- BESNIER Corentin
- BONGIBAULT Mathieu
- HESSMANN Chloé

Programmation Fonctionnelle : Devoir Maison

Sommaire

1. Mode d'emploi

- Comment définir un évaluateur ?

Notre outil nécessite la définition manuelle d'un évaluateur pour pouvoir fonctionner. Sans lui, la majorité des fonctions ne pourraient plus s'éxecuter, et le programme en lui-même n'aurait plus aucun sens. Pour définir notre évaluateur, nous utilisons une application partielle de la fonction comparer :

let evaluateur = comparer "mot";;

Ainsi, notre évaluateur sert de mémoire et nous permettra de ne pas avoir à répéter le mot choisit lorsque nous voudrons le comparer avec d'autres mots.
Il est impératif que <u>mot</u> soit du type String.

- Comment appeller les diverses fonctions ?

Pour appeller nos diverses fonctions, il faut passer par le module Mystere. En effet, ce module représente l'application du foncteur Evolution sur le module MotGenetique, et contient par conséquent leurs structures respectives (on entend ici qu'on peut accèder aux fonctions d'Evolution et de MotGenetique en passant par le module Mystere). Ainsi, pour appeller une fonction, on utilisera la syntaxe :

Mystere.<u>nomdelafonction</u> paramètres ;;

* *Toutes les fonctions de notre programme portent les mêmes noms qu'indiqués dans le sujet.*

2. Répartition du travail

- Module MotGenetique :

- typage : Chloé
- fonction créer : Chloé
- fonction muter : Chloé
- fonction croiser : Chloé & Mathieu

- Foncteur Evolution :

- typage : Corentin & Mathieu
- fonction reproduction : Corentin
- fonction mutation : Corentin & Mathieu
- fonction selection : Mathieu
- fonction generation : Mathieu
- fonction evolution : Mathieu

- Module Mystere :

- définition du module : Corentin

- Fonctions et modules annexes :

- fonction comparer : Mathieu
- module de typage de foncteur EvolutionType : Mathieu & Corentin

- Compte-rendu :

- rédaction : Corentin

3. Explication du code

- Le module MotGenetique :

```
module MotGenetique : GENOME with type individu = string and type parametre = int =
struct
  type individu = string
  type parametre = int

  let creer = fun parametre ->
    let rand_chr () = char_of_int (97 + (Random.int 26)) in
    let rec boucle l parametre =
      if parametre == 0 then l else boucle (l ^(String.make 1 (rand_chr()))) (parametre-1) in
    boucle (String.make 1 (rand_chr())) (parametre-1)


  let rec muter = fun l ->
    let longueur = String.length l in let
      rand_chr () = char_of_int (97 + (Random.int 26)) in
    if l="" then ""
    else if Random.int 100 > 5 then
      let l' = String.make 1 l.[0] in
      l'^ muter (String.sub l (1) (longueur-1))
    else
      let l' = String.make 1 (rand_chr()) in
      l'^ muter (String.sub l (1) (longueur-1))

  let croiser = fun mot1 mot2 ->
    let lg = String.length mot1 in
    if lg = String.length mot2 then let n = (Random.int (lg) ) in
      (String.sub mot1 0 n )^(String.sub mot2 n (lg-n))
    else mot2
end
```

Tout d'abord, parlons de la signature. Ici, MotGenetique représente un héritage du module signature GENOME, fournit dans l'énoncé. A cela nous rajoutons deux nouveaux types : `type individu = string and type parametre = int`.
Ceux-ci nous permettent de faire comprendre à Ocaml qu'un objet de type individu est aussi du type String, et qu'un objet de type parametre est aussi du type int, nous permettant de relier toutes les fonctions qui suivront à notre évaluateur (qui est du type individu → int). Ces types seront aussi définis dans la structure du module MotGenetique sinon il ne respectera plus sa signature imposée.

