Chapitre 2 : Mécanismes de programmation et conventions de description.

Ce chapitre précise les mécanismes et conventions utilisés en cours et en travaux dirigés, et propose quelques rappels sur les types de programmation que vous aurez l'occasion de rencontrer par la suite.

1. Conventions d'écriture pour les algorithmes

Commentaires

Tout algorithme commencera par une spécification informelle comportant :

- Entrées
- Sorties
- Ce qu'il doit faire (sans entrer dans les détails)

Format : {commentaires, spécifications} ou /* commentaires, spécifications */ ou //

Variables, instructions et blocs

L'algorithme manipule des variables dénotées par des étiquettes et susceptibles de figurer dans les instructions classiques.

- **Affectation** : machin ← truc.
  La flèche dénote l'affectation (on met dans machin la valeur contenue par bidule). On bannira absolument la notation issue du c (machin=truc), qui sera réservée à sa fonction mathématique de comparaison de valeurs.
  On refusera également les opérateurs d'affectation étendus tels que +=, -=, *=,… qui ne permettent pas de distinguer opération et affectation.
- **Structures de contrôle** :
  - Si… alors… sinon…
  - Tant que… faire…
  - Répéter… jusqu'à…
  - Pour… allant de… jusqu'à… faire
- **Blocs :**
  on délimitera les blocs d'instruction graphiquement avec une barre verticale et/ou une indentation.
  On pourra utiliser des notations début machin, fin truc afin de lever les éventuelles ambiguïtés.
- **Si… sinon** : attention à l'imbrication des blocs de condition (si) qui peut conduire à des ambiguïtés. On pourra utiliser l'instruction « sinon ne rien faire » pour clarifier les choses.

Booléens

Le type de variable booléen ne contient que deux constantes : VRAI et FAUX.
Certains langages les confondent avec des nombres, ce ne sera pas le cas ici.
Seront acceptées les opérations connues sur les booléens d'un point de vue mathématique :
"4=5" est une expression booléenne en mathématique, dont la valeur est FAUX.

Si B est une variable booléenne et A une variable entière, on pourra écrire : B←A=5, ce qui indique que B reçoit le résultat de la comparaison de la valeur contenue dans A avec l'entier 5.
On pourrait écrire : si A=5 alors B←VRAI sinon B←FAUX

Evaluation des connecteurs logiques

Confrontés à une expression du type *si (A ET B) alors…* où A et B sont des expressions, certains compilateurs évaluent A et B quelle que soit la valeur de A. D'autres fonctionnent de façon "paresseuse" et n'évaluent B que si A est vraie (c'est vrai également pour OU).

En règle générale, on préfèrera écrire comme si l'évaluation n'était pas paresseuse (il faut donc prendre garde à ce que B puisse être évaluée quelles que soient les circonstances), mais il arrivera qu'on considère l'évaluation comme paresseuse.

Classes, méthodes, types abstraits, procédures et fonctions

En programmation impérative (Algol, Fortran, Pascal, C, Ada, …), un programme est défini par une structure de données et un algorithme.
En programmation objet (Simula, SmallTalk, Objective-C, ObjVlisp, Java, C++, …), les structures de données sont empaquetées avec des outils qui les manipulent : c'est une forme de programmation modulaire. Il s'agit d'un style de programmation qui n'interagit pas fondamentalement avec l'algorithmique sous-jacente.
Nous utiliserons alternativement les deux points de vue :

- des procédures et fonctions agissant sur des données décrites par des types abstraits (style impératif)
- des classes comportant champs et méthodes (style objet)

Lorsqu'on utilise la forme « objet », on considère que les données sont les champs des classes et les procédures et fonctions sont leurs méthodes : rien ne change pour les algorithmes associés !

Exemple : on représenterait les disques des Tours de Hanoï comme suit

Version objet :	Version impérative :

Disque = classe{
Champs
Diamètre : flottant
Epaisseur : entier
Matière : {bois, fer, verre}
Peint : Booléen
 	
Méthodes
Fonction Volume() : flottant
Procédure Peindre()
}
	Type Disque = enregistrement
	Diamètre : flottant
	Epaisseur : entier
	Matière : {bois, fer, verre}
	Peint : Booléen
	
	Fonction Volume (d: flottant, e: entier) : flottant
	Procédure Peindre (var d : disque)

Procédures

Une procédure est un bloc d'instruction exécuté à un moment donné, sur appel de l'algorithme principal. Elle a un statut d'instruction. On précise systématiquement le passage des paramètres utilisés en entrée et en sortie.
Les variables internes aux procédures ne seront pas toujours déclarées, mais seront toujours considérées comme locales.
Les variables globales seront proscrites autant que possible.

