Annexe A

Le LDA

Dans cette annexe nous définissons le LDA (*le Langage de Description d'Algorithmes*) que nous allons utiliser. Nous ne nous attarderons pas sur les concepts ni sur certaines bonnes pratiques ; tout cela est vu dans les chapitres associés.

Nous utilisons un pseudo-code pour nous libérer des contraintes des langages de programmation.

— Un programme est une suite de lignes ne permettant pas d'utiliser pleinement les deux dimensions de la page (pensons à la mise en page des formules).

— Certaines constructions et règles n'existent que pour simplifier le travail du compilateur et/ou accélérer le code. C'est le cas, par exemple, de la syntaxe du *switch* ou encore des opérateurs de comparaisons qui sont uniquement binaires.

Dans vos réflexions, brouillons, premiers jets, nous vous encourageons à utiliser des notations qui vous sont propres et qui vous permettent de poser votre réflexion sur un papier et d'avancer vers une solution.

La version finale, toutefois, doit être lue par d'autres personnes. Il est **essentiel** qu'il n'y ait aucune ambigüité sur le sens de votre écrit. C'est pourquoi, nous devons définir une notation à la fois souple et précise.

Cette notation doit aussi être adaptée à des étudiants de première année. Ce qui nous amène à ne pas introduire des nuances qui leur échappent encore et, parfois, à imposer des contraintes qui seront relâchées plus tard mais qui permettent de cadrer l'apprentissage d'un débutant.

Remarque : Ce guide n'est pas universel. En dehors de l'école, d'autres notations sont utilisées, parfois proches, parfois plus lointaines. Votre professeur pourra également introduire quelques notations qui ne sont pas reprises ici. Lorsque vous changerez de professeur, soyez conscient que ces ajouts ne seront peut-être plus valables.

L'important est que le groupe qui doit communiquer au moyen d'algorithmes se soit préalablement mis d'accord sur des notations.

Note : Cette version ne concerne que les notions vues en DEV_1.

A.1 Les variables

```
var1, ... : Type
```

Uniquement en début de module. Où Type est à choisir parmi : entier, réel, booléen, chaine, structure et tableau.

Note : Plus de type *caractère*. Ca complique les choses sans rien apporter de fondamental et c'est très peu utilisé. À laisser pour les langages. Exit aussi les *énumérations* pour les mêmes raisons.

Note : Alternative 1 : Fusionner entier et réel en un seul type (nombre). L'avantage est que ça simplifie encore ; l'inconvénient est qu'il y a une perte d'expressivité pour le lecteur. Savoir qu'une variable ne va contenir que des valeurs entières aide à mieux comprendre son rôle et le reste de l'algorithme.

Note : Alternative 2 : Utiliser les notations mathématiques universelles pour les types numériques : $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{N}_0$... C'est concis et précis et ça permet de spécifier facilement qu'un paramètre doit être positif par exemple. L'inconvénient est que certains étudiants ne sont pas familiers de ces notations.

A.2 Les constantes

```
Constante nom = valeur
```

En début de module ou en dehors des modules si la constante est partagée par plusieurs modules.

A.3 Les expressions numériques

Comme le nom d'une variable est généralement composé de plusieurs lettres, nous écrirons explicitement la multiplication (via un $*$) afin d'éviter toute ambigüité.

Introduisons :

— DIV et MOD pour les entiers uniquement.
— hasard(n) pour obtenir un nombre aléatoire entre 1 et n.

Pour écrire les expressions, nous admettons toute notation mathématique puisqu'elle est concise et précise. L'écriture pourra être en deux dimensions.

Exemples :

$$\text{delta} \leftarrow \text{b}^2 - 4 * \text{a} * \text{c}, \qquad \text{racine} \leftarrow \frac{-\text{b} + \sqrt{\text{delta}}}{2 * \text{a}}$$

Ainsi, on pourra écrire :

— $\sin(x)$, $\cos(x)$ et toutes les fonctions trigonométriques.
— $|x|$ et $x!$ pour la valeur absolue et la factorielle.
— $\lfloor x \rfloor$ et $\lceil x \rceil$ pour la valeur plancher et la valeur plafond.

Bien sûr, ces notations ne seront pas admises en début d'année si le but de l'exercice est précisément de déterminer l'algorithme qui se cache derrière.

Note : Il est trop tôt en première pour leur permettre d'utiliser les opérateurs de somme et de produit (comme $\sum_{i=0}^{n} i$) mais on pourra les y encourager par la suite.

Il est possible d'assigner une valeur entière à une variable réelle et vice-versa. Un réel est *arrondi* avant d'être placé dans une variable entière.

A.4 Les expressions booléennes

Les valeurs possibles sont vrai et faux.

Les comparaisons ($<$, $\leq$, $>$ et $\geq$) peuvent être utilisées pour les types numériques uniquement. Pour augmenter leur expressivité, on considère qu'ils existent aussi en version *n-aire* comme en Math. On peut donc écrire : $1 \leq \text{nb} \leq 100$.

