qui stoppe le programme, et affiche l'emplacement où l'erreur est survenue (nom du fichier, numéro de ligne et de colonne).

Cette construction permet un développement itératif rapide : le programmeur n'écrit qu'une fois pour toutes les jeux d'essais ; à chaque modification de sa fonction, il lui suffit de réévaluer les `assert` pour vérifier qu'il n'y a pas de *régression* (se dit de l'apparition d'un bug qui fait qu'une fonction ne répond plus aux exigences spécifiées).



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

Pour que la programmation itérative non régressive soit opérationnelle, il est nécessaire que le nombre et la pertinence des jeux d'essais soit suffisant.

Le fichier de compte-rendu doit être conservé : il constituera un support de révision.

### 1.1.5 Fin de la séance

- N'oubliez pas avant de quitter l'éditeur de vérifier que le compte-rendu est syntaxiquement correct (directive `#use`). Pour fermer l'interpréteur OCaml, taper la commande `#quit;;` ou plus rapidement Ctrl D.
- Si l'enseignant vous le demande, déposez le compte-rendu sur la plateforme pédagogique de l'UE.
- Si vous êtes en salle de TP, n'oubliez pas de quitter également la session `Windows`.

# Chapitre 2

# Exercices et problèmes

**Sommaire**



Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TD$n$ ou TP$n$ , où $n$ est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n$^o$ $n$ sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2$^e$ créneau de TP (non supervisé).

## 2.1 TP1 Type et valeur d'une expression

Ce premier exercice a pour objectif de manipuler les types de base du langage OCaml. Pour chacune des expressions proposées, essayez de prédire son type et sa valeur, puis saisissez l'expression (terminée par `;;`) dans la fenêtre de l'interprète et observez la réponse. Vous pouvez bien sûr essayer d'autres expressions que celles proposées.

Lisez attentivement et conservez les messages d'erreur (en les copie-collant dnas le fichier compte rendu et en les mettant en commentaires, entre `(*` et `*)`) ; cela vous sera utile quand vous les retrouverez dans un contexte plus complexe.

### 2.1.1 Expressions basiques

**Q1.**
```
24;; 3+4;; 3+5.3;; 6.1+8.2;; 3.2+.6.1;; 6+.5;; 6.+.5.;;
```

**Q2.**
```
'f';; '5';; '3'+4;; '3+4';; 'x';; x;;
```

**Q3.**
```
true;; false;; true && false;; true || false;; not(false);;
```

**Q4.**
```
4 + 3 * 2 ;;
4 + 3 / 2 ;;
(4 + 3) / 2 ;;
4 + 3 /. 2 ;;
(4. +. 3.) /. 2. ;;
10 mod 5 ;;
10 mod 3 ;;
10 mod 0 ;;
```

Conclure en donnant la spécification des opérateurs `/`, `/.` et `mod`.

### 2.1.2 Opérateurs de comparaison

**Q5.**
```
2=3;; 'e'='t';; false=false;; 4=false;; '3'=3;; 6.=6.;; 8.1=7;;
```

**Q6.**
```
2<3;; 'e'<'t';; false<true;; true<false;; 4<false;; '4'<'6';;
2>= 3;; 2>  =3;; 2<>2;;
```

Essayons de les enchaîner :

**Q7.**
```
(2=3)=true ;;
not(2=3) ;;
(2=3)=false ;;
false=(2=3) ;;
```

**Q8.**
```
false=2=3 ;;
2=3=false ;;
```

Que pouvez-vous en conclure sur l'ordre d'évaluation des égalités successives ?

**Q9.**
```
2 < 3 < 4 ;;
2 = 3-1 = 4-2 ;;
```

Que pouvez-vous en conclure ?

**Q10.**
```
2<3 && 3<4 ;;
not (4<=2) || false ;;
not true && false ;;
true || true && false ;;
```

Que pouvez-vous en conclure concernant les priorités des opérateurs logiques `&&` (et), `||` (ou) et `not` (non) ?

### 2.1.3 Fabrication de jeux d'essai

La directive `assert` permet de fabriquer des jeux d'essai qui peuvent être ensuite réutilisés en les chargeant dans l'interprète.

**Q11.** Evaluer les expressions suivantes :

```
assert ((5=3) = false);;
assert ((5=5) = true);;
assert ((5=3) = true);;
```

Que pouvez-vous en conclure ?

**Q12.** Sauvegarder votre fichier `crtp1.ml`, taper dans la fenêtre dans laquelle vous avez lancé l'interprète OCaml `#use "crtp1.ml" ;;` et observez ...

### 2.1.4 Définition d'une constante

Tapez dans l'interprète `let a : int = 8;;`.

**Q13.** Evaluer les expressions suivantes :

```
a ;;
a + 5 ;;
a +. 9.1;;
```

Que pouvez-vous en conclure ?

### 2.1.5 Expressions conditionnelles

**Q14.** La constante `a` ayant été définie par `let a : int = 8;;`, parmi les expressions suivantes une seule est correctement typée. Laquelle ? :

```
if a < 10      if a < 10      if a < 10      if a < 10
then true      then a         then a         then true
else a         else false     ;;             else false
;;             ;;                            ;;
```

Déterminez la règle non respectée dans les autres implémentations, puis vérifiez vos réponses en utilisant l'interprète OCaml.

## 2.2 TP1 Maximum entre plusieurs entiers

Dans ce paragraphe nous définissons une fonction qui calcule le maximum entre deux entiers, nous la testons puis nous nous posons le même problème pour trois entiers.

### 2.2.1 Spécification, réalisation d'une fonction

On définit une fonction `max2` qui calcule le maximum de deux entiers :

```
(*                                                                    1
  |SPÉCIFICATION                                                      2
  |max2 : maximum de deux entiers                                     3
  | - Profil       : max2 : int -> int -> int                        4
  | - Sémantique : (max2 a b) est le plus grand des deux entiers a et b 5
```

```
  | - Exemples et propriétés :                                          6
  |      (a) (max2 3 4) = 4                                            7
  |      (b) (max2 4 3) = 4                                            8
  |      (c) ∀ a ∈ ℤ, (max2 a a) = a                                   9
  |RÉALISATION                                                         10
  | - Algorithme : le maximum est à mi-distance à droite du milieu     11
  | - Implémentation :                                                 12
*)                                                                     13
let max2 (a: int) (b: int): int =                                      14
  ( (a+b) + abs(a-b) ) / 2                                             15
```

**Q1.** Soumettre cette définition de fonction à l'interprète OCaml, et observer la réponse du système.

De manière générale, lors de la définition d'une fonction, OCamlrépond en donnant le profil de la fonction (c'est-à-dire `nom_fonction : types_paramètres -> type_résultat`). Par contre l'implémentation (le code qui définit la fonction) n'est pas réaffichée.

**Q2.** La valeur absolue est une fonction prédéfinie. Observer la réaction de l'interprète après la saisie de l'expression `abs ;;`. Donner la spécification de `abs`. Vérifier sur un jeu d'essai pertinent que cette fonction fait bien ce que vous avez décrit.

### 2.2.2 Utilisation d'une fonction

Pour connaître le profil d'une fonction il suffit de fournir l'expression
`nom_fonction ;;`
à l'interprète comme vous venez de le faire pour `abs`.

**Q3.** Appliquez la fonction `max2` à quelques jeux d'essai et en particulier des jeux ne satisfaisant pas le profil de la fonction, par exemple, `(max2 'd' 'a')`.

Pour élaborer des jeux d'essai qui seront facilement réutilisables au cours des séances suivantes, nous vous suggérons l'utilisation de la directive `assert`.

**Q4.** Taper les expressions suivantes :

```
assert ((max2 4 7) = 7) ;;
assert ((max2 4 7) = 5) ;;
assert ((max2 '4' 7) = 7) ;;
```

Observez la réponse de l'interpréteur lorsque le test donne un résultat correct.

### 2.2.3 Maximum de trois entiers

On veut réaliser une fonction qui détermine le maximum de trois entiers distincts deux à deux. Vous allez étudier plusieurs réalisations possibles selon la manière de conduire l'analyse par cas et de la formuler. On en considère quatre différentes.

| SPÉCIFICATION 1 — maximum de 3 entiers | |
|---|---|
| PROFIL | $max3 : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ |
| SÉMANTIQUE | $(max3\ a\ b\ c)$ est le maximum des 3 nombres $a, b, c$ distincts deux à deux |



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

**Réalisation basée sur une analyse du résultat (3 cas)**

ALGORITHME 1

$$(max3\ a\ b\ c) = \begin{cases} a & \textbf{si} \quad a \geq b\ \textbf{et}\ a \geq c \\ b & \textbf{si} \quad b \geq a\ \textbf{et}\ b \geq c \\ c & \textbf{si} \quad c \geq a\ \textbf{et}\ c \geq b \end{cases}$$

**Q5.** Compléter la définition, la fournir à l'interpréteur OCaml et tester la fonction ainsi définie.

```
let max3_v1 ............................. =
  if a>=b && a>=c then a
  else if ........... then ........
       else (* .......*) ........
```

**Réalisation basée sur des analyses par cas imbriquées**

ALGORITHME 2

$$(max3\ a\ b\ c) = \begin{cases} \textbf{si}\ (a > b) & \begin{cases} \textbf{si} & a > c \quad retournera \\ \textbf{si} & c \geq a \quad retournerc \end{cases} \\ \textbf{si}\ (b \geq a) & \begin{cases} \textbf{si} & b > c \quad retournerb \\ \textbf{si} & c \geq b \quad retournerc \end{cases} \end{cases}$$

**Remarque** Cet algorithme par étude de cas imbriquée correspond à une implémentation au moyen de `if then else` imbriqués.

**Q6.** Compléter la définition, la fournir à l'interpréteur OCaml et tester la fonction ainsi définie.

```
let max3_v2 ..................... =
  if .........
  then
       if .......... then ..... else .....
  else
       if .......... then .....else .....
```

**Réalisation par composition de fonctions**

Pour calculer la somme de trois valeurs on peut se ramener à la somme de deux valeurs, en écrivant par exemple $a + (b + c)$. On veut définir de manière analogue une fonction *max*3 pour calculer le maximum de 3 entiers.

ALGORITHME 3

$(max3\ \text{a b c}) = \ldots\ (\text{a},\ \ldots\ (\text{b,c}))$

**Q7.** Compléter la définition, la fournir à l'interpréteur OCaml et tester la fonction ainsi définie.

```
let max3_v3 ....................... =
  ..............
```



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

**Réalisation en nommant un calcul intermédiare**

On utilise la construction OCaml `let in` afin de donner un nom au calcul du maximum des entiers $b$ et $c$. Ce procédé correspond à ce que font les mathématiciens lorsqu'ils écrivent :

> « ***Posons*** $m$ = *le maximum de* $b$ *et* $c$ ***alors*** *le maximum des trois entiers* $a$, $b$ *et* $c$ *est* $max2\ (a,\ m)$. *Ainsi nous pouvons utiliser* $m$ ***dans*** *la suite ...* »

ALGORITHME 4

$$(max3\ a\ b\ c) = \left(\begin{array}{l} \textbf{\textit{posons}} \\ \quad m = (max2\ b\ c) \\ \textbf{\textit{dans}} \\ \quad (max2\ a\ m) \end{array}\right)$$

La construction

***posons*** *var* = *expr* ***dans*** *une expression utilisant var*

existe en `OCaml` et s'écrit :

`let` *var* `=` *expr* `in` *une expression utilisant var*

**Q8.** Compléter la définition, la fournir à l'interpréteur OCaml et tester la fonction ainsi définie.

```
let max3_v4 ..................................... =
  let m = ................ in
     .....................
```

**Q9.** On peut reprendre la même idée mais sans utiliser la fonction `max2`.

```
let max3_v4prime ........................ =
  let m = if ........... then ..... else ..... in
    if a>m then ..... else .....
```

## 2.3 TP1 Nommer une expression

**Q1.** Pour chacune des expressions suivantes, observer les réactions de l'interprète OCaml. En déduire la règle de liaisons des noms.

```
let x=3 and y=4 in x+y ;;

let x=3 and y=x+4 in x+y ;;

let x=10 in
let x=3 and y=x+4 in
   x+y ;;

let x=10 in
(let x=3 and y=x+4 in x+y) + x ;;

let x=10 in
let x=3 and y=x+4 in x+y + x ;;

x ;;
```



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

## 2.4 TP2 Ordre d'évaluation d'une expression

**Q1.** Définir les constantes `a` et `b` :

```
let a : int = 10 ;;
let b : int = 0 ;;
```

puis évaluer les expressions :

```
(b <> 0) && (a mod b = 0) ;;
(a mod b = 0) && (b <> 0) ;;
```

Préciser la règle d'évaluation de l'opérateur `&&`.