Exemple :
Procédure Ajoute (entrée, sortie S: Sac ; entrée X: Bille) {…}
/* S peut être modifiée en sortie, la procédure ajoute la bille X au sac S*/

Fonctions

Une fonction rassemble des instructions en prenant des paramètres en entrée et en manipulant des variables locales ; elle rend un résultat à la sortie, contrairement à la procédure.

Exemple :
Fonction compte (entrée T : Tas ; N : Entier ; y : Elément) : entier
/*Connaissant T de taille N, compte les occurrences de Y dans T. La sortie est le résultat rendu par la fonction*/
appel : si compte(T, N, y) = 5 alors…
ou bien total_de_y ← compte (T, N, y)

L'appel ne se fait pas comme une instruction mais en utilisant le résultat de la fonction dans une autre instruction. La fonction est du même type que le résultat qu'elle retourne.

2. Mode de passage des paramètres

Initialement, l'algorithmique distingue plusieurs modes de passage de paramètres, selon que les variables auxquelles ils sont identifiés peuvent être modifiées ou non.

L'avènement des langages objet a provoqué l'usage intensif de variables qui sont en réalité des pointeurs qui ne disent pas leur nom, ce qui peut rendre la distinction trompeuse.

Dans ce cours, nous avons fait le choix de bien préciser en en-tête les éléments qui doivent être modifiés ou non par l'algorithme : nous distinguerons donc deux types de passage de paramètre.

- **Passage par référence (par adresse, variable)**

Dans l'en-tête, le nom de la variable sera précédé du mot-clé var.
Exemple de syntaxe : Procédure ajoute (var A : Sac ; x : bille)
Ici, le paramètre A de la procédure ajoute est précédé de « var » : à l'exécution, il sera identifié à un nom de variable de même type, passée par référence (ou adresse).
La nouvelle valeur de A sera récupérable à la sortie dans la variable dont l'adresse a été passée.
La procédure va travailler physiquement à l'emplacement mémoire où se trouve la variable passée en paramètre

En clair : lors d'un appel ajoute(S, 5), la procédure va ajouter à l'adresse de la variable S la bille 5, formant ainsi un nouveau Sac à la place de S.

- **Passage par valeur (par copie)**

Exemple de syntaxe : Procédure ajoute (A : Sac ; x : bille)
Le paramètre A n'est pas précédé de « var » : seule une valeur (copie du paramètre) est donnée à la procédure au moment de l'appel. La zone mémoire où se trouve l'éventuelle variable passée ne sera pas modifiée.
Lors d'un appel ajoute(A, 5), la procédure va travailler sur une copie de A et ne pas la modifier elle-même : A est inchangé à la fin.

En résumé : si l'on veut modifier durablement une variable, on la passe par adresse (référence ou pointeur), sinon, on la passe par valeur (copie).

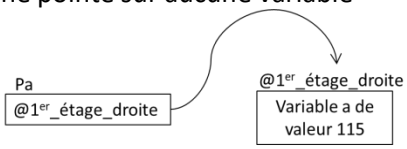
Attention, les choses ne sont pas aussi simples… certains langages rendent l'usage des pointeurs invisible (java, par exemple, qui du coup donne l'impression de passer des paramètres par valeur systématiquement), ou utilisent des types élaborés qui cachent des pointeurs (tableaux en C et C++, par exemple).

3. Pointeurs et adresses

En principe, les pointeurs ne font pas partie de l'algorithmique, mais ils seront utilisés pour l'implémentation de certaines structures.
Tous les langages les utilisent, mais de façon plus ou moins masquée selon les cas.

Dans ce cours, un pointeur est une variable dont la valeur est une adresse (ou Nil/Null)
Soit, pA, un pointeur.

- Si pA contient une adresse, à cette adresse se trouve la variable qu'il « pointe »
- Si pA contient Nil/Null, il ne pointe sur aucune variable



La variable pointée par pA se notera pA↑, ou parfois, par abus de langage, juste pA. Souvent, cette variable a des champs, que l'on manipulera.

Les déclarations de type pour les pointeurs varient d'un langage à l'autre.

Avantages des pointeurs :

- on peut faire de l'allocation dynamique de ressources (on ne réserve l'espace mémoire que pour sa durée d'usage, à l'exécution, par opposition à l'allocation statique), ce qui économise de la place
- à la déclaration d'un pointeur en statique, on ne réserve qu'une petite place mémoire pour l'adresse de l'élément pointé. En cours d'exécution, on allouera ensuite un espace plus grand destiné à contenir l'élément pointé (taille du type pointé).
- le pointeur étant une adresse, il est un type simple et la plupart des langages impératifs permettent de définir des fonctions retournant un type simple.

