

**UNIVERSITÉ
GRENOBLE
ALPES**

UFR IM²AG

**DÉPARTEMENT
LICENCE SCIENCES
ET TECHNOLOGIE**

LICENCE SCIENCES & TECHNOLOGIES, 1^{re} ANNÉE

UE [INF201](#)

ALGORITHMIQUE ET PROGRAMMATION FONCTIONNELLE

[2021-2022](#)

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

I	TYPES, EXPRESSIONS ET FONCTIONS	2
1	TP1 une séance d'expérimentation type	3
1.1	Déroulement d'une séance de TP	3
1.1.1	Installation d'un environnement pour OCAML	3
1.1.2	Édition du fichier de compte rendu	4
1.1.3	Utilisation de l'interpréteur OCAML	5
1.1.4	Utilisation conjointe de l'éditeur et de l'interpréteur OCAML	5
1.1.5	Construction du compte rendu de TP	7
1.1.6	Fin de la séance	8
2	Exercices et problèmes	9
2.1	TP1 Type et valeur d'une expression	10
2.1.1	Expressions basiques	10
2.1.2	Opérateurs de comparaison	10
2.1.3	Fabrication de jeux d'essai	11
2.1.4	Définition d'une constante	11
2.1.5	Expressions conditionnelles	11
2.2	TP1 Maximum entre plusieurs entiers	12
2.2.1	Spécification, réalisation d'une fonction	12
2.2.2	Utilisation d'une fonction	12
2.2.3	Maximum de trois entiers	13
2.3	TP1 Nommer une expression	15
2.4	TP2 Ordre d'évaluation	15
2.5	TP2 Moyenne de deux entiers	16
2.5.1	Types numériques : entiers (int) et réels (float)	16
2.5.2	Fonctions de conversion	16
2.6	TP2 Moyenne olympique	17
2.6.1	Trace de l'évaluation d'une fonction	17
2.7	TP3 Une date est-elle correcte ?	18
2.8	TP3 Relations sur des intervalles d'entiers	18
2.8.1	n-uplets	19
2.8.2	Points et intervalles	19
2.8.3	Intervalles, couples d'intervalles	20
2.9	TP3 Somme des chiffres d'un nombre	20
2.10	TP4 Permutation ordonnée d'un couple	22
2.10.1	Permutation ordonnée d'un couple d'entiers	22
2.10.2	Surcharge des opérateurs de comparaison	22
2.10.3	Fonctions génériques : paramètres dont le type est générique	22
2.11	TP4 Type Durée et opérations associées	23

2.11.1	Définition du type <i>duree</i> et des opérations associées	23
2.12	TP4 Codage des caractères	26
2.12.1	Le code ASCII d'un caractère	26
2.12.2	Valeur entière associée à l'écriture en base 10 d'un entier	27
2.13	TP4 Numération en base 16	27
2.13.1	Valeur entière associée à un chiffre hexadécimal	27
2.14	TP4 Chiffres d'un entier en base 16	27
 II DÉFINITIONS RÉCURSIVES		29
 III ORDRE SUPÉRIEUR		30
 IV STRUCTURES ARBORESCENTES		31

OCaml version 4.11.1

```
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
```

© INRIA

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'invite¹ #. L'utilisateur fournit l'expression à évaluer et appuie sur la touche Entrée². 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.

Rappels

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

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

¹En Anglais : prompt.

²Enter 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 avec l'environnement 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 débiter cette première séance, connectez-vous sur un serveur d'application Linux du DLST, en cliquant sur l'icône du bureau (en général, le serveur et l'icône s'appellent turing).

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 Linux¹, taper :

1. `gedit inf201-Puitg-Basset-Couillet-TP1.ml &`
`inf201-Puitg-Basset-Couillet-TP1.ml` est le nom du fichier (compte rendu de TP n° 1). Ne pas oublier le `&` final². Cet éditeur démarre parfois en mode plein écran, retailler sa fenêtre si besoin.

