AP2 : Algorithmique et programmation 2
Université de Lorraine, site de Nancy,
Licence première année, S2 : M-I et I-SPI

responsable de l'UE : Jean Lieber Dernière version : 7 décembre 2018

Sommaire

1 Introduction **3**

2 Les fonctions et les procédures **4**
2.1 Les fonctions en mathématiques (rappel) 4
2.2 Les fonctions en informatique 4
2.3 Les procédures 6
2.4 Les fonctions et procédures en C 8
2.5 Passage de paramêtres par valeur, passage de paramêtres par référence 11
2.5.1 Passage par valeur ou par référence en C 12

3 La récursivité **13**
3.1 Rappel : suite définie par une relation de récurrence 13
3.2 Les fonctions récursives 14

4 Les entiers naturels revisités et la démarche algorithmique d'AP2 **15**
4.1 Le type abstrait `entier_nat` 15
4.2 Algorithmique et programmation d'opérations non primitives sur les entiers naturels 16
4.2.1 Profil de `égaux` 16
4.2.2 Traitement d'exemples de calculs de `égaux`(e_1, e_2) 17
4.2.3 Jeu d'axiomes pour `égaux` 17
4.2.4 Algorithme récursif pour `égaux` 18
4.2.5 Algorithme itératif pour `égaux` 18
4.2.6 Programme récursif pour `égaux` 20
4.2.7 Programme itératif pour `égaux` 20
4.2.8 Test des programmes récursif et itératif implantant `égaux` 21
4.2.9 Autres étapes 21
4.3 Exercice 21

5 Les enregistrements (compléments au cours éponyme d'AP1) **22**
5.1 Type abstrait `complexe` 22
5.2 Manipulation algorithmique d'enregistrements 23
5.3 Implantation des enregistrements en C 24
5.4 Le principe d'encapsulation appliqué aux enregistrements 25

6 Les listes **26**
6.1 Les listes et les tableaux 26
6.2 Type abstrait `liste` 26
6.3 Implantation des listes par des enregistrements : les listes chaînées 27
6.3.1 Principe des listes chaînées 27
6.3.2 Les listes chaînées en C 27
6.4 Exercice 28
6.5 Opérations non destructrices et opérations destructrices 29

7 Piles, files et autres structures linéaires **30**
7.1 Les piles 30
7.1.1 Type abstrait `pile` 30
7.1.2 Implantation 30
7.1.3 Applications 31
7.1.4 Exercices 31

7.2 Les files . . . 31
7.2.1 Type abstrait `file` . . . 31
7.2.2 Implantation . . . 31
7.2.3 Applications . . . 32
7.2.4 Exercice . . . 32
7.3 Les ensembles finis . . . 32
7.3.1 Type abstrait `ensemble` . . . 32
7.3.2 Parcours d'un ensemble . . . 33
7.3.3 Implantations . . . 34
7.3.4 Exercice . . . 34
7.4 Les multiensembles finis . . . 34
7.5 Les listes circulaires . . . 34
7.6 Les listes bidirectionnelles . . . 35
7.7 Les listes hétérogènes . . . 35

8 Les tables 35
8.1 Notion de table . . . 35
8.2 Type abstrait . . . 35
8.3 Implantation par une liste d'association (« en vrac ») . . . 36
8.4 Implantation par un tableau trié . . . 36
8.5 Implantation par une table de hachage (*hash table*) . . . 37
8.6 Autres implantations . . . 37
8.7 Quelle implantation choisir ? . . . 37

9 Calcul numérique sur les réels et les flottants 37
9.1 Les flottants et les réels . . . 37
9.2 Les flottants en C . . . 37
9.3 Exercices . . . 38

10 Conclusion 39

A Algorithme et C : aide-mémoire 39
A.1 Les commentaires . . . 39
A.2 Les variables . . . 39
A.3 Les constantes . . . 40
A.4 Les opérations de base sur les booléens, les entiers et les réels . . . 40
A.5 La génération pseudo-aléatoire de nombres . . . 41
A.6 Les entrées/sorties . . . 41
A.7 Les conditionnelles . . . 41
A.8 Les boucles . . . 42
A.9 Les fonctions et les procédures . . . 42
A.9.1 Les fonctions et procédures avec passage de paramètre uniquement par valeur . . . 42
A.9.2 Les fonctions et procédures avec passage de paramètre par valeur ou par référence . . . 43
A.10 Les enregistrements . . . 43
A.10.1 Manipulation directe des enregistrements (à éviter) . . . 43
A.10.2 Manipulation des enregistrements par fonctions de lecture et d'écriture . . . 43
A.11 Les tableaux . . . 44
A.11.1 Les tableaux à une entrée . . . 44
A.11.2 Les tableaux à plusieurs entrées . . . 44

Statut de ce document

Ce document (accessible sous `homepages.loria.fr/JLieber/cours/ap2`) constitue des notes sur le cours d'AP2 (algorithmique et programmation 2) donné sur le site de Nancy pour le deuxième semestre de la première année des licences informatique et mathématiques.

Important : Ces notes de cours ne se substituent nullement au cours. Elles sont là pour indiquer certains points enseignés (mais pas tous !) et donner des énoncés d'exercices.

Informations pratiques

Les étudiants ont 60 heures de cours en présentiel et une quantité équivalente de préparation, que ce soit pour les enseignements intégrés (EI) ou pour les travaux pratiques (TP). Il y 40 heures d'EI et 20 heures de TP.

Les intervenants de ce cours font l'hypothèse suivante : les étudiants travaillent durant les séances et entre les séances de façon assidue et régulière. Ce travail est constitué de beaucoup d'exercices (il y aura peu d'apprentissage « par cœur ») et constituera essentiellement l'acquisition d'un savoir-faire. Les cours (cours intégrés et TP) sont obligatoires. Toute absence doit être justifiée auprès de l'administration (justificatif médical, etc.).

Le langage support de ce cours est le C, même si ce cours ne se veut pas un cours de C (seules les notions utiles de C seront utilisées dans ce cours). Il sera parfois fait allusion au langage Python, utilisé en AP1, et cela afin de mieux faire le lien entre les deux langages : beaucoup de notions sont indépendantes du langage de programmation choisi.

1 Introduction

Les ***prérequis*** de ce cours sont les notions suivantes (enseignées en AP1) :

- Notions de spécification (informelle), d'algorithme, de programme ;
- Notion de type et de variable (une variable est donnée par un nom — ou identifieur — et un type τ et a, à un instant donné, une valeur dont le type est τ) ;
- Déclaration d'une variable, affectation d'une variable (en particulier, initialisation d'une variable) ;
- Types simples (**booléen**, **caractère**, **entier naturel**, **entier**, **réel**) ;
- Types tableaux : déclaration d'une variable (p. ex., T : tableau de réel[10]) et notation $T[i]$;
- Type chaîne de caractères [1] ;
- Types enregistrements : déclaration du type (par ses champs et leurs types) et déclaration d'une variable de ce type ;
- Conditionnelles

  Si `<condition>` **Alors**
    | `<instruction(s) alors>`
  Sinon
    | `<instruction(s) sinon>`
  Finsi

  `<condition>` est une expression à valeur booléenne. `<instruction(s) alors>` est une séquence d'instructions (au moins une instruction). `<instruction(s) sinon>` est une séquence d'instructions éventuellement vide (et dans ce cas on enlève le mot-clef **Sinon**).

  Il existe des variantes, mais elles seront peu utilisées en AP2.
- Boucles « tant que »

  Tant que `<condition>` **Faire**
    | `<instruction(s)>`
  Fintantque

  `<condition>` est une expression à valeur booléenne.

  `<instruction(s)>` est une séquence d'instructions contenant dans la très grande majorité des cas au moins une instruction.
- Boucles « pour »

  Pour `<variable de type entier>` **allant de** `<valeur initiale>` **à** `<valeur finale>` **Faire**
    | `<instruction(s)>`
  Finpour

  `<valeur initiale>` et `<valeur finale>` sont deux expressions à valeurs entières telle que la première est inférieure ou égale à la seconde (dans le cas inverse, il faut rajouter *en descendant* Après la valeur finale).
  `<instruction(s)>` est une séquence d'instructions contenant dans la très grande majorité des cas au moins une instruction.
  Il existe des variantes, mais elles seront peu utilisées en AP2.
- Implantation de toutes ces notions en Python 3.

Connaître ces notions signifie savoir les appliquer de façon pertinente.

Le langage de programmation utilisé pour AP2 est C.