**Q2.** Définir la fonction `monExpression` de la façon suivante :

```
let monExpression (a:int) (b:int) : bool =
    (b <> 0) && (a mod b = 0) ;;
```

Évaluer l'expression : `monExpression 10 0 ;;`. Cette deuxième expérience devrait confirmer la conclusion de la précédente.

**Q3.** Définir la fonction `monEt` de la façon suivante :

```
let monEt (x:bool) (y:bool) : bool =
    x && y ;;
```

Évaluer les expressions :

```
monEt true false ;;
monEt false true ;;
monEt (b <> 0) (a mod b = 0) ;;
```

Que pouvez-vous conclure concernant l'ordre d'évaluation d'une expression faisant appel à une fonction ?

## 2.5 TP2 Moyenne de deux entiers

### 2.5.1 Types numériques : entiers (`int`) et réels (`float`)

**Q1.** Observer le type des valeurs suivantes : `3.5 ;;` `3,5 ;;` Observer le résultat (type et valeur) de l'évaluation de l'expression `(4 + 3) / 2 ;;`. Conclure en donnant la spécification des opérateurs `(+)` et `(/)`.

**Q2.** Observer la réaction de l'interprète pour chacune des expressions suivantes :

```
(4 + 3) /. 2 ;; (4.0 + 3.0) /. 2 ;; (4.0 +. 3.0) /. 2 ;;
(4.0 +. 3.0) /. 2.0 ;;
```

Conclure en donnant la spécification des opérateurs `(+.)` et `(/.)`

### 2.5.2 Fonctions de conversion

Observer la réaction de l'interprète après avoir saisi l'implémentation de la fonction définie ci-dessous :



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | $moyenne : \mathbb{Z} \to \mathbb{Z} \to \mathbb{R}$ |
| SÉMANTIQUE | $(moyenne\ a\ b)$ est la moyenne de $a$ et $b$ |
| **RÉALISATION 1** | |
| ALGORITHME | la moyenne est le milieu du segment [a,b] |
| IMPLÉMENT. | `let moyenne (a:int) (b:int) : float = (a +. b) /. 2.0` |

Pour convertir un entier en réel, OCaml fournit la fonction `float_of_int`.

**Q3.** Deviner le profil de cette fonction et vérifier la réponse en donnant `float_of_int ;;` à l'interprète. Évaluer les expressions `float_of_int 3` et `float_of_int 3.2`.

**Q4.** Donner une ou plusieurs réalisations correctes de la fonction *moyenne* en respectant la spécification ci-dessus. Tester avec des jeux d'essais significatifs.

**Q5.** En s'inspirant du nom de la fonction `float_of_int`, deviner le nom de la fonction permettant de convertir un réel en un entier. En donner une spécification, puis la tester.

## 2.6 TP2 Moyenne olympique

La moyenne olympique est la moyenne obtenue après avoir enlevé le nombre qui a la valeur maximale et celui qui a la valeur minimale. On demande de définir une fonction qui calcule la moyenne olympique de quatre entiers strictement positifs.

Par exemple, la moyenne olympique des quatre nombres $10, 8, 12, 24$ est $11$ ; celle des nombres $12, 12, 12, 12$ est $12$.

| SPÉCIFICATION 1 — moyenne olympique | |
|---|---|
| PROFIL | $moyol : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{R}^+$ |
| SÉMANTIQUE | $(moyol\ a\ b\ c\ d)$ est la moyenne olympique des nombres $a, b, c, d$ |
| PROFIL | $min4 : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ |
| SÉMANTIQUE | $(min4\ a\ b\ c\ d)$ est le minimum des nombres $a, b, c, d$ |
| PROFIL | $max4 : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ |
| SÉMANTIQUE | $(max4\ a\ b\ c\ d)$ est le maximum des nombres $a, b, c, d$ |
| PROFIL | $max2 : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ |
| SÉMANTIQUE | $(max2\ x\ y)$ est le maximum des 2 nombres $x, y$ |
| PROFIL | $min2 : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ |
| SÉMANTIQUE | $(min2\ x\ y)$ est le minimum des 2 nombres $x, y$ |
| **RÉALISATION 1 — moyenne olympique** | |



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

| | | | |
|---|---|---|---|
| ALGORITHME | (*max*2 *x y*) | : | *milieu de x et y + la moitié de la distance entre x et y* |
| | *min*2 | : | *utilisation de max*2 |
| | *max*4 | : | *utilisation de max*2 |
| | *min*4 | : | *utilisation de min*2 |
| | *moyol* | : | *utilisation des fonctions prcdentes* |

**Q1.** Implémenter les fonctions *min*2, *max*2, *min*4, *max*4 puis *moyol*. Les réalisations de ces fonctions ayant beaucoup de points communs et pour ne pas tout ré-écrire, utiliser les possibilités de copié-collé de l'éditeur.

**Q2.** Tester les fonctions *min*2, *max*2, *min*4, *max*4 et *moyol* avec des jeux d'essai significatifs.

### 2.6.1 Trace de l'évaluation d'une fonction

Pour que la fonction de nom `moyol` soit tracée chaque fois qu'elle est évaluée, il faut appliquer la directive `#trace` au nom de la fonction : `#trace moyol ;;`

**Remarque.** Les directives OCaml commencent toutes par `#` ; il faut taper un `#` en plus de celui qui correspond à l'invite de l'interprète (prompt).

Inversement, pour ne plus tracer l'évaluation, il faut appliquer la commande `#untrace` : `#untrace moyol;;`. Ici encore, ne pas oublier le `#`.

**Q3.** Tracer *moyol* ainsi que toutes les fonctions intermédiaires utilisées par *moyol*, puis appliquer *moyol* à un jeu d'essai.

**Q4.** Indenter la trace fournie par l'interprète (en utilisant des marges adéquates) de manière à faire apparaître l'emboîtement des appels successifs.

## 2.7 TP3 Une date est-elle correcte ?

Cet exercice fait suite à un exercice de TD.

**Q1.**

a) Définir les types `jour` et `mois`.

b) Implémenter la version 2 de la fonction `estJourDansMois_2`.

c) Définir en OCaml une liste de jeux d'essai significatifs pour tester cette fonction, sous la forme `assert ((estJourDansMois_2 28 1) = true);;`

d) Que répond OCaml à `(estJourDansMois_2 18 13)`, `(estJourDansMois_2 0 4)` ? Comment interpréter ces résultats ?



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

**Q2.**

a) Donner une troisième implémentation basée sur l'algorithme suivant : ALGORITHME composition conditionnelle sous forme de filtrage. L'utilisation d'expressions conditionnelles est interdite.

b) Vérifier que `estJourDansMois_3` donne les mêmes résultats que `estJourDansMois_2` sur les jeux d'essais définis plus haut. Quid pour `18 13` et `0 4` ?

**Q3.**

a) Donner une nouvelle implémentation (n°4) du type `Mois` en utilisant un *type énuméré*, et donc une nouvelle implémentation `estJourDansMois_4`.

b) Vérifier que `estJourDansMois_4` donne les mêmes résultats sur la même liste de jeux d'essais que `estJourDansMois_2`.

## 2.8 TP3 Relations sur des intervalles d'entiers

Cet exercice fait suite à un exercice de TD.

Rappels : en OCaml, les opérateurs de comparaison sont notés `=`, `<>`, `<`, `<=`, `>`, `>=`. Par ailleurs, `true` et `false` dénotent les deux valeurs booléennes. La conjonction (et) est notée `&&`, la disjonction (ou) `||`, et la négation (non) `not`.

**Q1.** Observer le résultat (type et valeur) des expressions suivantes :

```
true ;; false ;; vrai ;;
true and true ;; true && true ;;
2 < 3 ;; 2 >= 3 ;; 2 >  = 3 ;; 2 <> 2 ;;
2<3<4 ;; 2<3 && 3<4 ;; 2=3=true ;; 2=(3=true) ;;
not (4 <= 2) ;; not 4 <= 2 ;;
```

**Q2.** Implémenter les expressions booléennes suivantes en OCaml, en parenthésant les opérations de diverses manières. En déduire les priorités des opérateurs logiques `&&` (et), `||` (ou), `not` (non).

- non vrai et faux
- vrai ou vrai et faux

**Q3.** Mettre en évidence la règle d'évaluation des opérateurs `&&` et `||` : le deuxième terme est-il toujours évalué, ou au contraire n'est-il évalué que si nécessaire ? Pour cela observer la réaction de l'interprète avec les expressions suivantes :

```
10 mod 0 ;; 10 mod 5 ;;

let essaiEt1 (a:int) (b:int): bool =
 (b <> 0) && (a mod b = 0)
;;
essaiEt1 10 5 ;; essaiEt1 10 3 ;; essaiEt1 10 0 ;;

let monEt (x:bool) (y:bool): bool =
 x && y
;;
let essaiEt2 (a:int) (b:int): bool =
 monEt (b <> 0) (a mod b = 0)
;;
essaiEt2 10 0 ;;
```

Que calculent les fonctions `essaiEt1` et `essaiEt2` ?

### 2.8.1 n-uplets

**Rappels.** un n-uplet de valeurs est un vecteur à n composantes, séparées par une virgule. Le type d'un n-uplet est le produit cartésien ($\times$) des types de chacune des composantes.

**Q4.** Observez le résultat des expressions OCaml suivantes :

```
(10,20,30) ;; ((10,20),30) ;; 20,30.0 ;;
4,3 /. 2,0 ;; 4,3. /. 2,0 ;; 4,3. /. 2.,0 ;;
(4,0) /. (2,0) ;;
```

**Remarque.** Les virgules étant utilisées pour séparer les éléments d'un vecteur, un symbole différent de la virgule est utilisé pour séparer la partie entière de la partie décimale d'un réel, en l'occurrence un point. L'utilisation de parenthèses peut améliorer la lisibilité, comme par exemple : `(4, (3. /. 2.), 0) ;;`

### 2.8.2 Points et intervalles

**Q5.** Compléter l'implémentation de l'ensemble *intervalle* :

```
type intervalle = (* { (b_i,b_s) ∈ ℤ² tels que b_i ≤ b_s } *)
                  ..... ;;
```

**Q6.** Donner un ensemble significatif de jeux d'essais permettant de tester les fonctions *precede*, *dans* et *suit*. Nommer les jeux d'essai en utilisant la construction `let`, par exemple :

```
let interv_1 : intervalle = (-30, 50)
and x1 : int = -40 ;;
```

puis `assert (precede x1 interv_1) ;;`



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

**Q7.** Implémenter la fonction *precede*. Remarquer comment la définition de *intervalle* facilite la compréhension du profil de la fonction.

**Q8.** Observer l'évaluation des expressions :

```
precede 3 4 5 ;; precede (3,4,5) ;; precede 3 (4,5) ;;
```

Tester les trois fonctions avec les jeux d'essai définis plus haut.

### 2.8.3 Intervalles, couples d'intervalles

**Q9.** Donner un ensemble significatif de jeux d'essai permettant de tester les fonctions *coteAcote* et *chevauche* (voir énoncé exercice dans le polycopié de TD). Pour cela, fixer la valeur d'un intervalle I1, par exemple `let cst_I1 : intervalle = (10,20)`, puis énumérer diverses valeurs significatives d'intervalles I2, I3, etc, en plaçant les segments correspondants par rapport au segment correspondant à I1. Nommer chacun des jeux d'essai en utilisant la construction `let cst_I... : intervalle = ...`.

