Programmation Fonctionnelle Avancée

# Projet : Symbioz

# À rendre avant le vendredi 20 mars 2015, minuit

Le projet est à faire individuellement. Si toutefois vous le faites en binôme, une soutenance sera organisée le vendredi 27 mars après-midi.

Tout plagiat sera puni.

Le barème est donné à titre indicatif.

## 1 Symbioz

Sur la planète Symbioz vivent principalement trois espèces :

- Des *zherbs* qui poussent en groupe et prolifèrent vers des territoires adjacents.
- Des *krapits*, paisibles zherbivores qui paissent et cherchent toujours des nouveaux champs où se nourrir.
- Et des *kroguls*, terribles prédateurs aux sens affûtes qui traquent sans pitié les krapits pour les manger.

Tout ce beau monde tente de se nourrir, de se reproduire ou simplement de survivre. *Struggle for life.*

Librement inspiré du jeu *Symbioz* : `http://www.boardgamegeek.com/boardgame/5516/symbioz`

## 2 Présentation (2 pts)

Les fichiers sources (`ml`) doivent être lisibles et compréhensibles. Ceci inclue notamment (mais pas uniquement) les recommandations suivantes :

- **Aérer** : sauter des lignes entre les fonctions, aller à la ligne plutôt trop souvent que pas assez, espacer les expressions (par exemple de part et d'autre d'un `=` ou `->`), ...
- **Indenter** : le code doit être indenté pour être facilement lisible. Le `tuareg-mode` d'`emacs` (cf TP 01) fait du bon travail d'indentation. Les autres éditeurs sont un peu moins performant mais peuvent aussi faire du boulot correct.
- **Nommer explicitement** les fonctions et les variables. Pour les fonctions simples, le nom de la fonction et celui des variables devrait être suffisant pour comprendre ce que fait la fonction.
- **Commenter**. Idéalement, chaque fonction doit être précédée d'un court commentaire (une ou deux lignes) qui explique ce que fait la fonction. Pour les cas un peu plus compliqués, on peut détailler davantage. Penser en particulier à indiquer si il y a des conditions sur les arguments (par exemple "la liste ne doit pas être vide") ou à préciser ce que fait la fonction dans les cas limites (par exemple "si la liste est vide, on renvoie -1").
- **Commenter**. Les passages un peu "techniques" à l'intérieur d'une fonction doivent être expliqués brièvement (normalement, une ligne devrait suffire). En particulier, si il y a une formule un peu compliquée, rappeler à quoi elle sert (par exemple "ici on calcule le discriminant du polynôme").
- **Commenter** [1]. Utiliser des larges commentaires (toute la largeur de la ligne, voire plusieurs lignes) pour marquer des séparations visuelles fortes entre les différentes parties du programme à l'intérieur d'un même fichier.

## 3 L'épreuve du feu (0)

Tout projet qui ne compile pas **ne sera pas** corrigé et se verra donc attribuer la note 0.

1. Non, je ne me répète pas...

# 4 Rendu du projet (2 pts)

Pour rendre le projet, il faut envoyer **par mail** à l'adresse `Jean-Yves.Moyen@lipn.univ-paris13.fr` une archive (fichier `.tgz` de préférence) contenant **uniquement les sources** du projet (c'est-à-dire le fichier `.ml`).

Les sources **peuvent** être accompagnés (dans l'archive) d'une explication (supplémentaire) de 2 pages (1 recto-verso) **maximum**, sur papier ou au format `.pdf` ou OpenOffice uniquement. Normalement, les commentaires dans le code doivent suffire pour comprendre les programmes, mais si vous estimez avoir besoin de rajouter un schéma ou des explications plus poussées...

Le mail doit être envoyé avant le vendredi 20 mars 2015 à minuit (heure de Paris). La partie sur papier doit être rendue au plus tard au début du partiel le 24 mars 2014.

Le titre du mail doit être de la forme :
`[PFA] Projet - Nom de l'étudiant`

Par exemple : `[PFA] Projet - Moyen`

En cas d'envois multiples (parce que vous vous rendez compte après coup que la version précédente n'est pas correcte), précisez *dans le corps du mail* que ce nouvel envoi annule et remplace les précédents.