L'(in)égalité ($=$ et $\neq$) peut être testée pour tous les types.

Les opérateurs booléens s'écrivent NON, OU et ET. Les deux derniers sont *court-circuités*.

A.5 Les expressions avec des chaines

D'abord les fondamentaux qu'on ne pourraient écrire avec ce qu'on a.

Note : On peut considérer que le 1er caractère est en position 1 ou 0, à décider (cf. partie tableau)

```
taille(ch : chaine)  → entier                                  // donne la taille de la chaine.
sousChaine(ch : chaine, pos : entier, long : entier)  → chaine      // extrait une sous-chaine
ch[i]                                                  // raccourci pour sousChaine(ch,i,1)
concat(ch1, ch2, ..., chN : chaine)  → chaine                  // concatène des chaines
ch1 + ch2 + ...+ chN                                                          // idem
```

Les modules qui suivent sont donnés par facilité mais pourraient être écrits.

```
// Est-ce ?
estLettre(ch : chaine)  → booléen                          // ne contient que des lettres ?
estChiffre(ch : chaine)  → booléen                         // ne contient que des chiffres ?
estMajuscule(ch : chaine)  → booléen                       // ne contient que des majuscules ?
estMinuscule(ch : chaine)  → booléen                       // ne contient que des minuscules ?

// Conversions
majuscule(ch : chaine)  → chaine       // minuscules vers majuscules sans toucher au reste.
minuscule(ch : chaine)  → chaine       // majuscules vers minuscules sans toucher au reste.
numLettre(lettre : chaine)  → entier              // la position de la lettre dans l'alphabet.
lettreMaj(pos : entier)  → chaine       // la majuscule de position donnée dans l'alphabet.
lettreMin(pos : entier)  → chaine       // la minuscule de position donnée dans l'alphabet.
chaine(n : entier)  → chaine                          // convertit un entier en une chaine.
chaine(x : réel)  → chaine                            // convertit un réel en une chaine.
nombre(nb : chaine)  → réel                          // convertit une chaine en un nombre.

// Manipulations
estDansChaine(ch : chaine, sous-chaine : chaine)  → entier
            // dit où commence une sous-chaine dans une chaine donnée (-1 si pas trouvé)

// Dans ce ce qui précède :
// - lettre : doit être une chaine de taille 1 contenant une lettre. erreur sinon.
// - pos : erreur si pas entre 1 et 26.
// - nb : erreur si la chaine ne contient pas un nombre et rien que ça.
```

A.6 Les instructions de base

```
var ← expression var1, var2...
afficher expression1, expression2...
erreur "raison"    // Provoque l'arrêt de l'algorithme.
```

Note : Il y a quelques années, à la demande des professeurs de BD, le "écrire" est devenu "afficher" pour éviter des confusions dans l'esprit des étudiants avec les opérations sur les fichiers. Dans le même sens, il me semble que le "lire" devrait devenir un "demander". La modification dans le syllabus ne demanderait que quelques minutes mais dans nos têtes ça prendrait plus de temps. Est-ce que ça en vaut la peine ?

A.7 Les instructions de choix

Note : Chaque année, il y a une discussion à propos du "selon-que" avec conditons : est-ce que les conditions sont mutuellement exclusives ou pas ? D'un point de vue formel, ce serait mieux mais il faut bien constater que, en pratique, ça complique pas mal les conditions dans certains cas. De plus, ça demande un gros effort d'adaptation pour la plupart des langages. Je propose de l'écrire comme des sinon-si pour éviter toute ambigüité.

```
si condition alors
    Instructions
fin si
```

```
si condition alors
    Instructions
sinon
    Instructions
fin si
```

```
si condition alors
    Instructions
sinon si condition alors
    Instructions
sinon si condition alors
    ...
sinon
    Instructions
fin si
```

```
selon que expression vaut
 liste_1 de valeurs séparées par des virgules :
    Instructions
 liste_2 de valeurs séparées par des virgules :
    Instructions
    ...
 liste_k de valeurs séparées par des virgules :
    Instructions
 autres :
    Instructions
fin selon que
```

où l'expression peut être de type entier ou chaine (pas de réel) et les valeurs sont des constantes.

A.8 Les instructions de répétition

```
tant que condition faire
    Instructions
fin tant que
```

```
faire
    Instructions
jusqu'à ce que condition
```

```
pour indice de début à fin [par pas] faire
    Instructions
fin pour
```

Note : Transformer le "répéter...jusqu'à ce que" en "répéter...tant que" ? En Java, le "do-while" me gène car je crois systématiquement qu'un "while" commence quand je vois la fin du "do-while". Y-aurait-il le même problème ici ?

La boucle pour ne peut être utilisée que pour des entiers.

