GUIDE DE L'ÉTUDIANT POUR LE PSEUDOCODE

Charles-Antoine Brunet
Daniel Dalle
Frédéric Mailhot
Domingo Palao Munõz
04 octobre 2021

Département de génie électrique et de génie informatique
Faculté de génie
Université de Sherbrooke

Version : A21-01 (2021-10-04)

Ce document est réalisé avec l'aide de LaTeX et de la classe `gegi-minidoc`.

Table des matières

1 Introduction

La première question qui survient souvent est : *pourquoi faire du pseudocode ?* Plusieurs réponses existent, mais essentiellement, elles tournent autour de deux concepts. Le premier est qu'il faut faire du pseudocode lors de l'étape de conception afin d'avoir les idées claires lorsque l'on passe à l'étape d'implémentation, par exemple pour écrire du code en langage C ou en langage C++. Le deuxième est qu'il faut faire du pseudocode afin de documenter, dans un document de conception, ce qui est à faire et comment le faire. Ceci est très utile lorsque l'implémentation est réalisée par plusieurs personnes ou encore par d'autres que la personne qui a écrit le pseudocode. Le document de conception sera aussi très utile plus tard, par exemple, pour la maintenance du logiciel.

Le pseudocode est écrit en langage courant, le français dans le cas présent, pour la facilité de compréhension, mais aussi pour séparer la conception de l'implémentation. Le pseudocode est indépendant du langage qui sera utilisé lors de l'implémentation. Comme un algorithme est une suite d'actions, des verbes actifs à l'infinitif sont utilisés dans le pseudocode.

Les deux principaux soucis avec le pseudocode sont la *cohérence* et la *clarté*. La cohérence demande que lorsqu'un formalisme est choisi, il faut le respecter du début à la fin pour l'ensemble du pseudocode écrit. La clarté demande que le pseudocode ne contienne pas de lignes ambigües qui peuvent être mal interprétées ou encore dont on ignore la signification ou le but.

Dans ce sens, la clarté fait aussi référence au niveau de détail du pseudocode. *Quel est le niveau de détail approprié ?* Cela dépend de la personne qui écrit le pseudocode et du lecteur visé, mais aussi du contexte dans lequel il est utilisé. Ce qui est sûr, c'est que pour le lecteur du pseudocode, il ne doit pas y avoir de lignes ambigües, comme mentionné précédemment, ou de *lignes magiques*, des lignes de pseudocode dont on ignore comment les traduire dans le langage de programmation visé.

Ce document propose une forme de pseudocode pour les programmes du *Département de génie électrique et de génie informatique de l'Université de Sherbrooke* en fonction de critères pratiques liés au contexte de ces programmes. Ce document ne décrit pas une norme universelle. Aussi, des choix particuliers sont posés et ils reflètent les préférences des auteurs et s'écartent de la référence principale [1].

2 Pseudocode

2.1 Commentaires

Un commentaire permet de rendre plus claires certaines parties du pseudocode. Les commentaires sont une source d'information supplémentaire afin de faciliter sa compréhension et de lever certaines ambigüités. Une ligne de commentaire peut être placée n'importe où dans le pseudocode et elle débute par //, comme le montre l'exemple suivant.

```
// Ceci est un commentaire
```

2.2 Énoncés, opérateurs et bloc

Un énoncé, ou encore une instruction, permet de spécifier une action ou une opération à faire, comme par exemple solliciter l'usager pour une valeur. Il y a un énoncé par ligne.

Un énoncé est écrit en langage courant, mais il peut aussi être composé à partir de *mots-clés* prédéfinis. Certains de ces mots-clés sont introduits dans les sections suivantes, mais pour l'instant les opérateurs mathématiques usuels ou connus sont permis, dont les suivants.

Assignation : Le symbole de l'assignation est := et cette opération permet de donner une valeur à une variable.