# 5 Fonctions globales (3 pts)

## 5.1 Aléatoire, options

Pour initialiser le générateur aléatoire, utiliser (une seule fois, au début du fichier) :
`let _ = Random.self_init ()`
Pour tirer au hasard un entier entre 0 et $n-1$ (inclus), utiliser :
`Random.int n`

On utilisera beaucoup le type `'a option` qui représente "peut-être un objet de type `'a`". Concrètement, ce type est défini par

```
type 'a option =
  | None
  | Some of 'a
```

Il existe déjà de base dans `Ocaml`, il est donc inutile de le redéfinir.

## 5.2 Listes

**1**/Écrire une fonction `insert_at_pos : 'a -> 'a list -> int -> 'a list` qui insère un élément à une position donnée dans une liste. On suppose que l'entier n'est pas trop grand et que la position existe (il est inutile de faire les vérifications à ce sujet).
**2**/Écrire une fonction `insert_at_random` qui insère un élément à une position aléatoire dans une liste.
**3**/Écrire une fonction `shuffle` qui mélange une liste en prenant ses éléments un par un et en les insérant à une position aléatoire dans le résultat (initialement vide).
**4**/Écrire une fonction `random_get : 'a list -> 'a option` qui renvoie `None` si la liste est vide est `Some elt` où `elt` est un élément au hasard dans la liste si elle est non vide.
**5**/Écrire une fonction `clean_list : 'a option list -> 'a list` qui renvoie la liste des éléments qui ne sont pas `None`

## 5.3 Âge et sexe

Les individus seront de sexe soit masculin, soit féminin. Leur âge est groupé en trois classes d'âge : bébé, enfant ou adulte.
**6**/Écrire les deux types énumération correspondant.
**7**/Pour chacun, écrire une fonction `random_xxx` qui renvoie une valeur au hasard parmi celles possibles.
**8**/Pour chacun des deux types, écrire une fonction `string_of_xxx` qui transforme les valeurs en chaînes de caractères pour l'affichage.

# 6 La planète Symbioz (3 pts)

## 6.1 Signature

Une planète doit contenir :
- Un type (abstrait) `pos` qui représente une position sur la planète.
- Des fonctions `ouest`, `est`, `nord` et `sud` qui renvoient la position au (Ouest, Est, Nord ou Sud) de la position donnée en argument.
- Une fonction qui tire une position au hasard (attention ! Il faut que ce soit une fonction).

On aura aussi besoin de trier des objets encore inconnus pour le moment selon leur position (par exemple, connaître tous les krapits qui sont à une position donnée pour savoir qui le krogul peut manger). Comme on ne sait pas encore quels seront les objets à trier, ces fonctions doivent être paramétrées par la fonction qui donne la position d'un objet. On a donc :
- Une fonction `at_pos : ('a -> pos) -> pos -> 'a list -> 'a list` qui renvoie la liste des éléments présents à la position donnée. Le premier argument est une fonction qui indique comment calculer la position d'un élément.
- Une fonction `sort_by_pos : ('a -> pos) -> 'a list -> 'a list list` qui renvoie une liste de listes, chacune de ces listes étant l'ensemble de tous les éléments à une position donnée (on a donc autant de listes que de positions).

Pour afficher des bestioles, le même problème se produit : on veut pouvoir afficher des bestioles qu'on n'a pas encore définies mais la fonction d'affichage doit être présente ici car les bestioles n'ont pas à connaître la géographie de la planète... La fonction d'affichage est donc de type :
`display : (int -> int -> unit) -> pos -> unit`

et pour finir, il faut une fonction (sans argument ni résultat) `clear` qui efface l'affichage.

**9/**Écrire une signature de module `PLANETE` avec les bonnes définitions.

## 6.2 Module

Pour la planète Symbioz, on fera les choix suivants : la planète est un damier de $10 \times 10$ (utiliser des constantes `size_x` et `size_y`) et une position est représentée par un couple d'entiers. Quand on sort du damier d'un côté, on réapparaît de l'autre côté (gauche/droite ou haut/bas) [2].