Les instructions, fonctions, procédures et opérations possibles sur les variables associées aux pointeurs sont toutes celles qui sont autorisées pour les types concernés.
Quant aux variables pointeurs elles-mêmes, on peut les tester, les affecter et les passer en paramètre dans les procédures et les fonctions.

Comparaison de deux pointeurs
Si A et B sont deux pointeurs de même type, on peut écrire: si A = B alors… sinon… . On ne compare que les adresses. Si A et B contiennent la même adresse, ils pointent sur la même chose, mais l'inverse est faux : il est possible que A et B (pointeurs) référencent la même valeur mais soient différents (s'ils pointent sur des zones mémoire distinctes contenant les mêmes valeurs).

Affectation entre pointeurs
On peut affecter des pointeurs : mettre B dans A. Si B vaut Nil, A devient égal à Nil, c'est-à-dire ne pointe plus sur rien (en particulier, ce sur quoi pointait A est "détaché", c'est-à-dire perdu). Si B pointait sur une zone, A pointe maintenant sur la même zone et donc toute modification de A modifie B et vice-versa.

Il sera autorisé de tester, affecter, passer en paramètre les pointeurs dans les procédures et les fonctions, mais toute autre opération sera interdite.

4. Inventaire à la Prévert

- **Programmation impérative**

La programmation impérative est un paradigme de programmation décrivant les opérations sous forme de séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.
C, C++, Python, PHP, Java ou Javascript sont des langages impératifs… ce sont des langages que vous apprendrez lors de votre cursus.

- **Programmation déclarative**

La programmation déclarative consiste à créer des applications sur la base de composants logiciels indépendants du contexte et ne comportant aucun état interne.
L'appel d'un de ces composants avec les mêmes arguments produit exactement le même résultat, quel que soit le moment et le contexte de l'appel.
C'est une forme de programmation qui se rapproche de la logique mathématique.
Exemple de langages : HTML, Lisp, Caml,…

- **Programmation fonctionnelle**

La programmation fonctionnelle est un sous-ensemble de la programmation déclarative.
Le terme de programmation fonctionnelle désigne une famille de langages de programmation ayant un certain nombre de traits en commun, dont un rôle central donné aux fonctions. Les représentants les plus connus de cette famille se nomment Haskell, StandardML et Ocaml, etc...
La programmation fonctionnelle est un paradigme de programmation de type déclaratif qui considère le calcul en tant qu'évaluation de fonctions mathématiques.

Les langages fonctionnels utilisent des types et des structures de données de haut niveau et permettent l'usage de fonctions d'ordre supérieur, c'est- à-dire une fonction acceptant des fonctions comme arguments ou pouvant renvoyer une fonction comme résultat.

Petite introduction : https://www.lri.fr/~filliatr/publis/ipf.pdf

- **Programmation Orientée Objet**

La programmation orientée objet consiste à définir et faire interagir des briques logicielles (objets). Un objet correspond à un concept ou une entité du monde physique, possède une structure interne et un comportement associé.
C'est la modélisation objet qui génère un programme orienté objet : il n'est pas suffisant d'utiliser un langage compatible avec ce genre de représentation.
Il existe actuellement deux grandes catégories de langages à objets :

- les langages à classes, que ceux-ci soient sous forme fonctionnelle (Common Lisp Object System), impérative (C++, Java) ou les deux (Python, OCaml)
- les langages à prototypes (JavaScript, Lua)

- **Programmation dynamique**

La programmation dynamique est une méthode algorithmique définie pour résoudre des problèmes d'optimisation. Elle consiste à résoudre un problème en le décomposant en sous-problèmes. On résout ensuite les sous-problèmes du plus petit au plus grand en stockant les résultats intermédiaires. Il s'agit donc d'un paradigme algorithmique qui peut être rendu indépendant du langage de programmation.
Exemples d'applications : problème du plus court chemin, problème du sac à dos, alignement de séquences, répartition de ressources.

- **Programmation parallèle**

La programmation parallèle consiste à réaliser des programmes adaptés à l'exécution sur des machines parallèles (plusieurs processeurs traitent les informations de façon simultanée, de façon à augmenter la vitesse de traitement).
Les programmes parallèles se définissent par opposition aux programmes séquentiels, exécutant les opérations l'une après l'autre. Le parallélisme va consister à juxtaposer des processeurs séquentiels ou à exécuter simultanément des instructions indépendantes.
Ce type de programmation nécessite une adaptation des algorithmes, de façon à dégager les séquences d'instructions pouvant être parallélisées (sous-entendu les instructions n'ayant pas besoin du résultat d'autres instructions pour se dérouler).

Cette liste est évidemment non exhaustive !