Support de cours pour AP1-algo et AP2-algo

Christine Solnon

2007-2008

Table des matières

Chapitre 1

Introduction

1.1 Notion d'algorithme

L'algorithmique est une notion ancienne (apparue bien avant les premiers ordinateurs) qui peut se définir comme

> *"une méthode de résolution d'un problème sous la forme d'une suite d'opérations élémentaires obéissant à un enchaînement déterminé."*

Considérons par exemple le problème suivant :

> Plusieurs dossiers, classés par ordre alphabétique sur le nom, sont empilés sur une table. Il s'agit de déterminer si une personne de nom X a bien un dossier à son nom dans la pile.

Avant de chercher à résoudre ce problème, on commence par le spécifier plus précisément. Pour celà, on identifie :
- les données du problème (ce qu'on a en entrée avant de commencer la résolution) ;
- les résultats (ce qu'on fournit en sortie à la fin de la résolution) ;
- les éventuelles préconditions sur les données (des informations sur la nature des données en entrée) ;
- les postconditions (une relation définissant ce qu'on doit fournir en sortie en fonction de ce qu'on a en entrée).

Pour notre problème de recherche de dossier, on peut définir la spécification suivante :

Algorithme : Recherche linéaire
Entrées :
Une pile de dossiers P
Un nom X
Sorties :
Une réponse R
Précondition :
Les dossiers de P sont empilés par ordre alphabétique sur le nom, le dossier comportant le plus petit nom se trouvant au sommet de la pile.
Postcondition :
Si P contient un dossier au nom de X alors R doit être égal à *oui* sinon R doit être égal à *non*

On peut maintenant chercher une méthode pour résoudre ce problème. Une première solution consiste à regarder successivement chaque dossier de la pile de façon linéaire, à partir du dossier se trouvant au sommet de la pile, jusqu'à trouver soit le dossier ayant pour nom X, soit un dossier ayant un nom supérieur à X (auquel cas X n'a pas de dossier). L'algorithme correspondant peut s'exprimer de la façon suivante :

```
début
    tant que la pile de dossiers P n'est pas vide
    et le nom du dossier au sommet de P est inférieur à X faire
        Prendre le dossier au sommet de P
        Le poser à coté de P
    fintq
    si P est vide ou le nom du dossier au sommet de P est différent de X alors
        R ← non
    sinon
        R ← oui
    finsi
fin
```

La nature des opérations élémentaires utilisées pour résoudre le problème dépend des capacités de la personne (ou machine) qui va exécuter l'algorithme.

Ici, on suppose que l'algorithme s'adresse à une personne capable d'effectuer les opérations suivantes :
– comparer (selon l'ordre alphabétique) le nom du dossier au sommet de la pile avec X,
– prendre le dossier au sommet de la pile,
– poser un dossier à coté de la pile,
– regarder si la pile de dossiers est vide.
Cet algorithme introduit 3 types d'enchaînement d'opérations.

1. L'enchaînement en séquence consiste à effectuer les opérations en séquence les unes à la suite des autres. Par exemple, les lignes 4 et 5 s'enchaînent en séquence et, par conséquent, l'opération élémentaire de la ligne 5 n'est commencée qu'une fois que celle de la ligne 4 est terminée.
2. L'enchaînement alternatif consiste à effectuer alternativement soit une première suite d'opérations, soit une autre, en fonction d'une condition. Par exemple, en fonction de la condition de la ligne 7, on effectuera alternativement l'opération élémentaire de la ligne 8, ou celle de la ligne 10.
3. L'enchaînement répétitif consiste à effectuer plusieurs fois une même suite d'opérations élémentaires tant qu'une condition donnée est vérifiée. Par exemple, les opérations des lignes 4 et 5 sont répétées tant que les deux conditions des lignes 2 et 3 sont vérifiées.

Notons enfin que cet algorithme termine, c'est à dire qu'il s'exécute en un temps fini. En effet, si la pile contient n dossiers, les opérations des lignes 4 et 5 seront exécutées au plus n fois : à chaque fois, un dossier est enlevé de la pile ; au bout de n fois, la pile est vide et la condition de la ligne 1 n'est plus vérifiée.

L'algorithme de recherche séquentielle permet effectivement de résoudre notre problème. Il est cependant relativement inefficace : si la pile contient 1000 dossiers, il faudra en moyenne regarder le nom de 500 dossiers avant de pouvoir résoudre le problème, et dans le pire des cas, il faudra en regarder 1000.

De fait, une personne sensée se trouvant face à une pile de 1000 dossiers *triés par ordre alphabétique* ne regardera pas chaque dossier linéairement, du premier au dernier, mais partagera la pile de dossiers en 2 piles de tailles comparables et, en fonction du nom du dossier au sommet de la deuxième pile, continuera sa recherche soit dans la première pile, soit dans la seconde. Elle répètera ainsi ce processus "dichotomique" jusqu'à trouver le bon dossier ou bien jusqu'à ce que la pile soit vide. L'algorithme correspondant peut être formulé de la façon suivante :

```
début
    tant que la pile de dossiers P n'est pas vide
    et le nom du dossier au sommet de P est différent de X faire
        Couper la pile de dossiers P en 2 parties
        (On appellera Psup la moitié supérieure, et Pinf la moitié inférieure)
        si le nom du dossier au sommet de Pinf = X alors
            Ne garder que le premier dossier de Pinf dans P
        sinon si le nom du dossier au sommet de Pinf > X alors
            Enlever de P tous les dossiers de Pinf
        sinon
            Enlever de P le premier dossier de Pinf
            Enlever de P tous les dossiers de Psup
        finsi
    fintq
    si P est vide alors
        R ← non
    sinon
        R ← oui
    finsi
fin
```

Ce deuxième algorithme, plus compliqué dans son énoncé que le premier est néanmoins nettement plus efficace. En effet, si la pile contient 1000 dossiers, on la séparera en 2 piles de 500 dossiers puis une des 2 piles de 500 dossiers sera séparée en 2 piles de 255 dossiers, ... de telle sorte qu'en répétant (au plus) 10 fois ce processus dichotomique, soit on aura trouvé le dossier, soit la pile sera vide. Ainsi, alors que l'algorithme de recherche séquentiel demandera en moyenne de consulter 500 dossiers, celui de recherche dichotomique ne demandera que 10 consultations (dans le pire des cas).

Ce premier exemple permet d'introduire un point fondamental de l'algorithmique, à savoir la *complexité*. En effet, étant donné un même problème, il existe généralement plusieurs algorithmes le résolvant. Ces algorithmes seront notamment comparés sur leur complexité en temps, c'est à dire sur le nombre d'opérations élémentaires qui devront être effectuées pour résoudre un même problème. Cette complexité est calculée "approximativement" dans le sens où l'on donnera simplement un ordre de grandeur : sur l'exemple précédent on dira que l'algorithme de recherche linéaire est d'ordre linéaire, noté $\mathcal{O}(n)$, ce qui signifie que si l'on a n dossiers il faudra faire de l'ordre de n opérations, tandis que l'algorithme de recherche dichotomique est d'ordre logarithmique, noté $\mathcal{O}(log_2(n))$, ce qui signifie que si l'on a n dossiers il faudra faire de l'ordre de $log_2(n)$ opérations.

1.2 Introduction à la structure d'un ordinateur

Dans le contexte de ce cours, un algorithme est conçu pour être exécuté par un ordinateur. La notion d'opération élémentaire dépend donc des capacités d'un ordinateur. De façon très schématique, un ordinateur est composé des éléments suivants :

- une *mémoire*, chargée de conserver l'information ;
  La mémoire est composée d'une suite d'informations élémentaires appelées *bit*. Un bit peut prendre deux valeurs (généralement symbolisées par 0 et 1). Une suite de 8 bits forme un octet. Chaque octet a une adresse dans la mémoire permettant d'y accéder.
- une *unité centrale* (CPU), chargée de traiter l'information ;
  L'unité centrale est capable d'exécuter un nombre fini (et très limité) d'instructions simples. L'ensemble des instructions que l'unité centrale est capable d'exécuter forme le langage machine.
- plusieurs *unités d'entrée/sortie*, chargées d'échanger des informations avec l'environnement extérieur (le clavier, la souris, l'écran, ...) ;
- un *bus*, chargé de transférer l'information entre la mémoire, les unités d'entrée/sortie et l'unité centrale.

1.3 Démarche pour la résolution d'un problème

Pour résoudre un problème à l'aide d'un ordinateur, il est assez vite apparu indispensable d'ajouter des niveaux de description et d'abstraction entre l'homme, qui pose le problème, et la machine, qui est chargée de le résoudre. En effet, l'homme définit le problème en langue naturelle (par exemple en français), de façon informelle et souvent ambigüe et incomplète. A l'autre extrémité du processus, un ordinateur n'est capable d'exécuter que des instructions de très bas niveau, où l'information est codée sous la forme d'une suite de 0 et de 1.

Du problème à la spécification formelle

A partir de la description informelle, ambigüe et incomplète du problème, *l'analyste* élabore une *spécification formelle*, qui décrit de façon non ambigüe et (si possible) complète et correcte ce que doit faire le programme, c'est à dire le "quoi". Cette description est faite indépendemment d'une solution. Il s'agit notamment de préciser
- les paramètres en entrée, c'est à dire la nature des questions ou données du problème,
- les paramètres en sortie, c'est à dire la nature des solutions ou résultats au problème,
- les préconditions, c'est à dire les conditions portant sur les paramètres en entrée sous lesquelles le problème est défini,
- la relation entre les paramètres en entrée et les paramètres en sortie, c'est à dire la valeur des paramètres en sortie en fonction de celle des paramètres en entrée,
- les contraintes à respecter pour la résolution du problème (notamment les contraintes de ressources).

