UNIVERSITÉ
GRENOBLE
ALPES

UFR IM²AG

DÉPARTEMENT
LICENCE SCIENCES ET
TECHNOLOGIE

LICENCE SCIENCES & TECHNOLOGIES
1re ANNÉE

UE INF201, INF231
ALGORITHMIQUE ET PROGRAMMATION FONCTIONNELLE

2020 / 2021

EXERCICES DE TRAVAUX PRATIQUES

Dans le titre de chaque exercice, problème ou annale d'examen est indiqué TDn ou TPn, où n est un entier entre 1 et 12 correspondant au numéro de séance de travaux dirigés ou pratiques **à partir duquel** l'énoncé peut être abordé.

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.

Il est fortement recommandé de réviser chaque partie en cherchant à résoudre un ou deux énoncés d'annales.

Table des matières

Rappels

- *définir* = donner une définition,
  définition = spécification + réalisation ;
- *spécifier* = donner une spécification (le « quoi »),
  spécification = profil + sémantique + examples et/ou propriétés ;
- *réaliser* = donner une réalisation (le « comment »),
  réalisation = algorithme (langue naturelle) + implémentation (OCaml) ;
- *implémenter* = donner une inplémentation (OCaml).

Dans certains cas, certaines de ces rubriques peuvent être omises.

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 avec l'environnement OCaml. Il explique comment lancer un éditeur et l'interprèteur 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, utilisez un environnement local que vous avez installé sur votre machine (voir « OCaml sur votre ordi ») ou bien connectez-vous sur un serveur Linux du DLST, par exemple turing, si vous êtes en salle de TP.

1.1.1 Installation d'un environnement pour OCaml

Pour pouvoir programmer avec ce langage, nous suggérons l'utilisation d'un environnement composé de deux outils :

- *Un éditeur de texte* basique, par exemple Gedit. Cet outil, dont l'interface est simple et intuitive, vous permettra d'éditer vos fichiers sources. D'autres éditeurs, plus ou moins performants et complexes à installer, existent : Atom (avec le package language-ocaml), VSC (avec l'extension OCaml Platform), ...
- L'interpréteur OCaml (`ocaml`), pour évaluer votre code source et le mettre au point (voir la section précédente).

Dans la suite, nous détaillons uniquement l'utilisation de Gedit. Nous vous laissons adapter ces explications si vous utilisez un autre éditeur, et vous invitons à nous faire part de vos expérimentations afin d'améliorer ce document.

Lancer l'éditeur et l'interpréteur dans deux fenêtres différentes. Par exemple, dans un interpréteur de commandes Linux[1], taper :

1. `gedit inf201_Puitg_Basset_Couillet_TP1.ml &`

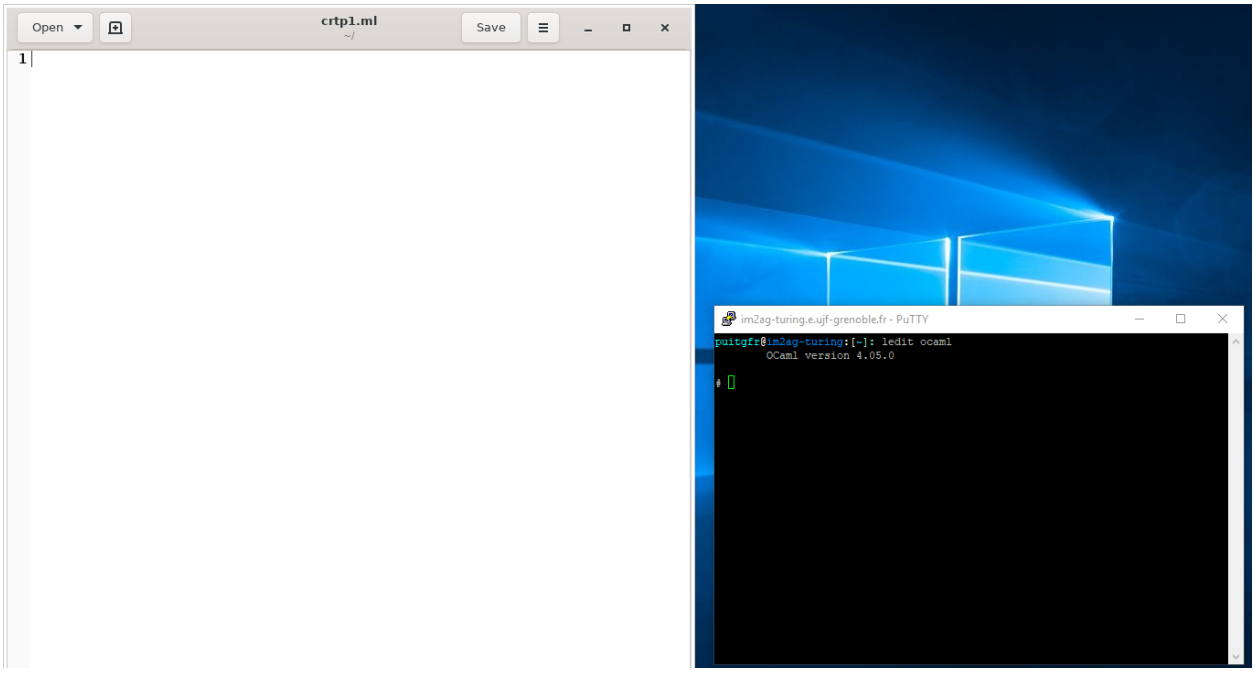
   `inf201_Puitg_Basset_Couillet_TP1.ml` est le nom du fichier (**c**ompte **r**endu de TP n° 1). Ne pas oublier le `&` final[2]. Cet éditeur démarre parfois en mode plein écran, retailler sa fenêtre si besoin.

1. Ne pas confondre avec l'interpréteur OCaml
2. La signification du &, et plus généralement des commandes Linux n'est pas détaillée dans cette UE.

2. `ledit ocaml`

   Nous vous conseillons la commande : `ledit ocaml`[3] plutôt que `ocaml` tout court ; vous pourrez ainsi rappeler une phrase précédente avec la touche ⇧ du clavier (flèche vers le haut).

Vous pouvez arranger les fenêtres pour obtenir un écran analogue à celui ci-dessous, dans lequel l'éditeur est placé en haut à gauche de l'écran ; il contiendra in fine le compte rendu. L'interprèteur OCaml apparaît en bas à droite de l'écran.



Il est agréable de travailler avec les deux fenêtres pleinement visibles. Pour cela :

- réduire la taille de la fenêtre de l'interpréteur ;

  un clic droit sur la barre de titre permet de choisir une taille de police de 9 dans le menu Change Settings > Appearance > Font.

- demander à Gedit d'afficher une marge à la colonne 80 ;

  menu burger ☰, Preferences, cocher « Display right margin at column 80 » ; retailler la fenêtre de Gedit de façon à laisser juste apparaître cette marge ; lors de l'édition du code, il est conseillé de ne pas trop dépasser cette limite.

