# Feuille d'exercices n°1

## Exercice I : Typage et évaluation d'expressions

**Q1** – Pour chacune des expressions suivantes, dire si elle est une expression valide du langage OCaml, dans le cas où l'expression est valide, donner son type et sa valeur, dans le cas contraire expliquer pourquoi elle n'est pas valide.

1) `true && false`
2) `fun (x: bool) -> x`
3) `fun x -> x`
4) `fun x -> false`
5) `true || (fun x -> false)`
6) `(fun f x -> f x) not true`
7) `(fun f x -> f x) (not true)`
8) `(fun x -> false) true false`
9) `let x = false in (x || (not x))`
10) `(fun x y -> x) (true, false) false`
11) 
```
let f = fun x -> not x in
let g = fun f x y -> (f x) && (f y) in
g f true
```

## Exercice II : N-uplets

**Q1** – Définir les fonctions `fst: ('a * 'b) -> 'a` (resp. `snd: ('a * 'b) -> 'b`) projetant une paire sur sa première (resp. seconde) composante.

**Q2** – Définir une fonction `paire: 'a -> 'b -> ('a * 'b)` qui, à partir de deux éléments $a$ et $b$ construit la paire $(a, b)$.

**Q3** – Déduire de la question précédente une fonction `paire_true: 'a -> (bool * 'a)` qui a un élément $a$ associe la paire (`true`, a).

**Q4** – Définir une fonction `curry: (('a * 'b) -> 'c) -> ('a -> 'b -> 'c)` prenant en argument une fonction `f` attendant une paire comme argument, et calculant une fonction de deux arguments `g` telle que pour tout `x`, `y`, on ait `f (x, y) = g x y`.

**Q5** – Définir une fonction `uncurry: ('a -> 'b -> 'c) -> (('a * 'b) -> 'c)` prenant en argument une fonction `g` à deux arguments, et calculant une fonction `f` attendant une paire telle que pour tout `x`, `y`, on ait `f (x, y) = g x y`.

## Exercice III : Table de vérité et opérateurs booléens

Les deux valeurs de type `bool` de OCaml sont `true` et `false`. On rappelle ici les tables de vérité des opérateurs booléens classiques : le produit, la somme et le complément. Dans cet exercice on ne s'autorisera pas à utiliser les opérateurs booléens (`&&`, `||`, `not`) prédéfinis par OCaml.

| $\cdot$ | true | false |
|---|---|---|
| true | true | false |
| false | false | false |

| $+$ | true | false |
|---|---|---|
| true | true | true |
| false | true | false |

| $\overline{\phantom{x}}$ | |
|---|---|
| true | false |
| false | true |

**Q1** – Définissez les opérateurs `et_bool` et `ou_bool`, de signature `bool -> bool -> bool` et `non_bool` de type `bool -> bool` effectuant respectivement les opérations ($\cdot$, $+$, $\overline{\phantom{x}}$).

On définit les opérateurs suivants :

- $A \to B \equiv \overline{A} + B$
- $A \leftrightarrow B \equiv (A \to B) \cdot (B \to A)$
- $A \oplus B \equiv (A + B) \cdot \overline{A \cdot B}$

**Q2** – Définissez les opérateurs `impl_bool`, `equiv` et `xor` effectuant respectivement les opérations ($\to$, $\leftrightarrow$, $\oplus$).

**Q3** – (Faire en TME) Comparez le résultat des appels : `ou_bool true (1/0=1)` et `true || (1/0=1)`, pouvez-vous expliquer cette différence?

## Exercice IV : Manipulation de fonctions

En programmation fonctionnelle, les fonctions sont des valeurs comme les autres et en particulier, il est possible d'écrire des fonctions prenant des fonctions en paramètres et/ou produisant des fonctions en sorties.

**Q1** – Écrire la fonction `application` qui prend en argument une fonction `f` et une valeur `x` et qui applique `f` à `x`.

**Q2** – Écrire la fonction `composition` prenant en argument deux fonctions `f` et `g` et calculant leur composée `f` $\circ$ `g`.

**Q3** – Écrire la fonction `f_ou_ident` prenant en argument une fonction `f` et un booléen `b` et calculant

- l'identité si `b` est vrai
- `f` si `b` est faux

## Exercice V : Manipulation de fonctions - 2

**Q1** – On rappelle qu'une fonction est dite injective si elle ne donne jamais le meme résultat pour deux arguments différents. Donnez un exemple de fonction booléenne injective, et un exemple de fonction booléenne non injective.

**Q2** – Définissez un predicat `est_injective` de signature `(bool -> bool) -> bool`, qui prend une fonction booléenne en paramètre et qui vérifie si celle-ci est injective.