De la spécification formelle à l'algorithme

A partir de la spécification formelle, *l'analyste/programmeur* élabore un *algorithme*, qui spécifie le "comment", c'est à dire l'enchaînement d'opérations élémentaires à effectuer pour résoudre le problème.

Un algorithme est généralement élaboré selon une *démarche descendante*, qui consiste à décomposer le problème en sous-problèmes, chaque sous-problème devant être de nouveau clairement spécifié puis résolu. Cette décomposition permet d'aborder le problème progressivement en créant des niveaux de description de plus en plus détaillés. Chacun des sous-problèmes sera d'autant plus compréhensible que dans sa description il n'y a que quelques idées simples. En général, la taille d'un algorithme ne doit pas dépasser une vingtaine de lignes.

Vérification de l'algorithme

Une fois conçu, un algorithme doit être validé. Il s'agit de vérifier que l'algorithme répond effectivement au problème spécifié, et plus particulièrement qu'il est
- *correct*, c'est-à-dire que les valeurs des paramètres de sortie calculées par l'algorithme sont effectivement celles que l'on souhaitait calculer ;
- *complet*, c'est-à-dire que les bonnes valeurs des paramètres de sortie sont trouvées pour toutes les valeurs possibles des paramètres en entrée (dans les limites spécifiées par les préconditions) ;
- *fini*, c'est-à-dire que le nombre d'opérations élémentaires à effectuer pour résoudre le problème est fini, autrement dit, que l'algorithme ne "boucle" pas indéfiniment sur une série d'instructions.

Cette vérification peut être effectuée de façon théorique par une preuve. Elle doit généralement être validée expérimentalement par des tests, c'est à dire une simulation de l'exécution de l'algorithme sur un certain nombre de données. Ces données utilisées pour tester la solution sont appelées jeux d'essai. Elles doivent être les plus complètes possibles, de façon à tester l'algorithme sur tous les cas possibles d'utilisation.

Il s'agit par ailleurs d'évaluer la *complexité en temps* de l'algorithme, c'est à dire un ordre de grandeur du nombre d'opérations élémentaires qui devront être effectuées en fonction des valeurs des paramètres en entrée. Enfin, il faut également évaluer la *complexité en espace* de l'algorithme, c'est à dire la taille mémoire nécessaire à l'exécution de l'algorithme.

De l'algorithme au programme

L'algorithme est ensuite codé par *le programmeur* dans *un langage de programmation*. On distingue usuellement 3 styles de langages de programmation, correspondant à 3 paradigmes de programmation :
- les langages impératifs (par exemple C, Ada, Cobol, Fortran, Pascal), où les opérations élémentaires sont des affectations et appels de procédures ;
- les langages fonctionnels (par exemple Lisp, Scheme, Haskell, CAML), où les opérations élémentaires sont des appels de fonctions ;
- les langages logiques (par exemple Prolog), où les opérations élémentaires sont des affirmations logiques.

Dans ces 3 catégories de langages de programmation, les programmes sont essentiellement structurés en termes de décomposition fonctionnelle et de traitements. Il existe un quatrième paradigme de programmation, orthogonal aux 3 premiers qui consiste à structurer les programmes en termes de données : la programmation orientée objet. Ainsi, il existe des langages impératifs orientés objet (par exemple C++, Java, Eiffel), des langages fonctionnels orientés objet (par exemple Clos, Smalltalk, Loops) et des langages logiques orientés objet (par exemple L&O, Login, Life).

Les opérations élémentaires utilisées pour coder un algorithme dépendent du langage de programmation choisi. Néanmoins, tous ces langages permettent de coder facilement les différents types d'enchaînements (en séquence, alternatif et répétitif) apparaissant dans un algorithme. La structuration et la gestion des données varie aussi d'un langage à l'autre. Il est alors nécessaire d'adapter en conséquence l'algorithme au moment du codage.

Du programme au code exécutable

Le code écrit dans un langage de programmation, appelé le code source, est ensuite traduit par un *compilateur* ou un *interpréteur* en un code en langage machine, appelé le code objet et exécutable par l'ordinateur. Un compilateur traduit la totalité du programme source pour générer un code objet qui sera ensuite exécuté, tandis qu'un interpréteur traduit une par une les instructions du code source et les exécute au fur et à mesure de la traduction. Cette phase de compilation ou d'interprétation est entièrement automatique et est effectuée par un programme. Vous en apprendrez plus sur ce sujet à l'occasion du cours de théorie des langages, en deuxième année.

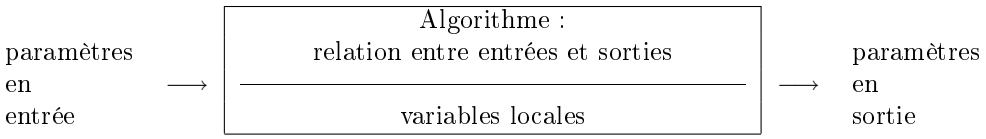
Chapitre 2

Variables, Expressions et Affectations

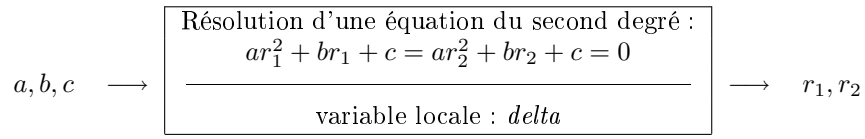
2.1 Les variables

Un problème est spécifié par une relation entre les valeurs des paramètres en entrée (les données) et les valeurs des paramètres en sortie (les résultats). Pour calculer les valeurs des paramètres en sortie à partir des valeurs en entrée, un algorithme peut utiliser des données intermédiaires, appelées variables locales.

Ainsi, un algorithme manipule 3 catégories de données, plus généralement appelées variables : les paramètres en entrée, les paramètres en sortie et les variables locales ; ce qui peut être schématisé de la façon suivante :



Par exemple, le problème consistant à résoudre une équation du second degré peut être décrit par :



Chaque variable (paramètre en entrée, paramètre en sortie ou variable locale) peut être vue comme une boite contenant une information. Elle est caractérisée par un nom, permettant de référencer la boite, un type, permettant de caractériser la nature des informations que l'on pourra déposer dans la boite, et une valeur, désignant l'information effectivement contenue dans la boite.

2.1.1 Nom d'une variable

Le nom d'une variable, aussi appelé identificateur, est la représentation symbolique de l'adresse où est stockée la valeur de la variable dans la mémoire. En effet, les données sont conservées dans la mémoire de l'ordinateur. Cette mémoire est constituée d'une suite de bits (informations binaires) regroupés par groupe de 8 en octets. Chaque octet a une adresse permettant de le retrouver. Au lieu de désigner une variable par son adresse en mémoire, on donne un nom symbolique à cette adresse.

Syntaxiquement, le nom d'une variable est une suite de caractères alpha-numériques, commençant par une lettre (majuscule ou minuscule). Le nom de la variable doit être choisi judicieusement afin de donner une indication sur la nature et le rôle de la donnée. Ainsi, si la variable correspond au total d'une facture, il est préférable de l'appeler `total-facture` plutôt que `toto` ou `X23`.

2.1.2 Type d'une variable

Toutes les données sont stockées en mémoire sous la forme d'une suite de 0 et de 1. En fonction du type (de la nature) de la donnée, cette suite de 0 et de 1 sera interprétée (décodée) différemment. Par exemple,
- un entier relatif peut être codé sur 2 octets consécutifs (soit 16 bits) de telle sorte que le bit le plus à gauche indique le signe de l'entier (positif si le bit est à 0, négatif sinon), tandis que les autres bits correspondent à l'entier codé en base 2 ;
- un nombre flottant peut être codé sur 4 octets consécutifs de telle sorte que l'octet le plus à gauche représente l'exposant (en base 2) et les 3 octets suivants la mantisse (en base 2) ;
- un caractère est codé sur un octet par son code ASCII en base 2 ;
- etc, ...

Pour connaître la valeur d'une variable, il est alors indispensable de connaître au préalable son type pour savoir sur combien d'octets elle est codée et comment interpréter correctement la suite correspondante de 0 et de 1.

Ainsi, il faut déclarer le type de chaque variable (paramètres et variables locales) avant de l'utiliser. A la suite de cette déclaration, on ne pourra mettre dans la variable que des informations appartenant au type déclaré.