1.1.2 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>       /
   ----------------------------------------------------------------------
*)
```

3. Ou `rlwrap ocaml` si vous n'avez pas le package `ledit`.

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.

1.1.3 Utilisation de l'interprèteur OCaml

L'interprèteur OCaml (sur la figure, en bas à droite de l'écran) affiche le caractère d'invite (prompt) suivi du curseur : **#** _

Lorsque vous taperez une expression terminée par deux points virgules (`;;`), celle-ci sera évaluée par l'interprèteur, qui en retour affichera un résultat. Tapez par exemple l'expression suivante dans la fenêtre de l'interprèteur 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 un caractère d'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 du système :

```
# x * 3
  ;;
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 ↑ (si vous avez lancé OCaml via `ledit` ou `rlwrap`).

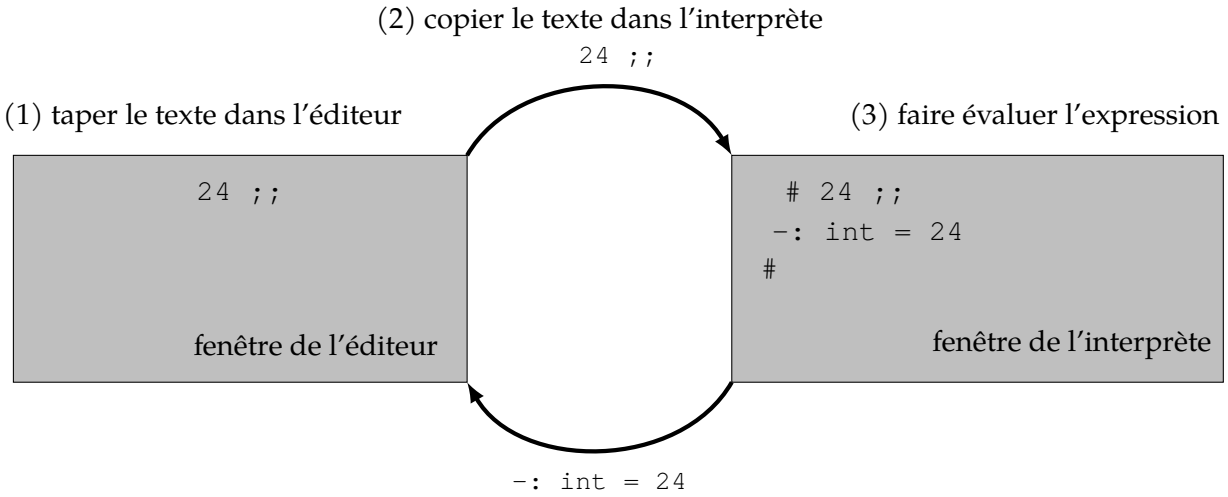
Par ailleurs, *lorsque vous quitterez l'interpréteur OCaml, tout ce que vous avez tapé sera perdu.* Il est donc nécessaire d'utiliser à la fois l'interpréteur OCaml et l'éditeur.

1.1.4 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 il sera sans doute préférable d'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, etc.).
2. Transférer le texte dans la fenêtre de l'interpréteur afin de l'évaluer.
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 : `(* ... *)`.

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



Pour réaliser les opérations de transfert entre les deux fenêtres, on utilise classiquement le copié-collé. Sous Linux, celui-ci peut se faire efficacement en utilisant uniquement la souris. Dans l'exemple suivant, on copie une expression simple de l'éditeur vers l'interprète.

1. Sélectionner l'expression, point-virgules compris ; dans le cas présent : `24 ;;`. Au relachement du clic gauche, le texte a été automatiquement copié.
2. Déplacer le pointeur de la souris dans la fenêtre de l'interpréteur.
3. Effectuer un clic droit[4] ; le texte précédemment sélectionné est alors collé (à l'endroit indiqué par le curseur).
4. Valider en appuyant sur Entrée. L'interpréteur OCaml affiche sa réponse.
5. Transférer cette réponse dans le compte rendu en utilisant la méthode inverse (copier dans l'interpréteur et coller dans l'éditeur).

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

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

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.

Remarque Pour aller plus vite, il est possible d'étendre la sélection jusqu'au retour à la ligne, ce qui évite d'avoir ensuite à taper sur Entrée dans l'interpréteur.

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";;
```

4. Selon l'éditeur ou le système d'exploitation utilisé, il peut s'agir plutôt d'un clic central, ou d'un clic des deux boutons si la souris ne comporte pas de troisième bouton

1.1.5 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";;`
NB : il faut taper le `#` ci-dessus **en plus** du `#` de l'interpréteur ; `#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.

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
  Exemples et propriétés : ...
  Algorithme : utilisation de +
*)
let somme3 (a:int) (b:int) (c:int) : int =
  a + b + c
;;
(* Tests : *)
assert ((somme3 1 2 3) = 6) ;;           (* cas général *)
assert ((somme3 (-1) 2 (-1)) = 0) ;;     (* entiers négatifs *)
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).

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.

1.1.6 Fin de la séance

- N'oubliez pas avant de quitter l'éditeur de vérifier que le compte-rendu est syntaxiquement correct (directive `#use`). Pour sortir de l'interpréteur OCaml taper la commande `#quit ; ;` ou plus rapidement `^D` (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

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 recopiant 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 `/.` 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;
```
Et si on essayait de les enchainer ...

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 implantations, 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 :

```
(*
  2.2.2
  |SPÉCIFICATION
  |max2 : maximum de deux entiers
  | - Profil      : max2 : int -> int -> int
  | - Sémantique : (max2 a b) est le plus grand des deux entiers a et b
  | - Exemples et propriétés :
  |     (a) (max2 3 4) = 4
  |     (b) (max2 4 3) = 4
  |     (c) ∀ a ∈ ℤ, (max2 a a) = a
  |RÉALISATION
  | - Algorithme : le maximum est à mi-distance à droite du milieu
  | - Implémentation :
*)
let max2 (a: int) (b: int): int =
  ( (a+b) + abs(a-b) ) / 2 ;;
```

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, OCaml répond en donnant le profil de la fonction (c'est-à-dire `nom_de_la_fonction : types des paramètres -> type_du_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_de_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 *maximum de 3 entiers*

Profil $max3 : \ Int \ \rightarrow \ Int \ \rightarrow \ Int \rightarrow Int$

Sémantique : $(max3\ a\ b\ c)$ *est le maximum des 3 nombres* a, b, c *distincts deux à deux*

Réalisation basée sur une analyse en fonction 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 : $(max3\ a\ b\ c) = \begin{cases} \textbf{si} \ (a > b) & \begin{cases} \textbf{si} & a > c \quad \textbf{rendre} \ a \\ \textbf{si} & c \geq a \quad \textbf{rendre} \ c \end{cases} \\ \textbf{si} \ (b \geq a) & \begin{cases} \textbf{si} & b > c \quad \textbf{rendre} \ b \\ \textbf{si} & c \geq b \quad \textbf{rendre} \ c \end{cases} \end{cases}$

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 *max3* pour calculer le maximum de 3 entiers.