**Q3** – Définissez un prédicat `est_commutatif` de signature `(bool -> bool -> bool) -> bool`, qui prend une fonction `f` sur les booléens et qui vérifie si `f` est commutative, *i.e.* pour toute paire de booléen `a` et `b`, `f a b = f b a`.

**Q4** – Définissez un predicat `est_reciproque` de signature `(bool -> bool) -> (bool -> bool) -> bool` qui prend deux fonctions booléennes en paramètre `f` et `g`, et vérifie si `f` est la reciproque de `g`, *i.e.* si pour tout argument `x`, `f (g x) = x`.

**Q5** – On a vu qu'il est possible en OCaml de passer une fonction en paramètre d'une autre fonction. Nous allons à présent calculer une fonction en sortie d'une autre fonction. Définissez une fonction `reciproque` de signature `(bool -> bool) -> (bool -> bool)` qui prend une fonction booléenne en paramètre et qui calcule sa réciproque. On supposera que la fonction passée en paramètre est injective.

# Travaux sur machines

## Mise en route

Pour installer sur votre compte le minimum vital pour développer en OCaml, veuillez taper dans le terminal les deux commandes suivantes :

```
$ sh /Infos/lmd/2019/licence/ue/LU2IN019-2019oct/install.sh
$ source ~/.bashrc
```

Durant les TMEs nous vous conseillons d'utiliser l'editeur de texte `Atom`. Il offre un bon support du langage `OCaml` et permettra de typer interactivement les expressions du langage. Il permet aussi de détecter les erreurs de type/syntaxe à la volée.

Vous pouvez toutefois utiliser un autre éditeur de texte mais il est alors très fortement recommandé d'en choisir un qui supporte OCaml (voir `https://github.com/ocaml/merlin/wiki`).

Si vous choisissez d'utiliser `Atom`, tapez dans une console la commande suivante pour installer les plug-ins `OCaml` pour Atom:

```
$ sh /Infos/lmd/2019/licence/ue/LU2IN019-2019oct/atom-setup.sh
```

### Toplevel

Pour ouvrir un interpréteur (toplevel) OCaml dans le terminal :

```
$ utop
```

*NB* : pour évaluer une expression dans `utop`, il faut la terminer par "`;;`".

Exemple 1 :

```
# print_int (2 + 1);;
```

Exemple 2 supposant la fonction `fois_deux` définie :

```
# print_int (fois_deux 21);;
```

Dans un terminal, on peut ouvrir un interpréteur (toplevel) OCaml en chargeant le contenu d'un fichier avec :

```
$ utop -init fichier.ml
```

Aussi, une fois `utop` ouvert, pour (ré)évaluer le contenu d'un fichier `toto.ml`, il faut faire :

```
# #use "toto.ml" ;;
```

***NB*** : pour tester des fonctions, il est fortement conseiller de définir un jeu de test et d'ajouter des lignes de vérification dans le fichier contenant les définitions de ces fonctions.

Par exemple, si dans un fichier est définie la fonction :

```
let f (x: int) (y:int) : int = x + y + 2
```

On peut ajouter un ensemble de tests à la suite de la définition de la fonction en utilisant `assert`. Par exemple :

```
let () = assert ((f 2 3) = 7)

let () = assert ((f (-1) (-1)) = 0)
```

Lorsque le fichier est chargé dans `utop`, si un test ne passe pas, un message d'erreur (avec le numéro de ligne correspondant) s'affiche.

***NB*** : vous êtes supposés tester toutes les fonctions demandées (même si non explicitement demandé). Cela nécessite de proposer un jeu de test pertinent.

## Exercice VI : Comparaison d'octets

Faire une copie du squelette du TME:

```
$ cp /Infos/lmd/2019/licence/ue/LU2IN019-2019oct/tds/td1/tme_etudiant.ml tme1.ml
```

**Q1** – Définir une fonction `xor: bool -> bool -> bool` calculant le ou exclusif de deux booléens.

On définit la relation d'ordre strict $\prec$ sur les booléens par: $x \prec y$ si et seulement si $x =$ `false` et $y =$ `true`.

**Q2** – Soit $x$ et $y$ deux booléens. Sur une feuille, donner 3 expressions booléennes n'utilisant que les opérateurs booléens + (ou logique), . (et logique), - (complément logique) et `xor` (ou exclusif), telles qu'elles sont respectivement vraies si et seulement si :

- $x \prec y$
- $x = y$
- $y \prec x$

On définit `type cmp = bool * bool * bool`

On souhaite définir une fonction `cmp1: bool -> bool -> cmp` tel que `cmp1 a b` = `(s1, s2, s3)` avec:

- `s1` = `true` si et seulement si `a` $\prec$ `b`;
- `s2` = `true` si et seulement si `a` = `b`;
- `s3` = `true` si et seulement si `b` $\prec$ `a`;

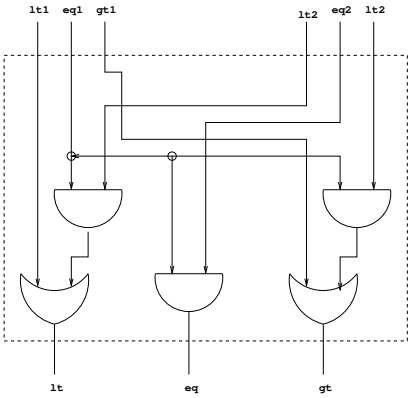
**Q3** – Combien de valeurs différentes de type `cmp`, la fonction `cmp1` peut-elle renvoyer ? Nommer ces différentes valeurs.

