**Rapport de la semaine de 11/6/2018 :**

# Application conversion degré Celsius/Fahrenheit :

Voici la liste détaillée des fonctions utilisées :

**Js_of_ocaml :**

```
clicked : buttonElement -> element -> unit

clicked2 : buttonElement -> element -> unit

converterC_handler : unit -> unit

converterF_handler : unit -> unit
```

**Obrowser :**

```
float_input : string -> string -> string -> (Node.t * Fragment.t)

button : string -> string -> ('a t -> 'a t) -> Fragment.t

p : string -> (Node.t * Fragment.t)
```

**BuckleScripts :**

```
clicked : unit -> unit

clicked2 : unit -> unit
```

Module `Element` :

Ce module contient toutes les fonctions qui permettent de manipuler les propriétés de tous les éléments `DomHtml`(tous les nœuds DOM).

Module `Document`:

Ce module contient toutes les fonctions qui permettent de manipuler le Document HTML pour récupérer des nœud etc ...

Module `Window` :

Ce module contient les fonctions qui permettent de manipuler les propriétés du nœud window.

# La concurrence en Ocaml :

Avant de pouvoir comparer les modèles de concurrence avec les trois méthodes (Js_of_ocaml,Obrowser et BuckleScript) il est nécessaire de comprendre comment la concurrence fonctionne généralement en Ocaml .

## Le module Lwt :

Lwt est un des projet de l'entreprise Ocsigen (le producteur de Js_of_ocaml) : Il s'agit d'un module Caml nommé « light weight threads » (Lwt) qui permet de gérer la concurrence dans un style monadique.

## Style :

*-Qu'est qu’une monade ?*

En théorie des langages fonctionnels typés, une monade est une structure permettant de manipuler des langages fonctionnels purs avec des traits impératifs.

*-Pour quoi une monade ?*

Les langages fonctionnels non stricts ne fournissent aucune garantie quant à l’ordre relatif d’évaluation des sous-expressions. Lorsqu’il est question, dans un tel langage, d’écrire des programmes dans un style séquentiel, on peut recourir à un style de programmation appelé monadique,qui impose une certaine séquentialité. Ce style repose sur des entités abstraites appelées monades, qui forment un outil très général pour combiner des calculs dans un langage typé.

*-Comment construire une monade ?*

Une monade peut se voir comme la donnée d'un couple constitué des deux éléments suivants :

- Une fonction nommée `return` qui construit à partir d'un élément de type sous-jacent un autre objet de type monadique . Cette fonction est alors de signature.
  `return : t -> calc(t)`
- Une fonction `bind`, (aussi représentée par l'opérateur infixe `>>==` ), associant à un type monadique et une fonction d'association un autre type monadique. Il permet de composer une fonction monadique à partir d'autres fonctions monadiques. Cet opérateur est de type
  `bind : calc(t) -> (t -> calc(u)) -> calc(u)`

En résume : La fonction `return` initialise un calcul, et l’opération `bind(e)(f)` va extraire la valeur calculée par `e` et l’envoyer à la fonction `f`.

Pour former une monade, ces deux opérations doivent satisfaire les trois propriétés suivantes :

```
bind (return(x)) (f) = f(x)
bind (m) (return) = m
bind (bind(m)(f)) (g) = bind (m) (fun x→bind (f(x)) (g))
```

## Installation et compilation:

Il suffit de passer la commande suivante en utilisant le gestionnaire de paquets Opam:

```
$opam install conf-libev lwt
```

Puis on passe cette deuxième commande afin d’installer la syntaxe ppx : (Pas obligatoire )

```
$opam install lwt_ppx
```

et pour compiler selon le contenu du fichier on va avoir besoin de l’une des commandes suivantes :

```
$ocamlfind ocamlopt exemple.ml -o exemple -linkpkg -package lwt lwt_ppx
$ocamlfind ocamlopt exemple.ml -o exemple -linkpkg -package lwt.unix lwt_ppx
```

**La syntaxe :**

Le modèle de concurrence de Lwt est un modèle monadique donc la syntaxe consiste à traduire les éléments/fonctions théoriques expliqués en langage Ocaml :

En premier lieu on possède le type suivant :

```
'a Lwt.t
```

Ce type est le type d'une monade (ici donc pour gérer une thread) qui retourne une valeur de type `'a`.Par exemple la fonction `Lwt_io.read_char` qui lit un caractère sur l'entrée standard et le renvoie a donc le type :

```
char Lwt.t
```

Une thread a trois états en Lwt :

- `Return x` :La thread a terminé et a bien renvoyé un résultat `x`.
- `Fail exn` : La thread a terminé mais au lieu de renvoyer un résultat c'est une exception `exn` qui a été déclenchée.
- `Sleep` : La thread dort et n'a pas encore renvoyé un résultat ou une exception.
- `Canceld` : La thread a été annulée.

Les trois fonctions suivantes permettent de créer respectivement des thread dans les trois états mentionnés avant (équivalant de `return` en théorie) :

- `Lwt.return : 'a -> 'a Lwt.t`
- `Lwt.fail : exn -> 'a Lwt.t`
- `Lwt.wait : unit -> 'a Lwt.t * 'a Lwt.u`

On peut utiliser les deux fonctions suivantes afin de réveiller une thread qui dort :

- `Lwt.wakeup : 'a Lwt.u -> 'a -> unit`
- `Lwt.wakeup_exn : 'a Lwt.u -> exn -> unit`

D'apres la documentation officielle de Ocsigene l'opération la plus importante à connaître est l'opération `bind` :

```
val bind : 'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t
```

cette opération permet de créer une thread ayant le type `'a Lwt.t` puis elle attend pour que cette thread se termine et renvoie un résultat, puis utilise ce résultat dans la fonction du type `'a -> 'b Lwt.t` afin de créer une nouvelle thread avec la valeur renvoyée par la première thread. Cette fonction ne commence donc que quand `'a Lwt.t` aura terminé si cette thread renvoie une exception l'opération `bind` 'fail' aussi et renvoie une exception.

On peut désormais composer plusieurs threads en simultané , pour cela il existe trois fonctions en Lwt :

- `val join : unit Lwt.t list -> unit Lwt.t`

- `val choose : 'a Lwt.t list -> 'a Lwt.t`
- `val pick : 'a Lwt.t list -> 'a Lwt.t`

`join` prend donc en paramètre une liste de threads et attend qu'elle se terminent toutes si il y a une seule thread qui 'fail' join 'fail' aussi en renvoyant la même exception.
`choose` prend aussi en paramètre une liste de threads et attend à ce que une des thread s'arrête et renvoie le résultat ou l'exception pour renvoyer ce même résultat.
On peut aussi utiliser les fonctions `new_key` et `with_value` afin d'avoir la possibilité de nommer les threads.

Références :

[1] "Programmation fonctionnelle", chapitre de Encyclopédie de l'informatique et des systèmes d'information, pp. 1016-1027, (Vuibert),2006
[2] Paul Rivier, "Les monades dans la programmations fonctionnelle",http://blog.demotera.com,http://blog.demotera.com,consulté le 15/06/2018