Algorithme 3 : $(max3\ a\ b\ c) = \ldots\ldots\ (a, \ldots\ldots\ (b,c))$

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

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

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 TP2 Nommer une expression

Q1. Pour chacune des expressions suivantes, observer les réactions de l'interprète. 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;;
```

2.4 TP2 Retour sur l'ordre d'évaluation d'une expression

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

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

puis évaluer les expressions :

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

Précisez 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);;
```

Evaluez 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 ; ;
```

Evaluez 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

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

SPÉCIFICATION

Profil *moyenne* : $\mathbb{Z} \to \mathbb{Z} \to \mathbb{R}$
Sémantique : *calcule la moyenne du couple d'entiers donné*

RÉALISATION

Algorithme : *la moyenne est le milieu du segment* $[a,b]$
Implémentation

```
let moyenne (a:int) (b:int) : float =
  (a +. b) /. 2.0
```

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

Q1. Devinez le profil de cette fonction et vérifiez votre réponse en donnant `float_of_int ; ;` à l'interprète. Évaluez les expressions `float_of_int 3` et `float_of_int 3.2`.

Q2. Donnez une ou plusieurs réalisations correctes de la fonction moyenne en respectant la spécification ci-dessus. Tester la fonction `moyenne` avec des jeux d'essais significatifs.

Q3. En s'inspirant du nom de la fonction `float_of_int`, devinez 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.

Exemple
La moyenne olympique des quatre nombres $10, 8, 12, 24$ est 11 ; celle des nombres $12, 12, 12, 12$ est 12.

SPÉCIFICATION *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 *moyenne olympique*

Algorithme : (*max2 x y*) : *milieu de x et y + la moitié de la distance entre x et y*
min2 : *utilisation de max2*
max4 : *utilisation de max2*
min4 : *utilisation de min2*
moyol : *utilisation des fonctions prcdentes*

Q1. Implémenter les fonctions *min2*, *max2*, *min4*, *max4* 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 *min2*, *max2*, *min4*, *max4* et *moyol* avec des jeux d'essai significatifs.

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) Implantez 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) Testez la fonction `estJourDansMois_2`.

e) Que répond OCaml à `(estJourDansMois_2 18 13)`, `(estJourDansMois_2 0 4)` ? Comment interprétez-vous ces résultats ?

Q2.

a) Donner une troisième implantation 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, y compris pour `18 13` et `0 4`.

Q3.

a) Donner une nouvelle implantation (n°4) du type `Mois` en utilisant un *type énuméré*. Modifier l'implantation de `estJourDansMois_3` en conséquence.

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

c) Que répond OCaml à `(estJourDansMois_4 18 13), (estJourDansMois_4 0 4)` ?

d) Quel jeu d'essai correspond au jour 0 du mois d'avril pour `estJourDansMois_4` ? Tester `estJourDansMois_4` sur ce jeu d'essai. Comment interprétez-vous les réponses de OCaml ?

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`.

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 (×) des types de chacune des composantes.

Q1. 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

Q2. Compléter l'implantation de l'ensemble Intervalle :

```
type intervalle = ..... (* { (bi,bs) ∈ ℝ² tels que bi ≤ bs } *) ;;
```

Q3. 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) ;; let x1 = -40 ;;
```

puis `assert ((precede x1 interv_1)=true);;`

Q4. Implémenter la fonction *precede*. Remarquer comment la définition de *intervalle* facilite la compréhension du profil de la fonction.

Q5. 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

Q6. Donner un ensemble significatif de jeux d'essai permettant de tester les fonctions *coteAcote* et *chevauche* (cf exercice polycopié TD). Pour cela, fixer la valeur d'un intervalle I1, par exemple `let cst_I1 = (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...) = ....`

Q7. Implanter 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 *somme des chiffres*

Profil $sc : \{0 \dots 9999\} \rightarrow \mathbb{Z}$

Sémantique : *$sc(n)$ est la somme des chiffres de la représentation de n en base 10.*

Exemple : $sc(9639) = 27$

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 *somme des chiffres*

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 *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`.
Implémentation `/!\ conditions utilisation sc(n) : n ∈ ..........`

```
let sc ............... =
    ....................................
    ....................................
    ....................................
    ....................................
```

Q3. Vous pouvez 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*. 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;;`.

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 donnéé : par exemple, (3, 4) est la permutation ordonnée de (4, 3). On introduit ainsi une fonction nommée *poCoupleE* :

SPÉCIFICATION

Profil *poCoupleE* : $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \times \mathbb{Z}$

Sémantique : *Soit* $(x',y') = poCoupleE(x,y)$. (x',y') *est la permutation ordonnée de* (x,y), *définie comme suit :*

$$(x',y') = \begin{cases} (x,y) & si\ x \leq y, \\ (y,x) & sinon. \end{cases}$$

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 ;;
2.0 < 3.0 ;;
```

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 =
     if x <= y then (x, y) else (y, x)
```

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

On observe que la réponse du système utilise la même convention, les noms des types donnés initialement étant ignorés (`'un_type * 'un_type` dans l'exemple) et remplacés par des lettres (`'a * 'a`). 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 α au lieu de `'a` (β au lieu de `'b`, ...) :

SPÉCIFICATION

Profil *poCouple* : $\alpha \times \alpha \rightarrow \alpha \times \alpha$

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

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

poCouple (3, 14.5) ;;
```

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

DÉFINITION (TYPE ABSTRAIT DE DONNÉES) *Lorsqu'on définit un type de donnée (par ex. le type duree) il est intéressant de définir les opérations associées à ce type (les constructeurs, les sélecteurs, les 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. [1]

Notre objectif ici 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 *Durée* et des fonctions associées

DÉFINITION D'ENSEMBLES

déf $jour = \mathbb{N}$

déf $heure = \{0 \dots 23\}$

déf $minute = \{0 \dots 59\}$

déf $seconde = \{0 \dots 59\}$

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

Spécifications des fonctions sur les durées

1. Nous utiliserons le type `list` de OCaml. C'est un *type abstrait de données*. Vous l'utiliserez ainsi que les opérations associées sans qu'il soit nécessaire de savoir comment les `list` sont représentées dans la machine.

SPÉCIFICATION *constructeurs*

Profil *sec_en_duree* : $\mathbb{Z} \rightarrow$ *duree*
Sémantique : *construit une donnée de type duree à partir d'un nombre de secondes*

Profil *vec_en_duree* : $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow$ *duree*
Sémantique : *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 *SÉLECTEURS*

Profil *jour* : *duree* $\rightarrow \mathbb{Z}$
Sémantique : *sélectionne la partie jour d'une durée*

Profil *heure* : *duree* $\rightarrow \mathbb{Z}$
Sémantique : *sélectionne la partie heure d'une durée*

Profil *minute* : *duree* $\rightarrow \mathbb{Z}$
Sémantique : *sélectionne la partie minute d'une durée*

Profil *seconde* : *duree* $\rightarrow \mathbb{Z}$
Sémantique : *sélectionne la partie seconde d'une durée*

SPÉCIFICATION *CONVERSION*

Profil *duree_en_sec* : *duree* $\rightarrow \mathbb{Z}$
Sémantique : *convertit une durée en un nombre de secondes*

SPÉCIFICATION *OPÉRATIONS*

Profil *som_duree* : *duree* $\rightarrow$ *duree* $\rightarrow$ *duree*
Sémantique : *effectue la somme de deux durées*

SPÉCIFICATION *PRÉDICATS*

Profil *eg_duree* : *duree* $\rightarrow$ *duree* $\rightarrow \mathbb{B}$
Sémantique : *teste l'égalité de deux durées*

Profil *inf_duree* : *duree* $\rightarrow$ *duree* $\rightarrow \mathbb{B}$
Sémantique : *teste la relation inférieur strict sur les durées*

Réalisation informatique du type *Durée* et des fonctions associées

Q1. Compléter les définitions de type ci-dessous :

DÉFINITION DE TYPES

