1 Introduction

De nombreux langages de programmation (des centaines...) existent ! Bien entendu, certains présentent des similarités, mais des fonctionnalités bien différentes peuvent aussi apparaître. Il existe notamment différents **paradigmes** de programmation qui correspondent à la façon d'envisager et de concevoir un programme (ceci sera développé plus précisément en classe de Terminale).

Au final, un langage de programmation finit toujours par être traduit en langage machine qui dépend de l'architecture matérielle sur laquelle le programme doit s'exécuter. Les langages de haut niveaux qui suivent le paradigme de programmation **impératif** (et procédural) se ressemblent assez et sont en quelque sorte une réécriture plus simple pour les humains du langage machine. Les langages qui suivent le paradigme **fonctionnel** s'écartent de l'aspect impératif du langage machine (suite d'instructions avec un aspect séquentiel) pour décrire *ce que sont* les choses plutôt que de décrire *comment* calculer les choses.

Au travers d'un exemple (mise en œuvre de l'algorithme de tri par sélection), nous verrons un aperçu de quelques langages pour mettre en avant leur similarités et différences.

Les langages que nous évoquerons ne sont pas à connaître (Python, C, Java, Erlang, Haskell). Ils ne servent qu'à illustrer le propos.

Pour approfondir cette découverte de la diversité langages, vous pouvez consulter très utilement le site suivant : `https://www.literateprograms.org/`

2 Tri pas sélection en langage impératif

Rappel : le tri par sélection repose sur l'idée qu'au cours de l'algorithme, la partie gauche du tableau est triée, et qu'il reste la partie droite à trier. On doit donc rechercher le plus petit élément non trié et l'échanger avec le premier élément non trié. On progresse ainsi jusqu'à ce que la partie non triée soit vide.
Voici ce qu'on peut écrire de façon condensée en pseudo-code :

```
Fonction TriSelection(tab):
  Pour i de 1 à Taille(tab):
     Chercher le minimum dans tab[i:fin]
     Permuter tab[i] et minimum
```

2.1 Implémentation en Python

```
def triSelection(tab):
  for i in range(len(tab)-1):
     # recherche du minimum :
     i_mini = i
     for j in range(i+1, len(tab)):
        if tab[j] < tab[i_mini]:
           i_mini = j
     # permutation de tab[i] et minimum :
     tab[i], tab[i_mini] = tab[i_mini], tab[i]
```

2.2 Implémentation en C

```
void triSelection(int tab[], int N) // N est la taille du tableau
{
   int i;
   for(i=0; i<N; i++)
   {
      // recherche du minimum :
      int i_mini = i;
      int mini = tab[i];
      int j;
      for(j=i+1; j<N; j++)
      {
         if (tab[j]<mini)
         {
            i_mini = j;
            mini = tab[j];
         }
      }
      // permutation de tab[i] et minimum :
      tab[i_mini] = tab[i];
      tab[i] = mini;
   }
}
```

2.3 Implémentation en Java

```
public static void triSelection(int[] tab)
{
   for (int i=0; i < tab.length; i++)
   {
      // recherche du minimum :
      int i_mini = i;
      int mini = tab[i];
      for (int j = i+1; j < tab.length; j++)
      {
         if (tab[j] < mini)
         {
            i_mini = j;
            mini = tab[j];
         }
      }
      // permutation de tab[i] et minimum :
      tab[i_mini] = tab[i];
      tab[i] = mini;
   }
}
```

2.4 Conclusion

Les structures de ces langages sont très similaires. Voici quelques différences principales :
- Les variables doivent être déclarées, et typées. Ceci est une différence fondamentale et très importante avec Python.
- Les blocs de code sont délimités par des accolades (et non pas avec des indentations comme en Python, même si on tend à les utiliser pour facilité la lecture du code).
- Les autres différences ne relèvent que de la syntaxe précise du langage, mais n'ont rien de fondamental.

