Informatique en MP(2)I

Marc de Falco

Introduction à la programmation (impérative)

Source de l'image : https://www.flickr.com/photos/binaryape/5151286161/

■ **Note 1 Roadmap** :
- finir l'écriture avec `Python`.
- rajouter `OCaml` et `C`.
- en profiter pour faire une introduction aux référence en `OCaml` quitte à rajouter une partie spécifique.
- rajouter beaucoup d'exemples et d'exercices.

■

I Introduction

Dans ce chapitre introductif sur la programmation, on va présenter celle-ci à travers quatre notions permettant de comprendre ce que signifie programmer. Un certain soin a été porté au fait de rendre cette présentation indépendante du langage tout en permettant qu'elle serve de support pour l'apprentissage d'un langage particulier. Ainsi, les éléments propres à un langage donnée sont clairement séparé.

Comme nous l'avons vu dans le chapitre d'introduction, la programmation consiste à décrire des algorithmes et la manière dont ils sont mis en œuvre sur une machine. En cela, le modèle que l'on suit ici est celui de la programmation dite impérative structurée. Pour autant, de nombreuses notions présentées ici restent valides dans d'autres paradigmes.

La présentation faite ici est indépendante du langage sous réserve que celui-ci comporte des traits impératifs structurés. Pour autant, selon les langages certaines notions sont plus ou moins riches. Des paragraphes spécifiques complètent ainsi le texte et on encourage le lecteur à basculer d'un langage à un autre tout au long de la lecture.

`OCaml` est un langage qui est avant tout fonctionnel déclaratif. Pour autant, il possède des traits impératifs et permet d'écrire aussi naturellement des programmes impératifs que dans des langages comme `C` ou `Python`. Cependant, cette couche impérative se rajoute par-dessus la couche fonctionnelle et on supposera que le lecteur la maîtrise déjà. Ainsi, dans la présentation qui suit, on se concentrera sur l'exposition spécifique des notions impératives.

OCaml

I.1 Les quatre étages de la programmation

Programmer c'est

- **représenter** des données, allant du booléen valant vrai ou faux, jusqu'aux bases de données permettant de modéliser des relations complexes
- **manipuler** ces données en lien avec un environnement, ce qui signifie autant de pouvoir les stocker que de pouvoir les transformer
- **répéter** ces manipulations élémentaires, car les données sont d'une taille finie, mais non connue à l'avance
- **abstraire** les notions précédentes pour ne pas avoir à se répéter, à -ravers la notion de fonctions

I.2 Exemple

Prenons un exemple simple pour comprendre les différents étages où le but est final est de pouvoir calculer des sommes comme $1 + 2 + \cdots + n$.

Représenter Un entier comme 3 est une donnée qu'on pourra représenter telle quelle dans la plupart des langages avec cependant des limitations à garder en tête : la mémoire étant finie, on ne peut pas représenter des entiers quelconques, car il y en a une infinité. Déjà, à ce stade, on peut se poser la question de savoir si on ne veut représenter que des entiers dans un intervalle fixe, afin de borner leur occupation mémoire, ou si on veut les représenter en arbitraire. On peut opérer sur ces entiers avec un opérateur tel que + pour l'addition, mais le choix de la représentation aura son importance sur ce qui se cache derrière cette opération. L'addition de deux très grands entiers en précision arbitraire prendra plus de temps.

A l'aide de cela, on peut ainsi écrire $1 + 2 + 3 + 4 + 5$.

Manipuler On peut aussi les stocker directement en mémoire à l'aide de variable. Ainsi

```c
int x = 3;
```
C

```python
x = 3
```
Python

```ocaml
let x = ref 3
```
OCaml

permettra de définir une variable nommée x et contenant la valeur entière 3. On peut alors manipuler cette variable en changeant sa valeur à l'aide des opérations :

```c
x = x + 2;
```
C

```python
x = x + 2
```
Python

```ocaml
x := !x + 2
```
OCaml

Avec ces instructions de manipulation, on peut reprendre le calcul précédent de $1 + 2 + \cdots + 5$ avec une suite d'instructions :

```c
int s = 0;
s = s + 1*1;
s = s + 2*2;
s = s + 3*3;
s = s + 4*4;
s = s + 5*5;
```
C

```python
s = 0
s = s + 1*1
s = s + 2*2
s = s + 3*3
s = s + 4*4
s = s + 5*5
```
Python

```ocaml
let s = ref 0 in
s := !s + 1*1;
s := !s + 2*2;
s := !s + 3*3;
s := !s + 4*4;
s := !s + 5*5
```
OCaml