2. `ledit ocaml`

Nous vous conseillons la commande : `ledit ocaml`³ plutôt que `ocaml` tout court; vous

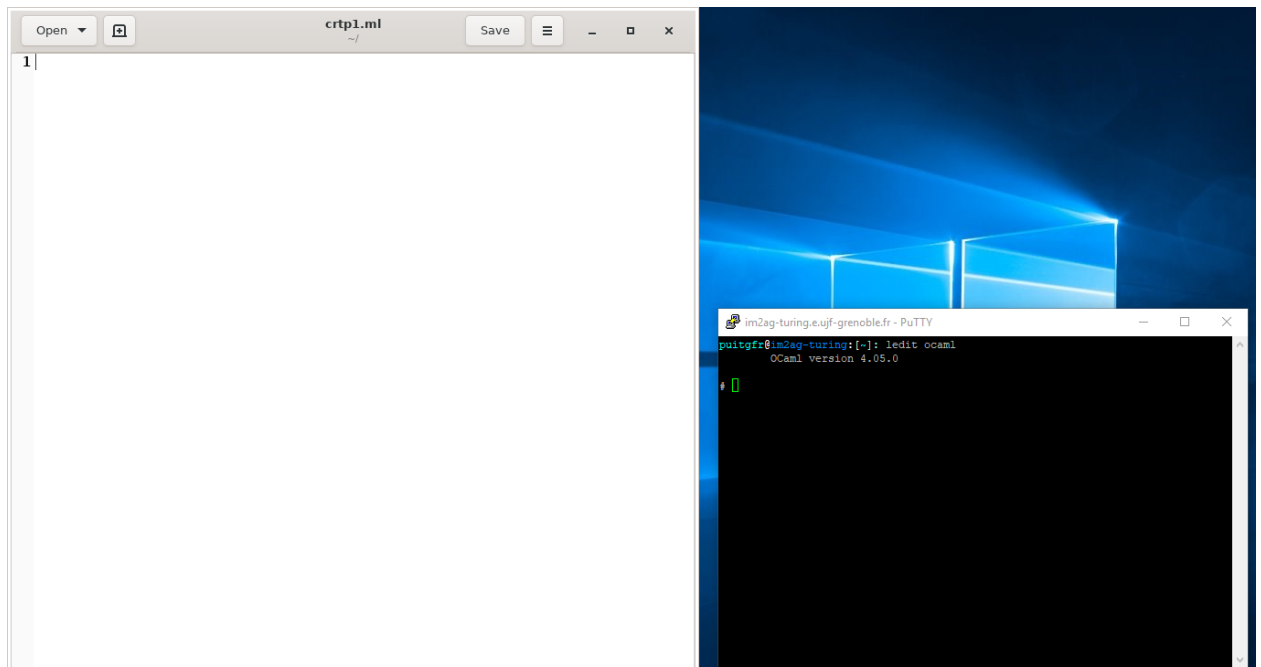
¹Ne pas confondre avec l'interpréteur OCAML

²La signification du `&`, et plus généralement des commandes Linux n'est pas détaillée dans cette UE.

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

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'interpréteur 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** » ; retailer 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>    /
-----
*)
```

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'interpréteur OCAML

L'interpréteur 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'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'interpréteur 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'interpréteur 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`).

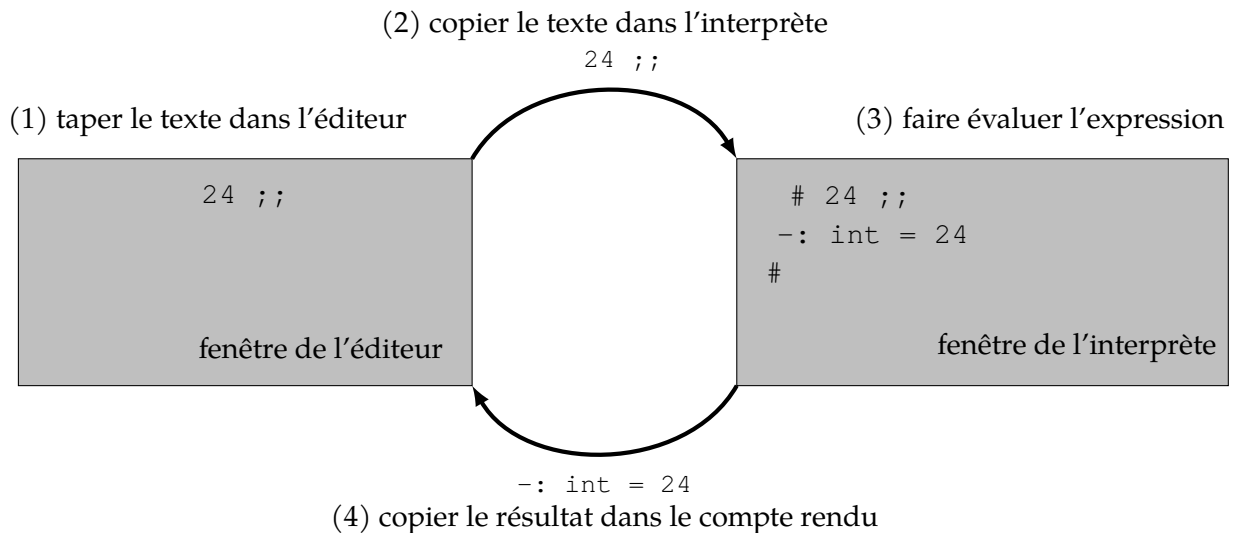
Par ailleurs, lorsque vous quitterez l'interpréteur OCAML, tout ce que vous avez tapé sera perdu. Il est donc nécessaire d'utiliser à la fois l'interpréteur OCAML et l'éditeur.

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'interpréteur.

1. Sélectionner l'expression, point-virgules compris ; dans le cas présent : `24 ; ;`. Au relâchement 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 droit⁴ ; 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 :

<code>(* Expressions</code>	<code>Réponses du système *)</code>	1
<code>24 ; ;</code>	<code>(* - : int = 24 *)</code>	2
<code>3+4 ; ;</code>	<code>(* ... *)</code>	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.

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" ; ;
```