La déclaration d'une donnée s'effectuera en faisant suivre le type de la variable de son nom. On utilisera dans la suite les types de données suivants :
- Les types de données *élémentaires* :
  - le type `entier`, désignant l'ensemble des entiers relatifs ;
    En pratique, le nombre d'entiers que l'on peut représenter est limité par le nombre d'octets utilisés pour les représenter, et les entiers trop grands (en valeur absolue) doivent être codés différemment.
  - le type `réel`, désignant l'ensemble des réels ;
    Là encore, il n'est pas possible de coder l'ensemble de tous les réels. En pratique, les réels sont approximés par des nombres flottants (ce qui peut provoquer des erreurs numériques).
  - le type `car`, désignant l'ensemble des caractères du clavier (caractères alpha-numériques et caractères spéciaux) ;
  - le type `logique` (aussi appelé booléen), désignant les deux valeurs `vrai` et `faux` ;
  - le type `pointeur`, désignant l'ensemble des adresses mémoires (ce type sera étudié au deuxième semestre) ;
  - le type `texte`, aussi appelé chaîne, désignant l'ensemble des chaînes de caractères[1]
- Les types de données *composés* :
  - le type `tableau`, désignant une suite indexée comportant un nombre variable mais borné de données de même type ;
  - le type `structure`, désignant une suite comportant un nombre fixé de données de types différents ;
  - le type `fichier`, désignant une suite comportant un nombre variable mais non borné a priori de données de même type (ce type sera étudié au deuxième semestre).

2.1.3 Valeur d'une variable

La valeur d'une variable est contenue dans l'emplacement mémoire se trouvant à l'adresse représentée par le nom de la variable. Ce contenu (une suite de 0 et de 1) doit être interprété correctement en fonction du type de la variable.

Attention : à la suite de sa déclaration, la valeur d'une variable n'est pas définie.

Données constantes : Certaines variables ont une valeur constante, qui ne change pas pendant toute l'exécution de l'algorithme. Ces variables sont appelées *constantes*. Elles sont déclarées au début de l'algorithme en faisant suivre le nom de la variable par sa valeur.

[1] Dans de nombreux langages, le type `texte` est représenté par un tableau de caractères.

2.2 Description des variables dans un algorithme

Par la suite, la partie statique d'un algorithme, spécifiant les paramètres et déclarant les variables, sera décrite selon le formalisme suivant :

Procédure : nom-de-la-procédure(<noms-des-paramètres>)
Entrées :
pour chaque paramètre en entrée, préciser son type et son nom
Sorties :
pour chaque paramètre en sortie, préciser son type et son nom
Précondition :
Conditions sur les paramètres en entrée
Postcondition :
Relation entre les paramètres en entrée et ceux en sortie
Déclarations :
pour chaque variable locale, préciser son type et son nom
const : pour chaque constante, préciser son nom et sa valeur
début
Suite d'opérations élémentaires permettant de calculer les paramètres en sortie en fonction des paramètres en entrée (partie dynamique de l'algorithme)
fin

Par exemple, l'algorithme calculant les racines d'une équation du second degré peut être spécifié de la façon suivante :

Procédure : racines(a, b, c, r_1, r_2)
Entrées :
réel a
réel b
réel c
Sorties :
réel r_1
réel r_2
Précondition :
$b^2 - 4ac \geq 0$ **et** $a \neq 0$
Postcondition :
$ar_1^2 + br_1 + c = ar_2^2 + br_2 + c = 0$ ou autrement dit, r_1 et r_2 sont les deux solutions de l'équation $ax^2 + bx + c = 0$
Déclarations :
réel *delta*
début
Suite d'opérations élémentaires
permettant de calculer r_1 et r_2 à partir de a, b et c.
fin

Pour abbréger, on pourra regrouper plusieurs déclarations de paramètres sur une même ligne lorsqu'ils ont le même mode de passage (entrée ou sortie) et le même type :

Procédure : racines(a, b, c, r_1, r_2)
Entrées :
réel a, b, c
Sorties :
réel r_1, r_2

2.3 L'affectation

Pour résoudre un problème, un algorithme décrit la suite d'instructions à effectuer pour calculer les valeurs des paramètres en sortie à partir des valeurs des paramètres en entrée. Il existe 2 instructions différentes : l'affectation, qui permet de changer la valeur d'une variable, et l'appel de procédure, qui permet d'appeler (d'utiliser) un algorithme dans un autre. On étudie ici l'affectation ; l'appel de procédure sera étudié au chapitre 4.

L'affectation permet de changer la valeur d'une variable, autrement dit de modifier le contenu de la mémoire à l'adresse symbolisée par le nom de la variable. Syntaxiquement, une affectation sera représentée par :

`nom-var` $\leftarrow$ `expr`

et a pour signification : la variable de nom `nom-var` prend pour valeur la valeur de l'expression `expr`, autrement dit, la valeur de l'expression `expr` est stockée dans la mémoire à l'adresse symbolisée par le nom `nom-var`. La valeur de l'expression `expr` doit appartenir au type déclaré pour la variable `nom-var`.

2.4 Définition d'une expression

L'expression affectée à une variable peut être une valeur explicite :

par exemple, l'instruction $a \leftarrow 25$ affecte la valeur 25 à la variable de nom a.

L'expression affectée à une variable peut également être la valeur contenue dans une autre variable :

par exemple, l'instruction $a \leftarrow b$ affecte à la variable de nom a la valeur contenue dans la variable de nom b.

Enfin, l'expression affectée à une variable peut être le résultat d'une opération entre d'autres expressions. Il existe 3 types d'opérations : les opérations arithmétiques, les opérations de comparaison et les opérations logiques.

2.4.1 Opérations arithmétiques

Les opérations arithmétiques binaires sont : l'addition, notée "+", la soustraction, notée "-", la multiplication, notée "*", la division réelle, notée "/", la division entière notée "div" et le modulo[2], noté "mod". Ces opérations prennent en argument deux expressions numériques et rendent la valeur numérique correspondant à l'application de l'opération sur les valeurs des expressions en argument.

Ces opérations sont définies sur les réels et sur les entiers, sauf "div" et "mod" qui ne sont définies que sur les entiers.

Enfin, les opérateurs de multiplication, de division et de modulo sont plus prioritaires que (et donc évalués avant) les opérateurs d'addition et de soustraction. Une expression arithmétique peut être parenthésée pour spécifier l'ordre de son évaluation.

Les opérations arithmétiques unaires sont le plus unaire, noté "+", et le moins unaire, noté "-". Ces opérations prennnent en argument une expression numérique et rendent une valeur numérique.

2.4.2 Opérations de comparaison

Une opération de comparaison prend en argument deux expressions de même type élémentaire (2 entiers, 2 réels, 2 caractères ou 2 chaînes de caractères), et rend une valeur logique (vrai ou faux). On utilisera les opérations de comparaison suivantes : $<, \leq, =, \neq, \geq$ et $>$.

Les entiers et les réels sont comparés selon l'ordre numérique usuel. Les caractères sont comparés selon l'ordre défini par le code ASCII : chaque caractère est codé par un entier compris entre 0 et 255, appelé code ASCII, et pour comparer deux caractères, on compare leur code ASCII. Dans ce code, les chiffres sont inférieurs aux caractères majuscules qui sont eux-mêmes inférieurs au caractères minuscules.

[2] On rappelle que le modulo est le reste de la division entière. Autrement dit, $x \bmod y = x - y * (x \text{ div } y)$ Par exemple, $23 \bmod 7 = 2$ car $23 \text{ div } 7 = 3$ et $23 - 3 * 7 = 2$.

Les chaînes de caractères sont comparées selon l'ordre lexicographique : un texte $t1$ est inférieur à un texte $t2$ s'il existe un caractère c de $t1$ tel que
– pour chaque caractère se trouvant avant c dans $t1$, le caractère correspondant de $t2$ soit identique,
– le caractère correspondant à c dans $t2$ soit supérieur à c.
Par exemple, "$aabcd$" < "aax" et "10" < "9".

2.4.3 Opérations logiques

Les opérations logiques binaires sont la disjonction, notée `ou`, et la conjonction, notée `et`. Ces opérateurs prennent en argument 2 expressions de valeur logique et rendent une valeur logique.

La signification des opérateurs logiques binaires est donnée par la table suivante :

A	B	A `et` B	A `ou` B
vrai	vrai	vrai	vrai
vrai	faux	faux	vrai
faux	vrai	faux	vrai
faux	faux	faux	faux

L'opération logique unaire est la négation, notée `non`. Cet opérateur prend en argument une expression de valeur logique et rend une valeur logique, selon la table suivante :

A	`non` A
vrai	faux
faux	vrai

2.4.4 Exemple

Soient les variables suivantes :

entier e_1, e_2,
réel r_1,
logique b_1, b_2

Après l'exécution de l'instruction

$e_1 \leftarrow (10 \ mod \ 3) + (5 \ div \ 2)$

la variable e_1 a pour valeur $1 + 2 = 3$;
après l'exécution de l'instruction

$e_2 \leftarrow e_1 * 2$

la variable e_2 a pour valeur $3 * 2 = 6$;
après l'exécution de l'instruction

$r_1 \leftarrow 4.5 * 2.0$

la variable r_1 a pour valeur 9.0 ;
après l'exécution de l'instruction

$r_1 \leftarrow r_1/3.0$

la variable r_1 a pour valeur $9.0/3.0 = 3.0$;
après l'exécution de l'instruction

$b_1 \leftarrow (r_1 > 4.2)$ `ou` $(e_1 = e_2)$

la variable b_1 a pour valeur $faux$;
après l'exécution de l'instruction

$b_2 \leftarrow$ `non`(b_1) `et` $r_1 \leq 4.2$

la variable b_2 a pour valeur $vrai$.