Ensuite vient la fonction creer :

```
let creer = fun parametre ->
  let rand_chr () = char_of_int (97 + (Random.int 26)) in
  let rec boucle l parametre =
    if parametre == 0 then l else boucle (l ^(String.make 1 (rand_chr()))) (parametre-1) in
  boucle (String.make 1 (rand_chr())) (parametre-1)
```

```
val creer : parametre -> individu
```

Cette fonction est utilisée afin de créer un individu aléatoirement. Ici comme dans le reste du programme, un individu est représenté par une chaine de caractère. Ainsi, la fonction creer prend en entrée un entier n et nous ressort un individu (= chaine de caractères) de longueur n. Pour cela, elle utilise la fonction rand_chr pour choisir un caractère aléatoire. La fonction rand_chr sélectionne un entier aléatoire entre 1 et 26 et retourne la lettre associée à ce numéro, selon l'ordre alphabétique (1 = a | 26 = z). Ensuite, nous avons la fonction récurssive boucle. Cette fonction prend en entrée une chaine de caractère l ainsi qu'un entier parametre, puis elle rajoute à la chaine l un caractère donné par rand_chr et reboucle en enlevant 1 à parametre, et ce jusqu'à ce que parametre = 0.

Viens maintenant la fonction muter :

```
let rec muter = fun l ->
  let longueur = String.length l in let
    rand_chr () = char_of_int (97 + (Random.int 26)) in
  if l="" then ""
  else if Random.int 100 > 5 then
    let l' = String.make 1 l.[0] in
    l'^ muter (String.sub l (1) (longueur-1))
  else
    let l' = String.make 1 (rand_chr()) in
    l'^ muter (String.sub l (1) (longueur-1))
```

```
val muter : individu -> individu
```

Cette fonction est utilisée afin de provoquer (ou non) une mutation sur un individu. Pour cela, elle prend en entrée un individu, et pour chaque caractères de la chaine, il y aura 5% chance qu'il soit altéré par un nouveau caractère, choisi aléatoirement, puis elle retournera le nouvel individu une fois tous les caractères parcourus.

Enfin vient la fonction croiser :

```
let croiser = fun mot1 mot2 ->
  let lg = String.length mot1 in
  if lg = String.length mot2 then let n = (Random.int (lg) ) in
    (String.sub mot1 0 n )^(String.sub mot2 n (lg-n))
  else mot2
val croiser : individu -> individu -> individu
```

Cette fonction est utilisée afin de croiser deux individus, donnant naissance à un nouvel individu. Pour cela, la fonction prend en entrée deux individus (les parents). Ensuite, cette fonction va comparer la longueur des deux chaines. Si elles sont différentes, elle renverra le deuxième individu, sinon elle choisira un entier aléatoire n compris entre 0 et la longeur de chaine des individus, puis elle fusionnera les n premiers caractères du premier individu au deuxième, en enlevant les k premiers caractères du deuxième individu, de sorte à ce que l'enfant soit de même longueur que ses parents, puis retournera l'enfant.

- Le foncteur Evolution :

```
module Evolution : EvolutionType = functor (Mot:GENOME) ->
struct
  type population = Mot.individu list
  type evaluateur = Mot.individu -> int
  let rec reproduction l n =
    let taille = List.length l in
    if l == [] then l
    else
    if n == 0 then l
    else
      let x = List.nth l (Random.int(taille)) in
      let y = List.nth l (Random.int(taille)) in
      let enfant = Mot.croiser x y in
      if x == y || x == enfant || y == enfant then reproduction l n else reproduction (l@[enfant]) (n-1)
  let mutation liste=
    let rec muterliste liste liste2 indice=
      if indice == List.length liste then fst (List.split liste2)
      else muterliste liste (liste2@[(Mot.muter (List.nth liste indice),List.nth liste indice)]) (indice+1) in
    muterliste liste [] 0
  let selection = fun evaluateur pop nbrsurvivant ->
    let appliquerliste liste evaluateur=
      let rec creerlisteappliquer liste liste2 indice=
        if indice == List.length liste then liste2
        else creerlisteappliquer liste (liste2@[(evaluateur (List.nth liste indice),List.nth liste indice)]) (indice+
      creerlisteappliquer liste [] 0 in
    let rec couperliste valeur liste = match liste with
      | [] -> failwith "liste vide couperliste"
      | y::liste -> if valeur=1 then [y] else y::couperliste (valeur-1) liste in
    snd (List.split (couperliste nbrsurvivant (List.sort compare (appliquerliste pop evaluateur))))
  let generation = fun evaluateur nbrvivant nbrsurvivant pop ->
    mutation (reproduction ( selection evaluateur pop nbrsurvivant ) (nbrvivant-nbrsurvivant))
  let evolution = fun evaluateur parametre nb_individus nb_meilleurs max_iter ->
    let rec creerpop parametre nb_individus pop= if nb_individus ==0 then pop
      else creerpop parametre (nb_individus-1) (pop@[Mot.creer parametre]) in
    let rec evoluer evaluateur nb_individus nb_meilleurs max_iter popfinal =
      if max_iter == 0 then popfinal
      else evoluer evaluateur nb_individus nb_meilleurs (max_iter-1) (generation evaluateur nb_individus nb_meilleurs
    evoluer evaluateur nb_individus nb_meilleurs max_iter (creerpop parametre nb_individus [])
end
;;
```