⁴Selon 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 débiter 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

```
( * -----1--
inf201_Puitg_Basset_Couillet_TP1.ml : cr exercices TP no1                2
                                                                           3
François Puitg <francois.puitg@univ-grenoble-alpes.fr> \                  4
Nicolas Basset <bassetni@univ-grenoble-alpes.fr>          >  Groupe boss1
Romain Couillet <Romain.Couillet@grenoble-inp.fr>         /                6
-----7--
```

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).

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` ().
- 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	10
2.1.1		Expressions basiques	10
2.1.2		Opérateurs de comparaison	10
2.1.3		Fabrication de jeux d'essai	11
2.1.4		Définition d'une constante	11
2.1.5		Expressions conditionnelles	11
2.2	TP1	Maximum entre plusieurs entiers	12
2.2.1		Spécification, réalisation d'une fonction	12
2.2.2		Utilisation d'une fonction	12
2.2.3		Maximum de trois entiers	13
2.3	TP1	Nommer une expression	15
2.4	TP2	Ordre d'évaluation	15
2.5	TP2	Moyenne de deux entiers	16
2.5.1		Types numériques : entiers (int) et réels (float)	16
2.5.2		Fonctions de conversion	16
2.6	TP2	Moyenne olympique	17
2.6.1		Trace de l'évaluation d'une fonction	17
2.7	TP3	Une date est-elle correcte ?	18
2.8	TP3	Relations sur des intervalles d'entiers	18
2.8.1		n-uplets	19
2.8.2		Points et intervalles	19
2.8.3		Intervalles, couples d'intervalles	20
2.9	TP3	Somme des chiffres d'un nombre	20
2.10	TP4	Permutation ordonnée d'un couple	22
2.10.1		Permutation ordonnée d'un couple d'entiers	22
2.10.2		Surcharge des opérateurs de comparaison	22
2.10.3		Fonctions génériques : paramètres dont le type est générique	22
2.11	TP4	Type Durée et opérations associées	23
2.11.1		Définition du type <i>duree</i> et des opérations associées	23
2.12	TP4	Codage des caractères	26
2.12.1		Le code ASCII d'un caractère	26

2.12.2	Valeur entière associée à l'écriture en base 10 d'un entier	27
2.13	TP4 Numération en base 16	27
2.13.1	Valeur entière associée à un chiffre hexadécimal	27
2.14	TP4 Chiffres d'un entier en base 16	27

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 copiant dans 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 `crtpl.ml`, taper dans la fenêtre dans laquelle vous avez lancé l'interprète OCAML `#use "crtpl.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 ? :

<code>if a < 10</code>	<code>if a < 10</code>	<code>if a < 10</code>	<code>if a < 10</code>
<code>then true</code>	<code>then a</code>	<code>then a</code>	<code>then true</code>
<code>else a</code>	<code>else false</code>	<code>;;</code>	<code>else false</code>
<code>;;</code>	<code>;;</code>		<code>;;</code>

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 :

```
( *
| 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)  $\forall a \in \mathbb{Z}, (\max2\ a\ a) = a$ 
| 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_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	$\text{max3} : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
SÉMANTIQUE	$(\text{max3 } a \ b \ c)$ 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

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

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

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

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 *max3* pour calculer le maximum de 3 entiers.

ALGORITHME 3

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

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

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

Réalisation en nommant un calcul intermédiaire

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

ALGORITHME 4

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

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 ;;
```

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 :

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

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

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.

2.6.1 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.

