éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

DIVERSITÉ ET UNITÉ DES LANGAGES DE PROGRAMMATION

1. Introduction

Il n'existe pas de langage de programmation « universel ». En effet, si l'on peut, en général, tout programmer avec n'importe quel langage, certains langages seront plus adaptés à la programmation de telle ou telle tâche. Le langage Python, par exemple, est très utilisé pour la mise en place rapide de programmes. Cependant, dès lors que l'on a besoin de performances plus élevées, ce n'est plus le langage le plus adapté. Un exemple typique est la bibliothèque `numpy` qui est programmée en langage C (bien qu'elle soit prévue pour être ensuite utilisée avec Python). En parallèle, l'évolution de techniques concernant différents aspects des langages de programmation (comme la programmation orientée objets, la gestion de la mémoire, des exceptions, des adresses mémoires) fait que l'on désire définir de nouveaux langages (ou d'en faire évoluer d'autres) pour inclure de nouvelles techniques.
Il en résulte que depuis la création de l'informatique, on a vu sans cesse des nouveaux langages être créés, souvent inspirés de langages déjà existant et incorporant ou développant de nouveaux aspects.
L'étude de l'évolution des langages de programmation permet ainsi d'illustrer le développement et l'évolution de différentes techniques ainsi que l'émergence de nouvelles idées. Déjà en 1976, Donald Knuth et Luis Trabb Pardo [1] avaient illustré ces évolutions en analysant et comparant un même algorithme dans plusieurs langages.

Nous reprenons ici cette idée en la modernisant et proposant quelques activités autour de l'implémentation d'un algorithme dans 15 langages de programmation.

2. Le mélange de Fischer-Yates

L' algorithme que nous allons utiliser est l'*algorithme de mélange de Fischer-Yates*, aussi appelé algorithme de Knuth. Cet algorithme prend en entrée un tableau (on considérera, au besoin, qu'il contient des entiers) et mélange l'ordre de ses éléments. Il ne renvoie rien, mais le tableau a été modifié (on parle d'algorithme *en place*).
Pour cela, l'algorithme procède très simplement : il parcourt le tableau de gauche à droite (en partant du deuxième indice) et, pour chaque indice `i`, il tire un indice `j` au hasard parmi tous les indices entre le premier et l'indice `i` et échange les valeurs des cases correspondantes (un test permet de ne pas faire d'échange lorsque `j` est égal à `i`).

Une implémentation possible en Python est la suivante :

```
1 def shuffle(tab):
2     for i in range(1, len(tab)):
```

```
        j = random.randint(0, i)
        if j < i:
            tab[i], tab[j] = tab[j], tab[i]
```

Le choix de cet algorithme est motivé par les points suivants :

1. il est très simple, court et facile à implémenter,
2. il comporte une boucle et un test,
3. il traîte un tableau,
4. il nécessite l'appel à une fonction prédéfinie pour engendrer un nombre aléatoire.

Il permet donc d'illustrer en peu de lignes les principaux aspects de la programmation impérative. Les figures 1 à 15 présentent des implémentations de cet algorithme dans des langages de programmation différents.

3. Questions et activités

Pour les différentes propositions d'activités qui suivent, il est suggéré de distribuer aux étudiants les différentes implémentations du même algorithme, en ayant éventuellement découpé les feuilles pour séparer les différents langages afin de faciliter les manipulations.

3.1. Lecture

Il est intéressant, tout d'abord, de lire les différents programmes pour repérer leur structure. Celle-ci est quasiment toujours la même : dans le corps de la fonction, on a une boucle `for` avec, à l'intérieur, le tirage aléatoire de la variable `j` suivi du test menant éventuellement à l'échange des valeurs de `tab` d'indices `i` et `j`.

3.2. Classification

Une fois repérée cette structure commune, on peut se pencher sur les variations, le même élément pouvant apparaître sous des formes différentes. Dans ces variations, on pourra de plus voir des similitudes entre différents langages, ce qui suggère que l'on peut les regrouper, que l'on peut opérer une classification parmi eux.
Pour orienter cette réflexion sur la classification entre langages de programmation, on pourra en particulier se pencher sur les points suivants :

- ▷ Comment les blocs d'instructions à l'intérieur d'un boucle ou d'un test sont-ils délimités ?
- ▷ Comment sont structurées les boucles `for` ? les tests ?
- ▷ Est-il nécessaire de déclarer les variables locales ? Si oui, où ?
- ▷ Les types de données sont-ils spécifiés explicitement ? Doit-on indiquer que telle variable correspond à un entier ou un tableau ?

Normalement, à l'issue de ces interrogations, vous devez avoir identifié des similitudes entre certains langages.

3.3. Chronologie

Un autre exercice intéressant que l'on peut faire est le suivant : trier les différents programmes par ordre chronologique de création du langage correspondant. Il est bien sûr illusoire d'espérer obtenir un classement exact, mais un certain nombre de questions permettent de réfléchir à l'évolution des langages. Nous présentons quelques pistes de réflexions, mais il faut garder à l'esprit que l'on n'a la plupart du temps que des tendances, et qu'il y a souvent des exceptions.