**10/**Écrire un module `Symbioz:PLANETE`. Voir ci-dessous pour l'affichage.

## 6.3 Affichage

Dans un premier temps, on peut se contenter d'un affichage purement textuel. La fonction `clear` ne fait rien et la fonction `display` se contente alors de décomposer la position en ses deux coordonnées :
`let display f (x,y) = f x y`

Si on veut, on peut faire un affichage graphique à l'aide du module `Graphics` :
`http://caml.inria.fr/pub/docs/manual-ocaml/libgraph.html`

Pour l'activer, taper `#load "graphics.cma";;` dans le toplevel.

Dans ce cas, on aura intérêt à faire des fonctions annexes cachées (par exemple, pour convertir les position sur la planètes en coordonnées à l'écran).

La fonction `display` devient alors plus compliquée. En effet, on ne veut plus afficher un objet de la case $(i, j)$ aux coordonnées graphiques $(i, j)$ et il faut donc au moins traduire les position sur la planète en coordonnées à l'écran. De plus, si plusieurs bestioles se trouvent à la même position sur la planète, il faut trouver le moyen de les afficher toutes sans qu'elles se recouvrent complètement...

# 7 Génétique élémentaire (5 pts)

Chaque espèce est représentée par deux modules. D'une part un module qui gère les individus (et les couples) et d'autre par un module qui gère l'ensemble de la population. On s'occupe ici de gérer les individus. Comme le comportement des individus dépend de la planète sur laquelle ils se trouvent, il s'agira de foncteurs prenant une `PLANETE` en argument.

2. Symbioz est donc un tore...

## 7.1 signature

La gestion des individus doit comporter :
- Des types (abstraits) `pos` et `individu`.
- Un test d'égalité (pour savoir si deux objets de type `individu` représentent le même individu ou deux différents), une fonction qui crée un individu aléatoire.
- Des fonctions qui renvoient la position, le sexe et la classe d'âge d'un individu (pour les deux derniers, utiliser les types génériques définis au début).

Remarque : si on veut, on peut rajouter un `type t=individu` pour pouvoir utiliser les foncteurs standards d'`Ocaml`.

Pour faire vivre les individus, il faut aussi :
- Une fonction `manger` qui prend un entier (la quantité de nourriture absorbée) et un `individu` et renvoie un `individu option` (qui vaudra `None` si l'individu meurt de faim).
- Une fonction `bouger : (pos -> int) -> individu -> individu`. La fonction sera une évaluation de la valeur des différentes positions pour décider comment on bouge.
- Une fonction `reproduire` qui prend un entier et deux `individu` (le père et la mère) et renvoie une liste d'`individu` (les enfants). L'entier servira de paramètre supplémentaire en cas de besoin.
- Une fonction `vieillir` qui fait vieillir un `individu`. Le résultat sera une `option`, puisqu'il est possible de mourir de vieillesse.

Pour visualiser les choses, il faut aussi prévoir une fonction `afficher : individu -> unit`.

**11/**Écrire une signature `INDIVIDU` avec tout ce qu'il faut.

**12/**Écrire une signature `MAKE_INDIVIDU` pour les foncteurs qui prennent une `PLANETE` et renvoient un `INDIVIDU`. Penser au `with type` pour le type `pos`.

Astuce : le type `individu` sera un enregistrement. On a tout intérêt à prévoir un champ `id : int` qui contiendra un numéro d'identification unique de chaque individu. C'est en particulier ce champ qu'il suffit de tester pour l'égalité, ... Ça impose de conserver dans le module une référence `last_id` qui indique le dernier numéro utilisé pour pouvoir créer de nouveaux individus.

## 7.2 Zherbs

Les zherbs ont les caractéristiques suivantes :
- Elles n'ont pas de sexe. En particulier, la fonction `sexe` lèvera toujours une exception.
- Leur âge est uniquement déterminé par leur classe d'âge. Autrement dit une zherb vivra 3 tours, 1 dans chaque classe d'âge.
- Une zherb ne mange pas (vive la photosynthèse). Elle ne meurt jamais de faim.
- Une zherb ne bouge pas.
- Quand un enfant né, il a 10% de chance de se trouver dans chacune des quatre cases adjacentes et 60% d'être dans la case de ses parents.
- La reproduction concerne une zherb adulte à la fois (le deuxième parent est ignoré). Une zherb fait un nombre de bébés tiré au hasard entre 0 et le nombre d'autres zherbs présentes dans la même case (utiliser le paramètre additionnel de `reproduire` pour ce nombre puisqu'on ne peut pas le connaître pour le moment).