**Q10.** Implémenter les deux fonctions et les tester.

## 2.9 TP3 Somme des chiffres d'un nombre

On veut définir une fonction *sc* qui à un entier compris entre 0 et 9999 associe la somme des chiffres qui le composent. Voici sa spécification :

| SPÉCIFICATION 1 — somme des chiffres | |
|---|---|
| PROFIL | $sc : 0, \ldots, 9999 \to \mathbb{Z}$ |
| SÉMANTIQUE | $sc(n)$ est la somme des chiffres de la représentation de $n$ en base 10. |
| EX. ET PROP. | (i) $sc(9639) = 27$ |

**Idée.** Le chiffre des unités d'un nombre peut être déterminé au moyen d'une division par 10, de la partie entière d'un nombre réel $r$ (notée $\lfloor r \rfloor$) et du reste de la division entière (opérateur modulo, noté mod) :

$$n = \underbrace{\left\lfloor \frac{n}{10} \right\rfloor}_{\text{nombre de dizaines dans } n} + \underbrace{n \bmod 10}_{\text{chiffre des units de } n}$$

| RÉALISATION 1 — somme des chiffres |
|---|



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

ALGORITHME Étant donné l'entier $n$, notons $m$ le chiffre des milliers, $c$ le chiffre des centaines, $d$ le chiffre des dizaines, $u$ le chiffre des unités, et *mil* le nombre de milliers, *cent* le nombre de centaines, *diz* le nombre de dizaines.

$$\begin{array}{rclrcl} diz & = & \left\lfloor \frac{n}{10} \right\rfloor & u & = & n \ mod\ 10 \\ cent & = & \left\lfloor \frac{diz}{10} \right\rfloor & d & = & diz \ mod\ 10 \\ mil & = & \left\lfloor \frac{cent}{10} \right\rfloor & c & = & cent \ mod\ 10 \\ & & & m & = & mil \ mod\ 10 \end{array}$$

NB : le nombre de dizaine de milliers, nul puisque $n < 9999$, n'est pas calculé.

Pour faciliter la réalisation de la fonction *sc*, on introduit une fonction *div* qui à deux nombres associe le quotient et le reste de leur division, dont voici une spécification :

| SPÉCIFICATION 2 — division euclidienne | |
|---|---|
| PROFIL | div : $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ |
| SÉMANTIQUE | posons $(q, r) = (div\ n\ d)$ ; $q$ est le quotient et $r$ le reste de la division de $n$ par $d$, de sorte que $n = q \times d + r$ avec $r < d$. |

**Q1.** Réaliser la fonction `div`.

Pour pouvoir exploiter le résultat de la fonction *div* – de manière plus générale, pour utiliser les éléments d'un n-uplet (ici n=2) résultat de l'application d'une fonction de nom *f* – il faut les *identifier* (leur donner un nom). En OCaml, cette identification est opérée par la construction `let ... in`. Par exemple, dans le cas de couples, la forme de la construction est :

```
let (a,b) = f(...) in
  expression utilisant a et b
```

**Q2.** Compléter la réalisation de la fonction *sc* ci-dessous.

ALGORITHME utilisation de la fonction *div* et de la construction `let ... in`.

IMPLÉMENT.
```
/!\ conditions utilisation sc(n): n ∈ ..........
let sc ............... =                                    1
    ........................................               2
    ........................................               3
    ........................................               4
    ........................................               5
```

On peut suivre l'exécution des fonctions `div` et `sc` lors de l'évaluation d'une expression calculant la somme des chiffres d'un entier. Pour cela, il faut tracer les fonctions *div* et *sc*.

**Q3.** Taper les commandes `#trace div ;;` et `#trace sc ;;` puis observer les appels successifs de *div* lors de l'application de *sc* à un jeu d'essai (par exemple, `sc 2345 ;;`). Pour plus de lisibilité, indenter la trace fournie par l'interprète de manière à faire apparaître les appels successifs. Inversement, pour ne plus tracer l'évaluation, il faut appliquer la commande `#untrace` : par exemple `#untrace div ;;`.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

## 2.10 TP4 Permutation ordonnée d'un couple d'entiers

### 2.10.1 Permutation ordonnée d'un couple d'entiers

On veut construire la permutation ordonnée d'un couple d'entiers donné : par exemple, $(3,4)$ est la permutation ordonnée de $(4,3)$. On introduit ainsi une fonction nommée *poCoupleE* :

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | $poCoupleE : \mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}\times\mathbb{Z}$ |
| SÉMANTIQUE | Posons $(x',y') = poCoupleE(x,y)$ ; $(x',y')$ est la permutation ordonnée de $(x,y)$, définie comme suit : $(x',y') = \begin{cases}(x,y) & \text{si } x \leq y,\\ (y,x) & \text{sinon.}\end{cases}$ |
| EX. ET PROP. | (i) $poCoupleE(3,4) = \ldots\ldots$<br>(ii) $poCoupleE(4,3) = \ldots\ldots$ |

**Q1.** Implémenter la fonction *poCoupleE*.

**Q2.** Observer la réaction de l'interprète lors de l'évaluation de l'expression suivante :

```
poCoupleE (33.3,14.5) ;;
```

### 2.10.2 Surcharge des opérateurs de comparaison

**Q3.** Observer le résultat (type et valeur) des expressions suivantes :

```
2 < 3 ;;                                    1
2.0 < 3.0 ;;                                2
```

Le même symbole $<$ est utilisé pour dénoter deux opérations de comparaison différentes l'une portant sur des entiers, l'autre sur des réels, alors que pour les opérations arithmétiques – l'addition par exemple – deux symboles différents sont utilisés. On dit que le symbole $<$ est *surchargé*. Les deux opérandes doivent avoir le même type et la signification précise du symbole en est déduite.

**Q4.** Vérifier sur plusieurs exemples que tous les opérateurs de comparaison sont surchargés (égalité, relation d'ordre), la seule règle étant que les opérandes soient de même type.

**Q5.** Réaliser la fonction *poCoupleR* pour construire la permutation ordonnée d'un couple de réels. Pour cela, exploiter la surcharge des opérateurs de comparaison : par rapport à la fonction *poCoupleE*, seuls le nom de la fonction et son profil doivent être changés.

### 2.10.3 Fonctions génériques : paramètres dont le type est générique

**Q6.** Observer la réaction du système pour la fonction suivante :

```
let poCouple (x,y: 'un_type*'un_type) : 'un_type*'un_type =      1
     if x <= y then x,y else y,x                                 2
```

Dans le profil de la fonction, les types sont indiqués par des identificateurs (autres que ceux des types définis par ailleurs), précédés par un (un seul) accent aigu (`'`). Ceci signifie que la spécification proposée ne dépend pas du type des paramètres : on parle de *fonctions génériques* ou de *polymorphisme*.

Souvent, des lettres minuscules (`'a`, `'b`, ...) sont utilisées à la place d'identificateurs longs (`'un_type`). Dans ce cas, attention à ne pas confondre `'a` (pour nommer un type polymorphe), `a` (pour nommer un paramètre) et `'a'` (la constante de type caractère première lettre de l'alphabet en minuscule).

Dans les spécifications, on note parfois $\alpha$ au lieu de `'a` ($\beta$ au lieu de `'b`, ...) :

| SPÉCIFICATION 2 | |
|---|---|
| PROFIL | poCouple : $\alpha \times \alpha \to \alpha \times \alpha$ |

**Q7.** Observer la réaction du système pour les expressions suivantes :

```
poCouple (3, 2) ;; poCouple (33.3, 14.5) ;;
poCouple (3, 14.5) ;;
```

1

**Q8.** Deviner le profil des opérateurs $<$, $\leq$, $>$, et $\geq$, puis vérifiez vos conjectures à l'aide de OCaml en tapant `(<)` ;;, `(<=)`, ...

**Q9.** Appliquer la fonction `poCouple` à des couples de caractères. Observer que la relation d'ordre sur les caractères, définie par OCaml, est celle sur les codes des caractères (voir exercice «Codage des caractères» ci-après). Ceci est vrai pour tous les opérateurs de comparaison ($<$, $\leq$, $>$, $\geq$).

## 2.11 TP4 Type Durée et opérations associées

### 2.11.1 Définition du type *duree* et des opérations associées

**Type abstrait de données.** Lorsqu'on définit un type de donnée, il est intéressant de définir les opérations associées à ce type (constructeurs, sélecteurs, opérations de calculs, ...). On dit qu'on a alors défini un *type abstrait de données* (ou TAD).

L'intérêt est qu'un programmeur pourra réutiliser le TAD et les opérations sans avoir besoin de connaître comment sont représentées les données ni comment sont réalisées les opérations.

Dans la suite de ce cours, nous utiliserons par exemple le type `list` d'OCaml. C'est un TAD ; nous l'utiliserons ainsi que les opérations associées sans qu'il soit nécessaire de savoir comment les `list` sont représentées dans l'interpréteur.

L'objectif de cet exercice est de définir le TAD *duree*, c'est-à-dire le type et les opérations associées.

On considère ici une durée exprimées en jour, heure, minute, seconde. On choisit de la représenter par un vecteur à 4 coordonnées $(j,h,m,s)$ (on dit aussi *quadruplet* ou *4-uplet*) où $j$ représente un nombre de jours, $h$ un nombre d'heures inférieur à une journée, $m$ un nombre de minutes inférieur à une heure et $s$ un nombre de secondes inférieur à une minute.

**Définition mathématique du type *Dure* et des fonctions associées**

***déf*** $jour = \mathbb{N}$

***déf*** $heure = \{0,\dots,23\}$

***déf*** $minute = \{0,\dots,59\}$

***déf*** $seconde = \{0,\dots,59\}$

***déf*** $duree = jour \times heure \times minute \times seconde$



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

**Spécifications des fonctions sur les durées**

| SPÉCIFICATION 1 — constructeurs |
|---|
| PROFIL *sec_en_duree* : $\mathbb{Z} \to$ *duree*<br>construit une donnée de type *duree* à partir d'un nombre de secondes<br>PROFIL *vec_en_duree* : $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to$ *duree*<br>construit une donnée de type *duree* à partir d'un 4-uplets représentant un nombre de jours, d'heures, de minutes et de secondes (sans limite de taille) |

| SPÉCIFICATION 2 — sélecteurs |
|---|
| PROFIL *jour* : *duree* $\to \mathbb{Z}$<br>sélectionne la partie jour d'une durée PROFIL *heure* : *duree* $\to \mathbb{Z}$<br>sélectionne la partie heure d'une durée PROFIL *minute* : *duree* $\to \mathbb{Z}$<br>sélectionne la partie minute d'une durée PROFIL *seconde* : *duree* $\to \mathbb{Z}$<br>sélectionne la partie seconde d'une durée |

| SPÉCIFICATION 3 — conversion |
|---|
| PROFIL *duree_en_sec* : *duree* $\to \mathbb{Z}$<br>convertit une durée en un nombre de secondes |

| SPÉCIFICATION 4 — opérations |
|---|
| PROFIL *som_duree* : *duree* $\to$ *duree* $\to$ *duree*<br>effectue la somme de deux durées |

| SPÉCIFICATION 5 — prédicats |
|---|
| PROFIL *eg_duree* : *duree* $\to$ *duree* $\to \mathbb{B}$<br>teste l'égalité de deux durées PROFIL *inf_duree* : *duree* $\to$ *duree* $\to \mathbb{B}$<br>teste la relation inférieur strict sur les durées |

**Réalisation informatique du type *Dure* et des fonctions associées**

**Q1.** Compléter les définitions de type ci-dessous :

```
type jour    = ..................................... ;; (* {0,...,31} *)
type heure   = ..................................... ;; (* {0,...,23} *)
type de0a59  = ..................................... ;; (* {0,...,59} *)
type minute  = ..................................... ;;
type seconde = ..................................... ;;
type duree   = ..................................... ;;
```

**Q1.** Réaliser la fonction *sec_en_duree*.

| RÉALISATION 5 |
|---|
| ALGORITHME utilisation de *div* réalisée dans un exercice précédent.<br>IMPLÉMENT. ... |

On considère la fonction suivante :



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

SPÉCIFICATION 6

PROFIL nb_total_sec : .....

*nb_total_sec*$(j,h,m,s)$ est le nombre total de secondes que représentent
$j$ jours + $h$ heures + $m$ minutes + $s$ secondes EX. ET PROP.

(i) *nb_total_sec*$(1,0,0,0) =$ .....

(ii) *nb_total_sec*$(0,1,0,0) =$ ....

(iii) *nb_total_sec*$(.,.,1,..) = 90$

(iv) *nb_total_sec*$(.,.,0,...) = 120$

**Q2.** Compléter la spécification de *nb_total_sec* et l'implémenter.

**Q3.** En déduire une réalisation de *vec_en_duree* :

RÉALISATION 6

ALGORITHME On calcule le nombre de secondes que représente le vecteur $(j,h,m,s)$ puis on utilise la fonction *sec_en_duree* pour construire une donnée de type *duree*.