3 Tri pas sélection en langage fonctionnel

Rappel : les langages fonctionnels reposent beaucoup sur la notion de récursivité (qui sera étudiée en détail en classe de Terminale) : càd qu'une fonction peut s'appeler elle-même.

3.1 Implémentation en Erlang

Ce langage n'étant pas forcément compréhensible directement avec vos connaissances de Python, nous décrirons étape par étape le code en commentant grossièrement les idées directrices.

Le code de la fonction de tri repose principalement sur 2 variables à chaque étape :
- une liste qui contient les éléments non triés (listeNonTriee)
- une liste qui contient les éléments déjà triés (listeTriee)

On écrit un module *triSelection* qui exporte la fonction *tri* avec 1 paramètre, donc on doit définir une fonction interne à 2 paramètres :

```
-module(triSelection).
-export([tri/1]).
tri(L) when is_list(L) -> tri(L, []).
```

L'idée directrice est : si on appelle *tri* avec 1 paramètre L qui est une liste, on appellera en fait *tri* avec 2 paramètres : L et une liste vide.

Le tri se termine lorsque la liste non triée est devenue vide. (*Remarque* : il faudra renverser la liste triée à la fin, parce que les éléments auront été ajoutés en tête) :

```
tri([], ListeTriee) -> lists:reverse(ListeTriee);
```

L'idée directrice est : si on appelle tri avec 2 paramètres qui respectent le motif d'une liste vide et d'une liste non vide, on renvoie une copie de cette liste renversée.

S'il reste des éléments dans la liste non triée, on extrait le plus petit élément de cette liste et on l'ajoute en tête de la liste triée. Puis on rappelle la fonction de tri avec le reste de la liste non triée et la liste triée mise à jour :

```
tri(ListeNonTriee, ListeTriee) ->
    {Mini, Reste} = extraire_mini(ListeNonTriee),
    tri(Reste, [Mini | ListeTriee]).
```

Fonction de sélection :
Il faut donc définir une fonction d'extraction du minimum d'une liste. Cette fonction `extraire_mini` prend une liste en paramètre et renvoie l'élément minimum de cette liste, et le reste de la liste privé de cet élément.

Ici, nous avons besoin de 3 variables :
- la liste des éléments non analysés (éléments de la liste qui pourraient contenir le miminum),
- la valeur du minimum en cours,
- la liste des éléments déjà analysés (ce sera le reste de la liste à renvoyer).

Nous commençons par indiquer que cette fonction prend un paramètre qui est une liste respectant le motif (ÉlémentDeTête-QueueDeListe) et comment "découper" ce paramètre en 3 paramètres distincts (QueueDeListe, ÉlémentDeTête, ListeVide) :

```
extraire_mini([Tete | Queue]) -> extraire_mini(Queue, Tete, []).
```

S'il n'y a aucun élément non analysé, cela signifie qu'on a trouvé le minimum et le reste de la liste, et donc qu'on peut s'arrêter :

```
extraire_mini([], Mini, Reste) -> {Mini, Reste};
```

Si nous trouvons un plus petit élément dans la liste des éléments non analysés, il devient le nouveau minimum et l'ancien minium doit être ajouté dans la liste des éléments déjà analysés (le Reste) :

```
extraire_mini([Tete | Queue], Mini, Reste)
    when Tete < Mini ->
    extraire_mini(Queue, Tete, [Mini | Reste]);
```

Sinon, il suffit d'ajouter l'élément de tête en cours d'analyse à la liste des éléments déjà analysés :

```
extraire_mini([Tete | Queue], Mini, Reste) ->
    extraire_mini(Queue, Mini, [Tete | Reste]).
```

Tous ces éléments assemblés donnent le code complet :

```
-module(triSelection).
-export([tri/1, extraire_mini/1]).

tri(L) when is_list(L) -> tri(L, []).
tri([], ListeTriee) -> lists:reverse(ListeTriee);
tri(ListeNonTriee, ListeTriee) ->
    {Mini, Reste} = extraire_mini(ListeNonTriee),
    tri(Reste, [Mini | ListeTriee]).

extraire_mini([Tete | Queue]) -> extraire_mini(Queue, Tete, []).
extraire_mini([], Mini, Reste) -> {Mini, Reste};
extraire_mini([Tete | Queue], Mini, Reste)
    when Tete < Mini ->
    extraire_mini(Queue, Tete, [Mini | Reste]);
extraire_mini([Tete | Queue], Mini, Reste) ->
    extraire_mini(Queue, Mini, [Tete | Reste]).
```

3.2 Implémentation en Haskell

Voici une autre vision du même programme.