Il n'est **pas nécessaire** de déclarer l'indice. Il ne peut être utilisé en dehors de la boucle et ne peut pas être modifié à l'intérieur de la boucle. De même, le début, la fin et le pas ne peuvent pas être modifiés dans la boucle.

A.9 Les modules

```
module nom(paramètres)
    Instructions
fin module
```

```
module nom(paramètres) → Type
    Instructions
    retourner expression
fin module
```

Note : Je propose ici une définition purement sémantique du passage de paramètre, indépendamment de toute considération technique d'implémentation dans les langages.

Pour les paramètres, on utilise des flèches pour indiquer l'**utilisation** du paramètre. Il s'agit d'une information sémantique qui n'a aucun lien avec le fonctionnement dans les langages

- ↓ pour une donnée pure (paramètre en entrée). Le paramètre doit avoir une valeur au début du module car cette valeur va être utilisée. À l'appel, on peut donner une expression. Par facilité, on admet que le paramètre soit modifié dans le module mais sans incidence sur ce qui est donné à l'appel.
- ↑ pour un résultat pur (paramètre en sortie). Le paramètre donné à l'appel doit être une variable. Cette variable ne doit pas avoir une valeur au début mais le module va lui assigner une valeur.
- ↓↑ pour une combinaison des deux (paramètre en entrée/sortie). Le paramètre donné à l'appel doit être une variable initialisée. Sa valeur sera utilisée par le module et pourra être modifiée.

On peut omettre les flèches si tous les paramètres sont en entrée.

A.10 Les structures

La structure doit être définie une fois pour toute, *quelque part*.

```
structure NomDeLaStructure
    nomChamp1 : type1
    nomChamp2 : type2
    ...
    nomChampN : typeN
fin structure
```

Quelques manipulations.

```
var : NomDeLaStructure
var ← {champ1, champ2...}                          // Assigner tous les champs
var1 ← var2                          // Tous les champs de var2 sont assignés à var1
lire var                                                  // Lit tous les champs de var
afficher var                                        // Affiche tous les champs de var
var1 = var2                                    // Teste l'égalité de tous les champs
```

On écrit var.nomChamp pour accéder à un champ. Ce champ est manipulé comme on le ferait de n'importe quelle variable. On peut donc, l'utiliser, l'assigner, le lire, l'afficher, le passer en paramètre...

Pour les structures en paramètre, attention à l'utilisation des flèches. On met une flèche en sortie pour indiquer que des champs vont être modifiés.

On suppose les modules suivants :

```
module date() → Date                                    // Donne la date du jour
module moment() → Moment                              // Donne le moment actuel
```

A.11 Les tableaux

```
nomTableau : tableau [borneMin à borneMax] de TypeElément
```

où borneMin et borneMax doivent être connus au moment où la déclaration sera effective. C'est une expression pouvant faire intervenir des constantes et des paramètres de la méthode. Les éléments n'ont pas de valeur initiale.

Quelques manipulations.

```
tab1 ← tab2          // les éléments de tab1 sont copiés dans tab2 (de même taille).
lire tab                                          // lit tous les élements du tableau
afficher tab                                      // affiche tous les éléments du tableau
tab1 = tab2          // les 2 tableaux ont la même taille et des éléments identiques
```

On écrit tab[indice] pour accéder à un élément du tableau. Cet élément est manipulé comme on le ferait de n'importe quelle variable. On peut donc, l'utiliser, l'assigner, le lire, l'afficher, le passer en paramètre...

Note : Certains voudraient que les tableaux commencent à 0 pour faciliter la traduction vers les langages courants comme Java et C. Ce serait bien qu'ils puissent faire les deux. On pourrait envisager de commencer à 0 pour les exercices qui vont être traduits en Java.

Il faudra peut-être écrire **tableau** [0 à N-1] de ... pour que N reste l'équivalent du tab.length.

A.11.1 Tableaux et paramètres

Quelques utilisations licites.

```
module brol( tab↓ : tableau [1 à N] d'entiers)
module brol() → tableau [1 à 10] d'entiers
module brol( N↓ : entier) → tableau [1 à N] d'entiers
module brol( tab↓ : tableau [1 à N] d'entiers) → tableau [1 à N] d'entiers
```

Attention à l'utilisation des flèches. Il faut se rappeler la sémantique.

- ↓ indique que les cases seront lues mais pas modifiées.
- ↑ indique que les cases n'ont pas à être initialisées. Certaines seront modifiées.
- ↓↑ indique que les cases seront lues et/ou modifiées.

Dans les 3 cas, un tableau doit être fourni à l'appel.

A.12 La généricité

Parfois, un algorithme peut être valide quel que soit le type des données (le swap en est un exemple évident). Dans ces cas, on peut écrire T en lieu et place du type.

Exemple :

```
module fill( tab↓ : tableau [1 à N] de T, val↓ : T )
    pour i de 1 à N faire
        tab[i] ← val
    fin pour
fin module
```