**13/**Écrire un module `Make_Zherb : MAKE_INDIVIDU`.

Remarque : il est maintenant possible de travailler directement sur la population (en particulier la population de zherbs) sans nécessairement créer les krapits et les kroguls.

## 7.3 Krapits

Les krapits ont les caractéristiques suivantes :
- Ils passent 1 tour bébé, 1 tour enfant et de 1 à 4 tours adultes (tiré au hasard au moment du passage à l'âge adulte). Leur classe d'âge ne suffit donc pas à déterminer leur vrai âge.
- Ils ont un certain nombre de *points de vie*. Quand un krapit mange une zherb, sa vie revient au maximum possible. Quand il ne mange pas, il consomme des points de vie. On pourra prendre par exemple 6 points de vie maximum et -1 point à chaque tour sans manger (créer des constantes nommées pour pouvoir changer facilement ces valeurs).

– Un krapit ne se déplace que si il a faim (si sa vie est inférieure à un seuil prédéfini). Dans se cas, il se déplace d'une case dans une direction tirée au hasard (il ignore la fonction d'évaluation des positions). On pourra prendre par exemple 3 comme seuil de faim.
– Deux krapits ne peuvent se reproduire que si ils sont tous les deux adultes et dans la même case. Ils font des portées de 1 à 5 bébés. Chaque bébé à 20% de chance de naître dans la case de ses parents et 20% de chances dans chacune des cases adjacentes.

**14/**Écrire un module `Make_Krapit : MAKE_INDIVIDU`.

Remarque : en pratique, on n'a pas besoin de tester ici si les krapits qui essayent de se reproduire sont bien adultes et dans la même case. C'est le module de gestion de la population qui s'en chargera.

### 7.4 Kroguls

Les kroguls ont les caractéristiques suivantes :
– Ils passent 2 tours bébé, 2 tours enfant et de 2 à 5 tours adulte.
– Ils ont des points de vie. Un krogul ne mange que si il a faim et manger un krapit lui fait regagner un certain nombre (fixe) de points de vie. Il perdent des points de vie si ils ne mangent pas. Par exemple 10 points de vie max, -2/tour sans manger, +5 si il mange, il ne mange que si il a 6 points de vie ou moins.
– Un krogul ne se déplace que si il est affamé (par exemple, 4 points de vie ou moins). En chasseur aguerri, il va dans la case adjacente la plus intéressante (utiliser la fonction d'évaluation) (au hasard en cas d'égalité).
– Deux kroguls qui se reproduisent ont entre 0 et 2 enfants. Chaque enfant à 5% de chance de naître dans chaque case adjacente, et 80% de naître avec ses parents.

**15/**Écrire un module `Make_Krogul : MAKE_INDIVIDU`.

## 8 Population (4 pts)

### 8.1 Signatures

Pour gérer une population, on a besoin :
– Des types abstraits `pos`, `individu`, `population` et `nourriture` (qui représentera une population de nourriture).
– D'une fonction pour créer une population aléatoire (prend en argument un entier : le nombre d'individu) et d'une fonction pour prendre un individu au hasard dans la population (renvoie une option, pour le cas de la population vide).
– D'une fonction qui renvoie la sous-population (c'est aussi une `population`) qui est dans une case donnée.
– D'une fonction qui tue un individu (et renvoie la population privée de cet individu).
– Des itérateurs `map`, `iter` et `reduce`. Ce dernier est de type `(individu -> 'a -> 'a) -> population -> 'a -> 'a`.
– Des fonctions `vieillissement` et `reproduction` qui font vieillir et reproduisent l'ensemble de la population.
– D'une fonction `mouvement` qui prend en argument une `nourriture` (pour pouvoir calculer une stratégie de mouvement) et une `population` et fait bouger chaque individu.
– D'une fonction `nourriture` qui prend une `nourriture` et une `population` et renvoie un couple `population * nourriture` (de la nourriture a été mangée, donc il faut renvoyer la nouvelle population de nourriture).
– D'une fonction `affichage` qui affiche une population.