2.5 Saisie et affichage de données

Un algorithme calcule les valeurs des paramètres en sortie, en fonction des valeurs des paramètres en entrée. Les valeurs des paramètres en entrée sont "données" à l'algorithme, tandis que les valeurs des paramètres en sortie sont fournies comme résultat de l'algorithme à celui qui l'exécutera. Ainsi, on ne s'occupe pas *a priori* de la saisie des valeurs des paramètres en entrée ni de l'affichage des valeurs calculées pour les paramètres en sortie. De fait, il est possible que les valeurs calculées par un algorithme ne soient pas affichées, mais par exemple qu'elles soient fournies comme valeurs d'entrée d'un autre algorithme.

Dans certains cas, on peut cependant souhaiter saisir au clavier une valeur pour l'affecter à une variable. Pour cela, on pourra utiliser l'instruction "saisir(x)" qui lit au clavier une valeur de même type que la variable x et affecte cette valeur à x. De même, pour afficher à l'écran la valeur d'une variable x, on pourra utiliser l'instruction "afficher(x)".

Chapitre 3

Enchaînement d'instructions

Les instructions d'un algorithme sont enchainées selon un ordre déterminé. Il existe trois façons différentes d'enchaîner des instructions : en séquence, de façon alternative ou répétitive.

3.1 Enchaînement séquentiel

L'enchaînement séquentiel d'une suite d'instructions permet d'exécuter les instructions les unes à la suite des autres. Syntaxiquement, les instructions seront notées les unes en dessous des autres (une instruction par ligne). Dans le cas d'instructions courtes, on pourra noter plusieurs instructions les unes à coté des autres, séparées par un point virgule. Dans ce cas, les instructions sont tout naturellement exécutées de la gauche vers la droite.

Exercice 1 *Calcul du diamètre, du périmètre et de la surface d'un cercle à partir de son rayon*

Exercice 2 *Calcul des coefficients d'une droite à partir de deux points*

Exercice 3 *Résolution d'une équation du second degré admettant exactement 2 solutions*

3.2 Enchaînement alternatif

L'enchaînement alternatif permet d'exécuter alternativement une première suite d'instructions, si une certaine condition est vérifiée, ou bien une autre série d'instructions, si la condition n'est pas vérifiée.

La syntaxe d'un enchaînement alternatif est

si *condition* **alors**
| suite-1
sinon
| suite-2
finsi

où `condition` est une expression logique, et `suite-1` et `suite-2` sont des suites d'expressions. Cet énoncé alternatif est interprété de la façon suivante :

> "si l'expression logique `condition` est évaluée à `vrai`, alors la suite d'instructions `suite-1` est exécutée, sinon (si l'expression logique `condition` est évaluée à `faux`), la suite d'instructions `suite-2` est exécutée."

Une version simplifiée de ce schéma est

```
si condition alors
| suite-1
finsi
```

Cet énoncé est interprété de la façon suivante :

> "si l'expression logique `condition` est évaluée à `vrai`, alors la suite d'instructions `suite-1` est exécutée, sinon (si l'expression logique `condition` est évaluée à `faux`), on ne fait rien."

Enfin, dans le cas où l'on a plus de deux alternatives dont toutes les conditions sont exclusives, on pourra utiliser l'énoncé suivant :

```
si condition1 alors
| suite-1
sinon si condition2 alors
| suite-2
sinon si condition3 alors
| suite-3
sinon si condition4 alors
| suite-4
sinon
| suite-5
finsi
```

Cet énoncé est équivalent à l'énoncé suivant, imbriquant 4 énoncés alternatifs à deux alternatives :

```
si condition1 alors
| suite-1
sinon
| si condition2 alors
| | suite-2
| sinon
| | si condition3 alors
| | | suite-3
| | sinon
| | | si condition4 alors
| | | | suite-4
| | | sinon
| | | | suite-5
| | | finsi
| | finsi
| finsi
finsi
```

Exercice 4 *Recherche du nombre de solutions d'une équation du second degré quelconque*

Exercice 5 *Recherche du plus petit nombre parmi trois nombres*

3.3 Enchaînement répétitif

L'enchaînement répétitif permet d'exécuter plusieurs fois une même suite d'instructions ; le nombre de fois où la suite d'instructions est exécutée étant déterminé par une condition logique.

La syntaxe d'un enchaînement répétitif est

```
tant que cond faire
| suite-inst
fintq
```

Cet énoncé est interprêté par l'ordinateur de la façon suivante :

> "tant que l'expression logique `cond` est évaluée à `vrai`, exécuter les instructions `suite-inst`, et recommencer ; arrêter ce processus itératif quand `cond` est évaluée à `faux`."

Exemple 1 : répétition d'un traitement k fois

Parfois, le nombre de fois où le traitement doit être répété est connu dans le sens où une variable du programme contient le nombre d'itérations à faire. Dans ce cas, une solution consiste à utiliser un compteur de boucle, que l'on incrémente de 1 à chaque passage dans la boucle. La condition `cond` de continuation de la boucle dépend alors de la valeur initialement mise dans ce compteur : on peut par exemple initialiser le compteur à 1 et continuer tant qu'il est inférieur ou égal au nombre d'itérations souhaitées ; on peut tout aussi bien initialiser le compteur à 0 et continuer tant qu'il est strictement inférieur au nombre d'itérations souhaitées.

Considérons par exemple le problème consistant à calculer x à la puissance n, où n est un nombre entier naturel. Pour résoudre ce problème, il s'agit de faire n multiplications par x. Pour cela, on peut par exemple compter de 0 à $n-1$. On obtient l'algorithme suivant :

```
Procédure : puissance(n, x, p)
Entrées :
    entier n
    réel x
Sorties :
    réel p
Précondition :
    n ≥ 0
Postcondition :
    p = x^n
Déclarations :
    entier cpt
début
| p ← 1
| cpt ← 0
| tant que cpt < n faire
| | /* invariant : p = x^cpt                                          */
| | p ← p * x
| | cpt ← cpt + 1
| fintq
| /* Nombre de passages dans la boucle = n ; cpt = n et p = x^cpt = x^n */
fin
```

On peut simuler l'exécution de cet algorithme en remplissant un tableau donnant les valeurs successivement prises par les variables. Par exemple, si $x = 2$ et $n = 4$:

	p	cpt
Avant le premier passage	1	0
Après le premier passage	2	1
Après le deuxième passage	4	2
Après le troisième passage	8	3
Après le quatrième passage	16	4

On peut vérifier qu'à chaque passage on a bien $p = x^{cpt}$. On peut également vérifier qu'après le dernier passage, on a bien $cpt = n$ et $p = x^{cpt}$.

On peut imaginer d'autres algorithmes pour calculer x à la puissance n. On aurait par exemple pu compter de 1 à n...

```
Procédure : puissance(n, x, p)
...
Déclarations :
    entier cpt
début
    p ← 1
    cpt ← 1
    tant que cpt ≤ n faire
        /* invariant : p = x^(cpt-1)                                   */
        p ← p * x
        cpt ← cpt + 1
    fintq
    /* Nombre de passages dans la boucle = n ; cpt = n + 1 et p = x^(cpt-1) = x^n   */
fin
```

...ou encore de n à 1.

```
Procédure : puissance(n, x, p)
...
Déclarations :
    entier cpt
début
    p ← 1
    cpt ← n
    tant que cpt > 0 faire
        /* invariant : p = x^(n-cpt)                                   */
        p ← p * x
        cpt ← cpt - 1
    fintq
    /* Nombre de passages dans la boucle = n ; cpt = 0 et p = x^(n-cpt) = x^n   */
fin
```

Quelques conseils au sujet des enchaînements répétitifs

Quand, pour résoudre un problème, on se rend compte que l'on va avoir besoin d'un énoncé répétitif, on cherche à identifier les "blocs" suivants :

- Instructions d'initialisation : ce sont les instructions qui permettent d'initialiser les variables sur lesquelles on va travailler dans l'énoncé répétitif. Dans l'exemple précédent du calcul de x à la puissance n, ce sont les deux instructions qui initialisent p et cpt.
- Condition d'arrêt : c'est la condition qui doit être satisfaite pour arrêter de boucler. Dans l'exemple précédent, on doit arrêter de boucler lorsqu'on a exécuté n fois les instructions à répéter. Notons que la condition mise derrière le mot clé **tant que** est la négation de la condition d'arrêt : on doit continuer tant qu'on n'a *pas encore* exécuté n fois les instructions à répéter. On prendra l'habitude de mettre en commentaire après le **fintq** la condition d'arrêt (en vérifiant qu'il s'agit bien de la négation de la condition mise derrière **tant que**).

– Instructions à répéter : ce sont les instructions qui sont exécutées à chaque passage dans la boucle. On distinguera généralement deux sous blocs dans ce bloc d'instructions : les instructions "de traitement" et les instructions "de passage".
  – Les instructions "de traitement" sont celles à l'origine du fait que l'on écrit un enchaînement répétitif. Dans l'exemple précédent, il s'agit de l'instruction $p \leftarrow p * x$.
  – Les instructions "de passage" sont celles qui permettent de modifier les variables sur lesquelles porte la condition d'arrêt (les variables qui permettent de contrôler le nombre de passages dans la boucle). Dans l'exemple précédent, il s'agit de l'instruction $cpt \leftarrow cpt + 1$.