Les notions enseignées dans ce cours sont les suivantes :

1. Une chaîne de caractères peut être, en première approximation, assimilée à un tableau de caractères. Ainsi, si `ch` est la chaîne de caractères `"travail régulier"`, ch[0] = `'t'`, ch[1] = `'r'`, ch[7] = `'␣'`, où `'␣'` est le caractère espace (parfois noté simplement `' '`). En revanche, l'usage des chaînes de caractères peut différer de celui des tableaux, selon le langage de programmation utilisé. Ainsi, en Python, une chaîne de caractères est non modifiable (on ne peut pas faire l'affectation `ch[0] = 'T'`). En C, une chaîne de caractères est bien assimilable à un tableau de `char` (type des caractères en C), mais pour représenter la chaîne de caractères `"abc"` on a besoin d'un tableau d'au moins 4 caractères : `'a'`, `'b'`, `'c'` et `'\0'`, ce dernier étant un caractère spécial signifiant dans ce contexte « fin de la chaîne ».

— Les fonctions et les procédures ;
— La récursivité ;
— Les enregistrements (suite du cours d'AP1 sur ce thème) ;
— Le type des listes (e.g., $(a\ b\ c\ d\ e)$) ;
— Les types des ensembles finis, des piles, des files et autres types similaires ;
— Les tables.

Si le temps le permet, des notions supplémentaires seront abordées (notamment, la structure d'arbre).

2 Les fonctions et les procédures

2.1 Les fonctions en mathématiques (rappel)

Soit A et B, deux ensembles. Une fonction f de A dans B est une relation sur $A \times B$ telle que : pour tout $x \in A$, il existe exactement un $y \in B$ tels que x soit relié à y. On note alors $y = f(x)$. Par exemple la fonction sinus est une fonction de $\mathbb{R}$ dans $\mathbb{R}$ et la multiplication des entiers naturels est une fonction de $\mathbb{N} \times \mathbb{N}$ dans $\mathbb{N}$.

Considérons la *définition de fonction* suivante : $\boxed{\begin{array}{llll} f: & \mathbb{R} & \longrightarrow & \mathbb{R} \\ & x & \longmapsto & x^2 - 1 \end{array}}$. Autrement écrit : f est une fonction de $\mathbb{R}$ dans $\mathbb{R}$ qui à $x \in \mathbb{R}$ associe $x^2 - 1$. Cette fonction f peut être définie autrement, par exemple : $\boxed{\begin{array}{llll} f: & \mathbb{R} & \longrightarrow & \mathbb{R} \\ & x & \longmapsto & (x-1)(x+1) \end{array}}$.

Ces deux définitions encadrées correspondent à la *même* fonction f au sens mathématique. Plus généralement, pour toute fonction f il existe une infinité de définitions de cette fonction.

2.2 Les fonctions en informatique

En algorithmique et dans la plupart des langages de programmation, on peut écrire des définitions de fonctions. Cependant, par abus de langage, on utilisera généralement le terme « fonction » à la place de définition de fonctions et on aura alors parfois plusieurs « fonctions » réalisant la même spécification.

définition d'une fonction. Commençons par un exemple simple. Soit la fonction définie (en mathématiques) de la façon suivante :

$$\begin{array}{llll} f: & \mathbb{R} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} & \longrightarrow & \mathbb{R} \\ & (x, a, b, c) & \longmapsto & ax^2 + bx + c \end{array}$$

Une telle définition peut s'écrire dans un algorithme de la façon suivante :

```
(...) /* Début de l'algorithme */
Fonction f (x : réel, a : entier naturel, b : entier naturel, c : entier naturel) : réel
Début
    retourner a * x * x + b * x + c
Fin
(...) /* Fin de l'algorithme */
```

La première ligne de la (définition de la) fonction s'appelle la ***signature***[1]. Elle donne le nom de la fonction, la ***liste des paramètres*** et le type du résultat. Sur l'exemple :

— La signature est « **Fonction** f (x : `réel`, a : `entier naturel`, b : `entier naturel`, c : `entier naturel`) : `réel` ».
— La liste des paramètres est « x : `réel`, a : `entier naturel`, b : `entier naturel`, c : `entier naturel`) » : chaque élément de cette liste est un nom de variable, appelé paramètre de la fonction, suivi du symbole « : » et terminé par le type de cette variable.
— Le type du résultat est « `réel` ».

Le corps de la fonction est délimité par les mots-clefs **Début** et **Fin** et contient un bloc d'instructions. L'instruction « **retourner** e » (dans l'exemple, « **retourner** $a*x*x+b*x+c$ ») doit être telle que e est une expression dont le type est le type du résultat. L'effet de cette instruction est d'évaluer l'expression e en une valeur v, d'arrêter l'exécution de la fonction (toute instruction dans le même bloc qui la suit est donc inutile) et de « donner » à l'appelant de la fonction la valeur v.

Le corps de la fonction peut contenir plusieurs instructions, comme dans l'exemple suivant :

1. C'est le terme que nous utiliserons dans ce cours. Certains l'appellent l'*en-tête* de la fonction, d'autres, le *prototype* de la fonction.

```
/* Fonction testant si un entier naturel est premier */
Fonction est_premier (n : entier naturel) : booléen
Variables
  | d : entier naturel
Début
  | d ← 2
  | Tant que n mod d ≠ 0 et d < n Faire
  |   | d ← d + 1
  | Fintantque
  | retourner d = n
Fin
```

On notera le commentaire destiné à expliquer ce que fait la fonction (dans ce cours, les commentaires sont délimités par **/*** et ***/**).

On notera l'usage de variables déclarées entre la signature de la fonction et le début. Elles s'appellent ***variables locales*** de la fonction. Par opposition, les variables définies en-dehors du cadre d'une fonction sont appelées ***variables globales***.

Exercice 1 *écrire une fonction appelée* `somme_0_à_n` *qui calcule* $\sum_{i=0}^{n} i$*, où* n *est le paramètre de la fonction (un entier naturel).*

Comment modifier cette fonction en une fonction calculant $\prod_{i=1}^{n}(2i+1)$ *?*

Exercice 2 *écrire une fonction qui à un entier naturel* n *et à un tableau* T *de* n *réels associe la somme de ses éléments :* $\sum_{i=0}^{n-1} T[i]$.

Comment modifier cette fonction pour avoir le produit des éléments de T *?*
Comment modifier cette fonction pour avoir le maximum des éléments de T *?*
Comment modifier cette fonction pour avoir le minimum des éléments de T *?*

Utilisation d'une fonction. Si une fonction a été définie dans un algorithme, elle peut être utilisée dans le reste de l'algorithme. Elle peut même être utilisée par d'autres fonctions.

Pour utiliser une fonction dont la signature est « **Fonction** f $(x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n) : \tau$ » ($\tau_1, \tau_2, \ldots, \tau_n$ et τ sont des types) on écrit $f(e_1, e_2, \ldots, e_n)$ qui est une expression de type τ qui est syntaxiquement valide à condition que pour tout i, l'expression e_i prenne une valeur de type τ_i. Par exemple, on peut ainsi définir la fonction calculant le volume d'un cylindre de révolution en définissant tout d'abord la fonction calculant l'aire d'un disque[2] :

```
Fonction aire_disque (r : réel) : réel
Début
  | retourner π * r²
Fin

Fonction volume_cylindre_révolution (rayon_section : réel, hauteur : réel) : réel
Début
  | retourner aire_disque(rayon_section) * hauteur
Fin
```

Supposons à présent qu'on ait dans le même algorithme une variable v de type `réel` et l'instruction suivante :

```
| v ← volume_cylindre_révolution(4., 12.)
```

Le fonctionnement de cette instruction est le suivant :

— On fait appel à la fonction `volume_cylindre_révolution` avec la liste de paramètres $(4., 2.)$ ce qui revient à considérer qu'on effectue les affectations de variables suivantes :

2. Pour rappel, le volume d'un cyclindre dont la section a une aire de A et dont la hauteur est h est $V = Ah$. L'aire d'un disque de rayon r est πr^2.

$rayon_section \leftarrow 4.$
$hauteur \leftarrow 2.$

et qu'on exécute ensuite l'instruction dans le corps de la fonction `volume_cylindre_révolution`.

— Cette instruction fait appel à la fonction `aire_disque` avec le paramètre $4.$ (puisqu'à ce moment, la valeur de $rayon_section$ est 4.). Cela signifie qu'on effectue l'affectation de variable suivante :

$r \leftarrow 4.$

avant d'effectuer l'instruction du corps de cette fonction.

— Cette instruction consiste à calculer $\pi * r^2$ avec $r = 4.$ (i.e., 16π, qui est remplacée par la valeur approchée 50.265482) et à « retourner » cette valeur, i.e., à quitter la fonction et à remplacer dans l'instruction de la fonction appelante `volume_cylindre_révolution` l'expression `aire_disque`($rayon_section$) par 50.265482. Cela fait que l'instruction de `volume_cylindre_révolution` peut à présent se lire :

retourner $50.265482 * 2.$

puisque $hauteur$ a pour valeur 2.. La valeur à retourner est donc 100.53096.