```
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

Algorithme : *utilisation de div réalisée dans un exercice précédent.*

Implémentation ...

On considère la fonction suivante :

SPÉCIFICATION

Profil *nb_total_sec* : .....

Sémantique : *nb_total_sec*(j,h,m,s) *est le nombre total de secondes que représentent* j *jours* $+$ h *heures* $+$ m *minutes* $+$ s *secondes*

Exemples :

1. *nb_total_sec*$(1,0,0,0) = \ldots\ldots$
2. *nb_total_sec*$(0,1,0,0) = \ldots.$
3. *nb_total_sec*$(.,.,1,..) = 90$
4. *nb_total_sec*$(.,.,0,\ldots) = 120$

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

Q3. En déduire une réalisation de *vec_en_duree* :

RÉALISATION

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 .*

Implémentation ...

Q4. Implanter les sélecteurs.

Q5. Réaliser *duree_en_sec* :

RÉALISATION

Algorithme : *On réutilise l'algorithme de la fonction nb_total_sec et les sélecteurs.*

Implémentation ...

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

RÉALISATION *som_duree* (opération)

Algorithme 1 : *On réutilise les fonction duree_en_sec et sec_en_duree.*

Implémentation 1 ...

Algorithme 2 : *On fait l'addition des vecteurs composante par composante en tenant compte des retenues.*

Implémentation 2 ...

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

RÉALISATION

Implémentation 1: en utilisant la fonction *duree_en_sec*

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

Implémentation 3: en comparant les vecteurs composante par composante

Implémentation 4: en utilisant les sélecteurs

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

RÉALISATION

Algorithme 1 : *En utilisant la fonction duree_en_sec.*

Algorithme 2 : *En utilisant des expressions conditionnelles imbriquées.*

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

DÉFINITION D'ENSEMBLES

déf *chiffre* = {'0' ... '9'}
déf *base10* = {0 ... 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. Implanter les types *chiffre* et *base10*, puis les fonctions *chiffreVbase10* et *base10Vchiffre*.

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 ... 9} et les 6 premières lettres de l'alphabet {*A* ... *F*}.

On définit les types *carhex* et *base16* :

DÉFINITION D'ENSEMBLES

déf *carhex* = {'0' ... '9'} ∪ {'*A*' ... '*F*'}
déf *base16* = {0 ... 15}

Q1. Donner le profil et la sémantique d'une fonction *carhexVbase16*, qui associe un élement de *base16* à un *carhex*.

Q2. Donner une implantation naïve des types *carhex* et *base16* sous forme respectivement d'un caractère et d'un entier.

Q3. Réaliser et implanter 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 implantant *carhex* et *base16* 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 :

DÉFINITION D'ENSEMBLES

déf $hexa4 = \{0 \dots 16^4 - 1\}$

Q1. Définir le type `hexa4`.

Q2. Dans cet exercice nous nous restreignons à des entiers qui peuvent être codés sur 4 caractères hexadécimaux. Définir le type `rep_hexa4` représentant les quadruplets de caractères hexadécimaux *carhex* à l'aide d'un type produit.

Q3. Donner le profil et la sémantique d'une fonction *ecriture_hex*, qui convertit un nombre de *hexa4* en un quadruplet de caractères hexadécimaux.

Q4. Spécifier et réaliser la fonction *base16Vcarhex* , inverse de la fonction *carhexVbase16* 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* en respectant la donnée fournie (entier compris entre 0 et $16^4 - 1$).

Deuxième partie

DÉFINITIONS RÉCURSIVES

Chapitre 3

Fonctions recursives sur les entiers

Sommaire

3.1 TP5 factorielle

On considère la fonction factorielle, spécifiée ainsi :

SPÉCIFICATION *factorielle*

Profil *factorielle* : $\mathbb{N} \to \mathbb{N}$

Sémantique : *fac* (n) *est l'entier* $n \times (n-1) \times \cdots \times 2 \times 1$ *; par convention* $fac(0) = 1$.

Q1. La fonction factorielle peut être définie par des équations récursives :

RÉALISATION

Définition récursive de la fonction par des équations

$$(1) \quad fact\ (0) = 1$$
$$(2) \quad fact\ (p+1) = (p+1) \times fact\ (p)$$

Algorithme : Codez cette fonction en OCaml en utilisant la construction `let rec fact ...` puis le filtrage. Noter dans l'implantation de la question **le changement de variable** lors du passage de l'équation (2) à l'implémentation.

Q2. Testez `fact` sur des données respectant les contraintes. Observez l'évaluation en traçant la fonction (`#trace fact`). Supprimer la trace de la fonction (`#untrace fact`).

Q3. Testez `fact` sur des données ne respectant les contraintes comme par exemple `fact (-1) ;;`. Expliquez. Vous pouvez essayer de tracer à nouveau si vous ne comprenez pas.

Q4. Que se passe-t-=il si on oublie le `rec` ? Implémentez une fonction appelée `fact2` identique à `fact` en la nommant avec `let fact2 ...` au lieu de `let rec` (attention à bien utiliser `fact2` pour l'appel récursif). Observez le message de l'interprète.

Contrairement aux équations récursives, l'ordre des motifs dans une construction filtrante est important.

Q5. Saisissez et complétez l'implémentation suivante dans laquelle on inserse les deux cas du filtrage. Observez les messages de l'interprète lors de la définition et lors du test de cette fonction. Expliquez pourquoi on obtient ces message quelle que soit la donnée.

```
let rec fact3 ... =
  match n with
    | pp1  -> ...
    | 0  -> ...
```

Q6. Donner une autre implémentation $fact3$ de factorielle en termes d'une expression conditionnelle.

3.2 TP5 somme d'entiers bâton

On définit le type *natPeano* qui représente un entier. Pour cela on dispose d'une constante Z pour représenter 0 et d'un constructeur S pour successeur.

Q1. Définir le type *natPeano* en OCaml.

Par ailleurs on considère une autre représentation des entiers par une séquence de bâtons. On définit pour cela le type *natBaton*. Le constructeur Nil reprisente la séquence vide et le constructeur de séquence est noté Cons. Un bâton est le caractère `'|'`.