Comme nous pouvons le remarquer dans la définition du foncteur, il est écrit qu'Evolution hérite d'un module EvolutionType. Ce module nous est nécessaire

afin de définir les types ainsi que les valeurs des fonctions que nous utilisons, de sorte à respecter avec exactitude le typage demandé dans l'énoncé. EvolutionType est la signature d'Evolution.

La première fonction est reproduction :

```
let rec reproduction l n =
  let taille = List.length l in
  if l == [] then l
  else
  if n == 0 then l
  else
    let x = List.nth l (Random.int(taille)) in
    let y = List.nth l (Random.int(taille)) in
    let enfant = Mot.croiser x y in
    if x == y || x == enfant || y == enfant then reproduction l n else reproduction (l@[enfant]) (n-1)
val reproduction : population -> int -> population
```

Cette fonction prend en entrée une population (= liste d'individus) ainsi qu'un entier n, et rajoute à la population n enfants, dont les parents sont choisis aléatoirement parmis la population. Pour cela, la fonction vérifie d'abord que la liste fournie n'est pas vide et que l'entier donné est différent de 0, conditions ne permettant pas le bon fonctionnement du programme. Ensuite, elle sélectionne deux individus aléatoirement dans la population, puis effectue leur croisement via la fonction Mot.croiser. Pour finir, elle vérifie que les deux parents choisis ne sont pas les mêmes et qu'ils soient différents de l'enfant (afin d'éviter qu'un individu puisse se reproduire avec lui même). Si c'est le cas, elle rajoute l'enfant à la population puis se réitère en enlevant 1 à n, jusqu'à ce que n = 0. Sinon, elle ne rajoute pas l'enfant et se réitère (sans modifier n). Elle retournera la nouvelle population à la fin de son itération.

La deuxième fonction est mutation :

```
let mutation liste=
  let rec muterliste liste liste2 indice=
    if indice == List.length liste then fst (List.split liste2)
    else muterliste liste (liste2@[(Mot.muter (List.nth liste indice),List.nth liste indice)]) (indice+1) in
  muterliste liste [] 0
val mutation : population -> population
```

Cette fonction prend en entrée une population et ressort une nouvelle population ayant subit des mutations. Pour cela, à chaque itération de muterliste, on rajoute dans liste2 le mot d'indice n dans liste (n correspondant à la valeur de indice) ayant subit une mutation via la fonction Mot.muter, puis on réitère jusqu'à ce que indice soit égal à la longueur de liste, avant de renvoyer liste 2.

La troisième fonction est selection :

```
let selection = fun evaluateur pop nbrsurvivant ->
   let appliquerliste liste evaluateur=
     let rec creerlisteappliquer liste liste2 indice=
       if indice == List.length liste then liste2
       else creerlisteappliquer liste (liste2@[(evaluateur (List.nth liste indice),List.nth liste indice)]) (indice+1) in
     creerlisteappliquer liste [] 0 in
   let rec couperliste valeur liste = match liste with
     | [] -> failwith "liste vide couperliste"
     | y::liste -> if valeur=1 then [y] else y::couperliste (valeur-1) liste in
   snd (List.split (couperliste nbrsurvivant (List.sort compare (appliquerliste pop evaluateur))))
val selection : evaluateur -> population -> int -> population
```

Cette fonction permet de sélectionner parmis une population d'individus ceux remplissants le plus les critères de l'évaluateur. En effet, en définissant notre évaluateur au début, nous lui avions donné une chaine de caractère caractérisant un critère. Ici, nous comparons l'évaluateur aux individus d'une population, selectionnant les nbrsurvivant individus remplissant le plus les conditions de l'évaluateur. Dans notre cas, le critère attribué à l'évaluateur est un individu où l'on compare son taux de ressemblance avec un autre individu, via la fonction comparer. Ainsi, la fonction selection effectue une comparaison des différences entre chaque individus de la population et notre individu évaluateur, et renvoit les nbrsurvivant individus ressemblants le plus à l'individu évaluateur. Pour cela, la fonction parcours la liste et crée une autre liste de tuples appelée liste2, dans laquelle elle stocke le mot ainsi que le nombre de différences avec l'individu évaluateur. Ensuite, elle trie liste2 en rangeant le nombre de différences par ordre croissant, puis elle retourne les x premiers individus de liste2 (x = nbrsurvivant)