— En fin de compte, l'affectation de v revient à faire :

$v \leftarrow 100.53096$

Exercice 3 *Ce qui précède (la définition des fonctions* `aire_disque` *et* `volume_cylindre_révolution` *ainsi que l'affectation de la variable* v*) aurait pu s'écrire de façon plus simple :*

$$v \leftarrow (\pi * 4.^2) * 2.$$

Donnez (au moins) un avantage de cette façon de faire.
Donnez (au moins) deux avantages de la première façon de faire (qui utilise la définition de deux fonctions).

écriture modulaire d'un algorithme. Un algorithme écrit de façon modulaire est constitué de nombreuses unités, appelées modules, chaque module étant de « petite taille ». Dans le cadre de ce cours, les modules considérés sont les fonctions et les procédures, mais il existe d'autres types de modules.

Un algorithme consistant en une seule suite d'instructions n'est donc pas modulaire (en AP1, comme la notion de fonction n'était pas introduite, il n'était pas possible de programmer de façon modulaire).

Parmi les avantages de l'écriture modulaire des algorithmes et des programmes, notons les suivants :

Décomposition du problème Un problème d'algorithmique complexe peut souvent se décomposer en plusieurs problèmes, chacun de ces problèmes consistant à réaliser un module.

Lisibilité Un algorithme écrit de façon modulaire est généralement plus lisible qu'un algorithme non modulaire, d'une part parce qu'il permet de décomposer la lecture en parties ayant des objectifs bien identifiés (surtout si on met des commentaires), d'autre part parce que les noms de modules (p. ex., les noms de fonctions) aident à cette lisibilité.

Réutilisabilité Si, dans un algorithme, on a écrit une fonction pour calculer, par exemple, le périmètre d'une ellipse (ce qui n'est pas trivial), cette fonction peut être réutilisée pour un autre algorithme.

Travail en groupe Le développement de programmes se fait souvent en groupe, chaque développeur étant responsable d'un ou plusieurs modules dont la spécification a été définie précédemment.

Les effets de bord. Le corps d'une fonction contient une ou plusieurs instructions. Parmi ces instructions, il y en a qui ont « un effet de bord », ce qui signifie que cette instruction a un effet qui n'est pas relatif au simple calcul du résultat de la fonction. À titre d'exemples, les instructions d'affichage et de saisie sont des instructions qui ont un effet de bord.

En général, ***les instructions ayant des effets de bord sont à proscrire dans les fonctions*** : ils ne facilitent pas la lisibilité de l'algorithme ni sa modularité. En revanche, elles peuvent être utilisées dans les procédures.

2.3 Les procédures

Une procédure est « une fonction qui ne retourne pas de valeur ». Elle contient donc une séquence d'instructions dont l'exécution dépend de la valeur de ces paramètres.

Par exemple, supposons qu'on veuille, en fonction du paramètre n de type `entier`, une instruction qui :

— Affiche n fois `"Bonjour !"` si $n \geqslant 0$;
— Affiche $-n$ fois `"Au revoir !"` si $n < 0$.

Pour ce faire, on peut définir la procédure `bonjour_au_revoir` de la façon suivante :

```
Procédure bonjour_au_revoir (n : entier)
Variables
  | i : entier naturel
Début
  | Pour i allant de 1 à |n| Faire
  |   | Si n ⩾ 0 Alors
  |   |   | afficher("Bonjour !")
  |   | Sinon
  |   |   | afficher("Au revoir !")
  |   | Finsi
  | Finpour
Fin
```

Ainsi, les deux instructions suivantes

```
bonjour_au_revoir(2)
bonjour_au_revoir(−1)
```

auront l'effet suivant

Bonjour !
Bonjour !
Au revoir !

On notera qu'il n'y a pas de type de sortie pour les procédures (puisqu'il n'y a pas de sortie). Par ailleurs, on n'a pas d'instruction de type **retourner** *valeur* (puisqu'on ne retourne pas de valeur. En revanche, on peut utiliser l'instruction **retourner** (« retourner sans valeur ») dont l'effet est de quitter la procédure.

Exercice 4 *On suppose que la fonction* `est_premier`*, qui teste que son paramètre (un entier naturel) est premier, est définie.*

écrire une procédure nommée `premiers_inférieurs_à` *qui prend en paramètre un entier naturel n et affiche tous les entiers naturels premiers inférieurs ou égaux à n.*

Par exemple, l'instruction `premiers_inférieurs_à(100)` *donnera le résultat suivant :*

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Exercice 5 *On considère les équations sur $\mathbb{R}$ de la forme suivante :*

$$ax^2 + bx + c = 0 \qquad \text{pour } a, b, c \text{ : trois réels}$$

Écrire une procédure dont le nom est `équation_2ème_degré` *qui résout sur $\mathbb{R}$ les équations du second degré en affichant les solutions. Cette procédure fera appel, directement ou indirectement, à 4 autres procédures :*

— `équation_1er_degré` *qui résout les équations de la forme*

$$ax + b = 0 \qquad \text{pour } a, b \text{ : deux réels}$$

— `équation_vrai_1er_degré` *qui résout $ax + b = 0$ avec $a \neq 0$.*

— `équation_0ème_degré` *qui résout les équations de la forme $a = 0$ (a est une constante et l'inconnue est x, par exemple : l'ensemble des solutions de l'équation $4 = 0$ est $\emptyset$ alors que l'ensemble des solutions de $0 = 0$ est $\mathbb{R}$).*

— `équation_vrai_2ème_degré` *qui résout $ax^2 + bx + c = 0$ avec $a \neq 0$, en distinguant les 3 cas habituels (pas de solution, racine double et deux solutions).*

---

procédures pour tester une fonction. La plupart des procédures utilisées dans ce cours le sont pour effectuer des tests de fonctions.

Supposons qu'on ait écrit la fonction suivante, calculant le plus grand commun diviseur de deux entiers naturels :

```
Fonction pgcd (a : entier naturel, b : entier naturel) : entier naturel
Variables
  | x, y, tmp : entier naturel
Début
  | x ← a
  | y ← b
  | Tant que y ≠ 0 Faire
  |   | Si x < y Alors
  |   |   | /* échanger x et y */
  |   |   | tmp ← x
  |   |   | x ← y
  |   |   | y ← tmp
  |   | Sinon
  |   |   | x ← x − y
  |   | Finsi
  | Fintantque
  | retourner x
Fin
```

Pour trouver comment tester cette fonction, deux questions se posent :

1. a et b étant fixé, comment tester pgcd(a, b) ?
2. Comment choisir le jeu de test, à savoir l'ensemble des couples (a, b) à tester ?

La procédure test_pgcd_param a pour objectif de répondre à la question 1. Sa liste de paramètres est celle de $pgcd$:

```
Procédure test_pgcd_param (a : entier naturel, b : entier naturel)
Début
  | afficher("pgcd(", a, b, ") = ", pgcd(a, b))
Fin
```

La procédure test_pgcd a pour objectif de répondre à la question 2. Sa liste de paramètres est vide :

```
Procédure test_pgcd ()
Début
  | test_pgcd_param (36, 27)
  | test_pgcd_param (27, 36)
  | test_pgcd_param (27000, 36000)
  | test_pgcd_param (0, 5)
  | test_pgcd_param (0, 0)
Fin
```

La traduction de cette fonction et de ces deux procédures en C ou dans un autre langage de programmation, suivi de son exécution, doit donner un résultat qu'il faudra comparer à l'attente du développeur :

```
pgcd(36, 27) = 9
pgcd(27, 36) = 9
pgcd(27000, 36000) = 9000
pgcd(0, 5) = 5
pgcd(0, 0) = 0
```

2.4 Les fonctions et procédures en C

Dans cette section, la syntaxe des fonctions et procédures en C est décrite en s'appuyant sur des exemples montrant comment divers constructions algorithmiques sont traduites en C (conditionnelles, boucles, etc.).