Remarque : plusieurs de ces fonctions seront faites très facilement à l'aide des itérateurs...

Remarque : on peut rajouter les types `elt = individu` et `t = population` pour utiliser les foncteurs usuels d'`Ocaml`.

Remarque : en pratique, on choisira toujours des `individu list` comme `population`. Dans ce cas, les itérateurs sont directement ceux du module `List`, en particulier `reduce` est exactement `List.fold_right`

**16/**Écrire une signature `POPULATION`.

**17/**Écrire une signature `MAKE_PLANTES` (pour les populations en bas de la chaîne alimentaire qui n'ont pas de nourriture). C'est des foncteurs qui prennent comme argument une `PLANETE` et un `MAKE_INDIVIDU`

et renvoient une `POPULATION`. Outre les `with type` évident, rajouter `nourriture = unit` pour bien indiquer qu'il n'y a pas de nourriture.
**18/**Écrire une signature `MAKE_ANIMAUX`. C'est des foncteurs qui prennent en argument une `PLANETE`, une `POPULATION` (de proies) et un `MAKE_INDIVIDU` et renvoient une `POPULATION`. Penser à tours les `with type`.

### 8.2 Zherbs

**19/**Écrire un foncteur `Make_Zherbs : MAKE_PLANTES`.

Pour les action qui peuvent mener à la mort et renvoient des `option` (`manger` et `vieillir`), penser à utiliser `clean_list` pour nettoyer le résultat.

Seule la reproduction est un peu difficile. Le reste est en gros l'application des itérateurs avec les actions individuelles sur la population. Pour la reproduction, on doit connaître le nombre d'individus dans une case donnée, utiliser `at_pos` de la `PLANETE`. Pour la reproduction, ne pas oublier de renvoyer à la fin une population qui contient aussi bien les individus déjà présents que les nouveaux nés.
**20/**Créer le module `Zherbs` en appliquant ce foncteur aux bons arguments.

### 8.3 Animaux

**21/**Écrire un foncteur `Make_Bestioles : MAKE_ANIMAUX`.

Pour le mouvement, on ignorera ici la stratégie de mouvement, c'est-à-dire qu'on prend une fonction d'évaluation des positions qui est constante.

Pour la nutrition, une bestiole mange un individu de sa nourriture présent dans la même case (si il y en a). L'individu mangé en question est tiré au hasard. Penser à le tuer !

Là aussi, seule la reproduction est un peu technique. Il faut procéder par étapes (fonctions intermédiaires) :

- Ne garder que les adultes.
- Séparer les individus par case (utiliser `sort_by_pos`)
- Séparer dans chaque case les individus par sexes.
- Mélanger (séparément) les mâles et les femelles pour former des couples au hasard.
- Faire se reproduire chaque couple (attention à gérer le cas où il n'y a pas le même nombre de mâles et de femelles).

**22/**Créer les modules `Krapits` puis `Kroguls` en appliquant ce foncteur.

Remarque : les kroguls sont ici aussi peu organisés que les krapits dans leurs déplacements. Et ce n'est pas vraiment un compliment...

## 9 Jeu complet (2 pts)

Pour le jeu complet, on fera un foncteur qui prend en argument les bons modules. Avec les `with type` correct, il doit ressembler à :

```
module Make_Game =
  functor (P:PLANETE) ->
  functor (Plantes:POPULATION with type pos=P.pos
                              and type nourriture = unit) ->
  functor (Herbivores:POPULATION with type pos=P.pos
                                  and type nourriture = Plantes.population) ->
  functor (Carnivores:POPULATION with type pos=P.pos
                                  and type nourriture = Herbivores.population) ->
  struct

  ...

  end
```

On n'a pas besoin de faire de signature ici.

Le jeu se déroule par tour. Chaque tour se déroule en 4 étapes :

1. Les zherbs mangent, se reproduisent et bougent (même si dans l'exemple ici elles ne mangent pas, il faut les faire manger car on pourrait avoir une espèce de zherbs qui a besoin de se nourrir).
2. Les kroguls mangent, se reproduisent et bougent.
3. Les krapits mangent, se reproduisent et bougent.
4. Tout le monde vieillit.