En général, on commence par identifier les instructions "de traitement". Ensuite, il s'agit d'écrire les instructions d'initialisation, la condition d'arrêt et les instructions de passage. Ces trois blocs dépendent les uns des autres, comme l'illustre l'exemple précédent (où l'on a proposé 3 algorithmes différents pour résoudre un même problème). Ils doivent donc être conçus en même temps. A ce moment, on se posera les questions suivantes :

– Est-on certain que la condition d'arrêt est atteinte ?
– Combien de fois les instructions à répéter sont-elles exécutées ?
  Cette information est à mettre en commentaires.
– Quelles sont les valeurs des variables lorsqu'on sort de la boucle.
  Cette information est à mettre en commentaires.

Exemple 2 : calcul du produit des n premiers entiers positifs

Il s'agit maintenant de calculer factorielle n, c'est-à-dire, $1 * 2 * 3 * \ldots * n$. On peut spécifier ce problème de la façon suivante :

Procédure : factorielle(n, f)
Entrées :
  entier n
Sorties :
  entier f
Précondition :
  $n \geq 0$
Postcondition :
  $f = n!$ où $n!$ est la fonction récursivement définie par :
  $0! = 1$
  $n! = n * (n-1)!$, $\forall n \geq 1$
  ou, autrement dit, $f = 1 * 2 * 3 * ... * n$

Pour cela, on se rend compte que l'on a besoin de faire n multiplications, mais contrairement à l'exemple précédent (puissance), le facteur multiplicatif change à chaque fois : il faut d'abord multiplier par 1, puis par 2, puis par 3, ... jusque n. On va donc d'abord écrire une boucle où une variable (par exemple i) va prendre successivement les valeurs 1, 2, 3, ... jusque n, soit :

$i \leftarrow 1$
tant que $i \leq n$ **faire**
| $i \leftarrow i + 1$
fintq
`/* Nombre de passages dans la boucle = n ;` $i = n + 1$ `*/`

On peut ensuite ajouter les instructions permettant de multiplier une variable f par les valeurs successivement prises par i. On obtient l'algorithme suivant :

```
Procédure : factorielle(n, f)
Déclarations :
    entier i
début
    f ← 1
    i ← 1
    tant que i ≤ n faire
        /* invariant : f = (i − 1)!                                    */
        f ← f ∗ i
        i ← i + 1
    fintq
    /* Nombre de passages dans la boucle = n ; i = n + 1 et p = (i − 1)! = n!   */
fin
```

Exemple d'exécution pour $n = 4$:

nb de passages dans la boucle	0	1	2	3	4
valeur de i	1	2	3	4	5
valeur de f	1	1	2	6	24

Terminaison : La terminaison de l'algorithme est assurée par le fait que la valeur de i augmente de 1 à chaque passage dans la boucle "tant que" et que l'on s'arrête quand elle devient supérieure ou égale à celle de n.

Complexité : Pour calculer f à partir de n, il faut d'abord effectuer 2 affectations, puis il faut répéter n fois la boucle "tant que". A chaque passage dans la boucle, on effectue 1 test ($i \leq n$), 1 addition, 1 multiplication, et 2 affectations. Au total, on effectuera donc $2 * n + 2$ affectations, n additions, n multiplications et n tests. Par conséquent, la complexité de factorielle est linéaire, en $\mathcal{O}(n)$.

Correction : La correction de l'algorithme peut être démontrée à l'aide de la propriété invariante. En effet, à chaque passage dans la boucle, la propriété invariante $f = (i-1)!$ est vérifiée (on vérifie facilement qu'elle est vraie au premier passage, et que si elle est vraie à un passage, alors elle est encore vraie au passage suivant). On arrête de boucler lorsque la condition $i \leq n$ n'est plus vérifiée, autrement dit lorsque $i = n + 1$. A ce moment, du fait de la propriété invariante, on sait que $fact = (i - 1)! = n!$.

Enoncé pour

Quand il y a une seule instruction d'initialisation et une seule instruction de passage *concernant une variable sur laquelle porte la condition d'arrêt*, alors on peut utiliser l'énoncé "pour" suivant :

```
pour (init ; cond ; passage) faire
    traitement
finpour
```

Cet énoncé est équivalent à l'énoncé suivant :

```
init
tant que cond faire
    traitement
    passage
fintq
```

Par exemple, les 3 algorithmes permettant de calculer x à la puissance n peuvent être écrits :

Procédure : puissance(n, x, p)
...
début
  $p \leftarrow 1$
  pour ($cpt \leftarrow 0$; $cpt < n$; $cpt \leftarrow cpt + 1$) **faire**
    /* invariant : $p = x^{cpt}$ */
    $p \leftarrow p * x$
  finpour
  /* Nombre de passages dans la boucle = n ; $cpt = n$ et $p = x^{cpt} = x^n$ */
fin

Procédure : puissance(n, x, p)
...
début
  $p \leftarrow 1$
  pour ($cpt \leftarrow 1$; $cpt \leq n$; $cpt \leftarrow cpt + 1$) **faire**
    /* invariant : $p = x^{cpt-1}$ */
    $p \leftarrow p * x$
  finpour
  /* Nombre de passages dans la boucle = n ; $cpt = n + 1$ et $p = x^{cpt-1} = x^n$ */
fin

Procédure : puissance(n, x, p)
...
début
  $p \leftarrow 1$
  pour ($cpt \leftarrow n$; $cpt > 0$; $cpt \leftarrow cpt - 1$) **faire**
    /* invariant : $p = x^{n-cpt}$ */
    $p \leftarrow p * x$
  finpour
  /* Nombre de passages dans la boucle = n ; $cpt = 0$ et $p = x^{n-cpt} = x^n$ */
fin

L'intéret d'un énoncé *pour* est de rendre plus lisible l'algorithme en regroupant sur une même ligne les instructions et la condition déterminant le nombre de passages dans une boucle : on peut, sans lire les instructions de traitement à répéter, savoir combien de fois sera exécutée la boucle... sous réserve que les instructions de traitement ne modifient pas des variables sur lesquelles porte la condition d'arrêt (dans ce cas, il est probablement plus lisible de ne pas utiliser une boucle *pour*).

Exercice 6 *Calcul de la somme des n premiers entiers*

Exercice 7 *Déterminer si un nombre est premier*

Exercice 8 *Calcul de la n-ème valeur de la suite de Fibonacci*

Exercice 9 *Calcul de $cos(x)$*

Chapitre 4

Appels de procédures et fonctions

Un algorithme est généralement élaboré par une *démarche descendante*, qui consiste à décomposer le problème en sous-problèmes, chaque sous-problème devant être de nouveau spécifié puis résolu. Cette décomposition permet d'aborder le problème progressivement en créant des niveaux de description de plus en plus détaillés. Elle permet également de réutiliser la résolution de certains sous-problèmes pour résoudre de nouveaux problèmes.

Ainsi, un algorithme peut être "appelé" dans le corps d'un autre algorithme afin de résoudre un sous-problème.

4.1 Paramètres effectifs

Lors de l'appel d'un algorithme, il faut préciser les valeurs des paramètres en entrée et, en retour, récupérer les valeurs des paramètres en sortie. Les paramètres utilisés pour cela sont appelés *paramètres effectifs*.

Exemple : Considérons le problème "somme-cos", qui consiste à calculer la somme $\cos(1) + \cos(2) + \cos(3) + \ldots + \cos(n)$.

Procédure : somme-cos(n, sc)
Entrées :
entier n
Sorties :
entier sc
Précondition :
$n \geq 1$
Postcondition :
$sc = \sum_{i=1}^{n} \cos(i)$
ou, autrement dit, $sc = \cos(1) + \cos(2) + \cos(3) + \ldots + \cos(n)$

Pour résoudre ce problème, on a successivement besoin de calculer cos(1), cos(2), ..., jusque cos(n). Or on a justement déjà résolu ce problème, qui avait été spécifié de la façon suivante au chapitre précédent :

Procédure : cos($x, eps, cosx$)
Entrées :
réel x, eps
Sorties :
réel $cosx$
Précondition :
$eps > 0$
Postcondition :
$cosx = cos(x)$, avec une précision de eps
(utilise le développement en série : $cos(x) = \sum_{i=0}^{\infty}(-1)^i \frac{x^{2i}}{(2i)!}$)

On va donc réutiliser la résolution de *cos* pour résoudre *somme-cos*. On dira que la procédure *somme-cos* *appelle* la procédure *cos*. Lors de cet appel, *somme-cos* doit préciser à *cos* les paramètres effectifs, c'est à dire :

1. la valeur de x pour laquelle on souhaite calculer cos(x),
2. la valeur de *eps*, donnant la précision avec laquelle on veut calculer cos(x),
3. la variable de *somme-cos* dans laquelle on souhaite récupérer le résultat du calcul de cos(x) (résultat contenu dans le paramètre *cosx* de *cos*).

