# Introduction au Pseudo-langage pour IF121

Pour pouvoir décrire des programmes sans être trop gênés par des détails syntaxiques (oubli de points virgule, syntaxe horrible comme le != utilisé en C++ pour dire "différent", ou la syntaxe C++ des boucles for), on introduit un "pseudo-langage". Il est suffisamment formel pour pouvoir être pris comme description précise du programme que l'on souhaite ensuite écrire en C++, et c'est pour cela que c'est un "langage". Mais il n'est pas vraiment exécuté par une machine, et on se permettra des libertés telles qu'il est bien approprié de lui ajouter l'étiquette de "pseudo".

Notamment, ce pseudo-langage nous servira pour pouvoir décrire des programmes au niveau d'abstraction souhaité.

Par exemple, au debout un passera du temps pour découvrir comment lire les éléments d'un tableau sur l'entrée, en produisant un fragment de programme en pseudo-langage qui aura cette allure

```
pour i<--1 a 10 faire
écrire "Entrez l'élément" i " du tableau"
        lire t[i]
fin pour
```

alors que plus avant, en écrivant des programmes plus complexes, ou notre intérêt est focalise sur des aspects de plus haut niveau, on pourra tout simplement se permettre d'écrire

```
lire les 10 éléments du tableau t
```

même si cela n'est pas une "instruction"

## Définition du Pseudo-langage

Attention: dans ce qui suit, ce qui est écrit en *italique* est un (pseudo-)mot-cle (qui pourra être souligne plutôt), alors que les mots ou phrases entre crochets, comme <nom> <variable> <liste d'instructions> etc. indiquent qu'a leur place on doit insérer l'objet décrit entre parenthèse (un nom, une variable, une liste d'instructions).

Un (pseudo-)programme est constitue, dans l'ordre, de:

- une série de déclarations de constantes
- une série de déclarations de types
- une série de déclarations de fonctions ou procédures
- la déclaration du corps principal du programme

Plus précisément, un (pseudo-)programme commence de la manière suivante:

```
constantes <nom> = <valeur>
           <nom> = <valeur>
...
types <nom> = <définition de type>
      <nom> = <définition de type>
...
```

Où les définition des types font intervenir soit un des types de base predéfinis, soit des types complexes comme les tableaux, les structures et les types énumérés introduit plus tard dans ce document. Les types de base disponibles sont

```
entier
caractère
booléen
réel
chaîne de caractères
```

avec les opérations vues en cours

**entiers** : + - * / % N.B.: division entière, et reste de la division entière, donc 7/3 fait 2 et non pas 2.33333....

**réels** : + - * /

**caractères** : aucun

**chaînes de caractères** : + (concatenation)

**booléens** :

- opérations logiques: *et ou non*
- opérateurs de relation: `> < >= <= =/= =`

On n'utilisera jamais de types imbriques, donc <nom de type> est un nom et pas par exemple un *tableau de ...* Idem dans la suite.

Suivent ensuite les déclarations de fonctions, de procédures et de programme:

```
fonction <nom de fonction>(<liste de paramètres>):<nom du type>
         <bloc>

procédure <nom de procédure>(<liste de paramètres>)
         <bloc>

programme <nom de programme>
        <bloc>
```

La <liste de paramètres> a la forme suivante :

```
valeur <nom> : <nom de type>
 <nom> : <nom de type>
 ...

référence <nom> : <nom de type>
    <nom> : <nom de type>

    ...
```

Un <bloc> a la forme suivante :

```
variables <nom> : <nom de type>

début <nom de la fonction ou de la procédure ou du programme>

<liste d'instructions>

fin <nom de la fonction ou de la procédure ou du programme>
```

Une <liste d'instructions> est une séquence d'instructions, qui seront exécutés dans l'ordre.
Les instructions que l'on retrouve peuvent être classes comme suit:

- instructions simples
- structures de contrôle
- appel de fonctions et procédures

## Les instructions simples sont:

**des instructions d'entrée et sortie**

sortie

```
écrire <nom de variable>
écrire <expression>
écrire <chaî ne de caractère>
```

on peut écrire plusieurs valeurs sur une même ligne, par exemple

```
écrire "La variable i vaut" i "et j vaut" j
```

entrée

```
lire <nom de variable>
```

on peut aussi se permettre des lectures multiples

```
lire i,j,k
```

**des affectations** , notées comme suit:

```
<nom> <-- <expression>
```

( `<--` est une flèche vers la gauche...). Notez qu'on se permettra d'écrire

```
incrementer i
```

pour

```
i <-- i+1
```

et

```
decrementer i
```

pour

```
i <-- i-1
```

## Les structures de contrôle dont on disposera seront les suivantes, où *condition* est une expression booléenne:

**Expressions conditionnelles :**

```
si <condition> alors <liste d'instructions>
 sinon <liste d'instructions>
fin si

si <condition> alors <liste d'instructions>fin si
```

**Analyse par cas :**

```
selon
      <condition> : <liste d'instructions>
      ...
fin selon
```

**Boucles:**

```
pour <nom> <-- <expression> a <expression> faire
     <liste d'instructions>
fin pour
```

La variable sur laquelle s'effectue la boucle ne doit pas être modifiée à l'intérieur de la boucle.

```
tant que <condition> faire
     <liste d'instructions>
fin tant que
```

```
faire
     <liste d'instructions>
tant que <condition>
```

```
répéter
     <liste d'instructions>
jusqu'à <condition>
```

## Appel de fonctions et procédures:

Si `nom` est un nom de fonction ou procédure, qui a été déclaré avec une <liste de paramètres> de longueur $n$, alors

```
nom(v1,...,vn)
```

est un appel de la fonction ou procédure, qui sera exécutée avec les $n$ valeurs donnés entre parenthèses "substitués à la place des" paramètres présents dans le corps de la fonction ou procédure. Le mécanisme précis de substitution sera exposé en cours.

## Tableaux, structures et commentaires

Les *tableaux* sont introduits par la définition de type:

```
<nom> = tableau de <valeur> <nom de type>
```

(exemple `type t = tableau de 5 entiers`). On utilise la syntaxe *t[i]* pour indicer la i-ème valeur du tableau t.

Les *structures* sont des enregistrement avec plusieurs champs introduites par des déclaration de type de la forme:

```
<nom> = structure
           <nom du champ> : <nom du type>
           <nom du champ> : <nom du type>
                    ...
        fin structure
```

On utilise la syntaxe *a.b* pour designer le champ b de la variable a

Les types *énumérés* sont des types pouvant assumer un nombre fini de valeurs, introduits par des déclaration de type de la forme:

```
<nom> = énuméré <valeur1>, <valeur2>, ...<valeurn>
```

(exemple `couleurs = enumere Bleu, Blanc, Vert`).
Pour faire des commentaires, on peut utiliser le mot clef *commentaire*, ou par abus de notation, les notations C++

```
/* ceci est un commentaire */
// ceci l'est aussi
```

### REMARQUE

Globalement, pour écrire les algorithmes en pseudo-langage, on ne focalise pas sur la syntaxe, notamment on n'utilise pas de point-virgules, et on ne décomptera pas des points si quelques un écrit

```
imprimer i
```

plutôt que

```
écrire i
```

dans son programme. Ce qu'il faut est que l'on puisse clairement et sans ambiguïté comprendre ce que le (pseudo-)programme fait.