À titre d'exemple, voici l'algorithme d'une fonction suivie de sa traduction en C :

Fonction `nb_chiffres` (n : `entier naturel`) : `entier naturel`
Variables
  x, nbc : `entier naturel`
Début
  $x \leftarrow n$
  $nbc \leftarrow 0$
  Tant que $x \neq 0$ **Faire**
    $x \leftarrow x \mathbin{//} 10$
    $nbc \leftarrow nbc + 1$
  Fintantque
  retourner nbc
Fin

```
unsigned int nb_chiffres (unsigned int n)
{
  unsigned int x, nbc ; /* déclaration des variables
  x = n ;               /* initialisation */
  nbc = 0 ;             /* des variables */
  while (x != 0)
  {
    x = x / 10 ;        /* division entière de x par 10 */
    nbc = nbc + 1 ;
  }
  return nbc ;
}
```

La signature d'une fonction en C correspond à la syntaxe suivante :

`<type de sortie> <nom de la fonction> (<liste des paramêtres>)`

La liste des paramêtres est une séquence d'éléments de la forme « `<type> <variable>` » séparées par des virgules.

Le corps d'une fonction est un bloc d'instructions, à savoir une séquence d'instructions précédée du symbole `{` et terminée par le symbole `}`. Chaque instruction se termine par le symbole `;`. Une variable locale est déclarée au début du bloc selon la syntaxe « `<type> <variable1>`, ... `<variable`n`>;` » (la déclaration des variables a donc la même syntaxe qu'une instruction).

Voici un deuxième exemple :

Fonction `puissance` (x : `réel`, n : `entier`) : `réel`
Variables
  a, p : `réel`
  i, k : `entier naturel`
Début
  $a \leftarrow x$
  $p \leftarrow 1$
  $k \leftarrow n$
  Si $x = 0$ **Alors**
    retourner 0
  Finsi
  Si $n < 0$ **Alors**
    $a \leftarrow 1/x$
    $k \leftarrow -k$
  Finsi
  Pour i **allant de** 1 **à** k **Faire**
    $p \leftarrow p * a$
  Finpour
  retourner p
Fin

```
double puissance (double x, int n)
{
```

```
  double a, p ;
  unsigned int i, k ;
  a = x ;
  p = 1 ;
  k = n ;
  if (x == 0)
    {
      return 0 ;
    }
  if (n < 0)
    {
      a = 1. / x ;
      k = -n ;
    }
  for (i = 1 ; i <= k ; i = i + 1)
    {
      p = p * a ;
    }
  return p ;
}
```

Une procédure en C suit la même syntaxe qu'une fonction en C, sauf que le type de sortie est remplacé par le mot-clef `void`, ainsi que l'exemple suivant le montre :

Procédure `test_puissance_param` (x : `réel`, n : `entier`)
Début
  afficher(x, `"↑"`, n, `" = "`, `puissance`(x, n))
Fin

Procédure `test_puissance` ()
Variables
  i, d, f : `entier`
  k : `entier naturel`
Début
  $d \leftarrow -4$
  $f \leftarrow 4$
  $k \leftarrow n$
  `test_puissance_param`(0., 4)
  Pour i **allant de** d **à** f **Faire**
    `test_puissance_param`(2., i)
  Finpour

```
void test_puissance_param (double x, int n)
{
  printf ("%f^%d = %f\n", x, n, puissance(x, n)) ;
}

void test_puissance ()
{
  int i, d, f ;
  d = -4 ;
  f = 4 ;
  test_puissance_param (0., 4) ;
  for (i = d ; i <= f ; i = i + 1)
    {
      test_puissance_param (2., i) ;
    }
}
```

La fonction `main` est la fonction qui est appelée en premier quand on fait appel au programme. On l'appelle pour cela la fonction principale. Dans le cadre de ce cours, elle aura toujours la structure suivante :

```
int main ()
{
  <instruction(s)>
  return EXIT_SUCCESS ;
}
```

Par exemple :

```
int main ()
{
  test_puissance () ;
  return EXIT_SUCCESS ;
}
```

Enfin, il faut noter qu'un programme C utilisant une instruction `printf` et la constante `EXIT_SUCCESS` doit contenir les lignes suivantes, en tête du fichier (juste Après les commentaires indiquant ce que fait le programme, quel est son auteur et la date de sa dernière version), sachant que les commentaires ajoutés ci-dessous sont inutiles :

```
#include <stdio.h>  /* Fait appel au fichier stdio.h de la bibliothèque standard */
#include <stdlib.h> /* Fait appel au fichier stdlib.h de la bibliothèque standard */
```

Exercice 6 *Reprenez tous les algorithmes de fonction et de procédures de ce chapitre et traduisez-les en C.*

2.5 Passage de paramètres par valeur, passage de paramètres par référence

Considérons le problème suivant : on veut écrire une procédure pour échanger la valeur de deux variables de type `entier`. Une première idée pourrait être la suivante :

/ Un algorithme qui ne fait pas ce qu'on attend de lui ! */*
Procédure `échanger` (a : `entier`, b : `entier`)
Variables
  tmp : `entier`
Début
  $tmp \leftarrow a$
  $a \leftarrow b$
  $b \leftarrow tmp$
Fin

Si x et y sont deux variables de type `entier` dans un algorithme dans lequel la fonction `échanger` a été définie, considérons la séquence d'instructions suivante :

$$\begin{aligned} &x \leftarrow 2 \\ &y \leftarrow 8 \\ &\texttt{échanger}(x, y) \\ &\textbf{afficher}(\texttt{"x = "}, x) \\ &\textbf{afficher}(\texttt{"y = "}, y) \end{aligned} \tag{2.1}$$

L'exécution de cette séquence donnera le résultat suivant :

```
x = 2
y = 8
```

qui montre que l'échange des variables ne s'est pas fait. Que s'est-il passé ?

La réponse est la suivante. Quand on fait appel à une fonction et une procédure, on fait en général des « passages de paramètres par valeur » ce qui veut dire que l'expression dans le passage de paramètre est substituée par sa valeur. Ainsi, si on a $f(e_1, e_2, \ldots, e_n)$ où les e_i sont des expressions dont la valeur est v_i, cela revient à faire $f(v_1, v_2, \ldots, v_n)$. Dans l'exemple ci-dessus, l'instruction `échanger`(x, y) revient à faire `échanger`$(2, 8)$, ce qui n'a aucun effet sur x et sur y (et qui, bien sûr, n'échangera pas ces deux entiers l'un avec l'autre !).

Pour résoudre ce problème, on peut préciser, dans la liste des paramètres, que pour un (ou plusieurs) de ces paramètres, l'identification doit se faire non pas avec la valeur de la variable mais avec la variable elle-même. La notation dans ce cours

consiste à écrire $\boxed{x}$ au lieu de x dans la liste des paramètres, si le passage de paramètre de x se fait par référence (on dit aussi « par variable » ou « par adresse »).

Par exemple, reprenons l'algorithme précédent avec des passages de paramètres par référence :

/ Un algorithme qui fait ce qu'on attend de lui. */*
Procédure `échanger` ($\boxed{a}$: `entier`, $\boxed{b}$: `entier`)
Variables
  tmp : `entier`
Début
  $tmp \leftarrow a$
  $a \leftarrow b$
  $b \leftarrow tmp$
Fin

Dans ce cas, si on reprend la séquence d'instructions (2.1), le résultat sera bien le résultat désiré :

```
x = 8
y = 2
```

2.5.1 Passage par valeur ou par référence en C

Voici un fichier C complet, présentant une implantation en C de la procédure `échanger` dont l'algorithme est présenté ci-dessous, suivi d'une procédure de test et de la fonction `main` :

```
/* procédure d'échange de variables
   auteur : Jean Lieber
   date : 24/01/14
*/

#include <stdio.h>
#include <stdlib.h>

void echanger (int *a, int *b)
{
  int tmp ;
  tmp = *a ;
  *a = *b ;
  *b = tmp ;
}

void test_echanger ()
{
  int x, y ;
  x = 2 ;
  y = 4 ;
  printf ("Avant l'échange : x = %d, y = %d\n", x, y) ;
  echanger (&x, &y) ;
  printf ("Après l'échange : x = %d, y = %d\n", x, y) ;
}

int main ()
{
  test_echanger () ;
  return EXIT_SUCCESS ;
}
```

La compilation de ce programme sous Linux par la commande

```
gcc -Wall echange.c -o echange
```

produit un fichier exécutable `echange` dont l'exécution donne

```
Avant l'échange : x = 2, y = 4
Après l'échange : x = 4, y = 2
```

donc bien ce qui était attendu.