Calcul du minimum d'une liste :
Cette fonction renvoie le plus petit élément d'une liste :

```
mini:: (a->a->Bool)->[a]->a
mini (<=) [] = undefined
mini (<=) [x] = x
mini (<=) (x:xs)
  | x <= (mini (<=) xs) = x
  | otherwise = mini (<=) xs
```

Quelques explications :

- La signature (1ère ligne) indique que cette fonction prend 2 paramètres :
  1) Une fonction qui prend elle-même 2 paramètres de type identique a (ex : des entiers) et renvoie un booléen.
  2) une liste d'éléments du type a (ex : des entiers).
- Pour savoir ce que renvoie cette fonction (la signature indique que c'est un élément toujours du type a), on retrouve la notion d'application de cette fonction à différents motifs comme en Erlang.
  1) le 1er paramètre est toujours la fonction booléenne de comparaison (<=).
  2a) si le 2nd paramètre est une liste vide : renvoie un valeur indéfinie
  2b) si le 2nd paramètre est une liste d'un seul élément x : renvoie cet élément x
  2c) si le 2nd paramètre est une liste dont la tête est x et la queue xs :
  -> renvoie x si x est inférieur au minimum de xs
  -> sinon renvoie le minium de xs

Suppression d'un élément d'une liste :
Cette fonction prend un élément et une liste en entrée et renvoie une liste qui ne contient plus cet élément (sa 1ère occurrence) :

```
supprime:: (Eq a) => a->[a]->[a]
supprime a [] = []
supprime a (x:xs)
  | a==x = xs
  | otherwise = x:(supprime a xs)
```

Quelques explications :

- Cette fonction prend 2 paramètres :
  1) un paramètre de type a (qui supporte les opérateurs de comparaison).
  2) une liste d'éléments du type a (ex : des entiers).
- La signature indique qu'elle renvoie une liste d'éléments du type a. Analysons les motifs pour savoir ce qu'elle renvoie exactement :
  1) appliquée à un élément a et une liste vide : renvoie une liste vide
  2) appliquée à un élément a et une liste dont la tête est x et la queue xs :
  -> renvoie xs si a = x
  -> sinon renvoie une liste dont la tête est x et la queue est la liste xs dont on a supprimé a.

Algorithme de tri complet :
Cette fonction implémente le tri par sélection de façon complète :

```
tri:: (Eq a) => (a->a->Bool)->[a]->[a]
tri (<=) [] = []
tri (<=) xs = [x] ++ tri (<=) (supprime x xs) where x = mini (<=) xs
```

Quelques explications :

- La signature indique que cette fonction prend 2 paramètres :
  1) Une fonction qui prend elle-même 2 paramètres de type identique a (supportant les comparaisons) et renvoie un booléen.
  2) une liste d'éléments du type a (ex : des entiers).
- La signature indique qu'elle renvoie une liste d'éléments du type a. Analysons les motifs pour savoir ce qu'elle renvoie exactement :
  1) le 1er paramètre est toujours la fonction booléenne de comparaison (<=).
  2a) si le 2nd paramètre est une liste vide : renvoie une liste vide.
  2b) si le 2nd paramètre est une liste xs : renvoie la concaténation :
  –> d'une liste d'un unique élément x,
  –> et d'une liste qui résulte du tri de la liste xs à laquelle on a retiré x ;
  l'élément x étant le minimum de xs !