Ces paramètres sont précisés dans l'ordre de leur déclaration. Ainsi, l'appel de "factorielle" se fera par l'instruction

cos(*expr*1,*expr*2,*var*)

où *expr*1 et *expr*2 sont deux expressions dont l'évaluation donne deux réels et *var* est une variable de type réel. L'exécution de cette instruction a pour conséquence d'affecter à *var* la valeur de *cosx* calculée par cos pour $x = expr1$ et $eps = expr2$. Le problème *somme-cos* peut alors être résolu par l'algorithme suivant :

Procédure : somme-cos(n, sc)
...
Déclarations :
réel $i, cosi$
const $eps = 0.0001$
début
  $sc \leftarrow 0$
  pour ($i \leftarrow 1$; $i \leq n$; $i \leftarrow i + 1$) **faire**
    `/* invariant :` $sc = \sum_{k=1}^{i-1} \cos(k)$ `*/`
    cos($i, eps, cosi$)
    $sc \leftarrow sc + cosi$
  finpour
  `/* Nombre de passages dans la boucle = n ;` $i = n + 1$ `et` $sc = \sum_{k=1}^{i-1} \cos(k)$ `*/`
fin

D'une façon plus générale, quand un algorithme est appelé dans un autre algorithme, on précise la liste des paramètres effectifs, en respectant l'ordre donné lors de la définition de l'algorithme appelé. Le type d'un paramètre effectif doit être le même que celui du paramètre formel correspondant. L'exécution d'un tel appel de procédure s'effectue alors en 3 étapes :

1. Affectation des valeurs des paramètres effectifs aux paramètres en entrée correspondants
2. Exécution de la procédure appelée dans un nouvel environnement
3. Affectation des valeurs des paramètres en sortie de la procédure appelée aux paramètres effectifs correspondants

Remarques.
– Le paramètre effectif correspondant à un paramètre en entrée peut être une valeur, une variable contenant une valeur ou une expression retournant une valeur. Dans tous les cas, la valeur doit être de même type que le paramètre formel correspondant.

– Le paramètre effectif correspondant à un paramètre en sortie doit toujours être une variable. De plus, s'il y a plusieurs paramètres en sortie, alors lors de l'appel de la procédure, il faudra utiliser des variables différentes comme paramètres effectifs.

Exercice 10 *Réutilisation d'algorithmes*

Exercice 11 *Appels successifs de 3 algorithmes*

Exercice 12 *Calcul de la somme des n premières factorielles.*

4.2 Paramètres en entrée et en sortie

Lors de l'appel d'un algorithme, une même variable peut être utilisée pour passer une valeur en entrée et en récupérer une autre en sortie. Considérons par exemple la suite d'instructions :

$n \leftarrow 4$
$factorielle(n, n)$

La valeur 4 de la variable n est passée comme paramètre en entrée de $factorielle$; tandis qu'à la fin de l'exécution de $factorielle$, la valeur du paramètre en sortie est affectée à n. Ainsi, à la suite de l'exécution de ces 2 instructions, la variable n aura pour valeur 24.

Il s'agit là d'une utilisation particulière de $factorielle$ qu'il n'est probablement pas intéressant de généraliser. En revanche, certains problèmes consistent systématiquement à modifier la valeur d'un paramètre, autrement dit, le paramètre doit être passé à la fois en entrée, pour connaître sa valeur de départ, et en sortie, pour modifier cette valeur. Dans ce cas, plutôt que de dupliquer ce paramètre (en le faisant apparaitre à la fois en entrée et en sortie), on dira qu'il s'agit d'un paramètre en entrée/sortie. Un paramètre en entrée/sortie sera déclaré par :

Procédure : nom-algo(var)
Entrée/Sortie :
type-de-var var

Cet algorithme pourra être appelé par :

nom-algo(x)

de telle sorte que le paramètre effectif x soit une variable de même type que la variable var. Lors de l'appel de "nom-algo(x)", la valeur du paramètre effectif x est affectée à var, tandis qu'au retour de l'appel, la valeur de var est affectée à x.

Pour spécifier le rôle d'un tel algorithme, c'est à dire la relation entre la valeur initiale du paramètre en entrée/sortie, et sa valeur finale, on a besoin de distinguer ces deux valeurs. Ainsi, on notera var^{in} la valeur du paramètre var en entrée et var^{out} sa valeur en sortie.

Exemple : Echanger les valeurs de 2 variables

Procédure : échanger(a, b)
Entrée/Sortie :
entier a, b
Postcondition :
$a^{out} = b^{in}$
$b^{out} = a^{in}$
Déclarations :
entier aux
début
$aux \leftarrow a$
$a \leftarrow b$
$b \leftarrow aux$
fin

4.3 Procédures et fonctions

On distingue deux types d'algorithmes :
– les algorithmes dont le but est de calculer une (et une seule) valeur à partir d'un certain nombre d'autres valeurs données en entrée ; ces algorithmes seront implémentés par des *fonctions*,
– les autres algorithmes, ayant un nombre quelconque de paramètres en entrée et en sortie ; ces algorithmes seront implémentés par des *procédures*.
Une fonction calcule une (et une seule) valeur à partir d'un certain nombre d'autres valeurs. Autrement dit, une fonction prend en entrée 0, 1 ou plusieurs paramètres et retourne en sortie une valeur. Par exemple, l'algorithme "factorielle" calcule $n!$ pour une valeur de n donnée ; l'algorithme "puissance" calcule x^n à partir de valeurs données pour x et n.

La déclaration d'une fonction s'effectuera de la façon suivante :

Fonction : type-fct nom-fct(< *paramètres* >)
Entrées :
liste des paramètres (types et noms)
Précondition :
Conditions sur les paramètres en entrée
Postcondition :
relation entre la valeur retournée par la fonction et ses paramètres en entrée

La valeur retournée en sortie par la fonction sera spécifiée dans le corps de la fonction par l'instruction

retourner *expr*

où *expr* est une expression (une valeur, une variable contenant une valeur, ou une opération entre expressions) de type *type-fct*.

Attention : les instructions se trouvant après une instruction de retour de fonction ne sont pas exécutées. Autrement dit, l'exécution de la fonction se termine avec la première instruction de retour trouvée.

Une fonction peut être appelée dans un autre algorithme. Un appel de fonction retourne une valeur et est donc assimilé à une valeur du type de la fonction.

Exemple 1 : reprise de factorielle.

```
Fonction : entier factorielle(n)
Entrées :
    entier n
Précondition :
    n ≥ 0
Postcondition :
    retourne n!
Déclarations :
    entier i, f
début
    f ← 1
    pour (i ← 1 ; i ≤ n ; i ← i + 1) faire
        /* invariant : f = (i − 1)!                                    */
        f ← f ∗ i
    finpour
    /* Nombre de passages = n ; i = n + 1 et f = (i − 1)! = n!         */
    retourner f
fin
```

La fonction "factorielle" peut alors être appelée dans une expression. Par exemple, après l'exécution des instructions

$x \leftarrow$ factorielle(4)
$y \leftarrow$ factorielle($x - 3 *$ factorielle(3))

la variable x a pour valeur $4! = 24$, et la variable y a pour valeur $(24 - 3 * 3!)! = 6! = 240$.

Exemple 2 : reprise de l'exercice "plus petit de trois nombres"

```
Fonction : entier plus-petit(n)
Entrées :
    entier a, b, c
Postcondition :
    retourne le plus petit nombre de l'ensemble {a, b, c}
début
    si a < b alors
        si a < c alors
            retourner a
        sinon
            retourner c
        finsi
    sinon
        si b < c alors
            retourner b
        sinon
            retourner c
        finsi
    finsi
fin
```

Sachant que les instructions suivant un **retourner** ne sont pas exécutées, on aurait tout aussi bien pu écrire :

```
Fonction : entier plus-petit(n)
Entrées :
    entier a, b, c
Postcondition :
    retourne le plus petit nombre de l'ensemble {a, b, c}
début
    si a < b alors
        si a < c alors
            retourner a
        finsi
        retourner c
    finsi
    si b < c alors
        retourner b
    finsi
    retourner c
fin
```

4.4 Modes de passage des paramètres

Lorsque l'on conçoit un algorithme, on a juste besoin de spécifier, pour chaque paramètre, s'il est donné en entrée, ou s'il est calculé en sortie, ou s'il est donné en entrée puis modifié par l'algorithme pour être ensuite retourné en sortie. Ainsi, au niveau de l'algorithme, on spécifie pour chaque paramètre son mode (entrée, sortie ou entrée/sortie). Au moment de programmer l'algorithme, il s'agit de trouver un mécanisme permettant d'implémenter le comportement correspondant au mode choisi :

- pour un paramètre en entrée : lors de l'appel de la procédure, il faut passer la valeur du paramètre effectif au paramètre formel ; pendant l'exécution de la procédure appelée, si le paramètre formel est modifié, il ne faut pas répercuter cette modification sur le paramètre effectif.
- pour un paramètre en sortie : à la fin de l'exécution de la procédure appelée, il faut transmettre la valeur du paramètre en sortie au paramètre effectif correspondant.
- pour un paramètre en entrée/sortie : lors de l'appel de la procédure, il faut passer la valeur du paramètre effectif au paramètre formel ; à la fin de l'exécution de la procédure appelée, il faut transmettre la nouvelle valeur du paramètre formel au paramètre effectif correspondant.

Pour implémenter ces comportements, on dispose (dans la plupart des langages de programmation) de deux mécanismes de passage de paramètres : le passage par valeur et le passage par référence.

Considérons par exemple une procédure q qui appelle une procédure p, et supposons que la procédure p a un seul paramètre x.