Les pointeurs en C. Une façon de comprendre cela s'appuie sur la notion de référence (ou de pointeur) en C. Si `x` est une variable de type `int` et dont la valeur à un instant donné est `4`, alors l'expression `x` au sein d'un programme C s'interprète comme la valeur de `x` (donc, `4`). Si on veut accéder à la variable elle-même (à savoir, la donnée de l'adresse mémoire et du type), on note cela `&x`. Le type de `&x` est « pointeur sur un `int` ». Pour déclarer une variable de ce type, on procède de la façon suivante :

```
int *p ; /* p est un pointeur sur un int : le " contenu " de p, *p, est de type int */
```

Par exemple, si on veut déclarer une variable `a` de type `int` et une variable `pa` de type « pointeur sur un `int` » puis initialiser `a` à `10` et `pa` à « pointeur sur `a` », on peut écrire (dans une fonction ou une procédure C) :

```
int a, *pa ;
a = 10 ;
pa = &a ;
```

Par la suite, on pourra modifier la valeur de `a` en utilisant `pa` : `pa` est un pointeur sur `a` donc, `*pa` et `a` sont identifés. Par exemple, on peut remplacer la valeur de `a` par `20` de la façon suivante :

```
*pa = 20 ;
```

Ppur revenir à l'exemple sur l'échange de deux variables, l'instruction C `echanger(&x, &y)` fait passer en paramètre un pointeur sur `x` et un pointeur sur `y`. La liste des paramètres de la fonction `echanger` est `(int *a, int *b)`, ce qui signifie que `a` et `b` sont des pointeurs sur des `int`. Le corps de cette fonction effectue ainsi un changement des contenus de `*a` et de `*b`.

Exercice 7 *Grâce au passage par référence en C, on peut remplacer n'importe quelle fonction par une procédure (ce qui n'est pas toujours une bonne idée). Par exemple, reprenons l'exemple de la fonction C* `puissance` *définie plus haut. écrivez une procédure appelée* `proc_puissance` *prenant en paramêtres* `x` *(de type* `double`*),* `n` *(de type* `int`*) et* `resultat` *(de type pointeur sur un* `double`*) tel que*

```
double r ;
r = puissance (1.4, 4) ;
```

soit équivalent à

```
double r ;
proc_puissance (1.4, 4, &r) ;
```

---

3 La récursivité

Le but de ce chapitre est double. Il est d'abord d'introduire la récursivité (algorithmes et programmes récursifs). Il est ensuite d'introduire une démarche d'algorithmique et programmation en plusieurs étapes, pour écrire un algorithme et l'implantation d'un programme.

3.1 Rappel : suite définie par une relation de récurrence

Une suite de réels est une fonction u de $\mathbb{N}$ dans $\mathbb{R}$. On note en général u_n à la place de $u(n)$ (pour un entier n). On peut définir une suite par une récurrence, ce qui signifie (en général) que la valeur de u_n est définie par une ou plusieurs valeurs u_i avec $i < n$ et par la valeur de u_0. Considérons par exemple la suite $(u_n)_n$ définie ainsi, pour $n \in \mathbb{N}$:

$$\begin{aligned} u_0 &= 0 \\ u_{n+1} &= (n+1)^2 + u_n \end{aligned} \tag{3.1}$$

Cette suite pourrait être définie sans relation de récurrence,

par exemple par $$u_n = 0 + 1^2 + 2^2 + \ldots + n^2 = \sum_{i=1}^{n} i^2$$

ou encore par $$u_n = \frac{n(n+1)(2n+1)}{6}$$

Le premier but de ce chapitre est de voir comment on peut définir des fonctions (ou des procédures) de façon récursive, de la même façon qu'on peut définir des suites par une relation de récurrence.

3.2 Les fonctions récursives

On rappelle l'abus de langage fréquemment fait en algorithmique et programmation qui fait qu'on utilise le terme « fonction » à la place du terme « définition de fonction ». Cela nous permet de parler de fonctions récursives au lieu de parler de définition récursive de fonction.

Une fonction récursive est une fonction qui fait appel à elle-même, que ce soit directement ou indirectement [1]. Par exemple, la définition (3.1) de la suite $(u_n)_n$ peut être traduite algorithmiquement par :

Fonction u $(a$: `entier naturel`$)$: `réel`
Début
  Si $a = 0$ **Alors**
    retourner 0.
  Finsi
  retourner $a^2 + u(a-1)$
Fin

Exercice 8 *Dans l'algorithme précédent, la conditionnelle n'utilise pas de* **Sinon**. *Pourquoi ?*

Exercice 9 *Considérons l'instruction suivante faisant appel à la fonction u définie par l'algorithme récursif ci-dessous*

$$x \leftarrow u(2)$$

Décrivez pas à pas le déroulement de cette instruction (appels à la fonction u, etc.).

Exercice 10 *Considérons les définitions par récurrence des suites ci-dessous :*

$$v_0 = 0 \qquad v_{n+1} = \cos(n) + v_n$$

$$f_0 = 1 \qquad f_1 = 1 \qquad f_{n+2} = f_{n+1} + f_n$$

$$w_0 = 2 \qquad w_{n+1} = w_0 \times w_1 \times \ldots \times w_n = \prod_{i=0}^{n} w_i$$

Pour chacune de ces définitions, traduisez-la en un algorithme récursif. Pour la suite $(w_n)_n$, vous pouvez utiliser une boucle pour.

Théoriquement, on peut se passer des boucles en utilisant la récursivité et, inversement, on peut se passer de la récursivité en utilisant des boucles. Cependant, en pratique, l'écriture de certaine fonction est beaucoup plus facile en utilisant la récursivité et vice-versa.

Un algorithme *itératif* est, par définition, un algorithme non récursif. En général, il fait appel à une ou des boucle(s), mais on peut aussi faire appel à des boucles dans un algorithme récursif.

À titre d'exemple, considérons l'algorithme itératif suivant, calculant la somme des éléments d'un tableau de réels :

Fonction `somme_tableau` $(T$: `tableau de réel[n]`$)$: `réel`
Variables
  i : `entier naturel`
  S : `réel`
Début
  $S \leftarrow 0.$
  Pour i **allant de** 0 **à** $n-1$ **Faire**
    $S \leftarrow S + T[i]$
  Finpour
  retourner S
Fin

On peut redéfinir cette fonction en faisant appel à une *fonction auxiliaire* (appelée `somme_tableau_aux`) définie récursivement :

1. Ce deuxième cas, appelé « récursivité croisée », apparaîtra moins souvent dans ce cours. Il consiste par exemple à avoir une fonction f faisant appel à une fonction g, laquelle fait appel à f (plus généralement, f peut faire appel à g_1, qui fait appel à g_2, ..., qui fait appel à g_n qui fait appel à f).

```
Fonction somme_tableau (T : tableau de réel[n]) : réel
Début
  | retourner somme_tableau_aux(n − 1, T)
Fin

Fonction somme_tableau_aux (i : entier naturel, T : tableau de réel[n]) : réel
Début
  | Si i ⩾ n Alors
  |   | retourner 0.
  | Finsi
  | retourner T[i] + somme_tableau_aux(i + 1, T)
Fin
```

Exercice 11 *Décrivez pas à pas le déroulement d'un appel à la fonction* `somme_tableau` *dans sa version récursive, pour un tableau de* 4 *réels.*

Exercice 12 *L'algorithme récursif de la fonction* u *donné en début de section peut être transformé en algorithme itératif. Donnez un tel algorithme itératif.*

---

4 Les entiers naturels revisités et la démarche algorithmique d'AP2

Cette section redécrit l'algorithmique des entiers naturels (déjà vue en AP1) en s'appuyant sur la notion de type abstrait. Puis elle décrit, à travers un exemple, une démarche algorithmique en 8 points pour l'algorithmique et la programmation récursives et itératives.

Remarque 1 Dans la suite de ce chapitre et du cours, on notera généralement `entier_nat` le type des entiers naturels, plutôt que `entier naturel` (pour éviter d'éventuelles confusion dues à la présence d'une espace [1] dans le nom du type).

4.1 Le type abstrait `entier_nat`

En mathématiques, une structure algébrique est généralement la donnée d'un ensemble E et d'opérations/fonctions/relations/lois sur E. Par exemple, $(\mathbb{R}, +, \times)$ est le corps des réels : il est donné par l'ensemble des réels et par les opérations $+$ et $\times$ sur cet ensemble.

De la même façon, un type abstrait est la description d'un ensemble et d'opérations « de base » sur cet ensemble. Ces opérations sont appelées « opérations primitives », car les autres opérations peuvent être définies grâce à elles. Ces opérations sont décrites par leurs profils (le profil d'une opération donnae les types de ses paramètres et le type de son résultat) et par un ensemble d'axiomes liant ces opérations.

Le type abstrait `entier_nat` s'appuie sur l'intuition des « entiers bâtons » et est défini ainsi :

Nom : `entier_nat` ;

Type abstrait importé `booléen` (un type abstrait importé est un type abstrait sur lequel le type abstrait en cours de définition est construit) ;

opérations primitives

- Profils :
  - Constructeurs (l'ensemble des valeurs du type est obtenue par toutes les expressions s'appuyant sur les constructeurs) :
    `zéro` : $\longrightarrow$ `entier_nat`
    `succ` : `entier_nat` $\longrightarrow$ `entier_nat`
    (`Erreur_entier_nat` : $\longrightarrow$ `entier_nat`)
    Explications : `zéro` est une opération sans paramètre, qu'on peut assimiler à une constante. On écrira `zéro` au lieu de `zéro`() pour simplifier.