Répeter En fait, les 5 instructions d'ajout qu'on vient d'écrire ont la même structure : il s'agit d'ajouter une valeur i^2 à la variable `s` avec i prenant tour à tour les valeurs 1, 2, 3, 4 et 5. Une structure comme une boucle va permettre d'éviter cette répétition et d'écrire qu'il existe une variable `i` prenant ces valeurs et les instructions à réaliser pour chaque valeur de `i`. On pourra donc écrire :

```c
int s = 0;
for(int i = 1; i <= 5; i++)
{
  s = s + i*i;
}
```
C

```python
s = 0
for i in range(1,5+1):
  s = s + i*i
```
Python

```ocaml
let s = ref 0 in
for i = 1 to 5 do
  s := !s + i*i
done
```
OCaml

C'est en fait très proche de la notation $1 + 2 + \cdots + 5 = \sum_{i=1}^{5} i$. En fait, on cette notation mathématique est très proche d'une boucle.

Abstraire imaginons qu'on ait besoin d'effectuer le calcul précédent $\sum_{i=1}^{n} i^2$ à plusieurs reprises pour différentes valeurs de n, on pourrait à chaque fois recopier ces lignes en changeant la valeur maximale prise par i. Mais c'est inefficace pour plusieurs raisons :

- en recopiant du code, on risque de propager et répéter des erreurs. Si jamais on découvre une meilleure manière de faire le calcul, il faudra ainsi changer le code à chaque endroit où on l'a copié.
- là où on a besoin de ce calcul, il est possible que ce soit pour faire d'autres choses avec et donc qu'il s'inscrive dans une logique complexe. En copiant le code, on rend le programme plus compliqué à comprendre, car la boucle pour calculer cette somme est mise sur le même plan que le code qui nous intéresse.

Pour faire un parallèle avec les mathématiques, c'est comme si on recopiait une preuve dans un cas particulier chaque fois qu'on a besoin d'appliquer un théorème.

On introduit ainsi une notion d'abstraction *les fonctions* qui vont nous permettre de faire un code générique de calcul en prenant n en paramètre.

```c
int somme_carres(int n)
{
  int s = 0;
  for(int i = 1; i <= n; i++)
  {
    s = s + i*i;
  }
  return s;
}
```
C

```python
def somme_carres(n):
  s = 0
  for i in range(1,n+1):
    s = s + i*i
  return s
```
Python

```ocaml
let somme_carres n =
  let s = ref 0 in
  for i = 1 to n do
    s := !s + i*i
  done;
  !s
```
OCaml

Comme on le verra, une telle fonction est caractérisée par son nom qui nous permet d'y faire référence, comme lorsqu'on applique le théorème de Thalès, ainsi que des arguments qui seront, à l'exécution, remplacé par les valeurs qui nous intéressent, comme le fait que Thalès est démontré dans un triangle *générique* mais on l'applique sur un triangle particulier et une valeur de retour qui correspond à ce que la fonction calcule.

On peut alors, quand on a besoin de faire le calcul de $\sum_{i=1}^{5} i^2$ juste écrire :

```c
somme_carres(5)
```
C

```python
somme_carres(5)
```
Python

```ocaml
somme_carres 5
```
OCaml

I.3 Se remettre sans cesse à l'ouvrage

De notre présentation *étagée*, on pourrait retenir une hiérarchisation des différents étages. Il n'en est rien.

Ainsi, autant il peut être assez direct de représenter des données, comme dans le cas précédent, autant cela peut être une étape cruciale que de choisir la meilleure représentation. C'est le cas quand on s'intéresse au choix d'une structure de données la plus adaptée ou quand on conçoit un schéma de base de données.

La programmation, et plus largement l'informatique, sont ainsi à rapprocher d'un art martial comme l'Aïkido où on travaille tout au long de sa pratique les mêmes gestes simples en se perfectionnant sans cesse. Aller au bout de ce chapitre ne sera donc pas le signe qu'on maîtrise la programmation, mais juste qu'on a fait le premier pas sur la voie du perfectionnement.

Tout au long des autres chapitres, on verra ainsi des méthodes, des notions, qui permettent de mieux comprendre et de mieux pratiquer la programmation. Mais au moment où se retrouvera à programmer, on ne sortira pas de ces quatre étages.