**Q4** – À l'aide des questions précédentes, définir la fonction `cmp1: bool -> bool -> cmp` telle que spécifiée ci-avant puis tester la fonction `cmp1` de manière exhaustive.

On veut maintenant comparer des vecteurs de bits (n-uplets de booléens) de même taille $a$ et $b$, en étendant de manière lexicographique (l'ordre du dictionnaire) l'ordre $\prec$. On notera cette ordre $\sqsubset$. Pour comparer 2 vecteurs de bits $a$ et $b$ de même taille (supérieurs ou égale à 2), on peut couper en deux chaque vecteur – disons $a$ en $a_1$ et $a_2$ et $b$ en $b_1$ et $b_2$; de manière à ce que $a_1$ et $b_1$ soient de même taille et donc $a_2$ et $b_2$ sont de même taille et ces tailles sont non nulles. On peut déduire le résultat pour la comparaison entre $a$ et $b$ en raisonnant comme suit:

- si $a_1 \sqsubset b_1$ ou si $a_1 = b_1$ et $a_2 \sqsubset b_2$ alors $a \sqsubset b$;
- si $a_1 = b_1$ et $a_2 = b_2$ alors $a = b$;
- si $b_1 \sqsubset a_1$ ou si $a_1 = b_1$ et $b_2 \sqsubset a_2$ alors $b \sqsubset a$

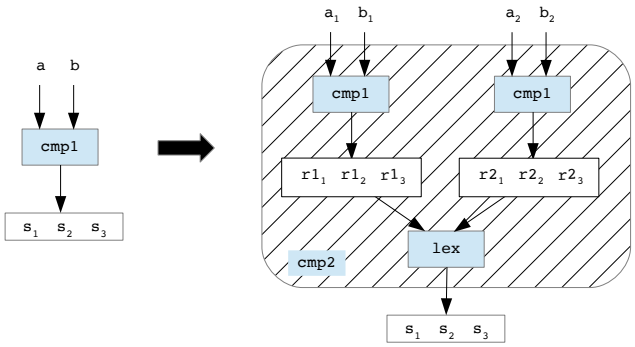
On réalise la combinaison lexicographique de deux comparaisons avec le circuit logique ci-dessous




**Q5** – En déduire une définition de la fonction `lex: cmp -> cmp -> cmp` telle que si `r1` est le résultat de la comparaison de `a1` et `b1` (de même taille) et `r2` le résultat de la comparaison de `a2` et `b2` (de même taille) alors `lex r1 r2` est le résultat de la comparaison des concaténations de `a1` avec `a2` et de `b1` avec `b2`.

**Q6** – Définir un type de vecteurs de 2 bits nommé `duet`, définir une fonction `cmp2: duet -> duet -> cmp` calculant $\sqsubset$ sur des duets.

Schématiquement, on passe de `cmp1` à `cmp2` en combinant `cmp1` et `lex` de la manière suivante:




**Q7** – Définir un type de vecteurs de 4 bits nommé `quartet`, définir une fonction `cmp4: quartet -> quartet -> cmp` calculant $\sqsubset$ sur des quartets.

**Q8** – Définir un type de vecteurs de 8 bits nommé `octet`, définir une fonction `cmp8: octet -> octet -> cmp` calculant $\sqsubset$ sur des octets.

Le fichier fournit une fonction `i2q` (resp. `i2o`) transformant un entier OCaml (type `int`) compris entre 0 et 15 (resp. 0 et 255) en quartet (resp. octet).

**Q9** – Compléter les annotations de type des fonctions `i2b`, `i2q`, `i2o` en utilisant les types définis pendant le tme.

**Q10** – Ajouter des tests pertinents pour les fonctions `i2b`, `i2q`, `i2o`.

**Remarque :** l'encodage des nombres (via `i2q` et `i2o`) et l'ordre lexicographique ci-avant défini sont tels que `i` $<$ `j` si et seulement si `(i2q i)` $\sqsubset$ `(i2q j)` (resp. de même avec `i2o`).

**Q11** – Utiliser la remarque ci-dessus pour tester (de manière exhaustive si possible ou pertinente sinon) les fonctions `cmp4` et `cmp8`.