Notre quatrième fonction est generation :

```
let generation = fun evaluateur nbrvivant nbrsurvivant pop ->
  mutation (reproduction ( selection evaluateur pop nbrsurvivant ) (nbrvivant-nbrsurvivant))
val generation : evaluateur -> int -> int -> population -> population
```

Cette fonction applique la fonction selection à une population1, puis reproduction à cette nouvelle population2, terminant par une mutation de population2, avant de nous retourner cette nouvelle population2. Afin de toujours conserver un nombre suffisant d'individus dans la population pour permettre une nouvelle sélection, reproduction donnera toujours m–n enfants (m étant le nombre de vivants souhaité à la fin de génération et n le nombre de survivants à la sélection de génération)

Enfin, la dernière fonction de notre foncteur est evolution :

```
let evolution = fun evaluateur parametre nb_individus nb_meilleurs max_iter ->
    let rec creerpop parametre nb_individus pop= if nb_individus ==0 then pop
      else creerpop parametre (nb_individus-1) (pop@[Mot.creer parametre]) in
    let rec evoluer evaluateur nb_individus nb_meilleurs max_iter popfinal =
      if max_iter == 0 then popfinal
      else evoluer evaluateur nb_individus nb_meilleurs (max_iter-1) (generation evaluateur nb_individus nb_meilleurs popfinal)in
    evoluer evaluateur nb_individus nb_meilleurs max_iter (creerpop parametre nb_individus [])
val evolution :
  evaluateur -> Mot.parametre -> int -> int -> int -> population
```

Cette fonction permet de donner une population résultante du passage d'un certain nombre de générations (défini par l'utilisateur) sur une population générée à partir d'un parametre (lui aussi fournit par l'utilisateur). Pour cela, elle va d'abord créer une population via la fonction creerpop, puis applique x fois la fonction generation à cette population (x = max_iter).

- Le module Mystère :

Ce module symbolise l'application du foncteur Evolution au module MotGenetique. Ainsi, il ne comporte pas de structure propre et hérite de la signature du foncteur :

```
module Mystere :
  sig
    type population = MotGenetique.individu list
    type evaluateur = MotGenetique.individu -> int
    val reproduction : population -> int -> population
    val mutation : population -> population
    val selection : evaluateur -> population -> int -> population
    val generation : evaluateur -> int -> int -> population -> population
    val evolution :
      evaluateur -> MotGenetique.parametre -> int -> int -> int -> population
  end
```

- La fonction comparer :

Il nous reste une dernière fonction à expliquer dans notre programme : la fonction comparer.

```
let comparer = fun mot1 -> fun mot2 ->
  if String.length mot1 != String.length mot2 then
    failwith "les gênes ne font pas la même taille"
  else let rec enumerer x y z =
         if x = "" then
           z
         else
         if String.sub x 0 1 = String.sub y 0 1 then
           enumerer (String.sub x 1 ((String.length x)-1)) (String.sub y 1 ((String.length y)-1)) z else
           enumerer (String.sub x 1 ((String.length x)-1)) (String.sub y 1 ((String.length y)-1)) z+1
    in enumerer mot1 mot2 0 ;;
val comparer : string -> string -> int = <fun>
```

Cette fonction prend en paramètre deux String et retourne un entier symbolisant le nombre de différences entre ces deux mots. Pour cela, elle vérifie d'abord que les deux mots font la même longueur, condition indispensable pour pouvoir faire la comparaison (elle retournera une erreur dans le cas où ils ne feraient pas la même longueur). Ensuite, elle va regarder si le premier caractère de ces deux mots sont les mêmes. Si oui, elle effaçera ces deux caractères et se réitèrera jusqu'à ce qu'il ne reste plus rien. Si les deux caractères sont différents, elle incrémentera un entier

nommé z de 1 avant d’effectuer la même opération. A la fin, la fonction retournera z.