II Représenter

Les données sont regroupées en informatique autour de la notion de types. Un type de données peut être vu en première approximation comme un ensemble de données de même nature. On considère, par exemple, usuellement le type des entiers naturels ou celui des nombres à virgule flottante.

II.1 Les données simples

Les données les plus simples sont celles qui correspondent à des valeurs numériques. Elles dépendent assez souvent des langages de programmation, mais on retrouve toujours un type pour les booléens, un type pour les entiers et un type pour les nombres à virgule flottante, c'est-à-dire pour des représentations de certains nombres réels.

```ocaml
```
OCaml

II.2 Les données composées ou structurées

Les données simples permettent de tout représenter. En effet, l'élément de donnée le plus primitif est le bit qui correspond à un booléen. Ainsi, la mémoire d'un ordinateur est entièrement constituée de booléens. Mais en disant cela, on ne dit pas grand-chose, car il ne s'agit pas d'une soupe informe de booléens, mais d'une organisation structurée. C'est ainsi qu'on considère des données composées comme les tableaux ou les couples.

On distingue deux types de données composées :

- les données *immuables*, c'est-à-dire celles qui ne permettent pas de changer les valeurs qu'elles regroupent ni leur structure après leur création
- les données *mutables* qui le permettent.

```ocaml
```
OCaml

III Manipuler

III.1 Variables

L'élément clé permettant de manipuler des valeurs est de pouvoir les placer à un endroit et de changer la valeur qui s'y trouve. C'est ce qu'on appelle une **variable**. Même si cela ne correspond pas à une réalité selon les langages, il est d'usage de considérer une variable comme une case mémoire disposant d'un nom et dans laquelle on place des valeurs.

On a alors trois éléments :

- définir une variable, souvent en la limitant à un certain type de donnée, on parle de **déclaration**
- accéder à sa valeur
- changer sa valeur, on parle d'**assignation**

Les noms de variables sont le plus souvent composés de lettres, de chiffres et du symbole `_`, ils doivent commencer par une lettre.

```
OCaml
```

III.2 État d'exécution

L'ensemble des variables et des valeurs auxquelles elles sont associées à un moment de l'exécution d'un programme est un état d'exécution. Les instructions de déclaration et d'assignation modifient ainsi l'état.

Suite aux instructions

```c
int a = 3;
int b = 2;
```

```python
a = 3
b = 2
```

```
let a = ref 3
let b = ref 2
```

l'état est alors

variable	valeur
a	3
b	2

Si on exécute ensuite l'instruction

```c
b = a;
```

```python
b = a
```

```
b := !a
```

il devient

a	3
b	3

Dans la suite, on pourra noter `a=3, b=3` un tel état. Ici, le = permet une notation intuitive.

III.3 Instructions et blocs

Les programmes qu'on va écrire vont maintenant être des suites d'instructions, comme

```c
int x = 3;
int y = x + 2;
x = y;
```

```python
x = 3
y = x + 2
x = y
```
Python

```ocaml
let x = ref 3 in
let y = ref (!x + 3) in
x := !y;
y := !x + 1
```
OCaml

Ces instructions sont regroupées ensemble sous une notion de blocs. Suivant les langages, ces blocs sont plus ou moins explicites.

En `OCaml`, il n'y a pas à proprement parler de notion de blocs car tout est une expression. Cependant, l'opérateur `;` permet d'évaluer successivement deux expressions. On pourra donc écrire

```ocaml
x := 3;
y := !x + 2;
x := !y
```

pour exécuter successivement ces instructions. On remarque que, contrairement à certains langages, `;` n'est pas un terminateur mais un séparateur : il est inutile de terminer par `;`.
Comme la valeur des expressions à gauche de `;` est perdue, on suppose qu'on ne le fait que pour des expressions dont la valeur est `() : unit`, c'est-à-dire pour les instructions. Si jamais on utilise `;` avec une expression ayant une autre valeur à gauche, un avertissement nous indique que l'usage est problématique :

```ocaml
# 3 ; 2;;
Warning 10 [non-unit-statement]: this expression should have type unit.
- : int = 2
```

Les blocs ont souvent besoin d'être séparés du reste du programme comme on le verra dans la suite. Pour cela, comme il s'agit de simples expressions, on peut utiliser les parenthèses comme ici :

```ocaml
(x := 3;
y := !x + 2;
x := !y)
```

mais il est souvent plus clair d'utiliser la paire `begin/end` qui, bien qu'elle soit strictement équivalente à une paire de parenthèses, permet de faire apparaître le fait qu'on a un bloc :

```ocaml
begin
  x := 3;
  y := !x + 2;
  x := !y
end
```
OCaml

Derrière cette notion de bloc on trouve une notion implicite qui est celle de flot d'exécution et qui régit la manière dont les instructions sont exécutées. En accord avec la manière dont on écrit les programmes, les instructions sont exécutées de haut en bas. L'état d'exécution est alors modifié le long du flot.