- ▷ **Verbosité** Les langages de programmation les plus anciens sont souvent assez verbeux : beaucoup de lignes sont nécessaires pour décrire la structure, préciser les variables, les types de données. Avec les progrès des analyseurs de code, un grand nombre de ces informations peut être inféré, ce qui permet de ne pas les écrire explicitement.
- ▷ **Variables locales** Avec le temps, la gestion des variables locales se simplifie considérablement. Au début, elles doivent toutes être déclarées au début de la fonction (parfois dans une section spécifique) avec leur type. Ensuite, deux évolutions apparaissent : elles peuvent être déclarées « en cours de route » (voire pas du tout) et la spécification de leur type n'est plus systématiquement nécessaire.

```ada
procedure Shuffle (Tab : in out Array_Type) is
   package Discrete_Random is new Ada.Numerics.Discrete_Random(Result_Subtype =>
   ↪ Integer);
   use Discrete_Random;
   J : Integer;
   G : Generator;
   TMP : Element_Type;
begin
   Reset (G);
   for I in Tab'Range loop
      J := (Random(G) mod I) + 1;
      if J < I then
        TMP := Tab(I);
        Tab(I) := Tab(J);
        Tab(J) := TMP;
      end if;
   end loop;
end Shuffle;
```

FIGURE 1 – Ada

▷ **Tableaux** Au départ, ce ne sont que des plages mémoires allouées à la bonne taille. Ensuite, avec le temps, ils deviennent des structures de données représentées de façon un peu plus sophistiquées. En particulier, une information importante comme la taille du tableau devenant disponible à l'aide de fonctions ou de méthodes, il n'est plus nécessaire de la passer explicitement en argument. De même, au début, il y avait deux conventions pour indexer les cases d'un tableau : en partant de 1 ou de 0. De nos jours, la première convention a quasiment disparu au profit de la seconde.

4. Un arbre généalogique de langages de programmation

En guise de solutions aux activités précédents, on trouvera en figure 16 un arbre généalogique succinct indiquant les liens de parenté entre les différents langages présentés précédemment.

Références

[1] Donald E. Knuth and Luis Trabb Pardo. The early development of programming languages. http://bitsavers.org/pdf/stanford/cs_techReports/STAN-CS-76-562_EarlyDevelPgmgLang_Aug76.pdf.

```
procedure shuffle (tab, n);
  array tab; integer n;
  begin
    integer i, j, tmp;
    for i := 1 step 1 until n do
    begin
      j := random(i + 1);
      if j < i then
      begin
        tmp := tab[i];
        tab[i] := tab[j];
        tab[j] := tmp;
      end
    end
  end shuffle
```

FIGURE 2 – Algol

```
void shuffle(int tab[], int n) {
  int i;
  for (i = 1; i < n; ++i) {
    int j = random(i + 1);
    if (j < i) {
      int tmp = tab[i];
      tab[i] = tab[j];
      tab[j] = tmp;
    }
  }
}
```

FIGURE 3 – C

```
100 FOR I = 1 TO LONGUEUR
110 J = INT(RND(1) * I + 1)
120 IF I = J THEN GOTO 160
130 TMP = TAB(I)
140 TAB(I) = TAB(J)
150 TAB(J) = TMP
160 NEXT I
170 END
```

FIGURE 4 – Basic

```
let shuffle tab =
  for i = 1 to Array.length tab - 1 do
    let j = Random.int (i + 1) in
    if j < i then (
      let temp = tab.(i) in
      tab.(i) <- tab.(j);
      tab.(j) <- temp
    )
  done
;;
```

FIGURE 5 – Caml

```
IDENTIFICATION DIVISION.
PROGRAM-ID. shuffle.

DATA DIVISION.
LOCAL-STORAGE SECTION.
01  i                      PIC 9(8).
01  j                      PIC 9(8).
01  temp                   PIC 9(8).

LINKAGE SECTION.
78  Table-Len              VALUE 10.
01  ttable-area.
    03  ttable             PIC 9(8) OCCURS Table-Len TIMES.

PROCEDURE DIVISION USING ttable-area.
    PERFORM VARYING i FROM 2 BY 1 UNTIL i = Table-Len
        COMPUTE j =
            FUNCTION MOD(FUNCTION RANDOM * 10000, Table-Len) + 1
        If j < i
          MOVE ttable (i) TO temp
          MOVE ttable (j) TO ttable (i)
          MOVE temp TO ttable (j)
        END-IF
    END-PERFORM

    GOBACK
    .
```

FIGURE 6 – Cobol

```ruby
def shuffle(tab)
  tab.each_index do |i|
    j = rand(i + 1)
    if j < i
      tab[i], tab[j] = tab[j], tab[i]
    end
  end
end
```

