# TP n° 1

Paradigmes et interprétation
Licence Informatique
Université Côte d'Azur

## Installation de `plait` et présentation de *DrRacket*

Suivez les consignes d'installation de l'environnement disponible sur la page du cours. *DrRacket* contient deux fenêtres principales.


La **fenêtre de définition** est la fenêtre dans laquelle vous écrirez votre code. En appuyant sur le bouton **Exécuter**, vous réinitialisez la mémoire. Le code de la fenêtre de définition est ensuite évalué. Dans la **fenêtre d'interaction** (également appelée top-level), vous pourrez réaliser des tests rapides de votre code. La sauvegarde ne concerne **que** la fenêtre de définition.

## Echauffements

1. Tapez quelques expressions au top-level, vérifiez leur résultat et le type associé.
2. Définissez une fonction `somme` qui prend en paramètre une liste de nombres et renvoie leur somme.

```
(define (somme [L : (Listof Number)]) : Number
  ... )
```

```
(define (somme [L : (Listof Number)]) : Number
  (if (empty? L)
      0
      (+ (first L) (somme (rest L)))))
```

ou

```
(define (somme [L : (Listof Number)]) : Number
  (foldl + 0 L))
```

3. Définissez une fonction polymorphe `Append` qui prend en paramètre deux listes et qui renvoie leur concaténation.

```
(define (Append [LG : (Listof 'a)]
                [LD : (Listof 'a)]) : (Listof 'a)
  ... )
```

```
(define (Append [LG : (Listof 'a)]
                [LD : (Listof 'a)]) : (Listof 'a)
  (if (empty? LG)
      LD
      (cons (first LG) (Append (rest LG) LD))))
```

ou

```
(define (Append [LG : (Listof 'a)]
                [LD : (Listof 'a)]) : (Listof 'a)
  (foldr cons LD LG))
```

4. Au top-level, demandez le type des fonctions `map`, `foldl` et `foldr`. Vous paraissent-ils cohérents ?

```
map   ; (('a -> 'b) (Listof 'a) -> (Listof 'b))
foldl ; (('a 'b -> 'b) 'b (Listof 'a) -> 'b)
foldr ; (('a 'b -> 'b) 'b (Listof 'a) -> 'b)
```

5. Sans utiliser les fonctions `map`, `foldl` et `foldr`, définissez des fonctions `Map`, `Foldl` et `Foldr` qui réalisent le même comportement.

```
(define (Map [f : ('a -> 'b)]
             [L : (Listof 'a)]) : (Listof 'b)
  (if (empty? L)
      L
      (cons (f (first L)) (Map f (rest L)))))
```

```
(define (Foldl [f : ('a 'b -> 'b)] [acc : 'b]
               [L : (Listof 'a)]) : 'b
  (if (empty? L)
      acc
      (Foldl f (f (first L) acc) (rest L))))

(define (Foldr [f : ('a 'b -> 'b)] [acc : 'b]
               [L : (Listof 'a)]) : 'b
  (if (empty? L)
      acc
      (f (first L) (Foldr f acc (rest L)))))
```

## Listes d'associations

1. Définissez un type polymorphe `Couple` qui représente un couple d'éléments de types quelconques (et potentiellement différents). Ce type n'aura qu'une seule variante `couple`. [1]

```
(define-type (Couple 'a 'b)
  [couple (fst : 'a) (snd : 'b)])
```

2. Définissez un type polymorphe `Option` qui possède deux variantes. L'une `vide` ne contient aucun champ et indique l'absence d'élément. L'autre `element` a un champ de type quelconque et dénote la présence d'un élément. [2]

```
(define-type (Option 'a)
  [vide]
  [element (value : 'a)])
```

3. Une liste d'association est une liste de couples clé/valeur. C'est un moyen simple de représenter un dictionnaire. On peut chercher une clé dans cette liste et renvoyer la valeur associée. Si une clé recherchée est absente, il faut également pouvoir l'indiquer.

   Définissez une fonction `find` qui prend en paramètres une liste d'association, une clé et qui renvoie une option contenant la valeur si elle a été trouvée ou `(vide)` dans le cas contraire.

```
(define couples (list (couple 'x 3) (couple 'y 7)))
(test (find couples 'y) (element 7))
(test (find couples 'z) (vide))
```

1. En `plait`, il existe une fonction `pair` qui construit une paire et des fonctions `fst` et `snd` qui projettent les composantes d'une paire.
2. En `plait`, ce type s'appelle `Optionof` et ses variantes sont `none` et `some`.

```
(define (find [L : (Listof (Couple 'a 'b))]
              [key : 'a]) : (Option 'b)
  (if (empty? L)
      (vide)
      (type-case (Couple 'a 'b) (first L)
        [(couple k v) (if (equal? k key)
                          (element v)
                          (find (rest L) key))])))
```

## Arbres binaires

1. Définissez un type récursif `ArbreBinaire` de sorte que :
   — Les noeuds internes sont étiquetés par une fonction `(Number Number -> Number)` et ont un fils gauche et un fils droit.
   — Les feuilles sont étiquetées par un nombre.

```
(define-type ArbreBinaire
  [feuille (val : Number)]
  [noeud (op : (Number Number -> Number))
         (fg : ArbreBinaire)
         (fd : ArbreBinaire)])
```

2. Définissez une fonction `eval` qui prend un arbre binaire en paramètre et renvoie sa valeur. L'évaluation d'une feuille est son étiquette. L'évaluation d'un noeud interne est l'application de son étiquette à la valeur de ses fils.

```
(define (eval [arbre : ArbreBinaire]) : Number
  (type-case ArbreBinaire arbre
    [(feuille val) val]
    [(noeud op fg fd) (op (eval fg) (eval fd))]))
```