Q2. Définir le type *natPaton* en OCaml.

On souhaite effectuer la somme de deux entiers en écriture bâton ; par exemple, la somme de || et ||| donnera |||||.

Q3. Spécifier, réaliser puis tester les fonctions suivantes :

- *addP* : une fonction additionnant deux entiers de Péano,
- *peanoVbaton* : une fonction convertissant un entier de Péano en un entier bâton.
- Tester ces fonctions.

Définissons une fonction appelée *somme_baton* qui donne la somme bâton de deux entiers de Peano.

Q4.

a) Donner la spécification de *somme_baton*.

b) Donner la réalisation non récursive de cette fonction.

c) Tester la fonction `somme_batons` sur un jeu d'essais significatif.

Avant de pouvoir additionner des bâtons, il faut être capable de transformer des entiers bâton en entiers de Peano.

Q5.

a) Donner la spécification de la fonction *batonsVpeano*.

b) Donner les équations récursives de sa réalisation.

c) Implémenter cette réalisation.

d) Discuter sur le message d'avertissement de l'interprète.

e) Tester cette fonction.

On peut maintenant définir la somme de deux entiers bâtons.

Q6. Implémenter la fonction *somme_natbatons* qui calcule la somme de deux entiers bâtons, et la tester sur un jeu d'essais significatif.

3.3 TP5 quotient et reste de la division entière

3.3.1 Quotient et reste

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

SPÉCIFICATION

Profil $(\_\ /\ \_), (\_\ mod\ \_) : \mathbb{N} \to \mathbb{N}^* \to \mathbb{N}$
Sémantique : *a*/*b* : *le quotient de la division entière de a par b*
a mod b : reste de la division entière de a par b

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

Q1.

a) Réaliser la fonction *quotient* qui calcule le quotient de la division de deux entiers.

> INDICATION *Comme on utilise l'algorithme des soustractions successives, il faut exprimer* $quotient(a, b)$ *en termes de* $(quotient\ (a - b)\ b)$ *(ce n'est pas toujours possible ; quand est-ce possible ?)*

b) La tester sur un jeux de tests pertinants.

c) Observer la réaction de l'interprète quand on ne respecte pas la contrainte sur *b*. Expliquer.

d) Observer la trace de l'appel `(quotient 17 5)`. Structurer la trace fournie par l'interprète en l'*indentant* de façon à mettre en évidence l'emboîtement des appels récursifs.

Q2. Réaliser puis tester la fonction *reste* dont la spécification a été donnée plus haut (*mod*), en utilisant l'algorithme des soustractions successives (donner les équations récursives puis l'implémentation).

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

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

Q3.

a) Donner la sémantique d'une fonction *qr*1 calculant le couple quotient/reste d'une division.

b) En donner la réalisation (en utilisant les deux fonctions `reste` et `quotient`).

c) Implémenter cette fonction.

Q4. Observer la trace de `(qr1 17 5)` ; il faut aussi tracer les deux fonctions utilisées par `qr1`. Comparer les listes des appels récursifs de ces deux fonctions. Qu'en conclure ?

Q5. En s'inspirant des réalisations de *quotient* et de *reste*, donner l'implémentation récursive *qr*2 qui effectue en même temps le calcul du quotient et du reste, puis la tester.

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

Q6. Observer maintenant la trace de `(qr2 17 5)`. Conclure.

Chapitre 4

Fonctions récursives sur les séquences

Sommaire

4.1 TP6 flux de circulation sur une voie routière

Quelques fonctions *classiques* 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 donnée, 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 d'entiers naturels : 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.

Dans ce qui suit, nous utilisons donc le type suivant :

DÉFINITION D'UN ENSEMBLE *relevés d'observation*

déf $releve = séq\,(\mathbb{N})$, où $\mathbb{N}$ est l'ensemble des entiers naturels

Sémantique : *Le ième élément d'un relevé est le nombre de véhicules observés pendant le ième jour de la période associée au relevé.*

Q1. Définir en OCaml les types `seq_int` et `releve` implémentant respectivement les ensembles *séq* $(\mathbb{N})$ et *releve*.

4.1.1 Nombre de jours sans véhicules

On souhaite disposer d'une fonction déterminant, pour un relevé donné, le nombre de jours pendant lesquels aucun véhicule n'est passé devant le point d'observation (flux nul).

Q2. Donner la spécification de la fonction *nbj_sans*.

Réalisation récursive

Q3. Définir 4 constantes de types `releve` nommées :

- `cst_R0` : test du cas où l'appareil de détection des voitures était en panne,
- `cst_R1` : test du cas où il n'y a pas de jours à flux nul,
- `cst_R2` : test du cas où il y a un jour à flux nul,
- `cst_R3` : test du cas où il y a plusieurs jours à flux nul, par exemple 4.

NB : `csr_R1`, `csr_R2` et `csr_R3` peuvent être déclinées de plusieurs façons. Ne pas hésiter à compléter ce jeu de constantes.

Q4. Réaliser la fonction *nbj_sans*, et la tester.

Généralisation

Maintenant, on veut connaître le nombre de jours pendant lesquels x véhicules sont passés devant le point d'observation.

Q5. Donner la spécification de la 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.

a) Donner les équations récursives de *nbj_avec*.