- Si x est passé par valeur :
  - lors de l'appel de p, on crée un nouvel environnement au dessus de l'environnement de q. Ce nouvel environnement contient une nouvelle variable, de nom x, et on recopie la valeur du paramètre effectif correspondant à x dans cette nouvelle variable.
  - Lors de l'exécution de p, on utilise cette nouvelle variable dont la valeur peut éventuellement être modifiée.
  - A la fin de l'exécution de p, son environnement est détruit (et la valeur du paramètre aussi). En revanche, l'environnement de la procédure q n'est pas modifié.
- Si x est passé par référence :
  - lors de l'appel de p, on ne crée pas un nouvel emplacement mémoire pour stocker x. A la place, on crée un lien (une référence !) entre le paramètre formel de nom x et le paramètre effectif correspondant, de sorte que p peut accéder au paramètre effectif correspondant à x.
  - Lors de l'exécution de p, les modifications faites sur x sont directement effectuées dans l'environnement de q sur le paramètre effectif correspondant à x.
  - A la fin de l'exécution de p, on détruit le lien entre x et le paramètre effectif de q correspondant, mais les modifications faites par p sur ce paramètre effectif restent.

Ainsi, lors de l'implémentation d'un algorithme dans un langage de programmation,
– un paramètre en sortie ou en entrée/sortie sera nécessairement passé par référence ;
– un paramètre en entrée sera généralement passé par valeur.
Cependant, lorsque le paramètre en entrée est codé sur un grand nombre d'octets (ce sera le cas des tableaux par exemple), on ne le passera pas par valeur mais par référence afin d'éviter d'avoir à recopier sa valeur, ce qui peut être long et couteux en place mémoire.

Chapitre 5

Les tableaux

Un tableau est une suite de n données de même type, rangées consécutivement. On dira que n est la *taille* du tableau et que les données sont ses *éléments*. Chaque élément est repéré dans le tableau par son *indice*. Un indice correspond à un numéro d'ordre dans le tableau, les éléments étant rangés par ordre d'indices consécutifs croissants.

Déclaration d'un tableau : une variable de type tableau est déclarée selon la syntaxe suivante

```
type-elem nom-tableau[d..f]
```

où `type-elem` est le type des éléments du tableau, `nom-tableau` est le nom du tableau, et `d` et `f` représentent respectivement les indices du premier et du dernier élément du tableau. Si $d > f$ alors le tableau est vide (il n'a aucun élément), sinon il a `f-d+1` éléments de type `type-elem`.

Par exemple, à la suite des déclarations suivantes :

```
Déclarations :
    entier tab1[1..10]
    réel tab2[12..44]
```

tab1 est un tableau de 10 entiers indicés de 1 à 10 et *tab2* est un tableau de 33 réels indicés de 12 à 44.

Remarque : quand on déclare un tableau en C, on ne donne que le nombre d'éléments du tableau, l'indice du premier élément du tableau étant toujours 0. Ainsi, l'instruction C suivante

```
int tab[10];
```

déclare un tableau `tab` de 10 entiers, dont le premier élément est à l'indice 0 et le dernier à l'indice 9, ce qui correspond à la déclaration algorithmique suivante

```
Déclarations :
    entier tab[0..9]
```

Accès à un élément du tableau : on accède à un élément dans un tableau à partir de son indice, la valeur de cet indice devant être comprise entre les indices de début et de fin du tableau. Par exemple, *tab1[8]* désigne l'élément d'indice 8 dans le tableau *tab1*, autrement dit le 8ème élément du tableau ; tandis que *tab2[14]* désigne l'élément d'indice 14 dans le tableau *tab2*, autrement dit le 3ème élément du tableau.

Précondition implicite aux algorithmes ayant des tableaux passés en paramètres : quand un tableau est déclaré en paramètre formel d'une procédure, les indices du premier et du dernier élément du tableau ne seront pas toujours précisés dans le type mais passés en paramètre. Ainsi, dans les exercices suivants, on déclarera souvent un paramètre de type tableau de la façon suivante :

```
Procédure : nom-algo(tab, d, f)
Entrées :
    Telt tab[?..?]
    entier d, f
```

Dans ce cas, on supposera en précondition que le tableau physique passé en paramêtre effectif à la place de *tab* sera défini pour les indices compris entre d et f. Notons que le tableau passé en paramètre effectif pourra avoir été défini pour un intervalle d'indices $[d_e..f_e]$ supérieur (tel que $d - e \leq d$ et $f \leq f_e$). Dans ce cas, la procédure appelée ne travaillera que sur une partie du tableau passé en paramètre (la partie entre d et f).

Parcours des éléments d'un tableau : dans de nombreux algorithmes, on doit parcourir le tableau pour faire un traitement sur chaque élément du tableau. L'algorithme générique pour le parcours d'un tableau *tab* indicé de d à f est le suivant :

```
Déclarations :
    entier i
début
    i ← d
    tant que i ≤ f faire
        /* invariant : les éléments d'indice j < i ont déjà été traités      */
        /* Traitement de l'élément d'indice i                                */
        i ← i + 1
    fintq
    /* Nombre de passages = f − d + 1 ; i = f + 1                            */
fin
```

Cet algorithme peut être exprimé de façon équivalente de la façon suivante :

```
Déclarations :
    entier i
début
    pour (i ← d ; i ≤ f ; i ← i + 1) faire
        /* invariant : les éléments d'indice j < i ont déjà été traités      */
        /* Traitement de l'élément d'indice i                                */
    finpour
    /* Nombre de passages = f − d + 1 ; i = f + 1                            */
fin
```

Exercice 13 *Calcul de la somme des éléments d'un tableau*

Exercice 14 *Calcul de la moyenne des éléments d'un tableau*

Exercice 15 *Calcul des moyennes par groupe pour un tableau contenant toutes les notes d'une promotion d'étudiants*

Exercice 16 *Compter le nombre d'occurrences d'un élément dans un tableau*

Exercice 17 *Déterminer si un tableau est un palindrome*

Exercice 18 *Calcul des n premières valeurs de la suite de Fibonacci*

Exercice 19 *Inversion des éléments d'un tableau*

Exercice 20 *Crible d'Eratosthème*

Chapitre 6

Etude de quelques algorithmes sur les tableaux

Les tableaux ont été introduits au chapitre précédant. On étudie maintenant un certain nombre d'algorithmes "classiques" sur les tableaux.

Comme au chapitre 5, ces algorithmes impliqueront souvent de parcourir le tableau pour faire un traitement sur chaque élément du tableau. Cependant, dans certain cas on ne devra pas parcourir le tableau jusqu'à son dernier élément, mais jusqu'à trouver un élément satisfaisant une certaine condition. On peut alors écrire

```
Déclarations :
    entier i
début
    i ← d
    tant que tab[i] ne satisfait pas la condition d'arrêt faire
        /* Les éléments d'indice j < i ont déjà été traités                    */
        /* Traitement de l'élément d'indice i                                  */
        i ← i + 1
    fintq
    /* Nombre de passages ≤ f − d + 1 ; i ≤ f                                  */
fin
```

Cependant, il peut arriver qu'aucun élément du tableau ne satisfasse la condition d'arrêt (par exemple, on cherche un élément de valeur donnée... et aucun élément du tableau n'a cette valeur). Dans ce cas, l'algorithme précédent va provoquer une erreur à l'exécution (une erreur de segmentation en C). En effet, quand $i = f$, la condition d'arrêt n'étant pas satisfaite par $tab[f]$, on entre dans la boucle et on incrémente i. A l'itération suivante, $i = f + 1$, et une erreur survient quand on accède à $tab[i]$. En C, il peut même arriver que le problème ne soit pas détecté, et que l'ordinateur accède aux octets suivant le dernier élément du tableau, en croyant qu'il s'agit encore d'un élément du tableau. On peut alors avoir des résultats d'exécution particulièrement déroutants, où des variables sont modifiées sans que l'on ne comprenne pourquoi !

Ainsi, quand on parcourt un tableau, on s'assure toujours que l'on ne peut pas sortir des bornes du tableau ; s'il est possible que la condition d'arrêt ne soit satisfaite par aucun élément, on ajoute un test pour vérifier que l'on n'est pas sorti des bornes du tableau. Ce test devra être fait AVANT d'accéder à l'élément :

```
Déclarations :
    entier i
début
    i ← d
    tant que i ≤ f et tab[i] ne satisfait pas la condition d'arrêt faire
        /* Les éléments d'indice j < i ont déjà été traités                    */
        /* Traitement de l'élément d'indice i                                  */
        i ← i + 1
    fintq
    /* Nombre de passages ≤ f − d + 1 ; i > f ou tab[i] satisfait la condition d'arrêt
       */
fin
```

Exercice 21 *Recherche de l'indice du plus petit élément d'un tableau*

Exercice 22 *Recherche séquentielle de l'indice de la première occurrence d'un élément dans un tableau*

Exercice 23 *Recherche dichotomique de l'indice d'une occurrence (quelconque) d'un élément dans un tableau trié*

Exercice 24 *Insertion d'un élément dans un tableau trié*

Exercice 25 *Suppression d'un élément à un indice donné*

Exercice 26 *Suppression de tous les doublons d'un tableau trié*

Exercice 27 *Interclassement de 2 tableaux*

Chapitre 7

La récursivité

La récursivité peut se définir comme la résolution d'un problème à partir de versions plus simples de lui-même. De fait, de nombreux problèmes se définissent tout naturellement de façon récursive. Par exemple, la fonction factorielle se définit récursivement par :

1. définition de factorielle pour un cas élémentaire (règle de base)

$$0! = 1$$

2. définition de factorielle pour un cas non élémentaire, en fonction de la définition de factorielle pour un cas plus simple (règle récursive)

$$n! = n * (n-1)! \quad \text{pour} \quad n \geq 1$$

De même, la suite de Fibonacci se définit récursivement par les 3 règles suivantes :

1. première règle de base
$$fibo(0) = 1$$
2. deuxième règle de base
$$fibo(1) = 1$$
3. règle récursive
$$fibo(n) = fibo(n-1) + fibo(n-2) \quad \text{pour} \quad n \geq 2$$

La terminaison de ce genre de définition est garantie par le fait qu'un "critère" (en l'occurrence la valeur de n) est modifié d'un appel récursif à l'autre et converge (en un nombre d'étapes finies) vers un des cas de base.