1. Le mot « espace » utilisé en typographie est un substantif féminin.

Dans un premier temps, on néglige le troisième constructeur (d'où les parenthèses). Du coup, cela signifie que l'ensemble des valeurs possibles est :

$$\{\texttt{zéro},\ \texttt{succ}(\texttt{zéro}),\ \texttt{succ}(\texttt{succ}(\texttt{zéro})),\ \texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zéro}))),\ \ldots\}$$

Si on interprète `zéro` par l'entier naturel 0 et $\texttt{succ}(x)$ par l'entier naturel $1+x$, on obtient $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$. De façon générale, on peut noter l'affirmation (récursive) suivante :

Un entier naturel est soit `zéro` soit $\texttt{succ}(x)$, où x est un entier naturel

La constante `Erreur_entier_nat` sert à indiquer une erreur.

— Accès :
`est_nul` : `entier_nat` $\longrightarrow$ `booléen`
`préc` : `entier_nat` $\longrightarrow$ `entier_nat`

— Axiomes (où x est de type `entier_nat`) :

[1] $\texttt{est_nul}(\texttt{zéro}) = \texttt{vrai}$

[2] $\texttt{est_nul}(\texttt{succ}(x)) = \texttt{faux}$

[3] $\texttt{préc}(\texttt{zéro}) = \texttt{Erreur_entier_nat}$

[4] $\texttt{préc}(\texttt{succ}(x)) = x$

Notons que, pour être complet, il faudrait aussi deux axiomes donnant les valeurs pour $\texttt{est_nul}(\texttt{Erreur_entier_nat})$ et $\texttt{préc}(\texttt{Erreur_entier_nat})$. En s'appuyant sur le « principe de propagation de l'erreur » qui dit que toute expression contenant une erreur est une erreur (éventuellement d'un autre type), on peut construire les axiomes suivants (qu'on négligera de donner pour les autres types abstrait du cours, en s'appuyant sur ce principe de propagation de l'erreur) :

[5] $\texttt{est_nul}(\texttt{Erreur_entier_nat}) = \texttt{Erreur_booléen}$

[6] $\texttt{préc}(\texttt{Erreur_entier_nat}) = \texttt{Erreur_entier_nat}$

4.2 Algorithmique et programmation d'opérations non primitives sur les entiers naturels

Sur la base de ce type abstrait, on peut définir d'autres opérations, dites « non primitives ». Plus précisément, pour une telle opération, on va suivre dans ce cours la démarche en 8 points suivante :

1. définir le profil de l'opération (sous la forme `<nom de l'opération>` : `<type_1>` $\times \ldots \times$ `<type_n>` $\longrightarrow$ `<type>`).
2. Traiter un ou plusieurs exemples, détaillant en plusieurs étapes comment calculer la valeur issue de l'opération pour des paramètres donnés.
3. définir un jeu d'axiomes en s'appuyant sur les exemples (les axiomes généralisent les exemples). Ce jeu d'axiomes constitue à la fois une définition formelle de l'opération et un moyen de calcul.
4. Traduire ce jeu d'axiomes en algorithme récursif.
5. écrire un algorithme itératif, en s'appuyant sur la reformulation des exemples sous la forme d'un tableau de variables (dépendant du temps).
6. Traduire l'algorithme récursif en programme C.
7. Traduire l'algorithme itératif en programme C.
8. écrire les procédures de test pour tester les deux fonctions.

Cette démarche va être détaillée ci-dessous pour l'opération `égaux` qui permet de tester que deux entiers naturels sont égaux.

4.2.1 Profil de `égaux`

Le profil est le suivant :
`égaux` : `entier_nat` $\times$ `entier_nat` $\longrightarrow$ `booléen`

Cela signifie que l'expression $\texttt{égaux}(e_1, e_2)$ est correcte en terme de type si e_1 et e_2 sont des expressions de type `entier_nat` (i.e., e_1 peut s'écrire `zéro` ou $\texttt{succ}(x)$ où x est un entier naturel et de même pour e_2). Si c'est le cas, $\texttt{égaux}(e_1, e_2)$ est une expression à valeur booléenne.

4.2.2 Traitement d'exemples de calculs de $\texttt{égaux}(e_1, e_2)$

Dans le cadre de l'exemple, `zéro` sera noté par le symbole 0 et $\texttt{succ}(x)$ sera noté Sx. Ainsi, $SSS0 = \texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{zéro})))$.

Comme le type de sortie de `égaux` est `booléen`, il est bon d'avoir au moins deux exemples : l'un dont le résultat sera `vrai`, l'autre dont le résultat sera `faux`.

Premier exemple. On va évaluer l'expression $\texttt{égaux}(SSSS0, SSS0)$ (on s'attend à la réponse `faux`). Pour ce faire, on ramène la question « Que vaut $\texttt{égaux}(SSSS0, SSS0)$?» à une question juste un peu plus simple, à savoir « Que vaut $\texttt{égaux}(SSS0, SS0)$?». On repart de cette question, qu'on simplifie à nouveau. Cela donne :

$$\texttt{égaux}(SSSS0, SSS0) = \texttt{égaux}(SSS0, SS0) = \texttt{égaux}(SS0, S0) = \texttt{égaux}(S0, 0) = \texttt{faux}$$

On peut noter que les étapes du calcul symbolisées par $=$ correspondent à la même action (enlever un S à chacun des paramètres), à l'exception de la dernière étape.

On a donc deux sortes d'actions, qu'on va numéroter [1] et [2] (qui sont les numéros de futurs axiomes). Ce faisant, on peut « décorer » l'exemple de la façon suivante :

$$\texttt{égaux}(SSSS0, SSS0) \overset{[1]}{=} \texttt{égaux}(SSS0, SS0) \overset{[1]}{=} \texttt{égaux}(SS0, S0) \overset{[1]}{=} \texttt{égaux}(S0, 0) \overset{[2]}{=} \texttt{faux}$$

Pour poursuivre la « décoration » de l'exemple, il faut indiquer, pour (presque) chaque valeur en partie droite d'une équation comment elle a été calculée sur la base de valeurs en partie gauche de l'équation :

$$\texttt{égaux}(SSSS0, SSS0) \overset{[1]}{=} \texttt{égaux}(SSS0, SS0) \overset{[1]}{=} \texttt{égaux}(SS0, S0) \overset{[1]}{=} \texttt{égaux}(S0, 0) \overset{[2]}{=} \texttt{faux}$$

Deuxième exemple. En suivant le même principe, on peut traiter l'exemple $\texttt{égaux}(SSS0, SSS0)$ et le décorer de la façon suivante :

$$\texttt{égaux}(SSS0, SSS0) \overset{[1]}{=} \texttt{égaux}(SS0, SS0) \overset{[1]}{=} \texttt{égaux}(S0, S0) \overset{[1]}{=} \texttt{égaux}(0, 0) \overset{[3]}{=} \texttt{vrai}$$

4.2.3 Jeu d'axiomes pour `égaux`

L'analyse des exemples ci-dessus montre que [1] s'applique quand les deux arguments sont non nuls (i.e., ce sont des $\texttt{succ}(\textit{quelque chose})$). On arrive à l'axiome suivant :

[1] $\texttt{égaux}(\texttt{succ}(x), \texttt{succ}(y)) = \texttt{égaux}(x, y)$

(sous-entendu « pour tout x et y de type `entier_nat` », étant donné que c'est le type qui correspond au paramètre de `succ` — cf. le profil de cette opération primitive).

[2] s'applique quand le premier paramètre est un $\texttt{succ}(\textit{quelque chose})$ alors que le second est `zéro`, d'où l'axiome :

[2] $\texttt{égaux}(\texttt{succ}(x), \texttt{zéro}) = \texttt{faux}$

[3] s'applique quand les deux paramètres valent `zéro`, d'où :

[3] $\texttt{égaux}(\texttt{zéro}, \texttt{zéro}) = \texttt{vrai}$

Ces axiomes sont-ils suffisants ? Non, car il existe des couples de paramètres qui ne sont couvert par aucun axiome. Il s'agit des couples dont le premier est `zéro` et le second est un $\texttt{succ}(\textit{quelque chose})$. D'où le dernier axiome :

[4] $\texttt{égaux}(\texttt{zéro}, \texttt{succ}(x)) = \texttt{faux}$

On peut voir que ces axiomes sont à la fois une définition de l'opération d'égalité et qu'ils permettent de définir un processus de calcul pour tester l'égalité : il suffit d'appliquer *de gauche à droite* ces axiomes et on arrivera fatalement sur une des deux valeurs `vrai` et `faux`. Pour s'en convaincre, on peut reprendre un exemple.