**23/**Écrire le foncteur. L'appliquer avec les bons arguments pour créer un jeu... et admirer le résultat !

# 10 Bonus : évolution

Quelques possibilités de personnalisation de l'ensemble qui sont autant de pistes de réflexions et d'ouvertures possible pour un système plus complexe.

## 10.1 Chasse

Les kroguls peuvent sentir les krapits pour les chasser. Pour cela, il faut modifier leur stratégie de mouvement (dans la population seulement) pour donner à chaque case une valeur, par exemple le nombre de krapits. Ou la différence entre le nombre de krapits et le nombre de kroguls (il vaut mieux aller sur un territoire sans compétition).

Du coup, il faut maintenant séparer `Make_Bestioles` en deux foncteurs puisque les krapits et les kroguls ne se comportent plus pareil.

Peut-on facilement changer la stratégie de déplacement des krapits pour qu'ils fuient les kroguls ? Que faudrait-il faire pour autoriser une telle stratégie ?

## 10.2 Planète

Et si la planète n'est plus torique ? Comment représenter une planète un peu plus réaliste ? Qu'est-ce que ça change ?

Comment créer des continents et des océans sur la planète ? Des espèces qui ne savent pas nager, qui peuvent traverser des petits bras de mer ou d'autre capables de voler très loin au dessus de l'océan ?

Comment mettre des climats sur la planète (différent dans chaque case et plus ou moins favorables à la survie des espèces) ?

## 10.3 ADN

Chaque individu peut avoir un ADN. Par exemple, l'ADN des krapits peut donner des fonctions de déplacements différentes (pas les même probabilités de chaque côté, pas le même seuil de faim, ...) On peut aussi rajouter le numéro du tour en paramètre pour avoir par exemple un mouvement circulaire Nord/Est/Sud/Ouest et on recommence.

L'ADN des kroguls peut aussi concerner la stratégie de mouvement en changeant l'évaluation des cases. Dans ce cas, on peut vouloir que la fonction d'évaluation renvoie quelque chose de plus compliqué qu'un entier, sans doute un nouveau type à définir pour les individus et les populations.

L'ADN peut aussi coder des éléments de longévité (durée de vie dans chaque classe d'âge) ou de fertilité (nombre d'enfants à chaque reproduction), ou la position de naissance des enfants, ... Normalement, l'ADN ne devrait modifier en profondeur que des éléments du foncteur `MAKE_INDIVIDU` (et la marge des éléments du foncteur `MAKE_PLANTES` ou `MAKE_ANIMAUX` pour répercuter certaines de ces modifications comme le type de retour de la fonction d'évaluation des positions).

Lors de la reproduction, chaque parents transmet une partie de son ADN à ses enfants. On peut expérimenter différentes règles de transmissions (par exemple, tout l'ADN de la mère ; moitié/moitié au hasard ; une transmission qui dépend du sexe de l'enfant pour représenter des caractères "liés à l'X" ; ...)

Comment gérer des mutations de l'ADN lors de la reproduction ?

Si on visualise différemment les individus d'ADN différent, est-ce qu'on peut arriver à voir certains ADN se répandre plus vite parce qu'ils procurent un avantage évolutif ?

## 10.4 Compétition

En modifiant seulement le foncteur `Make_Game` (en lui ajoutant un argument), on peut avoir 2 espèces de carnivores qui sont en compétition (ils mangent les mêmes herbivores). En leur donnant des stratégie de chasse (ou des ADN) différentes, est-ce qu'on voit des situation intéressantes (création de territoires de chasses séparés ou conflit sur le même territoire ? Extermination d'une espèce au profit de l'autre ? Autre chose ?)

Que faut-il modifier pour pouvoir avoir 2 espèces de zherbivores (toutes les deux mangées par les carnivores) ? Pour avoir 2 espèces de zherbs ? Est-ce qu'on voit des situation intéressantes de cette manière ?

Si on veut gérer un nombre quelconque d'espèces de chaque type, il faut pouvoir passer un nombre quelconque de modules en argument à un foncteur. Comme souvent en `Ocaml`, ça se fait avec une liste (de modules !) qui est possible avec des *modules citoyens de première classe*. Voir :
`http://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec230`
... et bon courage pour se lancer là-dedans...