b) Tester l'implémentation ci-dessous et observer la réponse de l'interprète.

```
let rec nbj_avec  (x : int) (r : releve) : int =
  match (x,r) with
    | (_,Nil)           -> 0
    | (x,Cons(x,fin))   -> 1 + (nbj_avec x fin)
    | (x,Cons(pr,fin)) -> (* 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,Cons_int(x,fin))` est donc interdit.

c) Corriger et tester l'implémentation ci-dessus.

Q7. En déduire une implémentation non récursive de *nbj_sans*, et tester cette nouvelle implémentation.

4.1.2 Appartenance

La relecture de l'exercice de TD « Un élément apparait-il dans une séquence » peut aider.
On souhaite savoir si un relevé comporte au moins un certain flux donné.

Q8. Donner la spécification de la fonction $(flux\_app\ x\ r)$ qui renvoie vrai si et seulement si x apparaît dans r.

Réalisation d'une fonction booléenne

Q9. Réaliser la fonction *flux_app* en en donnant une implémentation par filtrage sans utiliser d'expression conditionnelle, et tester cette implémentation.

Q10. Observer, puis indenter la trace de l'évaluation de l'expression : `flux_app 194 Cons_int(125 , Cons_int( 142 , Cons_int( 253 , Cons_int( 194 , Cons_int( 155 , Cons_int( 45 , Cons_int( 62, Nil_int)))))));;`. Combien d'éléments de la séquence sont-ils comparés à la valeur 194 ?

Q11. Généraliser le résultat pour l'évaluation de l'expression (*flux_app x r*) en examinant les diverses situations significatives.

Généralisation

Les réalisations de *flux_app* ne font intervenir que l'opération = sur les entiers (outre les connecteurs logiques). Comme ce symbole est polymorphe (voir partie 1), cette fonction peut être définie de manière générique sur toute séquence d'éléments.

Q12. Donner la spécification de la fonction *app* qui teste l'appartenance d'un élément à une séquence.

Q13. Implémenter cette fonction polymorphe, en introduisant un type général de séquence : `type 'elt seqG`. Tester cette fonction en l'appliquant à des listes d'éléments de types variés.

Valeurs mini et maxi des flux journaliers

On souhaite connaître les extréma des flux journaliers d'un relevé non vide. Ce sont les valeurs minimum et maximum de la séquence d'entiers représentant le relevé.
La relecture de l'exercice de TD « Maximum de n entiers » peut aider.

On considère maintenant des relevés non vides :

DÉFINITION D'UN ENSEMBLE

déf $releveNV \quad = \quad releve^* = releve \setminus \{Nil\}$

Q14. Implémenter le type des relevés non vide `releveNV`.

Q15. Spécifier les fonctions *fluxmin* et *fluxmax*.

Q16. Réaliser la fonction *fluxmax*.

Q17. Que signifie l'avertissement de l'interpréteur sur ces deux dernières fonctions ? Pourquoi cet avertissement ? Que doit en penser le programmeur sérieux qui a spécifié proprement sa fonction avant de la réaliser ?

Une des façons de résoudre ce problème consiste à remplacer `Nil`, constructeur sans argument, par un constructeur unaire *Single* (pour « singleton »), de type $\mathbb{N} \rightarrow releveNV$. On ne peut alors construire que des séquences non vides.

Q18. Définir le type `releveNV2` de cette façon.

Q19. Réimplémenter la fonctions *fluxmax* avec le type `releveNV2`.

Flux observé au jour J

Étant donnés un relevé non vide r et un entier positif $j \leq |r|$, on veut connaître le flux observé le jour j.

Q20. Spécifier la fonction $(flux\_obs\ r\ j)$, qui renvoie le flux observé le jour j du relevé r.

Q21. Réaliser $flux\_obs$ et la tester.

4.2 TP7 polynômes

On considère des polynômes à coefficients entiers comme par exemple $-3x^4 + 2x^2 + 10x - 12$. Dans cet exercice, $3x^4$ (noté aussi `3x4`) signifie 3 fois x puissance 4.

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*.

Dans cette représentation, *les monômes à coefficient nul ne sont pas représentés*. 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* (α) : `'a list`.

Q1. Définir les ensembles *monôme* et *polynôme*, ainsi que les types OCaml correspondants.

Q2.

a) Définir des constantes `cst_M1`, `cst_M2`, `cst_M3`, `cst_M4` et `cst_M5` représentant les monômes : 10 (monôme constant), $7x$, $-3x^2$, x^4 et $-x^4$.

b) En réutilisant `cst_M1`, `cst_M2`, `cst_M3`, `cst_M4` et `cst_M5`, définir des constantes `cst_P1`, `cst_P2`, `cst_P3`, `cst_P4` et `cst_P5` – qui serviront de jeux d'essais pour la suite – représentant les polynômes : 10 (polynome constant), $7x + 10$, $-3x^2 + 7x$, $x^4 - 3x^2 + 7x$, $-x^4$.

Q3. Définir une fonction nommée *derivMono* qui, étant donné un monôme en x, lui associe le monôme dérivé par rapport à x. Par exemple, le monôme dérivé de `cst_M4` est $4x^3$. On conviendra que la puissance du monôme nul est 0. Tester cette fonction.

Q4. Définir une fonction, nommée *derivPoly* qui, étant donné un polynôme en x, lui associe le polynôme dérivé par rapport à x. Par exemple, le polynôme dérivé de `cst_P4` est $4x^3 - 6x + 7$.

Le type Caml *unit* est particulier : il ne contient qu'un élément, noté `()`. Mathématiquement, on peut donc le noter sous forme du singleton `{()}`. Ce type est utilisé pour les procédures, c-a-d les « fonctions » qui ne renvoient pas de résultat. Remarquons que dire « les procédures ne renvoient rien » est un abus de langage : elles renvoient toujours une seule et même valeur : `().` `{()}` est également utilisé pour les fonctions (ou procédures) ne prenant pas d'argument, d'où la notation `()`.

La construction `assert(expr)` permet d'assurer que l'expression booléenne `expr` est vraie, auquel cas `assert` renvoie la valeur `()` ; sinon, une erreur est signalée.

Q5. Tester *derivPoly* sur les polynômes `[cst_M1]` à `[cst_M5]`, puis `cst_P1` à `cst_P4` en utilisant `assert`. Dans le cas présent, si `assert` ne renvoie pas `()`, c'est qu'il y a une erreur dans *derivPoly*.

Q6. Définir une fonction, nommée *sommePoly*, qui étant donnés deux polynômes, leur associe le polynôme somme. Par exemple, la somme des polynômes `cst_P4` et `cst_P5` est `cst_P3`. On tiendra compte du fait que les séquences représentant les polynômes sont en ordre décroissant des puissances et que les monômes nuls ne sont pas représentés.

Q7. Tester sommePoly, par exemple grâce aux expressions suivantes :

CODE CAML

```
sommePoly cst_P1 cst_P1 ;;
sommePoly cst_P1 cst_P2 ;;
sommePoly cst_P2 cst_P1 ;;
assert (sommePoly cst_P4 cst_P5 = cst_P3 &&
  sommePoly cst_P5 cst_P4 = cst_P3 );;
```

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

CODE CAML

```
assert (derivPoly (sommePoly cst_P1 cst_P1) =
sommePoly (derivPoly cst_P1) (derivPoly cst_P1));;

derivPoly (sommePoly cst_P1 cst_P2) =
sommePoly (derivPoly cst_P1) (derivPoly cst_P2);;

derivPoly (sommePoly cst_P2 cst_P1) =
sommePoly (derivPoly cst_P2) (derivPoly cst_P1);;

derivPoly (sommePoly cst_P4 cst_P5) =
sommePoly (derivPoly cst_P4) (derivPoly cst_P5);;
```

4.3 TP8 somme d'une suite de nombres

Il est impératif de tester les fonctions au fur et à mesure de leur réalisation.

On considère des textes (séquences de caractères) constitués de chiffres et d'espaces. 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. 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 types suivants :

DÉFINITION D'ENSEMBLES

déf $chiffre = \{0,1,2,3,4,5,6,7,8,9\}$
déf $chiffreCar = \{'0','1','2','3','4','5','6','7','8','9'\}$
déf $nombre = séq\ (chiffreCar) \setminus \{[\,]\}$
déf $txtnb = séq\ (car)$, où les car sont des *chiffreCars* ou des espaces

Q1. Définir ces ensembles en OCaml.

Q2. Donner les spécifications des fonctions suivantes :

- *somme_txtnb* qui donne la somme de nombres réprésentés en tant que *txtnb*,
- *les_nb* qui donne la séquence de nombres réprésentés en tant que *txtnb*,
- *snbVsnat* qui a partir d'une séquence de nombres renvoie une séquence d'entiers,
- *somme* qui renvoie la somme d'une séquence d'entiers.

Q3. Étape 3
Réaliser la fonction *somme*.

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

- *ccVc* qui associe un *chiffre* au *chiffreCar* passé en argument,
- *nbVnat* qui associe un entier naturel au nombre passé en argument

  INDICATION *Le découpage à droite est plus indiqué ici.*

- *snbVsnat* qui converti une séquence de nombres en une séquence d'entiers naturel.

Q4. Étape 2
Définir ces fonctions.

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

- *sup_esp* qui supprime les espaces en début de *txtnb*,
- ******* $prnb\_r(txt) = (nb, reste)$ tel que *nb* est le premier nombre de *txt* (s'il existe) et *reste* est le reste de *txt*. Exemple :