4.2.4 Algorithme récursif pour `égaux`

La traduction de ces axiomes en un algorithme récursif donne alors :

```
Fonction égaux (a : entier_nat, b : entier_nat) : entier_nat
Début
  | /* Traduction de l'axiome [3] */
  | Si est_nul(a) et est_nul(b) Alors
  |   | retourner vrai
  | Finsi
  | /* Ci-dessous, on a donc non(est_nul(a) et est_nul(b)), i.e., l'un au moins de a et b est non nul */
  | /* Traduction de l'axiome [2] */
  | Si est_nul(b) Alors
  |   | retourner faux
  | Finsi
  | /* Ci-dessous, on a donc b est non nul */
  | /* Traduction de l'axiome [4] */
  | Si est_nul(a) Alors
  |   | retourner faux
  | Finsi
  | /* Ci-dessous, on a donc a et b non nuls */
  | /* Traduction de l'axiome [1] */
  | retourner égaux(préc(a), préc(b))
Fin
```

L'ordre dans lequel les axiomes sont traduits est théoriquement quelconque. Cependant, une bonne pratique consiste à les considérer « du plus simple au plus compliqué ».

Notons que ce type de traduction a déjà été fait au début de ce chapitre (partant de définition par récurrence de suites pour arriver à des algorithmes récursifs).

Cet algorithme peut se simplifier en :

```
Fonction égaux (a : entier_nat, b : entier_nat) : entier_nat
Début
  | Si est_nul(a) et est_nul(b) Alors
  |   | retourner vrai
  | Finsi
  | Si est_nul(b) ou est_nul(b) Alors
  |   | retourner faux
  | Finsi
  | retourner égaux(préc(a), préc(b))
Fin
```

Exercice 13 *Expliquez comment cette simplification a été effectuée.*

4.2.5 Algorithme itératif pour `égaux`

On peut reformuler le premier exemple ci-dessous — `égaux`$(SSSS0, SSS0)$ — en utilisant un tableau de variables :

t	0	1	2	3
x	$SSSS0$	$SSS0$	$SS0$	$S0$
y	$SSS0$	$SS0$	$S0$	0

Dans ce tableau, t indique le temps avec des valeurs entières. 0 est l'instant initial, 1 est l'instant de la fin de la première itération (de la future boucle), 2 est l'instant de la fin de la deuxième itération, etc. À noter que t n'est pas une variable informatique (t ne sera pas utilisé en tant que tel dans l'algorithme).

L'analyse de ce tableau va permettre de dégager des éléments pour construire l'algorithme itératif. Les éléments qui se dégagent de ce raisonnement seront encadrés comme cela. À ce stade, l'algorithme ressemble à cela :

```
Fonction égaux (a : entier_nat, b : entier_nat) : entier_nat
Variables
  | x, y : entier_nat
Début
  | /* Initialisation des variables : */
  | x ← ?
  | y ← ?
  | Tant que ? Faire
  |   | ?
  | Fintantque
  | retourner ?
Fin
```

x et y sont des variables et dans la suite, x_t et y_t denotent les valeurs de x et y à l'instant t. La valeur initiale de x est $SSSS0$, c'est-à-dire, en généralisant à la question $\texttt{égaux}(a,b) = ?$, qu'on a $\boxed{x_0 = a}$. De la même façon, $\boxed{y_0 = b}$. La question qui se pose ensuite est « Comment les valeurs x_t et y_t évoluent-elles ? ». Dans cet exemple, on a :

$$x_{t+1} = \texttt{préc}(x_t)$$
$$y_{t+1} = \texttt{préc}(y_t)$$

ce qui va correspondent aux deux instructions suivantes : $\boxed{\begin{array}{ll}\text{(a)} & x \leftarrow \texttt{préc}(x) \\ \text{(b)} & y \leftarrow \texttt{préc}(y)\end{array}}$. Ces deux instructions vont être le corps de la boucle tant que qui va être introduite dans l'algorithme itératif. La question qui se pose alors est « Dans quel ordre ? » : (a) avant (b), (b) avant (a) ou bien cela n'a pas d'importance ? Dans cet exemple, $\boxed{\text{l'ordre entre (a) et (b) n'a pas d'importance}}$ car il n'y a pas d'interactions entre les valeurs des variables. On prend un ordre quelconque (l'ordre lexicographique, par exemple).

La question suivante porte sur la condition d'arrêt de la boucle. En l'occurrence, la boucle s'arrête parce que la valeur de y est nulle, d'où la condition

$$\texttt{STOP}_1 = \texttt{est_nul}(y)$$

Est-ce la seule façon d'arrêter les itérations ? En l'occurrence, si on prend un autre exemple, on verra que non. Prenons l'exemple du calcul de $\texttt{égaux}(SS0, SSSSS0)$:

t	0	1	2
x	$SS0$	$S0$	0
y	$SSSSS0$	$SSSS0$	$SSS0$

. La raison de l'arrêt de cette boucle est

$$\texttt{STOP}_2 = \texttt{est_nul}(x)$$

Si on estime que ce sont les deux seuls cas d'arrêt, la condition d'arrêt sera $\texttt{STOP}_1$ **ou** $\texttt{STOP}_2$. Donc la condition du tant que est la négation de cette question, à savoir :

$$\textbf{non}(\texttt{STOP}_1\ \textbf{ou}\ \texttt{STOP}_2) = \textbf{non}\ \texttt{STOP}_1\ \textbf{et non}\ \texttt{STOP}_2 = \textbf{non}\ \texttt{est_nul}(y)\ \textbf{et non}\ \texttt{est_nul}(x)$$
$$= \boxed{\textbf{non}\ \texttt{est_nul}(x)\ \textbf{et non}\ \texttt{est_nul}(y)}$$

En fin de compte quelle est la valeur à retourner ? Souvent il est intéressant de savoir quelle est la condition qui suit la sortie du tant que, à savoir le fait que cette condition est fausse et donc, que sa négation est vrai, c'est-à-dire que x est nul ou y est nul (ou inclusif). En l'occurrence, la réponse à la question $\texttt{égaux}(a,b)$? s'est ramené à la question $\texttt{égaux}(x,y)$? pour toutes les valeurs successives de x et y. À la fin, on sait que l'un des deux au moins est nul et donc, l'égalité de x et de y revient à tester qu'ils sont tous les deux nuls, d'où la valeur à retourner : $\boxed{\texttt{est_nul}(x)\ \textbf{et}\ \texttt{est_nul}(y)}$.

On peut donc écrire l'algorithme itératif suivant pour égaux :

Fonction égaux (a : entier_nat, b : entier_nat) : entier_nat
Variables
  x, y : entier_nat
Début
  /* *Initialisation des variables :* */
  $x \leftarrow a$
  $y \leftarrow b$
  Tant que non est_nul(x) **et non** est_nul(y) **Faire**
    $x \leftarrow$ préc(x)
    $y \leftarrow$ préc(y)
  Fintantque
  /* *x est nul et/ou y est nul* */
  retourner est_nul(x) **et** est_nul(y)
Fin

4.2.6 Programme récursif pour égaux

Pour passer à la programmation en C, on peut soit définir les opérations primitives dans ces deux langages soit utiliser des correspondances suivantes :

Notation algorithmique	C
entier_nat	unsigned int
zéro, succ(x)	0, x + 1
0, x + 1 est_nul(x), préc(x)	x == 0, x - 1

On va appliquer ces correspondances pour traduire l'algorithme récursif en un algorithme itératif :

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

bool egaux_R (unsigned int a, unsigned int b)
{
  if (a == 0 && b == 0)
    {
      return true ;
    }
  if (a == 0 || b == 0)
    {
      return false ;
    }
  return egaux_R (a - 1, b - 1) ;
}
```

4.2.7 Programme itératif pour égaux

Suite du programme précédent :

```
bool egaux_I (unsigned int a, unsigned int b)
{
  unsigned int x, y ;
  x = a ;
  y = b ;
  while (x != 0 && y != 0)
    {
      x = x - 1 ;
      y = y - 1 ;
    }
  return x == 0 && y == 0 ;
}
```

4.2.8 Test des programmes récursif et itératif implantant `égaux`

Fin du programme précédent en C :