IMPLÉMENT. ...

**Q4.** Implémenter les sélecteurs.

**Q5.** Réaliser *duree_en_sec* :

RÉALISATION 6

ALGORITHME Réutilisation de *nb_total_sec* et des sélecteurs.

IMPLÉMENT. ...

**Q6.** Réaliser *som_duree* en utilisant les deux algorithmes suivants :

RÉALISATION 6 — *som_duree* (OPÉRATION)

ALGORITHME 1) On réutilise les fonction *duree_en_sec* et *sec_en_duree*.

IMPLÉMENT. [1] ...

ALGORITHME 2) On fait l'addition des vecteurs composante par composante en tenant compte des retenues.

IMPLÉMENT. [2] ...

**Q7.** Réaliser *eg_duree* en utilisant les quatre algorithmes suivants :

RÉALISATION 6

ALGORITHME 1) en utilisant la fonction *duree_en_sec*

IMPLÉMENT. ...

ALGORITHME 2) en décomposant les durées en vecteurs $(j,h,m,s)$ avec des `let`

IMPLÉMENT. ...

ALGORITHME 3) en comparant les vecteurs composante par composante



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

IMPLÉMENT. ...
ALGORITHME 4) en utilisant les sélecteurs
IMPLÉMENT. ...

**Q8.** Implémenter *inf_duree* en utilisant les deux algorithmes suivants :

RÉALISATION 6
ALGORITHME 1) en utilisant la fonction *duree_en_sec*
IMPLÉMENT. ...
ALGORITHME 2) en utilisant des expressions conditionnelles imbriquées
IMPLÉMENT. ...

## 2.12 TP4 Codage des caractères

### 2.12.1 Le code ASCII d'un caractère

La fonction `int_of_char` associe à tout caractère l'entier qui lui est associé dans le code ASCII qui sert à représenter le caractère en machine.

**Q1.** En utilisant cette fonction, observez les codes des chiffres, des lettres majuscules et des lettres minuscules, entre autres sur les expressions suivantes : `int_of_char(8);;` `int_of_char('8') ;;`

OCaml offre également une fonction nommée `char_of_int` la fonction qui permet de retrouver un caractère à partir de son code ASCII.

**Q2.** L'appliquer sur des exemples simples : entiers négatifs, entiers positifs. Sachant que la borne supérieure est de la forme $2^k - 1$, trouver le domaine de définition de la fonction. Quelle est la valeur de $k$ ?

### 2.12.2 Valeur entière associée à l'écriture en base 10 d'un entier

***déf*** $chiffre = \{'0', \ldots, '9'\}$
***déf*** $base10 = \{0, \ldots, 9\}$

**Q3.** Donner le profil et la sémantique d'une fonction $chiffreVbase10$, qui associe un élement de $base10$ à un $chiffre$.

**Q4.** Donner le profil et la sémantique d'une fonction $base10Vchiffre$, qui associe un $chiffre$ à un élement de $base10$.

**Q5.** Déduire des deux questions précedentes une manière de réaliser les fonctions $chiffreVbase10$ et $base10Vchiffre$, sans utiliser de tests sur l'écriture de l'entier.

INDICATION *Observer les expressions suivantes :*

```
int_of_char('8') - int_of_char('0') ;;
char_of_int(8 + int_of_char('0')) ;;
```

**Q6.** Implémenter les types $chiffre$ et $base10$, puis les fonctions $chiffreVbase10$ et $base10Vchiffre$.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

## 2.13 TP4 Numération en base 16

### 2.13.1 Valeur entière associée à un chiffre hexadécimal

Les chiffres hexadécimaux sont les symboles élémentaires utilisés pour noter les nombres dans la numération par position en base 16 : on utilise les chiffres arabes $0, \ldots, 9$ et les 6 premières lettres de l'alphabet $A, \ldots, F$.

On définit les ensembles suivants :

$$carhex \stackrel{def}{=} \{'0', \ldots, '9'\} \cup \{'A', \ldots, 'F'\}$$
$$base16 \stackrel{def}{=} \{0, \ldots, 15\}$$

**Q1.** Donner le profil et la sémantique d'une fonction *carhexVbase*16, qui associe un élement de *base*16 à un *carhex*.

**Q2.** Donner une implémentation naïve des types *carhex* et *base*16 sous forme respectivement d'un caractère et d'un entier, avec les restrictions convenables.

**Q3.** Réaliser cette fonction en veillant à ne pas réécrire du code déjà écrit. On peut réutiliser la fonction de l'exercice 2.12 pour les chiffres entre 0 et 9, et le codage de `'A'` en ASCII pour calculer le décalage des chiffres entre `'A'` et `'F'`.

**Q4.** Refaire la question précédente en implémentant *carhex* et *base*16 à l'aide de constructeurs et d'un type somme, et tester.

## 2.14 TP4 Chiffres d'un entier en base 16

On considère l'ensemble suivant :

$$hexa4 \stackrel{def}{=} \{0, \ldots, 16^4 - 1\}$$

**Q1.** Définir le type `hexa4`.

Dans cet exercice nous nous restreignons à des entiers qui peuvent être codés sur 4 caractères hexadécimaux.

**Q2.** Définir le type `rep_hexa4` représentant les quadruplets de caractères hexadécimaux *carhex* à l'aide d'un type produit.

**Q3.** Donner le profil, la sémantique et un exemple d'une fonction *ecriture_hex*, qui convertit un nombre de *hexa*4 en un quadruplet de caractères hexadécimaux.

**Q4.** Spécifier et réaliser la fonction *base*16*Vcarhex*, inverse de la fonction *carhexVbase*16 étudiée dans l'exercice 2.13.1 «Numération en base 16». On prendra soin de ne pas réécrire du code déjà écrit.

**Q5.** En se basant sur la structure générale de la fonction `sc` de l'exercice 2.9 «Somme des chiffres d'un nombre», réaliser la fonction *ecriture_hex*.

**Q6.** Tester la fonction *ecriture_hex*.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

## Deuxième partie

# DÉFINITIONS RÉCURSIVES

# Chapitre 3

# Fonctions recursives sur les entiers

**Sommaire**



Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TD$n$ ou TP$n$ , où $n$ est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n$^o$ $n$ sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2$^e$ créneau de TP (non supervisé).

## 3.1 TP5 Factorielle

On considère la factorielle, spécifiée ainsi :

**SPÉCIFICATION 1**

PROFIL $fact : \mathbb{N} \to \mathbb{N}^*$

SÉMANTIQUE Soit $n \in \mathbb{N}^*$, $fact(n)$ est l'entier $n \times (n-1) \times \cdots \times 2 \times 1$ ; par convention $fact(0) = 1$.

**Q1.** Compléter la réalisation de $fact$ ci-dessous :

**RÉALISATION 1**

- Équations de récurrence :
  (1) $fact(0) = 1$
  (2) $fact(p+1) = (p+1) \times fact(p)$

ALGORITHME analyse par cas par filtrage

IMPLÉMENT.
```
let rec                                    1
⋮
```

Noter *le changement de variable* lors du passage de l'équation (2) à l'implémentation.

**Q2.** Tester `fact` sur des données respectant les contraintes. Tracer l'évaluation (directive `#trace fact`), et indenter la trace obtenue de façon à faire apparaître l'enchaînement des appels récursifs ; `#untrace fact` permet de supprimer le traçage de la fonction.

**Q3.** Tester `fact` sur des données ne respectant les contraintes comme par exemple `fact (-1) ;;`. Justifier le phénomène observé. Tracer à nouveau pour valider la justification.

Que se passe-t-=il quand on oublie le `rec` ?

**Q4.** Implémenter une fonction appelée `fact2` identique à `fact` en la nommant avec `let fact2 ...` au lieu de `let rec fact2 ...` (attention à bien utiliser `fact2` pour l'appel récursif). Observer le message de l'interprète.

Contrairement aux équations récursives, l'ordre des motifs dans un filtrage est important.

**Q5.** Saisir et compléter l'implémentation suivante dans laquelle on inserse les deux cas du filtrage. Observer les messages de l'interprète lors de la définition et lors du test de cette fonction. Expliquer pourquoi on obtient ces message quelle que soit la donnée.

```
let rec fact3 ... =
  match n with
    | n  -> ...
    | 0  -> ...
```

**Q6.** Donner une autre implémentation `fact3` de la factorielle, par composition conditionnelle.

## 3.2 TP5 Somme d'entiers bâton

L'exercice de TD « additions des entiers de Peano » explore une définition des entiers grâce aux constructeurs $Z$ et $S$. Cet exercice propose une autre représentation des entiers, basée sur la *numération unaire*, qui date de 35000 av. J.-C. Le principe est évident : une unité correspond à un caillou, ou à une entaille dans un os, ou à un bâton.

On définit ainsi :

```
type baton = string (* dont les caractères sont uniquement des | *)
```

L'entier 0 est alors représenté par `""`, l'entier 1 par `"|"`, l'entier 2 par `"||"`, etc.

**Q1.** Implémenter l'addition sur les bâtons.

### 3.2.1 Fonctions de conversion

Au delà de |||| ou ||||| bâtons, il n'est pas évident de dire au premier coup d'œil combien il y en a. Une solution (il y en a beaucoup d'autres[1] ) consiste à regrouper les bâtons et à leur donner des noms, ce qui revient – ici – à définir des fonctions de conversion :

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | $natVbaton : \mathbb{N} \to baton$ |
| SÉMANTIQUE | $natVbaton(n)$ est l'entier bâton correspondant à $n$. |

[1] https://fr.wikipedia.org/wiki/Système_de_numération

| | |
|---|---|
| EX. ET PROP. | (i) $natVbaton(5) =$ "\|\|\|\|\|" |
| PROFIL | $batonVnat : baton \to \mathbb{N}$ |
| SÉMANTIQUE | $batonVnat(b)$ est l'entier naturel correspondant à $b$. |
| EX. ET PROP. | (i) $batonVnat($"\|\|\|\|\|"$) = 5$ |

**Q2.** Implémenter $\mathbb{N}$ par un type synonyme.

**Q3.** Implémenter *natVbaton* grâce à la fonction `make` du module `String` de la librairie standard d'OCaml :

| SPÉCIFICATION 2 | |
|---|---|
| PROFIL | `String.make` : $\mathbb{N} \to$ `char` $\to$ `string` |
| SÉMANTIQUE | `(String.make n c)` retourne une chaîne formée de `n` fois le caractère `c`. |

**Q4.** Implémenter récursivement *natVbaton* par composition conditionnelle (sans utiliser `String.make`).

**Q5.** Implémenter récursivement *batonVnat* par composition conditionnelle, en utilisant la fonction `sub` du module `String` de la librairie standard d'OCaml :

| SPÉCIFICATION 3 | |
|---|---|
| PROFIL | `String.sub` : `string` $\to \mathbb{N} \to \mathbb{N} \to$ `string` |
| SÉMANTIQUE | `(String.sub s i l)` retourne la sous-chaîne de `s` à partir du caractère d'index `i` de longueur `l`. Les index commencent à 0. |
| EX. ET PROP. | (i) `(String.sub "OCaml" 1 3) = "Cam"` |

Pour obtenir la longueur d'une chaîne, on utilisera `String.length`.

### 3.2.2 Multiplication d'entiers bâtons

**Q6.** Implémenter récursivement une fonction `mult` opérant la multiplication de bâtons par composition conditionnelle.

Pour tester `mult`, on pourra utiliser les fonctions de conversion du paragraphe précédent de la façon suivante :

```
let mult_nat (n1:nat) (n2:nat) : nat =                                  1
  batonVnat(mult (natVbaton n1) (natVbaton n2))                         2
```

**Q7.** Donner une autre implémentation de `mult`, non récursive.

## 3.3 TP5 Quotient et reste de la division entière

Pour calculer le quotient et le reste de la division entière, OCaml fournit les deux opérateurs `/` et *mod* spécifiés comme suit :



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | `(/), (mod)` : $\mathbb{N} \to \mathbb{N}^* \to \mathbb{N}$ |
| SÉMANTIQUE | $a$ / $b$ : quotient de la division entière de $a$ par $b$<br>$a$ *mod* $b$ : reste de la division entière de $a$ par $b$ |

Le but de cet exercice est de réaliser `/` et `mod`, que l'on appellera *quotient* et *reste* pour les distinguer des opérateurs prédéfinis par OCaml.