```
prnb_r(['1';'2';'3';' ';'4';'5';' ';'6']) =
  (['1';'2';'3'], [' ';'4';'5';' ';'6'])
```

- *les_nb* qui extrait une séquence de nombre d'un txtnb.

Q5. Étape 1
Définir ces fonctions.

Q6. Définir *somme_txtnb*.

Troisième partie

ORDRE SUPÉRIEUR

Chapitre 5

Utilisation/conception de fonctions d'ordre supérieur

Sommaire

5.1 TP9 Curryfication

Q1. Implémenter la fonction `add` de profil `int*int -> int` qui additionne les deux composantes d'un couple d'entier naturel.

Q2. Tester `add` sur les couples suivants : $(3,0)$, $(3,5)$, $(5,3)$, 3.

Remarquer que sur le dernier exemple, il n'est pas possible d'appliquer `add` partiellement sur un seul entier : `add` attend un argument de type couple d'entiers.

Q3. Implémenter une fonction `addC` de profil `int -> int -> int` et la tester. Comparer les deux implémentations et résultats de l'interpréteur. Quelles sont les différences ?

Comme on le voit sur le dernier exemple, et contrairement à `add`, il est possible d'appliquer partiellement `addC`.

Q4. Quelle est la nature de `(addC 3)` ? Peut-on appliquer `(addC 3)` ? Quelle(s) différence(s) y-a-t'il entre `(addC 3 5)` et `((addC 3) 5)` ?

Une autre façon de comprendre `addC` est de dire que pour tout entier `a`, `addC(a)` est une fonction qui pour tout entier b renvoie la somme de `a` et `b`.

Ceci se voit bien si on implémente `addC` de la façon suivante :

CODE CAML

```
let rec addCbis (a: int): int -> int =
  if a = 0 then
    function b -> b
  else
    function b -> 1 + (addCbis (a-1) b)
```

`addCbis` est une fonction qui teste son premier argument : si `a` est nul, on renvoie la fonction constante anonyme égale à `b` (`function b -> b`)[1]. Sinon, on renvoie la fonction anonyme qui à `b` associe `1 + (addC (a-1) b))` (`function b -> 1 + (addC (a-1) b)`). Cette implantation est parfaitement équivalente à la précédente, mais insiste sur la possibilité d'appliquer `addC` partiellement.

Q5. Évaluer et tester `addCbis`.

5.2 TP9 Composition de fonctions

Mathématiquement, on note $\circ$ la composition de fonctions. Pour toutes fonctions f et g telles que g soit définie sur l'ensemble des valeurs prises par f,

$$(g \circ f)(x) \stackrel{def}{=} g(f(x))$$

Q1. Spécifier puis implémenter la composition.

Q2. Définir une fonction `incr` qui ajoute 1 à un entier. En déduire l'implémentation d'une fonction `plus_deux` qui ajoute 2 à un entier, en utilisant uniquement les fonctions `comp` et `incr`.

Q3. Définir une fonction `fois_deux` qui multiplie un entier donné par 2. Puis, en utilisant les fonctions `comp`, `incr` et `fois_deux`, définir la fonction `f1` qui a chaque entier x associe $2 * x + 1$. Définir de manière similaire la fonction `f2` qui a chaque entier x associe l'entier $2 * (x + 1)$. Pour chacune de ces deux fonctions, proposer une variante utilisant `comp` et l'autre ne l'utilisant pas.

5.3 TP9 Dérivation de fonction

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

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

Q1. Spécifier puis réaliser une fonction `dérivée` qui dérive une fonction selon l'approximation décrite ci-dessus. Les arrondis sur les calculs entre flottants ont parfois des conséquences surprenantes ; ainsi, pour la valeur associée à `dx` vous pouvez choisir `0.001`.

Q2.

a) Calculer la dérivée de *sinus* en 0.

b) Calculer la dérivée seconde de *sinus* en 0.

c) Calculer la dérivée troisième de *sinus* en 0.

1. La notation `fun b -> b` existe également ; dans cet exercice, ces deux notations sont interchangeables.

Q3.

a) Implémenter la fonction `iteration`, qui prend un paramètre n et une fonction, et renvoie la fonction composée n fois.

INDICATION *Utiliser la fonction `comp` définie dans un exercice précédent.*

b) Appliquer `iteration` pour déterminer la dérivée troisième de sinus en 0 et retrouver le dernier résultat de la question précédente.

5.4 TP9 Quantificateurs

Q1. Spécifier puis implémenter la fonction d'ordre supérieur `non` qui étant donné un prédicat p renvoie la négation de p. Tester cette fonction en définissant un prédicat de votre choix.

Q2. Implémenter la fonction `pour_tous` qui teste si tous les éléments d'une séquence vérifient une propriété donnée.

Remarque Cette fonction existe dans la librairie standard : elle s'appelle `List.for_all`. Vous pouvez l'utiliser pour examiner son comportement.

Q3. Implémenterla fonction `il_existe` qui teste l'existence dans une séquence d'un élément vérifiant une propriété donnée.

Remarque Cette fonction existe dans la librairie standard : elle s'appelle `List.exists`. Vous pouvez l'utiliser pour examiner son comportement.

Q4. Donner une autre implantation – non récursive – de la fonction `il_existe` utilisant uniquement les fonctions `non` et `pour_tous`.

Considérons la liste `l` définie par `l = [1015; 4305; 728; 861].`

Q5. Tester à l'aide des fonctions précédentes si tous les entiers dans `l` sont des multiples de 7, puis s'il existe un multiple de 13 dans `l`.

5.5 TP10 Primalité et nombres de Mersenne

Les nombres de Mersenne sont les entiers de la forme $2^p - 1$, où p est un nombre premier. Le but des questions suivantes est de déterminer le premier nombre de Mersenne qui ne soit pas un nombre premier.

Q1. Définir une fonction *est_premier* qui teste si un entier supérieur ou égal à 2 est premier.

Q2. Implémenter une fonction *puiss* qui associe à un entier n l'entier 2^n.

Q3. Déterminer le premier nombre de Mersenne qui ne soit pas un nombre premier :

a) de façon récursive ;

b) en créant une liste de nombres premiers et en appliquant les fonctions `map` et `find`.

Comparer le nombre d'appel à la fonction `est_premier` dans les deux cas.

5.6 TP10 Tri générique par insertion

Le but de cet exercice est de réaliser une fonction de tri générique par insertion.

Cette fonction est d'ordre supérieur et son argument est un comparateur. Des comparateurs différents sur une même structure de donnée engendrent différentes instanciations de cette fonction de tri, et donc différents tris.

Q1. Définir une fonction polymorphe d'insertion `insertion_pos` d'un élément donné dans une séquence donnée s à une position donnée n. Contrainte : $0 \leq n \leq |s|$.

> **Remarque** Le premier élément d'une séquence est numéroté 0.

Q2. Définir une fonction `insertion` d'insertion d'un élément donné dans une liste déjà triée. Cette fonction utilise un comparateur `cmp` renvoyant un entier indiquant comment deux éléments x et y se comparent entre eux. Cet entier est nul si $x = y$, strictement positif si x est strictement supérieur à y et strictement négatif si x est strictement inférieur à y.

Q3. Spécifier et réaliser une fonction `tri_insertion` de tri générique par insertion d'une liste.

> INDICATION *Utiliser `fold_left`.*

Q4. Trier la liste d'entiers `[2; 3; 12; 3; 24; 1; 2; 4; 9; 6; 10]` dans l'ordre croissant et dans l'ordre décroissant.

Un relevé de notes est une liste de couples dont la première coordonnée est une chaîne de caractères et la seconde coordonnée un entier.

Q5. Après avoir défini deux comparateurs `cmp_notes` et `cmp_noms`, donner des expressions OCaml permettant de trier un relevé de notes par ordre croissant des notes puis par ordre alphabétique.

> **Indication** Pour `cmp_notes`, on pourra utiliser la fonction OCaml prédéfinie `compare` qui permet de comparer deux chaînes de caractère.

Considérons les points de coordonnées réelles dans le plan.

Q6. Définir le type `point` et la fonction `dist0` qui donne la distance d'un point du plan à l'origine.

Q7. Trier la séquence de points `[(0., 1.); (1., 0.); (0., 0.); (5.; -2.); (1., 1.)]` par ordre croissant de distance à l'origine.

Quatrième partie

STRUCTURES ARBORESCENTES

Chapitre 6

Structures arborescentes

Sommaire

6.1 TP11 Appropriation des notations

Q1. Définir un type arbre binaire d'entiers `ab_int`.

Q2. Implanter les deux arbres

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

en utilisant uniquement les constructeurs Caml et en définissant d'abord leurs sous-arbres.

Q3. Définir les constructeurs suivants :

- `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 implantation des deux arbres précédents en utilisant le plus possible les fonctions de construction `abS`, `abUNd`, `abUNg`.

6.2 TP11 Somme des entiers d'un arbre

Q1. Implanter la fonction `somme` qui à tout arbre binaire d'entiers associe la somme de tous les entiers de l'arbre.

Q2. Tester la fonction `somme`.

6.3 TP11 Hauteur d'un arbre binaire

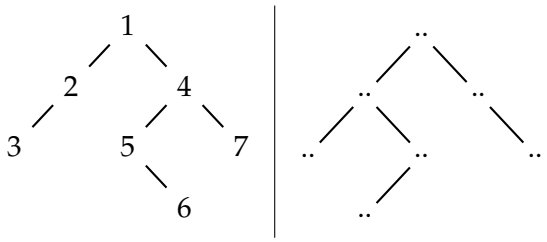
Q1. Définir le type polymorphe des arbres binaires.

Q2. On appelle hauteur d'un arbre la longueur maximale d'un chemin direct de la racine à un nœud de l'arbre. Implanter la fonction `hauteur` qui associe à tout arbre binaire sa hauteur.

Q3. Tester la fonction `hauteur` avec un arbre binaire d'entiers.

6.4 TP11 Arbres symétriques

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



Q1. Donner une définition (spécification, algorithme, terminaison) de la fonction *sym* qui donne le symétrique d'un arbre binaire.

Q2. Donner la définition (spécification, algorithme, terminaison) d'un prédicat *sontSym* qui indique si deux arbres sont symétriques.

INDICATION *Pour l'implémentation, réutiliser des fonctions des questions et/ou exercices précédents.*

6.5 TP12 Tri en arbre d'une sequence d'entiers

En informatique, 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 arbres binaires de recherche.

Par exemple :

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

Dans cet exercice, les ABR sont implémentés par le type suivant :