Une analogie qui sera utile dans la suite est de voir l'état d'un programme comme un train qui circule à travers les rails du flot de contrôle, en rajoutant ou modifiant des wagons à chaque instruction présentée comme un arrêt. Cette image assez naïve permet de faire passer une idée très importante qui est celle de l'ordre d'évaluation et du contexte associé.

OCaml

Un exemple important est celui de l'échange du contenu de deux variables `a` et `b`. On ne peut pas écrire

```c
b = a;
a = b;
```
C

```python
b = a
a = b
```
Python

```ocaml
b := !a;
a := !b
```
OCaml

car comme on vient de le voir, la valeur de `b` est déjà perdue après la première assignation. Une solution consiste

alors à introduire une variable temporaire qui ne va servir que de support pour pouvoir garder la valeur de `b`. On écrira ainsi :

```c
// ici, on ne peut pas le faire de manière
// générique, cela dépend du type de a et b.
int temp = b;
b = a;
a = temp;
```
C

```python
temp = b
b = a
a = temp
```
Python

```ocaml
let temp = !b in
b := !a;
a := temp (* pas besoin que temp soit une r
```
OCaml

III.4 Portée

Une variable a une durée de vie, souvent, elle n'existe que localement dans un programme. Par exemple, uniquement au sein de la fonction ou du bloc dans laquelle on l'a définie. On parle de portée d'une variable pour désigner l'ensemble des instructions dans lesquelles ont peut y accéder.

```ocaml
```
OCaml

III.5 Manipuler des données composées mutables

Les types de données mutables permettent de modifier leur valeur. Ce sont le plus souvent des **collections**, c'est-à-dire des arrangements structurés de valeurs, comme un tableau qui les arrange en ligne de la première à la dernière. Lorsqu'on peut faire un accès direct à une valeur, on peut alors la modifier comme si c'était une variable. Il peut également être possible de faire des modifications sur la structure de la collection elle-même, comme enlever ou rajouter des éléments.

III.5.i Modification des valeurs des éléments

```ocaml
```
OCaml

III.5.ii Partages de valeurs composées entre plusieurs variables

Il y a une différence fondamentale entre écrire

```python
a = 3
b = a
a = 2
```
Python

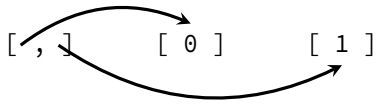
et écrire

```python
a = [ 3 ]
b = a
a[0] = 2
```
Python

Dans le premier cas, lors de l'assignation `b = a`, il n'y a pas de lien entre la valeur `3` dans `a` et celle dans `b`. Dans le second cas, après `b = a`, on a `b` et `a` qui pointent vers le même tableau. On notera `b, a = [ 3 ]` cet état. Ainsi, quand on exécute l'instruction `a[0] = 2` on modifie le tableau qui est également associé à `b`. Donc `a` **et** `b` sont associées au tableau `[ 2 ]`.

En fait, quand on écrit `a = [ 3 ]`, on ne **stocke** pas la valeur du tableau dans `a`. On crée le tableau `[ 3 ]` en mémoire et on place dans `a` une référence vers ce tableau. Quand on exécute alors `b = a`, on place dans `b` une autre référence mais vers le même tableau. Une manière simple de voir cela est de représenter les références par des flèches, ainsi l'état pourrait en fait se représenter ainsi :

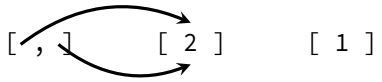
Si on considère un tableau de tableau comme `[ [ 0 ], [ 1 ] ]` on aura alors dans les cases du tableau principal des références vers les deux tableaux `[ 0 ]` et `[ 1 ]` :



Ainsi, si on écrit

```
t = [ [ 0 ], [ 1 ] ]
t[1] = t[0]
t[1][0] = 2
```
Python

on aura alors les deux références qui pointent vers le même tableau, et quand on modifie `t[1][0]`, on modifie donc forcément `t[0][0]` également pour aboutir à l'état :



On peut remarque que plus rien ne fait référence ici au tableau `[ 1 ]`. Il y a un mécanisme dans `Python` qui détecte cela et qui supprime de la mémoire le tableau.