**Q1.** Implémenter $\mathbb{N}$, sans oublier la contrainte appropriée sur le type.

### 3.3.1 Quotient et reste

**Q2.** Compléter la réalisation de la fonction *quotient* ci-dessous :

| RÉALISATION 1 | |
|---|---|
| ALGORITHME | On procède par soustractions successives ; il faut donc exprimer $(quotient\ a\ b)$ en termes de $(quotient\ (a-b)\ b)$ |

- Équations de récurrence :
  (1) $(quotient\ a\ b) = \ldots\ si\ \ldots$
  (2) $(quotient\ a\ b) = \ldots\ sinon$

IMPLÉMENT.

⋮

**Q3.**

a) Tester *quotient* sur un jeu d'essais pertinent.

b) Observer la réaction de l'interprète quand on ne respecte pas les contraintes sur $b$. Expliquer.

c) Observer la trace de l'appel $(quotient\ 17\ 5)$. Structurer cette trace en l'indentant de façon à mettre en évidence l'emboîtement des appels récursifs.

**Q4.** En utilisant le même algorithme que pour *quotient*, réaliser (équations récursives puis implémentation) et tester la fonction *reste*. Ne pas hésiter à tracer si besoin.

### 3.3.2 Réalisation d'une fonction à valeur n-uplet

On souhaite définir une fonction calculant le couple formé du quotient et du reste de la division de deux entiers.

**Q5.** Compléter sa spécification ci-dessous :

| SPÉCIFICATION 2 | |
|---|---|
| PROFIL | $qr_1$ : ..... |
| SÉMANTIQUE | *Posons* $\ldots = (qr_1\ a\ b)$ ;<br>.....<br>..... |
| EX. ET PROP. | (i) $(qr_1\ 17\ 5) = \ldots$ |



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

**Q6.** Donner une implémentation non récursive de $qr_1$.

**Q7.** Observer la trace de $(qr_1\ 17\ 5)$ ; il faut aussi tracer les fonctions utilisées par $qr_1$. Comparer les listes des appels récursifs de ces deux fonctions. Qu'en conclure ?

**Q8.** En s'inspirant des réalisations de *quotient* et de *reste*, implémenter une fonction récursive $qr_2$ qui effectue *en même temps* le calcul du quotient et du reste.

> **Indication** Utiliser la construction `let` pour nommer les composantes du résultat de l'appel récursif.

**Q9.** Vérifier l'abscence de calculs redondants sur la trace de $(qr_2\ 17\ 5)$



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

# Chapitre 4

# Fonctions récursives sur les séquences

**Sommaire**



Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TD$n$ ou TP$n$ , où $n$ est un entier entre 1 et 12 correspondant au numéro de séance de TD ou TP **à partir duquel** l'énoncé peut être abordé. En pratique, un exercice n$^o$ $n$ sera peut-être traité en séance $n+1$ ou $n+2$, et les exercices ne sont pas forcément traités dans l'ordre du polycopié.

Il y a plus d'énoncés que de séances ; ne pas hésiter à s'entraîner sur les énoncés qui n'auront pas été abordés en séance, notamment pendant le 2$^e$ créneau de TP (non supervisé).

## 4.1 TP6 Flux de circulation

Quelques fonctions classiques *à connaître* sont illustrées ici, à travers l'étude du flux de circulation sur une voie routière.

L'observation a lieu en un point précis de cette voie et fournit les « flux journaliers » d'une période, c'est-à-dire pour chaque jour de la période le nombre de véhicules qui sont passés ce jour-là.
On dispose ainsi de « relevés d'observation » sous forme de séquences de flux : dans un relevé, chaque entier correspond à un flux journalier pour un jour de la période considérée. L'ordre des éléments dans le relevé correspond à l'ordre chronologique sur la période d'observation.

On définit donc les ensembles suivants :

$$flux \stackrel{def}{=} \mathbb{N} \cup \{-1\}$$
$$releve \stackrel{def}{=} \{V\} \cup \{C(pr, fin) \text{ tel que } pr \in flux, fin \in releve\}$$

Un flux journalier de $-1$ indique une panne de l'appareil de comptage de véhicules ; $V$ est le relevé vide ; $C$ permet d'ajouter un flux à un relevé.
Soit $r$ un relevé non vide ; le ième élément de $r$ est le nombre de véhicules observés pendant le ième jour de la période associée à $r$.




[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

On ne cherchera pas à définir la longueur de la période d'observation, qui peut donc être différente selon le relevé étudié.

**Q1.** Implémenter les ensembles *flux* et *releve* en OCaml.

### 4.1.1 Statistiques sur les relevés

#### Nombre de jours sans véhicules

On souhaite définir une fonction *nbj_sans* déterminant le nombre de jours pendant lesquels aucun véhicule n'est passé devant le point d'observation :

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | *nbj_sans* : ..... |
| SÉMANTIQUE | *nbj_sans*$(r)$ est le nombre de jours à flux nul dans $r$. |

**Q2.** Compléter la spécification de la fonction *nbj_sans* ci-dessus.

**Q3.** Définir en OCaml trois relevés simulant trois cas différents :

- $R_1$ : cas où il n'y a pas de jours à flux nul,
- $R_2$ : cas où il y a un jour à flux nul,
- $R_3$ : cas où il y a plusieurs (par exemple 4) jours à flux nul.

NB : $R_1$, $R_2$ et $R_3$ peuvent être déclinées de plusieurs façons, par exemple en faisant varier la période d'observation. Ne pas hésiter à compléter ce jeu de de tests.

**Q4.** Implémenter *nbj_sans* récursivement.

#### Généralisation

On souhaite maintenant connaître le nombre de jours pendant lesquels $x$ véhicules sont passés devant le point d'observation.

**Q5.** Spécifier une fonction *nbj_avec* qui, étant donnés un entier naturel $x$ et un relevé $r$, donne le nombre de jours où le flux, dans $r$, est égal à $x$.

**Q6.** Donner des équations récursives définissant *nbj_avec*.

**Q7.** Évaluer l'implémentation ci-dessous, observer la réponse de l'interprète et analyser le message d'erreur.

```
let rec nbj_avec (x:flux) (r:releve) : int (* >= 0 *) =
  match x, r with
  | _, V -> 0
  | x, C(x,fin) -> 1 + nbj_avec x fin
  | x, C(pr,fin) -> (* on a donc ici x <> pr *) nbj_avec x fin
```

En OCaml, les motifs doivent être *linéaires*, ce qui signifie – ici – ne comporter aucune variable apparaissant plus d'une fois. Le motif `x, C(x,fin)` de la ligne 4 est donc interdit.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

**Q8.** Corriger l'implémentation ci-dessus ; on prendra soin de ne pas filtrer inutilement.

**Q9.** En déduire une implémentation non récursive d'une fonction *nbj_panne* qui donne le nombre de jours où les appareils de comptage ont été en panne.

### 4.1.2 Assertions et tests unitaires

Soit une expression booléenne dont on veut assurer qu'elle soit toujours vérifiée ; par exemple : $fact(4) = 24$, où $fact$ est une fonction implémentant la factorielle. En OCaml, on écrit `assert(fact 4 = 24).`

La construction `assert(expr)`, où $expr \in \mathbb{B}$, implémente la vérification d'assertions. OCaml évalue *expr* ; si le résultat est `true`, la constante `()` est renvoyée ; sinon, l'erreur `Assert_failure` est signalée :

```
# assert (fact 4 = 42) ;;
Exception: Assert_failure ("//toplevel//", 1, 0).
# assert (fact 4 = 24) ;;
- : unit = ()
```

> Le type OCaml `unit` est particulier : il ne contient qu'un élément, noté `()`. Ce type est utilisé pour les procédures, c'est-à-dire les « fonctions qui ne renvoient pas de résultat ». Remarquons l'abus de langage ; les procédures renvoient toujours une seule et même valeur : la constante `()`. Par exemple :
>
> ```
> # print_string ;;
> - : string -> unit = <fun>
> # print_string "inf2X1\n" ;;
> inf2X1
> - : unit = ()
> ```
>
> Le type `unit` est également utilisé pour les fonctions (ou procédures) ne prenant pas d'argument, d'où la notation `f()`. Par exemple :
>
> ```
> # Sys.time ;;
> - : unit -> float = <fun>
> # Sys.time() ;;
> - : float = 0.493159
> ```

Les assertions sont bien adaptées à la rédaction de tests unitaires [1], et aux développements itératifs en cycles rapides. Ainsi, elles peuvent figurer n'importe où dans le code et en particulier immédiatement après la définition d'une fonction afin de traquer les bugs et régressions [2], dans le cadre de réusinage [3]. Dans ce cas, on les rédige de la façon suivante :

```
let rec fact (n:int (* ⩾ 0 *)) =                                    1
  if n = 0 then 1 else n * fact(n-1)                                2

                                                                    3
let _ = assert(fact 0 = 1)                                          4
let _ = assert(fact 4 = 24)                                         5
```

Pour information, la librairie OCaml Alcotest [4] fournit une api[5] permettant de généraliser et d'automatiser les tests unitaires assertifs.

**Q10.** Asserter qu'aucune panne n'est à déplorer sur les relevés $R_1$, $R_2$ et $R_3$.

[1] https://fr.wikipedia.org/wiki/Extreme_programming#Tests_unitaires
[2] https://fr.wikipedia.org/wiki/Régression_logicielle
[3] https://fr.wikipedia.org/wiki/Réusinage_de_code
[4] https://github.com/mirage/alcotest
[5] application programming interface

### 4.1.3 Appartenance

La (re)lecture de l'exercice de TD « présence d'un élément » peut aider.

On souhaite savoir si un relevé comporte un certain flux :

| SPÉCIFICATION 3 | |
|---|---|
| PROFIL | $flux\_app$ : ..... |
| SÉMANTIQUE | $(flux\_app\ x\ r) = vrai$ si et seulement si $x \in r$. |

**Q11.** Compléter la spécification de $flux\_app$ ci-dessus.

**Q12.** Implémenter $flux\_app$ par filtrage *sans* utiliser d'expression conditionnelle (`if then else` interdit).

**Q13.** Tracer l'évaluation de l'expression `flux_app 7 cstR3`. Combien d'éléments du relevé sont-ils comparés à la valeur `7` ?

**Q14.** Généraliser ce résultat pour l'évaluation de l'expression $(flux\_app\ x\ r)$, en examinant les diverses situations significatives.

#### Généralisation

Outre les connecteurs logiques, la réalisation de $flux\_app$ ne fait intervenir que l'opérateur $=$, sur les entiers. Cet opérateur étant polymorphe, $flux\_app$ peut être définie de manière générique sur des séquences d'éléments de type quelconque.

**Q15.** Implémenter une fonction d'appartenance *app* polymorphe sur des séquences ($seq$) d'éléments de type quelconque ($\alpha$).

### 4.1.4 Extrema des flux journaliers

La (re)lecture de l'exercice de TD « maximum d'une séquence » peut aider.

On souhaite connaître les extrema, c'est-à-dire les valeurs minimum et maximum des flux journaliers d'un relevé. On définit donc :

$$releveNV \stackrel{def}{=} releve \setminus \{V\}$$

**Q16.** Implémenter $releveNV$ par un type synonyme.

**Q17.** Spécifier les fonctions $fluxmin$ et $fluxmax$.

**Q18.** Implémenter récursivement $fluxmin$ et/ou $fluxmax$.

**Q19.** Que signifie l'avertissement de l'interpréteur lors de l'évaluation de ces fonctions ? Pourquoi cet avertissement ? Que doit en penser le programmeur sérieux qui a spécifié proprement sa fonction avant de la réaliser ? Comment supprimer cet avertissement ?

Ignorer l'avertissement d'OCaml, le faire taire grâce à l'attribut `[@@warning "-8"]`, ou tout autre méthode plus ou moins propre (utilisation du motif anonyme `_`, génération d'exception



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

avec `invalid_arg "..."`) est risqué dans le cadre d'un développement complexe impliquant beaucoup de ré-usinage de code (refactoring).

On solutionne ce problème proprement en remplaçant $V$, constructeur sans argument, par un constructeur unaire $S$ (pour « singleton »), de type $flux \to releveNV$. Il devient alors impossible de construire des relevés vides.