FIGURE 7 – Ruby

```fortran
  subroutine shuffle (tab, n)
    integer n, tab(*)
    integer i, j, temp
    real r

    do 10 i = 2, n
      call random_number(r)
      j = int(r * i) + 1
      if (j < i) then
        temp = tab(j)
        tab(j) = tab(i)
        tab(i) = temp
      endif
10  continue
    return
  end
```

FIGURE 8 – Fortran

```go
func shuffle(tab []int) {
  for i := 1; i < len(tab); i++ {
    j := rand.Intn(i + 1)
    if j < i {
      tmp := tab[i]
      tab[i] := tab[j]
      tab[j] := tmp
    }
  }
}
```

FIGURE 9 – Go

```java
public static void shuffle (int[] tab) {
  for (int i = 1; i < tab.length; i++) {
    int j = gen.nextInt(i + 1);
    if (j < i) {
      int temp = tab[i];
      tab[i] = tab[j];
      tab[j] = temp;
    }
  }
}
```

FIGURE 10 – Java

```python
  def shuffle(tab):
    for i in range(1, len(tab)):
      j = random.randint(0, i)
      if j < i:
        tab[i], tab[j] = tab[j], tab[i]

```

FIGURE 11 – Python

```javascript
function shuffle(tab) {
  for (var i = 1; i < tab.length; i++) {
    var j = Math.floor((i + 1) * Math.random());
    if (j < i) {
      var temp = tab[rand];
      tab[rand] = tab[i];
      tab[i] = temp;
    }
  }
}
```

FIGURE 12 – Javascript

```kotlin
fun shuffle(tab: Array<Int>) {
  for (i in 1 until tab.size) {
    val j = (0..i).random()
    if (j < i) {
      val tmp = tab[i]
      tab[i] = tab[j]
      tab[j] = tmp
    }
  }
}
```

FIGURE 13 – Kotlin

```pascal
procedure shuffleList(var tab: tableau);
var
  i, j, tmp : integer;
begin
  for i := 1 to high(tab) - low(tab) do begin
    j := random(i + 1);
    if j < i then
      tmp := tab[i + low(tab)];
      tab[i + low(tab)] := tab[j + low(tab)];
      tab[j + low(tab)] := tmp
    end
  end
end;
```

FIGURE 14 – Pascal

```scala
def shuffle(tab: Array[Int]) = {
  for (i <- 1 to tab.size - 1) {
    val j = util.Random nextInt(i + 1)
    if (j < i) {
      val tmp = tab(i)
      tab(i) = tab(j)
      tab(j) = tmp
    }
  }
}
```

FIGURE 15 – Scala

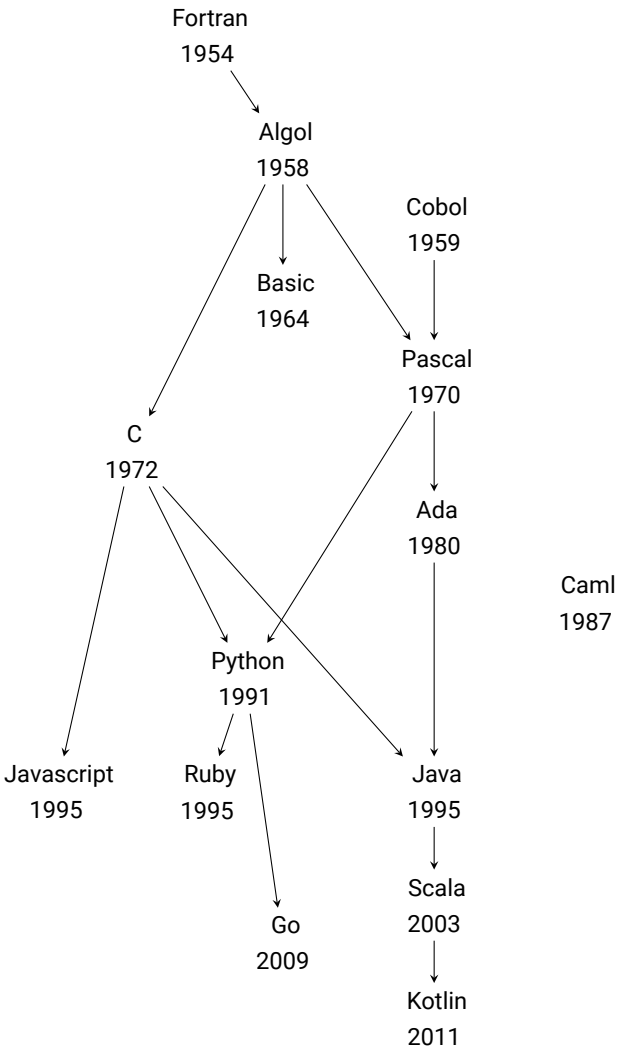

FIGURE 16 – Généalogie des langages de programmation