Arithmétique : Les opérateurs arithmétiques sont permis, comme l'addition $(+)$, la soustraction (-), la multiplication $(*)$, la division $(/)$, etc. Les parenthèses sont aussi permises afin de spécifier l'ordre d'évaluation d'une expression arithmétique.

Relationnel et logique : Les opérateurs relationnels et logiques sont permis sous formes textuelles ou mathématiques, comme plus-petit-que $(<)$, plus-grand-que $(>)$, plus-petit-ou-égal $(\leq)$, et $(\wedge)$, ou $(\vee)$, négation $(\neg)$, différent-de $(\neq)$, etc.

Incrément et décrément : L'incrément (ou le décrément) peut être écrit sous forme textuelle *Incrémenter i* ou sous forme mathématique $i := i+1$.

Quelques énoncés qui illustrent l'utilisation de ces d'opérateurs suivent et des exemples de pseudocode sont donnés dans la section 3.

```
a := b + c / (g - h)
c := (a < 3) ∧ (b ≠ 5)
```

Évidemment, un algorithme est typiquement composé de plusieurs énoncés qui forment une séquence. Dans certaines situations, afin d'éviter les ambigüités, il est parfois nécessaire de spécifier explicitement le début et la fin d'une séquence et ainsi définir ce qu'on peut appeler un *bloc*. Cela peut être fait avec les mots-clés **DÉBUT** et **FIN**, comme le montre l'exemple suivant. Afin d'améliorer la lisibilité, les énoncés à l'intérieur d'un bloc sont indentés.

```
DÉBUT
    // Commentaire
    Énoncé 1
    Énoncé 2
    ...
FIN
```

2.3 Variables et variables locales

Lorsqu'un algorithme utilise des variables, souvent appelées *variables locales*, elles doivent être déclarées, dans le sens d'annoncées au lecteur, au tout début de l'algorithme sous forme de commentaires, comme le montre l'exemple suivant.

```
// age (entier) : l'age du capitaine, valeur positive
// temperature (réel) : la température extérieure
// donnees (tableau d'entiers) : taille 10, les données lues du fichier
```

Déclarer une variable consiste à donner son nom, son type entre parenthèses et, si nécessaire, un deux-points (:) suivi d'une courte description. Cette description peut éventuellement contenir des contraintes sur la variable, comme indiquer qu'elle doit être strictement positive.

Dans certaines situations, il est permis de déclarer plusieurs variables **du même type** sur la même ligne. Évidemment, cela ne doit pas nuire à la compréhension ou la clarté du pseudocode et un bon jugement doit être utilisé. Un exemple de ce cas est le suivant :

```
// x, y et z (entier) : les coordonnées du point dans l'espace
```

Les types de base pour les variables sont : *entier*, *réel* et *caractère*. Évidemment, les types de données créés par le programmeur sont aussi admis. Les tableaux sont permis et ils peuvent être multidimensionnels. Les éléments d'un tableau sont référés par les caractères [et]. Des exemples de déclarations de variables suivent et d'autres exemples sont donnés à la section 3.

2.4 Opérations d'entrées et sorties

Les algorithmes exigent souvent d'obtenir des données de l'usager ou d'un fichier ainsi que d'afficher des données à l'écran ou de les écrire dans des fichiers. Les opérations suivantes sont celles disponibles pour les entrées et sorties. Typiquement, sur un ordinateur, les entrées et les sorties sont les suivantes : clavier, écran et fichier. Par contre, il est possible que dans certaines situations d'autres entrées et sorties existent, comme des capteurs ou des actuateurs, par exemple lire la température d'un thermomètre. Des exemples d'utilisation de ces opérations suivent et d'autres sont donnés à la section 3.

```
Lire nomVariable du clavier
Lire température du thermomètre
Lire nomVariable du fichier nomFichier
Écrire nomVariable à l'écran
Écrire voltage dans le contrôleur de tension
Écrire nomVariable dans nomFichier
```