**Q20.** Définir le type $releveNV_2$ de cette façon.

**Q21.** Réimplémenter la fonction $fluxmax$ avec le type $releveNV_2$, et observer l'abscence d'avertissement d'OCaml relativement au filtrage.

### 4.1.5 Flux observé au jour J

On souhaite connaître le flux observé un certain jour :

| SPÉCIFICATION 5 | |
|---|---|
| PROFIL | $fluxobs : \mathbb{N}^* \to releveNV_2 \to flux$ |
| SÉMANTIQUE | $(fluxobs\ j\ r)$ est le flux observé le jour $j$ dans le relevé $r$. Précondition : $j \leqslant \lvert r \rvert$. NB : le premier jour est le jour 1. |

**Q22.** Implémenter $fluxobs$.

## 4.2 TP7 Polynômes

On considère des polynômes à coefficients entiers comme par exemple $-3x^4 + 2x^2 + 10x - 12$.

Dans cet exercice, un monôme est caractérisé par un coefficient et une puissance (positive ou nulle). Par exemple, au monôme $-3x^4$ correspond le couple $(-3, 4)$. Un polynôme sera représenté par une séquence de monômes à coefficients non nuls *en ordre décroissant des puissances*. On définit donc :

$$monôme = \mathbb{Z} \times \mathbb{N}$$
$$polynôme = séq\ (monôme)\ \text{(non nuls et par ordre décroissant des puissances)}$$

Dans cette représentation, *les monôme à coefficient nul ne doivent pas apparaître*. C'est un invariant de la représentation, qui doit constamment être respecté. En particulier, le polynôme constant 0 est représenté par une séquence vide.

À partir de ce TP, on utilisera l'implémentation native OCaml des $séq\ (\alpha)$ : `'a list`.

**Q1.** Implémenter les ensembles définis ci-avant en OCaml.

**Q2.** Jeux de tests

a) Définir des constantes $M_1, M_2, M_3, M_4$ et $M_5$ représentant les monômes : 10 (monôme constant), $7x$, $-3x^2$, $x^4$ et $-x^4$.

b) En réutilisant $M_1, M_2, M_3, M_4$ et $M_5$, définir des constantes $P_1, P_2, P_3, P_4$ et $P_5$ – qui serviront de pour la suite – représentant les polynômes : 10 (polynôme constant), $7x + 10$, $-3x^2 + 7x$, $x^4 - 3x^2 + 7x$, $-x^4$.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

### 4.2.1 Dérivation

La dérivée d'un monôme est spécifiée par :

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | *derivMono* : *monôme* → *monôme* |
| SÉMANTIQUE | *derivMono*$(m)$ est le monôme dérivé de $m$. Par convention, la puissance du monôme nul est 0. |
| EX. ET PROP. | (i) *derivMono*$(M_4) = 4x^3$ |

**Q3.** Implémenter *derivMono*.

La dérivée d'un polynôme est spécifiée par :

| SPÉCIFICATION 2 | |
|---|---|
| PROFIL | *derivPoly* : *polynôme* → *polynôme* |
| SÉMANTIQUE | *derivPoly*$(m)$ est le polynôme dérivé de $m$. |
| EX. ET PROP. | (i) *derivPoly*$(P_4) = 4x^3 - 6x + 7$ |

**Q4.** Implémenter *derivPoly*.

**Q5.** Tester *derivPoly* sur les polynômes `[cstM1]` à `[cstM5]`, puis sur $P_2$, $P_3$ et $P_4$ en utilisant les assertions (relire le paragraphe 4.1.2 page 35).

### 4.2.2 Somme

La somme de deux polynômes est spécifiée ainsi :

| SPÉCIFICATION 3 | |
|---|---|
| PROFIL | *sommePoly* : *polynôme* → *polynôme* → *polynôme* |
| SÉMANTIQUE | (*sommePoly* $p_1$ $p_2$) est le polynôme $p_1 + p_2$. |
| EX. ET PROP. | (i) (*sommePoly* $P_4$ $P_5$) $= P_3$ |

**Q6.** Implémenter *sommePoly*.

> **Indication** On n'oubliera pas que les séquences représentant les polynômes sont triées par ordre décroissant des puissances et que les monômes nuls ne sont pas représentés.

**Q7.** Tester *sommePoly*, par exemple en calculant $P_1 + P_1$, $P_1 + P_2$, $P_2 + P_1$ ; asserter que $P_4 + P_5 = P_5 + P_4 = P_3$

**Q8.** Vérifier sur quelques exemples la propriété bien connue : « la dérivée de la somme est la somme des dérivées ».

## 4.3 TP8 Somme d'une suite de nombres

**Il est impératif de tester les fonctions au fur et à mesure de leur implémentation, particulièrement dans cet exercice.**

On considère des textes (séquences de caractères) constitués uniquement de chiffres et d'espaces. Par exemple : `[' ';'1';'2';'3';' ';'4';'5';' ';' ';'6';' ']`.

*Dans cet exercice, on appellera nombre toute suite non vide de chiffres écrits en base dix* (`'0'`, `'1'`, ..., `'9'`) délimitée par des espaces. Par exemple : `['1';'2';'3']`. Le texte peut commencer ou se terminer par un chiffre. Mais il peut aussi commencer ou se terminer par un ou plusieurs espaces. Deux nombres sont séparés par un ou plusieurs espaces. Chaque nombre est la représentation d'un entier en base 10 et le texte représente une suite d'entiers.

Étant donné un tel texte, on veut déterminer la somme des entiers qu'il représente.

**Principe de résolution.** On procède en trois étapes :

1. Obtention d'une séquence de nombres *sn* à partir des caractères du texte source *s*. Par exemple, si *s* est `['1';'2';'3';' ';' ';'4';'5';' ';'6']`,
   *sn* = `[['1';'2';'3'];['4';'5'];['6']]`.
2. Obtention d'une séquence d'entiers *se* à partir des nombres de *sn*. En reprenant l'exemple précédent, on obtient *se* = [123 ; 45 ; 6].
3. Obtention de la somme des entiers de *se*.

On définit ainsi les ensembles suivants :

$$\begin{aligned}
&\mathit{chiffre} = \{0,1,2,3,4,5,6,7,8,9\}\\
&\mathit{chiffreCar} = \{\texttt{'0'},\texttt{'1'},\texttt{'2'},\texttt{'3'},\texttt{'4'},\texttt{'5'},\texttt{'6'},\texttt{'7'},\texttt{'8'},\texttt{'9'}\}\\
&\mathit{nombre} = \mathit{séq}\,(\mathit{chiffreCar}) \setminus \{[\,]\}\\
&\mathit{txtnb} = \mathit{séq}\,(\mathit{char}), \text{où les car. sont des } \mathit{chiffreCar} \text{ ou des espaces}
\end{aligned}$$

**Q1.** Implémenter ces ensembles en OCaml, sans oublier de préciser les éventuelles restrictions sur les types.

**Q2.** Spécifier les fonctions suivantes :

- *somme_txtnb* : calcule la somme de nombres réprésentés en tant que *txtnb*,
- *les_nb* : construit la séquence de nombres réprésentés en tant que *txtnb*,
- *snbVsnat* : construit la séquence d'entiers correspondant à une séquence de nombres,
- *somme* : calcule la somme d'une séquence d'entiers.

### 4.3.1 Étape 3

**Q3.** Implémenter *somme*.

### 4.3.2 Étape 2

Pour implémenter la fonction *snbVsnat*, les fonctions intermédiaires suivantes sont nécessaires :



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

| SPÉCIFICATION 5 — Fonctions intermédiaires pour *snbVsnat* | |
|---|---|
| PROFIL | $ccVc$ : $chiffreCar \to chiffre$ |
| SÉMANTIQUE | $ccVc(cc)$ est le $chiffre$ correspondant à $cc$. |
| PROFIL | $nbVsnat$ : $nombre \to \mathbb{N}$ |
| SÉMANTIQUE | $nbVsnat(nb)$ est l'entier correspondant à $nb$. |

**Q4.** Implémenter les deux fonctions spécifiées ci-dessus ; en déduire une implémentation de *snbVsnat*.

### 4.3.3 Étape 1

Pour implémenter la fonction *les_nb*, les fonctions intermédiaires suivantes sont nécessaires :

| SPÉCIFICATION 6 — Fonctions intermédiaires pour *les_nb* | |
|---|---|
| PROFIL | $sup\_esp$ : $txtnb \to txtnb$ |
| SÉMANTIQUE | $sup\_esp(txt)$ supprime les espaces au début de $txt$. |
| PROFIL | $prnb\_r$ : $txtnb \to nombre \times txtnb$ |
| SÉMANTIQUE | Posons $(nb, reste) = prnb\_r(txt)$ ; $nb$ est le premier nombre de $txt$ (s'il existe) et $reste$ est le reste de $txt$. |
| EX. ET PROP. | (i) $prnb\_r($`['1';'2';'3';' ';'4';'5';' ';'6']`$) =$ (`['1';'2';'3']`, `[' ';'4';'5';' ';'6']`) |

**Q5.** Implémenter les deux fonctions spécifiées ci-dessus ; en déduire une implémentation de la fonction *les_nb*.

**Q6.** Implémenter *somme_txtnb*.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

# Troisième partie

# ORDRE SUPÉRIEUR

# Chapitre 5

# Schémas et fonctions d'ordre supérieur

**Sommaire**



## 5.1 TP9 Curryfication

**Q1.** Implémenter la fonction *add*, de profil $\mathbb{Z}^2 \to \mathbb{Z}$, qui additionne les deux composantes d'un couple d'entiers.

**Q2.** Tester *add* sur les données suivantes : $(3,5)$, $(5,3)$, 3. Expliquer le message d'erreur de l'interpréteur obtenu au dernier test à la lumière des concepts de curryfication et d'application partielle.

**Q3.** Implémenter une fonction $add_C$ à deux paramètres et la tester. Comparer les deux implémentations et résultats de l'interpréteur. Quelles sont les similitudes et les différences ?

**Q4.** Quelle est la nature de $(add_C\ 3)$ ? Peut-on utiliser $(add_C\ 3)$ ? Quelles similitudes et différences y-a-t'il entre $(add_C\ 3\ 5)$ et $((add_C\ 3)\ 5)$ ?

Une autre façon de comprendre *addC* est de remarquer que pour tout entier $x$, $addC(x)$ est une fonction qui pour tout entier $y$ renvoie la somme de $a$ et $b$. Cette interprétation saute aux yeux si on implémente *addC* de la façon suivante :

```
let addC2 (x:int) : int -> int =
  if x = 0 then
    fun y -> y    (* fonction (anonyme) *)
  else
    fun y -> x+y  (* fonction (anonyme) *)
```

Cette implémentation est parfaitement équivalente à la précédente, mais insiste sur la possibilité d'effectuer des applications partielles.

**Q5.** Observer le profil, évaluer et tester `addC2x`.

## 5.2 TP9 Dérivation de fonction

Considérons une fonction $f$ de $\mathbb{R}$ dans $\mathbb{R}$, dérivable. Étant donné un petit accroissement $dx$, la dérivée de $f$, notée $f'$, est la fonction définie pour tout réel $x$ par :

$$f'(x) = \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx}$$

Pour éviter les aléas classiques des calculs en virgule flottante inhérents à la norme IEEE 754[1], on prendra $dx = 0.001$.

**Q1.** Spécifier puis réaliser une fonction `dérivée` qui dérive une fonction selon l'approximation décrite ci-dessus.

**Q2.** En déduire une implémentation de la dérivée de la fonction $\begin{pmatrix} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x & \longmapsto & x^2 \end{pmatrix}$. Tester.

**Q3.** Implémenter des fonctions et des constantes permettant de calculer les dérivées 1re, 2° et 3e en 0. Comparer avec les valeurs «exactes» obtenues grâce aux fonctions `sin` et `cos` prédéfinies par OCaml. La fonction `abs_float`, qui donne la valeur absolue d'un réel, pourra être utile.