```
type abr =
  | V
  | N of abr * int * abr
```

Pour insérer un entier i 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 i doit être inséré, jusqu'à atteindre une feuille F ;
2. on ajoute un nouveau nœud d'étiquette i comme fils gauche ou comme fils droit de F selon que i est supérieur ou inférieur à l'étiquette de F.

Q1. (2pt) Implémenter une fonction `insert` qui, étant donnés un entier i et un ABR a, insère i dans a selon le principe décrit ci-dessus.
Dans cette question, l'utilisation de toute fonction intermédiaire est interdite.

Q2. (2pt) Implémenter une fonction `parcours_s` qui, étant donné un arbre, retourne la séquence d'entiers obtenue en utilisant le parcours symétrique (sous-arbre gauche, racine, sous-arbre droit). Par exemple :

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

Pour trier une liste d'entiers, on utilise une fonction auxiliaire `tri_aux` qui, étant donnés une liste `l` à trier et un ABR `a`, enlève le premier élément de la liste `l` et l'insère dans l'arbre à la bonne place, et cela tant qu'il y a encore des éléments dans `l`.

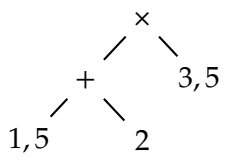
Q3. (2pt) Implémenter `tri_aux`.

Pour trier une liste d'entiers `l`, il suffit de démarrer avec la liste `l` et un arbre vide. On parcourt ensuite `l` en insérant tous les entiers dans le nouvel ABR qu'on construit à partir de l'arbre vide. On utilise enfin la fonction `parcours_s` pour parcourir de manière ordonnée l'arbre obtenu.

Q4. (2pt) Implémenter la fonction `tri` en suivant l'algorithme décrit ci-dessus.

6.6 TP12 Représentation des expressions arithmétiques

Les expressions arithmétiques peuvent se représenter sous forme d'arbres binaires dont les noeuds 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 noeuds : les feuilles, étiquetées par des nombres réels, et les noeuds 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 `expr`, nécessite donc deux constructeurs : un pour les feuilles et un pour les noeuds internes. Notons qu'il n'y a pas besoin de la notion d'arbre vide.

Q1. Définir un type appelé `operateur` permettant de symboliser les quatre opérations suivantes dans les entiers : addition, soustraction, multiplication et division. Définir ensuite le type `expr`.

Q2. Implémenter l'expression 1 + 1.

Q3. Implémenter la fonction `val_expr` telle que, pour tout arbre d'expression arithmétique `a`, `val_expr a` retourne la valeur de l'expression représentée par `a`.