Dans certaines situations, pareillement à la déclaration de variables, il est possible de lire ou d'écrire plusieurs variables ou valeurs en un seul énoncé. Évidemment, cela ne doit pas nuire à la compréhension ou la clarté du pseudocode et un bon jugement doit être utilisé. Un exemple de ce cas est le suivant :

```
Lire x, y et z du clavier
```

Il est important de bien comprendre l'interprétation de ces énoncés. Lire une variable du clavier signifie que la valeur donnée au clavier est stockée dans la variable. Ainsi, si l'usager donne la valeur 10 au clavier, la variable contiendra alors la valeur 10. Pareillement, écrire une variable à l'écran signifie en fait d'écrire la valeur qui est stockée dans la variable à l'écran.

Aussi, en admettant que la variable `température` contienne la valeur 10, il y a une différence entre les deux énoncés suivants :

Écrire température à l'écran
Écrire "température" à l'écran

Le premier affiche la valeur `10` à l'écran alors que le second écrit la chaine de caractères `température` à l'écran. Le premier affiche une valeur numérique, la valeur de la variable, alors que le second affiche une chaine de caractères, comme l'indiquent les guillemets.

2.5 Énoncés de sélection

Plusieurs énoncés de sélections, aussi appelés énoncés conditionnels, sont disponibles. Ces énoncés permettent de changer la séquence d'exécution exécutant certains énoncés ou non en accord avec les conditions logiques spécifiées. Dans tous les cas, un énoncé peut être remplacé par un bloc d'énoncés (section 2.2). Des exemples d'énoncés de sélection sont donnés à la section 3.

2.5.1 Énoncé si-alors

Cet énoncé de sélection permet d'exécuter un énoncé uniquement si la condition est vraie, autrement, l'énoncé n'est pas exécuté. L'équivalent anglais de cet énoncé est le *if-then*.

SI condition **ALORS**
    Énoncé

2.5.2 Énoncé si-alors-sinon

Cet énoncé de sélection permet d'exécuter un énoncé uniquement si la condition est vraie (Énoncé 1), autrement, un autre énoncé est exécuté (Énoncé 2). L'équivalent anglais de cet énoncé est le *if-then-else*.

SI condition **ALORS**
    Énoncé 1
SINON
    Énoncé 2

2.5.3 Énoncé si-alors-sinon-si

Cet énoncé de sélection permet d'exécuter un de plusieurs énoncés selon quelle est la première condition rencontrée qui est vraie. Si aucune condition n'est vraie, alors l'énoncé spécifié par la clause sinon est exécuté. La clause terminale sinon est optionnelle. Cet énoncé n'a pas vraiment d'équivalent anglais, mais si on veut lui en donner un, alors *if-then-else-if-else*.

SI condition 1 **ALORS**
    Énoncé 1
SINON SI condition 2 **ALORS**
    Énoncé 2

SINON SI ... **ALORS**
    ...
SINON
    Énoncé n

2.6 Énoncés de répétition

Plusieurs énoncés de répétition, aussi appelés énoncés de boucle, sont disponibles. Ces énoncés permettent de répéter un énoncé un certain nombre de fois, possiblement aucune. Dans tous les cas, un énoncé peut être remplacé par un bloc d'énoncés (section 2.2). Des exemples d'énoncés de répétition sont donnés à la section 3.

2.6.1 Énoncé tant-que

L'énoncé de boucle *tant que* permet de répéter un énoncé tant que la condition est vraie. Il faut remarquer que si la condition est fausse dès le départ, l'énoncé ne sera jamais exécuté. L'équivalent anglais de cet énoncé est le *while*.

TANT QUE condition
    Énoncé

2.6.2 Énoncé pour

L'énoncé de boucle *pour* permet de répéter un énoncé pour un intervalle de valeurs, les valeurs limites étant incluses, ou pour un ensemble de valeurs. Ce sont les deux exemples qui suivent.