Dans cet exercice, un *endomorphisme* est une fonction de profil $\alpha \to \alpha$. La *puissance* d'un endomorphisme est définie ainsi :

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | $puiss : \mathbb{N} \to (\alpha \to \alpha) \to \alpha \to \alpha$ |
| SÉMANTIQUE | $(puiss\ n\ f)$, notée $f^n$ en math., est la composée de $f$ avec elle-même $n$ fois :<br>• par convention, $f^0 \stackrel{def}{=} id_\alpha$ (fonction identité, dans $\alpha$)<br>• si $n > 0$, $f^n \stackrel{def}{=} \underbrace{f \circ \cdots \circ f}_{n\ fois}$ |
| EX. ET PROP. | (i) $\begin{pmatrix} \mathbb{Z} & \longrightarrow & \mathbb{Z} \\ x & \longmapsto & x+1 \end{pmatrix}^2 = \begin{pmatrix} \mathbb{Z} & \longrightarrow & \mathbb{Z} \\ x & \longmapsto & x+2 \end{pmatrix}$ |

**Q4.** Implémenter *puiss*.

a) directement, sans utiliser de fonction intermédiaire ;

b) à l'aide de $o$, dont on rappelle l'implémentation sous forme d'un opérateur infixe (cf cours ou exercice «composition») :

```
let (>>) (f:'a->'b) (g:'b->'c) : 'a->'c =
  fun x -> g (f x)
```

**Q5.** En déduire la dérivée troisième de sinus en 0 et retrouver le dernier résultat de la question précédente.

**Q6.** Quelle est la sémantique de l'expression `puiss 3 (derivee 0.001 sin) 0.` ?

[1] https://fr.wikipedia.org/wiki/IEEE_754



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

## 5.3 TD9 Tri par insertion

Considérons la séquence $s = pr :: fin$. Le principe du tri par insertion consiste à trier $fin$, puis y insérer $pr$ au bon endroit (selon $\leqslant$).

On spécifie l'insertion dans une liste triée de la façon suivante :

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | $inserer$ : $séq\,(\alpha)$ *triée* $\to \alpha \to séq\,(\alpha)$ |
| SÉMANTIQUE | $(inserer\ s\ x)$ est la séquence triée obtenue en insérant $x$ dans $s$, supposée triée, au bon endroit. |
| EX. ET PROP. | (i) $(inserer\ [1;3]\ 0) = [0;1;3]$<br>(ii) $(inserer\ [1;3]\ 1) = [1;1;3]$<br>(iii) $(inserer\ [1;3]\ 2) = [1;2;3]$<br>(iv) $(inserer\ [1;3]\ 3) = [1;3;3]$<br>(v) $(inserer\ [1;3]\ 4) = [1;3;4]$ |

**Q1.** En utilisant le schéma d'ordre supérieur $fold\_right$, donner une implémentation non récursive de $inserer$.

> **Indication** Ne pas chercher à insérer $x$ dans $s$, mais plutôt l'inverse : positionner les éléments de $s$ autour de l'accumulateur, en se rappelant que $s$ est triée.

**Q2.** En déduire une implémentation (non récursive) d'une fonction de tri par insertion.

## 5.4 TP9 Affixes

Un *affixe* est une séquence de lettres qui se placent avant (préfixe) ou après (suffixe) un mot pour en modifier le sens. Néanmoins, on n'imposera aucune restriction sur le type des éléments des séquences manipulées dans cet exercice.

### 5.4.1 Suffixes

Les suffixes d'une séquence sont spécifiés comme suit :

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | $suffixes$ : $séq\,(\alpha) \to séq\,(séq\,(\alpha))$ |
| SÉMANTIQUE | $suffixes\ [e_0;\ \ldots\ ;e_{n-1}] =$<br>$[\ [\ ]\ ;\ [e_{n-1}]\ ;\ [e_{n-2};e_{n-1}]\ ;\ \cdots\ ;\ [e_0;\cdots;e_{n-1}]\ ]$ |
| EX. ET PROP. | (i) $(suffixe\ [2;3;1]) = [\ [\ ]\ ;\ [1]\ ;\ [3;1]\ ;\ [2;3;1]\ ]$ |

**Q1.** Implémenter $suffixes$ grâce à un schéma $fold$.

> **Indication**
>
> On a : $(suffixes\ [e_0;e_1]) = [\ [\ ]\ ;\ [e_1]\ ;\ [e_0;e_1]\ ]$
>
> et : $(suffixes\ [e_0;e_1;e_2]) = [\ [\ ]\ ;\ [e_2]\ ;\ [e_1;e_2]\ ;\ [e_0;e_1;e_2]\ ]$
>
> Or : $[e_1;e_2] = (tl\ [e_0;e_1;e_2]);[e_2] = (tl\ [e_1;e_2]);[\ ] = (tl\ [e_2])$.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

### 5.4.2 Préfixes

Les préfixes d'une séquence sont spécifiés comme suit :

| SPÉCIFICATION 2 | |
|---|---|
| PROFIL | $prefixes\ :\ séq\ (\alpha) \rightarrow séq\ (séq\ (\alpha))$ |
| SÉMANTIQUE | $prefixes\ [e_0;\ \ldots\ ;e_{n-1}] = [\ [\ ]\ ;\ [e_0]\ ;\ [e_0;e_1]\ ;\ \cdots\ ;\ [e_0;\cdots;e_{n-1}]\ ]$ |
| EX. ET PROP. | (i) $(prefixe\ [2;3;1])\ =\ [\ [\ ]\ ;\ [2]\ ;\ [2;3]\ ;\ [2;3;1]\ ]$ |

#### 1re implémentation

**Q2.** Comparer les préfixes de $[2;3;1]$ aux suffixes de $[1;3;2]$

**Q3.** En déduire une impémentation (non récursive) de la fonction *prefixes*.

#### 2e implémentation

**Q4.** Quelle(s) opération(s) faut-il effectuer pour passer des préfixes de $[e_0;e_1]$ aux préfixes de $[e_0;e_1;e_2]$ ?

**Q5.** En déduire une impémentation (non récursive) de la fonction *prefixes*.

#### 3e implémentation

On spécifie la fonction auxiliaire suivante :

| SPÉCIFICATION 3 | |
|---|---|
| PROFIL | $ajoutAchq\ :\ \alpha \rightarrow séq\ (séq\ (\alpha)) \rightarrow séq\ (séq\ (\alpha))$ |
| SÉMANTIQUE | $(ajoutAchq\ x\ ss)$ est la séquence obtenue en ajoutant $x$ en tête de chaque séquence de $ss$. |
| EX. ET PROP. | (i) $(ajoutAchq\ 0\ [[2;3];[5];[\ ];[3;8]])\ =\ [[0;2;3];[0;5];[0],[0;3;8]]$ |

**Q6.** Donner une implémentation non récursive d'*ajoutAchq*.

**Q7.** En déduire une implémentation (non récursive) de *prefixes*.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

# Quatrième partie

# STRUCTURES ARBORESCENTES



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

# Chapitre 6

# Structures arborescentes

**Sommaire**



## 6.1 TP11 Appropriation des notations

Le but de cet exercice est d'examiner diverses méthodes de lecture / écriture d'arbres.

**Q1.** Implémenter l'ensemble *abin* ($\alpha$) des arbres binaires.

### 6.1.1 Méthode 1 : avec les constructeurs

**Q2.** Implémenter les deux arbres `ex1` et `ex2` :

```
  15          15
  /\          /\
 7  33       7 33
  \         /
  13      13
```

en utilisant uniquement les constructeurs OCaml et en définissant au préalable tous leurs sous-arbres.

### 6.1.2 Méthode 2 : avec les fonctions de construction

**Q3.** Définir les fonctions de construction suivantes :

- *abS* prend un entier et renvoie l'arbre formé uniquement de cet entier à la racine ;
- *abUNd* prend un couple $(n, A)$ formé d'un entier $n$ et d'un arbre $A$ et renvoie l'arbre avec $n$ à la racine et $A$ en sous-arbre droit ;
- *abUNg* prend un couple $(A, n)$ formé d'un entier $n$ et d'un arbre $A$ et renvoie l'arbre avec $n$ à la racine et $A$ en sous-arbre gauche.

**Q4.** Donner une autre implémentation des deux arbres précédents en utilisant le plus possible les fonctions de construction *abS*, *abUNd*, *abUNg*.

### 6.1.3 Méthode 3 : avec des opérateurs

Même avec les fonctions de construction qui permettent d'alléger un peu les notations, les implémentations des arbres ne sont ni très pratiques à utiliser, ni aisément lisibles. L'utilisation d'opérateurs – fonctions infixes très facilement définissables en OCaml – permet d'améliorer la lisibilité et facilite la création de jeux de tests.

Un opérateur se notera entre parenthèses au niveau des spécifications et des implémentations, et au milieu de ses arguments (ou devant son argument pour les opérateurs unaires) lors de l'utilisation. Par exemple :

```
# (+) ;;
- : int -> int -> int = <fun>
# 1 + 2 ;;
- : int = 3
```

**Q5.** Implémenter `(!)` tel que `!x` est l'arbre singleton de nœud `x`[1].

**Q6.** Implémenter `(->>)` tel que `r->>d` est l'arbre de racine `r` et de sous-arbre droit `d`, puis `(<<-)` tel que `g<<-r` est l'arbre de racine `r` et de sous-arbre gauche `g`.

**Q7.** Redéfinir les deux arbres de la question 2 en utilisant le plus possible les opérateurs précédents.

Pour un nœud binaire vrai, l'obtention d'un opérateur est plus complexe. Il faudrait un opérateur ternaire (racine, sous-arbre gauche, sous-arbre droit), ce qui pose des problèmes de sémantique et de notation : à supposer que `<-->` soit l'opérateur ternaire en question, où placerait-t'on les arguments `r`, `g` et `d`?

Une solution consiste à utiliser deux opérateurs binaires, `<--` et `-->`, dont voici la définition (les spécifications sont détaillées ci-dessou pour mémoire ; il n'est pas nécessaire de les comprendre en détails pour pouvoir utiliser les opérateurs).

**SPÉCIFICATION 1**

| | |
|---|---|
| PROFIL | $(--\!>) : \alpha \to abin(\alpha) \to (abin(\alpha) \to abin(\alpha))$ |
| SÉMANTIQUE | `r --> d` est un arbre en cours de construction, c'est-à-dire une fonction qui attend un arbre (le sous-arbre gauche). |
| PROFIL | $(<\!--) : abin(\alpha) \to (abin(\alpha) \to abin(\alpha)) \to abin(\alpha)$ |
| SÉMANTIQUE | `a <-- f` est l'arbre résultant de l'application de la fonction f sur a. Ici, f est un arbre en cours de construction, c'est donc la fonction qui attend le sous-arbre gauche pour construire l'arbre complet. |

**RÉALISATION 1**

```
let (-->) (r:'a) (d : 'a abin) : 'a abin -> 'a abin =                1
  fun (g : 'a abin) -> Ab(g,r,d)                                     2
let (<--) (g : 'a abin) (f : 'a abin -> 'a abin) : 'a abin =         3
```

[1] On écrase ainsi l'opérateur OCaml de déréférencement des pointeurs, ce qui n'est pas génant : d'une part, il n'est pas utilisé en INF2X1 ; d'autre part, on peut utiliser `r.contents` au lieu de `!r`.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

```
  f g                                                                    4
```

**Q8.** Après avoir copié-collé les implémentations ci-dessus, redéfinir les arbres de la question 2 à l'aide des opérateurs `(-->)` et `(<--)`.

On peut donc maintenant construire de manière concise et agréable des exemples d'`abin` sans avoir à recourir directement aux constructeurs de ce type.

Néanmoins, on ne perdra pas de vue que seule les expressions utilisant uniquement `Av` et `Ab` ont une réalité ; les opérateurs et fonctions de construction ne sont que du sucre syntaxique.