- Définir les types `jour` et `mois`.
- Implémenter la version 2 de la fonction `estJourDansMois-2`.
- Définir en OCAML une liste de jeux d'essai significatifs pour tester cette fonction, sous la forme `assert ((estJourDansMois-2 28 1) = true) ; ;`
- Que répond OCAML à `(estJourDansMois-2 18 13)`, `(estJourDansMois-2 0 4)` ? Comment interpréter ces résultats ?

Q2.

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

- Donner une nouvelle implémentation (n°4) du type `Mois` en utilisant un *type énuméré*, et donc une nouvelle implémentation `estJourDansMois-4`.
- 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 = (* { (bi, bs) ∈ ℤ2 tels que bi ≤ bs } *)
..... ;;
```

1
2

- 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)           1
and x1 : int = -40 ;;                             2
puis assert (precede x1 interv_1) ;;
```

- 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	$sc : 0 ; 9999 \rightarrow \mathbb{Z}$
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) :

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

RÉALISATION 1 — 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.

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

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 $\text{div} : \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \times \mathbb{Z}$

SÉMANTIQUE posons $(q, r) = (\text{div } n \ d)$; 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 $sc(n)$: $n \in \dots$

```
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 ;;`.

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	$poCoupleE : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \times \mathbb{Z}$
SÉMANTIQUE	Soit $(x', y') = poCoupleE(x, y)$. (x', y') est la permutation ordonnée de (x, y) , définie comme suit : $(x', y') = \begin{cases} (x, y) & \text{si } x \leq y, \\ (y, x) & \text{sinon.} \end{cases}$
EX. ET PROP.	(i) $poCoupleE(3, 4) = \dots\dots$ (ii) $poCoupleE(4, 3) = \dots\dots$

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

1

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 α au lieu de 'a' (β au lieu de 'b', ...):

SPÉCIFICATION 2	
PROFIL	<code>poCouple</code> : $\alpha \times \alpha \rightarrow \alpha \times \alpha$

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

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

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 exprimée 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

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**SPÉCIFICATION 1 — constructeurs**PROFIL $sec_en_duree : \mathbb{Z} \rightarrow duree$ SÉMANTIQUE construit une donnée de type *duree* à partir d'un nombre de secondesPROFIL $vec_en_duree : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow duree$ 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 2 — 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 3 — conversionPROFIL $duree_en_sec : duree \rightarrow \mathbb{Z}$

SÉMANTIQUE convertit une durée en un nombre de secondes

SPÉCIFICATION 4 — opérationsPROFIL $som_duree : duree \rightarrow duree \rightarrow duree$

SÉMANTIQUE effectue la somme de deux durées

SPÉCIFICATION 5 — prédicatsPROFIL $eg_duree : duree \rightarrow duree \rightarrow \mathbb{B}$

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 :

```

type jour      = ..... ;; (* {0, ..., 31} *)      1
type heure     = ..... ;; (* {0, ..., 23} *)      2
type de0a59    = ..... ;; (* {0, ..., 59} *)      3
type minute    = ..... ;;                        4
type seconde   = ..... ;;                        5
type duree     = ..... ;;                        6

```

Q1. Réaliser la fonction *sec_en_duree*.

RÉALISATION 5

ALGORITHME utilisation de *div* réalisée dans un exercice précédent.

IMPLÉMENT. ...

On considère la fonction suivante :

SPÉCIFICATION 6

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

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. ...

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

IMPLÉMENT. ...

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

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', ..., '9' }

déf *base10* = {0, ..., 9}

- Q3.** Donner le profil et la sémantique d'une fonction *chiffreVbase10*, qui associe un élément de *base10* à un *chiffre*.
- Q4.** Donner le profil et la sémantique d'une fonction *base10Vchiffre*, qui associe un *chiffre* à un élément de *base10*.
- Q5.** Dédurre des deux questions précédentes 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') ; ; 1
char_of_int(8 + int_of_char('0')) ; ; 2
```

- Q6.** Implémenter 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* :

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

déf *base16* = {0, ..., 15}

- Q1.** Donner le profil et la sémantique d'une fonction *carhexVbase16*, qui associe un élément de *base16* à un *carhex*.
- Q2.** Donner une implémentation naïve des types *carhex* et *base16* sous forme respectivement d'un caractère et d'un entier, avec les restrictions convenables.
- 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 implémentant *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 :

déf *hexa4* = 0 ; 16⁴ - 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

Troisième partie

ORDRE SUPÉRIEUR

Quatrième partie

STRUCTURES ARBORESCENTES