POUR i := x **À** y **PAR PAS DE** 2
    Énoncé

POUR toutes les valeurs de la liste
    Énoncé

Dans le premier exemple, la boucle est exécutée pour des valeurs de i allant de x à y inclusivement en incrémentant i de 2 à chaque itération. Avec l'énoncé *pour*, l'incrément est de 1 par défaut, s'il n'est pas spécifié avec la partie *par pas de*.

Il faut remarquer qu'il est possible que l'intervalle fasse en sorte que l'énoncé ne soit jamais exécuté ou encore que l'ensemble des valeurs soit vide et que l'énoncé ne soit jamais exécuté. L'équivalent anglais de cet énoncé est le *for*.

2.6.3 Énoncé faire

L'énoncé de boucle *faire* permet de répéter un énoncé tant que la condition est vraie. Cet énoncé, contrairement au *tant que*, assure que l'énoncé est exécuté au moins une fois. L'équivalent anglais de cet énoncé est le *do-while*.

FAIRE
    Énoncé
TANT QUE condition

2.7 Membres de structure ou de classe

Dans certaines situations, il est intéressant d'utiliser les concepts de structure ou de classe. Dans ce cas, l'accès aux membres de la structure ou de la classe est fait en utilisant le point comme opérateur. En voici quelques exemples d'utilisation :

structure.membre := valeur
Écrire structure.membre à l'écran

2.8 Fonctions et procédures

Il est intéressant dans certaines situations de séparer logiquement le pseudocode en plusieurs parties. Ces parties, par exemple un calcul de moyenne, peuvent alors être réutilisées à partir de différents endroits dans le pseudocode. Ces parties sont appelées *fonction* ou *procédure*. Historiquement, les deux se distinguent par le fait qu'une fonction retourne une valeur (une seule !), comme la moyenne calculée, alors que la procédure ne retourne pas de valeur, par exemple une procédure qui écrirait une valeur dans un fichier. Bien que certains auteurs distinguent l'une de l'autre, dans ce document, la différence n'est pas retenue et seules les fonctions, avec ou sans valeur de retour, sont permises.

2.8.1 Fonctions

Afin d'indiquer que le pseudocode définit une fonction, le mot-clé **FONCTION** est utilisé. Ce mot-clé permet de spécifier le *nom* de la fonction. Les *paramètres*, entre parenthèses, et la *valeur de retour*, précédée par un deux points (:), sont ensuite donnés. Ils suivent la même syntaxe que les variables locales (section 2.3) et sont indentés. Selon le contexte, une description sommaire de la tâche réalisée par la fonction peut être donnée. Le *corps* de la fonction suit ces informations et il est donné dans un bloc (section 2.2). Il est à noter que les variables locales sont déclarées dans le corps de la fonction et non pas avec les paramètres et la valeur de retour. Ceci permet de bien séparer ce qui constitue l'interface avec la fonction (paramètres et valeur de retour) de ce qui est interne à la fonction.

Le corps de la fonction est un bloc (section 2.2) qui comprend les énoncés nécessaires à la réalisation de la tâche. Afin d'invoquer (appeler) une fonction dans du pseudocode, il suffit d'utiliser son nom et de donner les paramètres requis entre parenthèses. Un exemple de définition de fonction suit, mais des exemples sont aussi sont donnés dans la section 3.

```
FONCTION CalculBidon(age) : valeur
    // Description sommaire de la tâche réalisée par la fonction
    // age (entier) : l'age du capitaine, valeur positive
    // valeur (réel) : la valeur bidon calculée et retournée
DÉBUT
    // température (réel) : la température extérieure
    // nbCarottes (entier) : le nombre de carottes, valeur positive
    // nbPatates (entier) : le nombre de patates, valeur positive
    Lire température du thermomètre
    valeur := age * température / (nbCarottes + nbPatates)
    Retourner valeur
FIN
```