Ce qui vient d'être dit pour les tableaux en Python est encore vrai avec le dictionnaires, et plus généralement avec la plupart des données.

III.5.iii Modification de la structure des données composées

Pour parler de modification de tableaux ou de dictionnaires, il est nécessaire de comprendre que `Python` est un langage objet. On se contentera ici de dire qu'un objet est une donnée munie d'opérations de manipulation sur cette donnée.

Ainsi, si `t` est un tableau, on pourra écrire `t.append(x)` pour ajouter la valeur `x` comme une nouvelle case à la fin du tableau. `t.append(x)` est une instruction, elle ne renvoie pas un nouveau tableau avec cette modification mais elle modifie directement `t`. On dit que `.append` est une **méthode** de la **classe** `list` de `t`. Il est tout à fait possible d'utiliser `Python` sans écrire de classes, mais dans la mesure où c'est un langage objet dans son cœur, on manipulera forcément des objets.

Par exemple, la suite d'instructions :

```
t = []
t.append(2)
t.append(3)
```
Python

va faire passer l'état de `t = []` à `t = [ 2 ]` puis `t = [ 2, 3 ]`.

Il existe beaucoup de méthodes pour les types `list` et `dict`. On les verra selon les besoins dans la suite. Il est toutefois possible, et souhaitable, de se référer à la documentation pour avoir une description détaillée de celles-ci.

III.6 Instruction conditionnelle

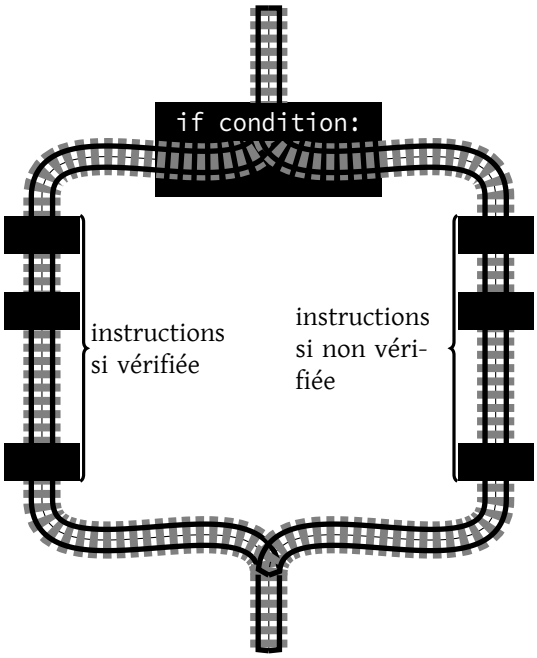
L'instruction conditionnelle va permettre d'enrichir le flot de contrôle avec des branchements. Elle permet d'orienter le flot d'exécution dans un bloc ou dans un autre selon qu'une condition soit vérifiée ou non.

On écrit

En `OCaml` il y a des expressions conditionnelles qui ont déjà été vues mais qui maintenant prennent une autre signification avec des expressions contenant des instructions. Dans le contexte de la programmation impérative, on pourra utiliser `begin/end` pour accentuer les différences :

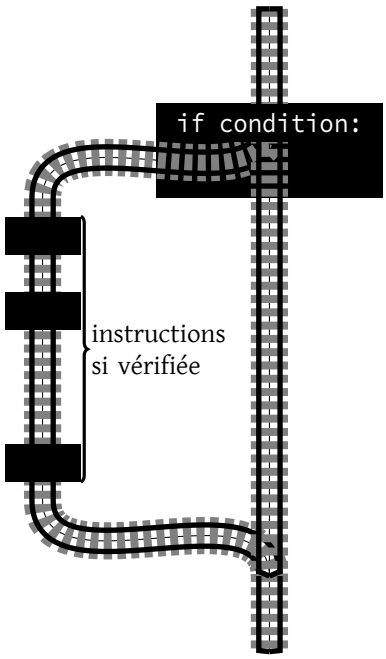
```OCaml
if condition
then
 begin
  (* bloc si la condition
  est vérifiée *)
 end
else
 begin
  (* bloc si la condition
  n'est pas vérifiée *)
 end
```

Si on reprend l'analogie des trains et des rails, une instruction conditionnelle est un échangeur qui, selon que le train vérifie ou pas la condition va le faire circuler dans une branche ou une autre. Il est important de comprendre qu'une fois une branche exécutée, les flots se rejoignent sur l'instruction qui suit l'instruction conditionnelle.