### 6.1.4 Affichage (un peu plus) agréable d'arbres

Lire l'affichage d'un arbre dans l'interpréteur est difficile, voire impossible à partir d'une dizaine de nœuds. À titre d'illustration, évaluer par exemple :

```
let ex12 : int abin = ex1 <-- 1 --> ex2                                  1
```

Heureusement, OCaml a prévu une infrastructure permettant à l'utilisateur de modifier ses routines internes d'affichage. La première étape consiste à définir une fonction (d'ordre supérieur) de conversion d'un arbre en chaîne de caractères, dont voici la définition pour information :

**SPÉCIFICATION 2**

| | |
|---|---|
| PROFIL | *arbreVchaine_op* : $(\alpha \to \textit{chaîne}) \to \textit{abin}\ (\alpha) \to \textit{chaîne}$ |
| SÉMANTIQUE | Étant donné une fonction de conversion *nVc* des nœuds d'un arbre en chaîne de caractères, (*arbreVchaine_op nVc a*) est une chaîne représentant *a*. |
| EX. ET PROP. | (i) (*arbreVchaine_op* `string_of_int ex1`) = `"(7->>!13)<--15-->!33"`<br>(ii) (*arbreVchaine_op* `string_of_int ex2`) = `"(!13<<-7)<--15-->!33"` |

**RÉALISATION 2**

```
let rec arbreVchaine_op (nVc:'a->string) (a : 'a abin) : string =        1
  let estSingle (a : 'a abin (* non vide *)) : bool =                    2
    match a with Ab(Av,_,Av) -> true | _ -> false                       3
  in                                                                     4
  let parenth (a : 'a abin) (str_a : string) =                           5
    if estSingle a then str_a else "(" ^str_a ^")"                       6
  in                                                                     7
  match a with                                                           8
  | Av       -> ""                                                       9
  | Ab(g,r,d) -> let str_r = nVc r in                                    10
    if g = Av && d = Av then                                             11
      "!" ^str_r                                                         12
    else (* g != Av ou d != Av ou les deux *)                            13
      let str_g = arbreVchaine_op nVc g                                  14
      and str_d = arbreVchaine_op nVc d in                               15
      if g = Av then                                                     16
        str_r ^"->>" ^parenth d str_d                                    17
      else if d = Av then                                                18
        (parenth g str_g) ^"<<-" ^str_r                                  19
      else (* g et d non vides *)                                        20
```



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

```
        (parenth g str_g) ^"<--" ^str_r ^"-->" ^parenth d str_d
```

La compréhension des détails du fonctionnement de *arbreVchaine_op* n'est pas demandée.

La deuxième étape remplace la routine standard d'affichage d'OCaml, via la directive `#install_printer`, par une fonction d'affichage de l'utilisateur (selon un format imposé) :

```
let affiche_intabin (a : int abin) : unit =
  Format.printf "@[%s@]" (arbreVchaine_op string_of_int a)
;;
#install_printer affiche_intabin
```

On notera l'utilisation de la fonction OCaml `string_of_int` comme fonction de conversion des nœuds en chaîne, ainsi que de la fonction `printf` du module `Format`, dont la sémantique sort du cadre de cette UE (le lecteur intéressé pourra néanmoins consulter la documentation du module `Format` [2] ).

**Q9.** Télécharger, puis évaluer (directive `#use`) le fichier `affiche_arbre_op.ml`. Observer l'effet du changement d'afficheur sur `ex1`, `ex2` et `ex12`.

Le code de `affiche_arbre_op.ml` ne fonctionnera que si le type *abin* ($\alpha$) a été correctement défini au préalable.

Pour revenir à l'affichage standard, il faut supprimer l'afficheur de l'utilisateur :

```
#remove_printer affiche_intabin ;;
```

Pour afficher d'autres types d'arbre, il suffit de remplacer l'afficheur courant par un nouvel afficheur. Par exemple, pour des arbres de caractères :

```
let affiche_charabin (a : char abin) : unit =
  Format.printf "@[%s@]" (arbreVchaine_op (String.make 1) a)
;;
#install_printer affiche_charabin
```

Noter l'application partielle de la fonction :

| SPÉCIFICATION 3 | |
|---|---|
| PROFIL | `String.make : int → char → string` |
| SÉMANTIQUE | `(String.make n c)` est la chaîne constituée de `n` fois le caractère `c`. |

pour constituer la fonction de conversion des nœuds en chaîne (paramètre *nVc* de *arbreVchaine_op*)

**Q10.** Après avoir installé l'afficheur `affiche_charabin`, observer l'évaluation du jeux d'essais suivant, puis dessiner (sur une feuille) l'arbre `ex3` :

```
let a4 : char abin = !'d' <-- 'e' --> !'g'
let a3 : char abin = !'a' <-- 'c' --> a4
let a2g : char abin = a3 <-- 'k' --> !'m'
let a2d : char abin = 'p' ->> !'q' <<- 'u'
let ex3 : char abin = a2g <-- 'n' --> a2d
```

On peut donc maintenant afficher de manière concise et agréable des exemples d'`abin` en cachant les constructeurs de ce type.

Néanmoins, on ne perdra pas de vue que seule les expressions utilisant uniquement `Av` et `Ab` ont une réalité ; les afficheurs ne sont que du sucre syntaxique.

[2] https://ocaml.org/api/Format.html

## 6.2 TP11 Sommes

**Q1.** Implémenter la fonction *somme* qui à tout arbre binaire d'entiers associe la somme de tous les nœuds de l'arbre.

**Q2.** En ré-implémentant les opérateurs de construction et les exemples `ex1` et `ex2` de l'exercice précédent (et éventuellement un afficheur d'arbre), tester *somme*.

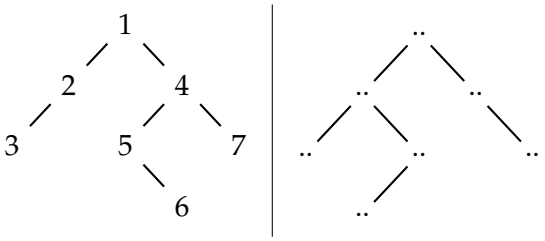
**Q3.** Tracer l'évaluation de *somme* sur `ex1` ou `ex2`. On prendra soin d'indenter la trace afin de mettre en évidence l'emboîtement des appels récursifs.

**Q4.** Implémenter la fonction suivante :

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | $somme\_pos\_neg\_0 : abin(\mathbb{Z}) \to \mathbb{Z}^+ \times \mathbb{Z}^- \times \mathbb{N}$ |
| SÉMANTIQUE | Posons $(sp, sn, nb_0) = (somme\_pos\_neg\_0\ a)$ :<br>• $sp$ est la somme des éléments strictement positifs ;<br>• $sn$ est la somme des éléments strictement négatifs ;<br>• $nb_0$ est le nombre d'éléments nuls ;<br>dans l'arbre $a$. |

## 6.3 TP11 Arbres symétriques

Le symétrique d'un arbre $a$ est l'image de $a$ dans un miroir.




**Q1.** Compléter l'exemple ci-dessus.

**Q2.** Implémenter une fonction *sym* qui donne le symétrique d'un arbre.

**Q3.** En déduire une implémentation non récursive d'un prédicat *sontSym* qui indique si deux arbres sont symétriques.

## 6.4 TP12 Tri en arbre d'une sequence

Un arbre binaire *de recherche* (ABR) est un arbre binaire $a$ dans lequel chaque nœud est étiqueté par un entier $x$ tel que :

- chaque nœud du sous-arbre gauche de $a$ est étiqueté par un entier inférieur ou égal à $x$ ;

- chaque nœud du sous-arbre droit de $a$ est étiqueté par un entier strictement supérieur à $x$ ;
- les sous-arbres gauche et droit de $a$ sont eux-mêmes des ABR.

Par exemple :

```
    5
   / \
  2   9
 / \
1   4
```

Les ABR seront implémentés par :

```
type 'a abr =
| Av
| Ab of 'a abr (* ABR *) * 'a  * 'a abr (* ABR *)
```

Il est donc de la responsabilité du programmeur / utilisateur d'assurer les trois propriétés faisant d'un arbre un ABR.

Pour insérer un entier $e$ dans un ABR $a$, on l'ajoute comme *nouvelle feuille* de la manière suivante :

1. on recherche (récursivement) dans quel sous-arbre gauche ou droit de $a$ l'entier $e$ doit être inséré, jusqu'à atteindre une feuille $f$ ;
2. on ajoute un nouveau nœud d'étiquette $e$ comme fils gauche ou comme fils droit de $f$ selon que $e$ est supérieur ou inférieur à l'étiquette de $f$.

**Q1.** Implémenter une fonction *insert* qui, étant donnés un entier $e$ et un ABR $a$, insère $e$ dans $a$ selon le principe décrit ci-dessus.
Dans cette question, l'utilisation de toute fonction intermédiaire est interdite.

**Q2.** Implémenter une fonction `parcours_sym` qui retourne la séquence d'entiers obtenue en parcourant symétriquement un arbre (sous-arbre gauche, racine, sous-arbre droit). Par exemple :

```
    5
   / \
  2   9   --->   [1 ; 2 ; 4 ; 5 ; 9]
 / \
1   4
```

Pour trier une séquence, on utilise une fonction auxiliaire *seqVabr* qui produit l'ABR correspondant à une séquence.

**Q3.** Donner une implémentation récursive de *seqVabr*.

**Q4.** En utilisant un schéma d'ordre supérieur, ré-implémenter *seqVabr* non récursivement.

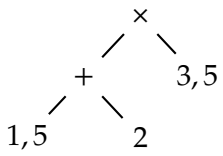
**Q5.** En déduire une implémentation non récursive d'une fonction de tri sur les séquences.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

## 6.5 TP12 Expressions arithmétiques

Les expressions arithmétiques peuvent être représentées sous forme d'arbres binaires dont les nœuds internes sont étiquetés par un symbole représentant l'opération à effectuer et les feuilles par les opérandes. De tels arbres sont appelés arbres d'expressions arithmétiques. Un arbre d'expression arithmétique comprend deux sortes de nœuds : les feuilles, étiquetées par des nombres réels, et les nœuds internes composés d'un sous-arbre gauche, d'un opérateur et d'un sous-arbre droit. Par exemple, l'expression arithmétique $(1,5+2)\times 3,5$ est représentée par l'arbre :




La définition du type des arbres d'expressions arithmétiques, notée `exprarith`, nécessite donc celle des nœuds : un nœud est soit une opérande, soit une des quatre opérations.

**Q1.** Définir les types `noeud` et `exprarith`.

**Q2.** Implémenter les expressions $1+2\times 3$ et $(1+2)\times 3$.

### 6.5.1 Valeur d'une expression arithmétique

**Q3.** Implémenter une fonction *val_expr* telle que, pour tout arbre d'expression arithmétique $a$, $val\_expr(a)$ est la valeur de l'expression représentée par $a$.

**Q4.** Tester *val_expr* sur les expressions $1+2\times 3$ et $(1+2)\times 3$.

### 6.5.2 Afficheur pour les expressions arithmétiques

**Q5.** Implémenter une fonction de conversion des nœuds en chaîne de caractères.

**Q6.** En s'inspirant de l'exercice « Appropriation des notations », améliorer la lisibilité de l'affichage des expressions arithmétiques.

### 6.5.3 Linéarisation d'expressions arithmétiques

Le paragraphe précédent utilise la fonction générale de conversion des $séq\,(\alpha)$ en chaîne de caractères.

Les `exprarith` ont la particularité de ne jamais comporter de sous-arbres unaires (gauche ou droit). Cette particularité permet d'améliorer un peu plus la représentation linéaire des expressions arithmétiques : les symboles `<--` et `-->` sont en effet inutiles.

On note également que tout nœud qui n'est pas un opérateur est nécessairement une opérande : les symboles `!` sont donc également inutiles.

Dans cet exercice, on ne cherche pas à optimiser le parenthésage en tenant compte de la priorité des opérateurs. Les linéarisation obtenues sont dites *totalement parenthésées*.

**Q7.** Implémenter une fonction *estFeuille* qui indique si une expression arithmétique non vide est réduite à une feuille.



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg

**Q8.** Implémenter la fonction suivante, qui parenthèse toutes les sous-expressions, à l'exception des opérandes :

| SPÉCIFICATION 1 | |
|---|---|
| PROFIL | *parenth* : *exprarith* → `string` → `string` |
| SÉMANTIQUE | (*parenth a str_a*) parenthèse l'expression arithmétique *a* dont *str_a* est la conversion en chaîne, à moins que *a* soit une opérande. |
| EX. ET PROP. | (i) (*parenth* `!(F 1.) "1."`) = `"1."`<br>(ii) (*parenth* `(!(F 1.)<- Plus -> !(F 2.)) "1. + 2.")` = `"(1. + 2.)"` |

**Q9.** En déduire une implémentation d'une fonction *exprVchaine* de conversion des expressions arithmétiques en chaîne de caractères.

**Q10.** Observer l'effet de l'afficheur suivant sur les exemples précédents :

```
let affiche_exprarith (ea:exprarith) : unit =
  Format.printf "@[%s@]" (exprVchaine ea)
;;
#install_printer affiche_exprarith
```



[git] • Branche master@7307b1b • Version v2.0.3-polys (11/01/2023) par François Puitg