2.8.2 Paramètres en entrée et en sortie

Pour chaque paramètre d'une fonction, il est possible de le qualifier et de spécifier s'il est en entrée ou en sortie avec les mots-clés **entrée** et **sortie** respectivement. Ne rien spécifier pour un paramètre est équivalent à spécifier **entrée**. Le mot-clé **entrée** spécifie qu'un paramètre est une valeur fournie par la fonction appelante. Le mot-clé **sortie** spécifie que la valeur de

ce paramètre sera donnée par la fonction et que la fonction appelante pourra récupérer et consulter cette valeur. Essentiellement, **entrée** et **sortie** correspondent au **passage par valeur** et au **passage par référence** dans les langages de programmation. L'exemple suivant montre l'utilisation des mots-clés **entrée** et **sortie**.

```
FONCTION FoisCinq(valeur : entrée, résultat : sortie) : succès
    // valeur (entier) : la valeur fournie par la fonction appelante
    // résultat (entier) : le résultat du calcul donné à la fonction appelante
    // succès (entier) : 0 si erreur, 1 sinon
DÉBUT
    succès := 1
    résultat := valeur * 5
    Retourner succès
FIN
```

Il est important de ne pas confondre un paramètre en sortie et la valeur de retour d'une fonction. L'exemple précédent illustre ce propos. En effet, la fonction donne comme valeur de retour un code indiquant le succès ou non d'une opération, mais le résultat de l'opération est donné par l'entremise d'un paramètre en sortie.

3 Exemples

3.1 Calcul du carré

L'algorithme qui suit est un exemple sans définition de fonction, ni appel de fonction.

```
DÉBUT
    // valeur (réel) : valeur donnée par l'usager
    // carré (réel) : résultat du calcul
    Lire valeur du clavier
    carré := valeur * valeur
    Écrire carré à l'écran
FIN
```

3.2 Menu avec aide

L'algorithme qui suit est un exemple sans définition de fonction, mais qui fait **appel à des fonctions** dont le pseudocode n'est pas donné. Ce sont les fonctions AfficherResultat, AfficherMoyenne, AfficherEcartType et AfficherAide. Il est important de ne pas confondre un appel de fonction, comme AfficherResultat, avec l'opération d'écriture de la section 2.4.

```
DÉBUT
    // L'usager choisi une fonctionnalité à exécuter.
    // Des fonctions sont appelées, mais leur pseudocode n'est pas donné.
    // choix (entier) : le choix fait par l'usager
    Lire choix du clavier
    SI choix est égal à 1 ALORS
        AfficherResultat()
    SINON SI choix est égal à 2 ALORS
        AfficherMoyenne()
```

```
    SINON SI choix est égal à 3 ALORS
        AfficherEcartType()
    SINON
        AfficherAide()
FIN
```

3.3 Calcul de puissance

L'algorithme qui suit est un exemple de définition de fonction. Il est à noter que les variables locales sont déclarées dans le corps de la fonction, comme énoncé à la section 2.8.

```
FONCTION CalculPuissance(valeur, exposant) : puissance
    // valeur (entier)
    // exposant (entier) : valeur positive ou nulle
    // puissance (entier) : le resultat du calcul et qui est retourné
DÉBUT
    // i (entier)
    puissance := 1
    POUR i := 1 À exposant
        puissance := puissance * valeur
    Retourner puissance
FIN
```

3.4 Nombre de valeurs positives

L'algorithme qui suit est un exemple de définition de fonction. Il est à noter que les variables locales sont déclarées dans le corps de la fonction, comme énoncé à la section 2.8.

```
FONCTION NombrePositives(valeurs, taille) : nombre
    // Trouve le nombre de valeurs positives dans un tableau
    // valeurs (tableau d'entiers)
    // taille (entier) : taille du tableau
    // nombre (entier) : nombre de valeurs positives
DÉBUT
    // i (entier)
    nombre := 0
    POUR i := 0 À taille-1
        SI valeurs[i] > 0 ALORS
            nombre := nombre + 1
    Retourner nombre
FIN
```

Liste des références

[1] K. H. Rosen, *Mathématiques discrètes, édition révisée*. Chenelière–McGraw-Hill, 2002.