Il arrive qu'on n'ait rien à faire dans une branche, on peut alors placer un bloc vide, mais, en général, on préfère, quitte à nier la condition, ne pas écrire de `else` comme dans :

```Python
if condition:
  # bloc si la
  # condition est vérifiée
# le flot saute directement ici si
# la condition n'est pas vérifiée
```



Remarque En `Python`, un bloc vide se note `pass` comme dans le programme suivant :

```python
if x == 3:
  y = y + 1 # incrémente y si x vaut 3
else:
  pass # ne fait rien sinon
```

qu'on aurait pu écrire :

```python
if x == 3:
  y = y + 1 # incrémente y si x vaut 3
```

On pourrait être tenter d'écrire l'instruction a priori inoffensive `y = y` dans le bloc du `else`, mais une telle assignation n'est jamais gratuite. En effet, pour certaines données, il peut se produire une duplication coûteuse pour la réaliser et c'est une bonne pratique de ne pas écrire des opérations qui, si pour nous elles ne font rien comme `y = y + 0`, peuvent en fait avoir un coût caché.

Une principe clé de la programmation est la compositionnalité des instructions de gestion de flot : on peut placer des instructions conditionnelles dans le corps d'une des branches. On pourra donc écrire

```python
if condition1:
  if condition2:
    # bloc si condition 1 et condition 2 sont vérifiées
  else:
    # bloc si condition 1 est vérifiée mais condition 2 ne l'est pas
elese:
  if condition2:
    # bloc si condition 1 n'est oas vérifiée et condition 2 l'est
  else:
    # bloc si condition 1 et condition 2 ne le sont pas
```

III.6.i Conditions et opérations sur les booléens

Le rôle des conditions est central dans l'usage des instructions conditionnelles et ainsi c'est très important de bien comprendre le fonctionnement des booléens et de leurs opérations.

Une condition est une formule logique constituée

- de formules atomiques portant sur l'état comme `x == 3`, `x < 2`, ...
- d'opérateurs booléens pour relier ces formules : `not`, `and` ou `or`

III.6.ii Inversion de point de vue par rapport aux mathématiques

En mathématiques, il est courant d'avoir des définitions de valeurs par cas. Par exemple, on pourrait écrire

$$x = \begin{cases} \frac{y}{2} & \text{si } y \text{ pair} \\ \frac{y-1}{2} & \text{sinon.} \end{cases}$$

On pourrait le traduire alors aisément ainsi :

```python
if y % 2 == 0:
  x = y / 2
else:
  x = (y-1) / 2
```

Mais en faisant cela, on a inversé le point de vue en plaçant la condition avant l'affection de `x`.

III.6.iii Instructions conditionnelles en cascade

III.7 Entrées et sorties

IV Répéter

Jusqu'ici, on a vu des instructions permettant de manipuler l'état d'exécution, éventuellement de manière différente selon sa valeur à l'aide d'instructions conditionnelles. Mais étant donné un état de départ, on connait déjà les instructions qui seront exécutées et surtout **on connait leur nombre** qui est majoré par le nombre total d'instructions. En effet, pour reprendre l'analogie des trains, ceux-ci ne font que descendre le long du flot et le nombre d'arrêts rencontrés est majoré par le nombre d'arrêts total.

Cependant, la force principale de l'informatique vient du fait qu'on peut, à l'aide d'un nombre d'instructions fini, avoir la potentialité d'exécuter un nombre arbitrairement grand d'instructions. A cette fin, nous allons introduire un élément fondamental : la notion de répétition ou de boucles.

On pourrait être étonné de la formulation précédente sur ce nombre arbitrairement grand car la mémoire, et même le temps d'exécution en pratique, étant fini, tout est borné. Si on considère un programme qui va calculer la somme des éléments d'un tableau et qu'on sait que la longueur d'un tel tableau est majorée, on pourrait imaginer une énorme boucle conditionnelle

IV.1 Répéter n fois

IV.2 Répéter pour chaque élément

IV.2.i Accumulateur

IV.2.ii Drapeau

IV.3 Répéter tant qu'une condition est vérifiée

IV.4 Choisir la structure de boucle adaptée

IV.5 Boucles imbriquées et dimensionnalité

V Abstraire

V.1 Définir et appeler des fonctions

V.2 Action d'une fonction et valeur de retour

V.3 Fonctions et portée

V.4 Structurer des programmes avec des fonctions

V.5 Les fonctions comme valeurs