D'une façon similaire, un algorithme peut résoudre un problème de façon récursive. Dans ce cas, on procèdera toujours selon les deux étapes suivantes :

1. résolution du problème dans les cas élémentaires (les cas de base) ;
2. résolution du problème pour les cas non élémentaires en faisant appel à la résolution du problème pour des cas plus simples.

Pour s'assurer de la terminaison de ce genre d'algorithme, il faut vérifier que d'un appel récursif à l'autre, les valeurs d'un ou plusieurs paramètres changent de telle sorte que l'on converge, en un nombre d'appels fini, vers un cas élémentaire.

Considérons par exemple l'algorithme de calcul récursif d'une factorielle.

```
Fonction : entier fact_rec(n)
Entrées :
    entier n
Précondition :
    n ≥ 0
Postcondition :
    retourne n!
début
    si n = 0 alors
        /* résolution pour le cas de base n = 0                    */
        retourner 1
    sinon
        /* résolution récursive                                    */
        retourner (n*fact_rec(n − 1))
    finsi
fin
```

Cet algorithme termine (sous la précondition que n soit effectivement positif ou nul au départ, autrement, ça boucle!!!). En effet, à chaque appel récursif la valeur de n est décrémentée de 1 de telle sorte que n converge (au bout de n appels successifs) vers le cas de base $n = 0$.

L'énoncé de cet algorithme est plus "élégant" que sa version répétitive, notamment parce qu'il est plus concis et qu'il reproduit la définition récursive usuelle de la fonction mathématique factorielle.

La complexité théorique de cet algorithme est la même que celle de sa version répétitive : en effet, pour calculer la factorielle de n, on effectuera n appels successifs à fact_rec, chaque appel se résumant à une affectation pour le passage du paramètre en entrée, un test $(n = 0)$, une multiplication et une affectation pour le passage du paramètre en sortie. Au total, on effectuera donc $2 * n$ affectations, n tests et n multiplications. Par conséquent, la complexité de fact_rec est linéaire en $\mathcal{O}(n)$.

En pratique, chaque appel récursif nécessite d'empiler un nouvel environnement au moment de l'exécution, cet environnement étant ensuite dépilé au retour de l'appel. Ces empilements et dépilements successifs prennent un peu de temps et de place mémoire. Par conséquent, un algorithme récursif pourra être très légèrement moins efficace qu'un algorithme itératif effectuant le même traitement. Cependant, de nombreux compilateurs peuvent optimiser un code récursif pour éviter ces empilements et dépilements d'environnements (on parle de "dérécursification"). Dans ce cas, le code exécutable optimisé obtenu à partir d'un algorithme récursif est aussi efficace que le code exécutable obtenu à partir d'un algorithme itératif.

Exercice 28 *Calcul récursif de la n-ième valeur de la suite de Fibonacci*

Exercice 29 *Affichage de caractères saisis au clavier dans l'ordre inverse de leur saisie*

Exercice 30 *Calcul récursif de* x^y

Exercice 31 *Palindrome récursif*

Exercice 32 *Recherche dichotomique récursive*

Exercice 33 *Afficher l'ensemble des nombres de* n *chiffres ne comportant que des* 1 *et des* 2. *Par exemple, l'ensemble des nombres de 3 chiffres ne comportant que des 1 et des 2 est*

$$\{222, 221, 212, 211, 122, 121, 112, 111\}$$

Chapitre 8

Etude de quelques tris

Les méthodes de tri sont très importantes en pratique et interviennent dans de nombreux problèmes. Le tri est également un bon exemple de problème pour lequel de nombreux algorithmes existent.

Spécification du problème de tri : on dispose en entrée d'une suite (une liste, une séquence, ...) de n éléments comparables ; le résultat en sortie est une suite dont les éléments sont une permutation des éléments de la suite donnée en entrée et telle que les éléments se succèdent par ordre croissant.

On étudie ici les tris sur les tableaux (la suite d'éléments à trier est rangée dans un tableau) ; on étudiera ultérieurement des tris sur des listes chaînées et des fichiers.

On supposera ici que les éléments du tableau à trier sont d'un type "Telt" inconnu pour lequel on dispose des opérations de comparaison $<, >, =, \leq, \geq$ usuelles. On dira que l'algorithme est *générique*, c'est-à-dire qu'il est *paramétré* sur le type des éléments du tableau et la fonction de comparaison.

La spécification d'une procédure de tri est la suivante :

Procédure : tri(tab, d, f)
Entrée/Sortie :
Telt $tab[?..?]$
Entrées :
entier d, f
Postcondition :
1)- Le tableau en sortie est une permutation du tableau en entrée.
2)- Les éléments du tableau en sortie sont triés par ordre croissant, i.e.,
$\forall i \in [d..f-1], tab^{out}[i] \leq tab^{out}[i+1]$

Exercice 34 *Tri par sélection*

On trie les éléments d'un tableau en sélectionnant un par un les éléments du tableau, du plus petit au plus grand :

1. on recherche le plus petit élément du tableau, et on l'échange avec le premier élément du tableau, puis
2. on recherche le plus petit élément du sous-tableau commencant au deuxième indice, et on l'échange avec le deuxième élément du tableau, puis
3. on recherche le plus petit élément du sous-tableau commencant au troisième indice, et on l'échange avec le troisième élément du tableau,
4. ... etc ...

D'une façon plus générale, on répète les 3 opérations suivantes :

1. chercher l'indice *ipp* du plus petit élément du sous-tableau commençant à l'indice i,
2. échanger l'élément d'indice *ipp* avec l'élément d'indice i du tableau,

3. incrémenter i

Au départ, i est initialisé à d; on arrête le processus lorsque i est égal à f.

Pour chercher l'indice du plus petit élément d'un sous-tableau, on peut utiliser la fonction "indice_plus_petit".

Exercice 35 *Tri par insertion*

Le tri par insertion procède de la même façon qu'un joueur de carte pour trier ses cartes, en insérant successivement chaque élément du tableau dans un sous-tableau déjà trié : à la i^{eme} étape, les $i-1$ premiers éléments du tableau sont déjà triés et on insère le i^{eme} élément du tableau dans ce sous-tableau trié.

Pour insérer le i^{eme} élément dans le sous-tableau trié, on peut utiliser la procédure "insere_trie".

Exercice 36 *Tri rapide ou quicksort*

Le tri rapide est un exemple de tri par dichotomie. L'idée est de récursivement

1. partitionner le tableau à trier en deux sous-tableaux tels que tous les éléments du premier sous-tableau soient inférieurs à tous les élements du second tableau,
2. recommencer ce processus sur chacun des sous-tableaux

jusqu'à ce que les sous-tableaux à trier ne contiennent plus qu'un seul élément.

Chapitre 9

Les structures

Le type `structure` désigne une suite comportant un nombre déterminé de données de types différents, chaque composant de la structure étant appelé *champ*.

Déclaration d'une structure : une variable de type structure est déclarée selon la syntaxe suivante

struct
  T_1 $champ_1$
  T_2 $champ_2$
  ...
  T_n $champ_n$
fstruct *nom_var*

où n est le nombre de champs de la variable *nom_var*. Pour chacun des champs, on précise son type T_i et son nom *champ_i*. Par exemple, la variable *date* déclarée de la façon suivante :

struct
  entier *jour*
  texte(10) *mois*
  entier *annee*
fstruct *date*

est une structure composée de 3 champs : un premier champ de nom *jour* et de type entier, un deuxième champ de nom *mois* et de type texte et un troisième champ de nom *annee* et de type entier.

Déclaration d'un type structure : Quand plusieurs variables sont d'un même type structure, il est vivement recommandé de définir un nouveau type, puis de déclarer les variables comme appartenant à ce type.

Par exemple, on peut déclarer les types Tdate et Tpersonne de la façon suivante :

```
type struct
  entier jour
  texte(10) mois
  entier annee
fstruct Tdate
type struct
  texte(20) nom
  texte(20) prenom
  car sexe
  Tdate date_naissance
fstruct Tpersonne
```

et ensuite déclarer les variables *toto* et *durand* comme étant de type Tpersonne :

```
Déclarations : Tpersonne toto, durand
```

Accès à un champ d'une structure : on accède à un champ d'une structure en faisant suivre le nom de la structure d'un point '.' et du nom du champ. Ainsi, *toto.nom* désigne le champ *nom* de la variable structurée *toto*, tandis que *durand.date_naissance.annee* désigne le champ *annee* du champ *date_naissance* de la variable structurée *durand*.

Exercice 37 *Comparaison de dates*

Exercice 38 *Etant données une date d, une adresse a et une personne p, déterminer si p a plus de 18 ans à la date d, et si elle habite la même ville que a.*

Exercice 39 *Tri indirect d'un tableau de personnes, par age croissant*