```
void test_egaux_param (unsigned int a, unsigned int b)
{
  printf ("égaux (%u, %u) ? ", a, b) ;
  if (egaux_R (a, b))
    {
      printf ("oui (R) ") ;
    }
  else {
    printf ("non (R) ") ;
  }
  if (egaux_I (a, b))
    {
      printf ("oui (I)\n") ;
    }
  else {
    printf ("non (I)\n") ;
  }
}

void test_egaux ()
{
  test_egaux_param (4, 3) ;
  test_egaux_param (2, 5) ;
  test_egaux_param (6, 6) ;
}

int main ()
{
  test_egaux () ;
  return EXIT_SUCCESS ;
}
```

4.2.9 Autres étapes

On peut étendre la démarche par les 3 points suivants (hors-programme d'AP2) :

9. Preuve d'algorithme ou de programme : prouver que l'algorithme ou le programme réalise bien la spécification (on peut, par exemple, prouver que le résultat donné par l'algorithme itératif est cohérent avec les axiomes).
10. Preuve de terminaison : prouver que quel que soit les valeurs des paramètres, l'algorithme ou le programme va s'exécuter en un temps fini.
11. Étude de la complexité : évaluation de l'ordre de grandeur asymptotique du temps de calcul de l'exécution d'un algorithme.

4.3 Exercice

Pour faire l'exercice ci-dessous, vous aurez besoin de vous appuyer sur le type abstrait `booléen`, dont les opérations ont déjà été étudiées en AP1 : `vrai`, `faux`, **non**, **et**, **ou**, et sur l'opération « valeur conditionnelle », non étudiée (*a priori*) en AP1 et qui est définie comme suit :

- τ est un type donné ;
- Profil : **Si-Alors-Sinon-Finsi**$_\tau$: $\texttt{booléen} \times \tau \times \tau \longrightarrow \tau$
- Axiomes :
  - [1] **Si-Alors-Sinon-Finsi**$_\tau(\texttt{vrai}, v_1, v_2) = v_1$
  - [2] **Si-Alors-Sinon-Finsi**$_\tau(\texttt{faux}, v_1, v_2) = v_2$
- Notation : au lieu de **Si-Alors-Sinon-Finsi**(c, v_1, v_2) on écrit
  Si c **Alors**
  | v_1
  Sinon
  | v_2
  Finsi .
- Attention à ne pas confondre avec les *instructions* conditionnelles.

Exercice 14 *On considère les opérations non primitives suivantes :*

- $*$ `ppq` *est l'opération qui aux entiers naturels x et y associe* `vrai` *si $x < y$ et* `faux` *sinon* (`ppq` *est l'acronyme de « plus petit que »).*
- $*$ `plus` *est l'opération qui à deux entiers naturels x et y associe la somme de x et y ($x + y$).*
- $*$ `moins` *est l'opération qui à deux entiers naturels x et y associe la somme de x et y ($x - y$). à noter que si $x < y$, le résultat est une erreur.*
- $*$ `mult` *est l'opération qui à deux entiers naturels x et y associe le produit de x et y ($x \times y$).*
- $*$ `fact` *est l'opération qui à l'entier naturel n associe la factorielle de n : $n! = \prod_{i=1}^{i=n} i$.*
- $*$ `somme_0_à_n` *est l'opération qui à l'entier naturel n associe la somme des entiers naturels i inférieurs ou égaux à n :* $\texttt{somme_0_à_n}(n) = \sum_{i=0}^{i=n} i$.
- $**$ `div` *est l'opération qui à deux entiers naturels x et y associe le quotient de la division entière de x par y ($x \,\texttt{div}\, y$).*
- $**$ `mod` *est l'opération qui à deux entiers naturels x et y associe le reste de la division entière de x par y ($x \bmod y$).*
- $*$ `puissance` *est l'opération qui à deux entiers naturels x et y associe x^y.*
- $**$ `pgcd` *est l'opération qui à deux entiers naturels x et y associe le plus grand commun diviseur de x et y.*
- $***$ `est_premier` *est l'opération qui teste si un entier naturel est premier (on rappelle que l'entier naturel x est premier si $x \geqslant 2$ et si les seuls diviseurs de x sont 1 et x).*
- $***$ `est_carré` *est l'opération qui teste si un entier naturel est un carré parfait :* $\texttt{est_carré}(x) = \texttt{vrai}$ *ssi il existe y tel que* $\texttt{carré}(y) = x$, *où* `carré` *est la fonction définie par l'axiome* $\texttt{carré}(x) = \texttt{mult}(x, x)$.
- $***$ `racine_carrée` *est l'opération qui à un entier naturel associe sa racine carrée entière, à savoir le plus grand entier naturel r tel que* $\texttt{carré}(r) \leqslant x$.
- $***$ `n_ème_premier` *est l'opération qui à un entier naturel n associe le $n^{\text{ème}}$ nombre premier (comme les premiers nombres premiers sont, dans l'ordre, 2, 3, 5, 7, 11, etc.,* $\texttt{n_ème_premier}(4) = 7$*).*
- $**$ $\texttt{nb_chiffres}(x)$ *est le nombre de chiffres de x en numérotation décimale (p. ex.,* $\texttt{nb_chiffres}(486) = 3$*).*
- $**$ $\texttt{n_ème_chiffre}(n, x)$ *est le chiffre « en 10^n » de x, ainsi* $\texttt{n_ème_chiffre}(0, x)$ *est l'unité de x,* $\texttt{n_ème_chiffre}(1, x)$ *est sa dizaine,* $\texttt{n_ème_chiffre}(2, x)$ *est sa centaine, etc.*

Pour chacune d'elle, il est demandé de suivre les huit points de la démarche ci-dessous. Ces opérations peuvent s'appuyer sur les opérations sur les booléens, sur les opérations primitives du type `entier_nat` *et sur les opérations non primitives déjà définies auparavant.*

Le nombre d'étoiles est caractéristique de la difficulté de l'exercice, selon le principe suivant :

$*$ *: très facile* $\quad$ $**$ *: facile* $\quad$ $***$ *: assez facile* $\quad$ $****$ *: relativement facile*

Indications *Pour les opérations* `est_premier`, `est_carré`, `racine_carrée` *et* `n_ème_premier`, *la partie récursive (jeu d'axiomes, algorithme récursifs, programme récursif) devra faire appel à une fonction auxiliaire.*

---

5 Les enregistrements (compléments au cours éponyme d'AP1)

Ce cours revient sur la notion d'enregistrement introduite en AP1. Plus précisément, il revient sur la notion d'enregistrement sous l'angle des types abstraits, à travers un exemple suivi : celui du type des nombres complexes.

5.1 Type abstrait `complexe`

On va représenter un nombre complexe $z \in \mathbb{C}$ par deux réels a (la partie réelle — champ `re`) et b (la partie imaginaire — champ `im`) : $z = a + ib$. Cela donne le type abstrait suivant :

Nom : `complexe`;

Type abstrait importé `réel`;

Opérations primitives

- Profils :
  - Constructeurs :
    - `zéro` : $\longrightarrow$ `complexe` (*la valeur par défaut du type est $0 + i0$*)
    - `écrire_re` : `réel` $\times$ `complexe` $\longrightarrow$ `complexe`
    - `écrire_im` : `réel` $\times$ `complexe` $\longrightarrow$ `complexe`
  - Accès :
    - `lire_re` : `complexe` $\longrightarrow$ `réel`
    - `lire_im` : `complexe` $\longrightarrow$ `réel`
- Axiomes :
  - [1] $\texttt{lire_re}(\texttt{zéro}) = 0$
  - [2] $\texttt{lire_re}(\texttt{écrire_re}(a, z)) = a$
  - [3] $\texttt{lire_re}(\texttt{écrire_im}(b, z)) = \texttt{lire_re}(z)$
  - [4] $\texttt{lire_im}(\texttt{zéro}) = 0$
  - [5] $\texttt{lire_im}(\texttt{écrire_re}(a, z)) = \texttt{lire_im}(z)$
  - [6] $\texttt{lire_im}(\texttt{écrire_im}(b, z)) = b$

Une fois ce type abstrait défini, on peut définir des opérations non primitives (et suivre les étapes (1) à (3) de la démarche algorithmique décrite dans le chapitre précédent). Par exemple, considérons l'opération calculant le conjugué d'un nombre complexe (on rappelle que si $z = a + ib \in \mathbb{C}$ avec $a, b \in \mathbb{R}$, le conjugué de z est $\overline{z} = a - ib$).

(1) Son profil est `conjugué : complexe` $\longrightarrow$ `complexe` .
(2) À titre d'exemple, si $z = a+ib$ avec $a =$ `lire_re`(z) et $b =$ `lire_im`(z), alors `conjugué`$(z) = a-ib =$ `écrire_im`$(-$`lire_im`$(z), z)$.
(3) Un jeu d'axiome(s) pour cette opération est :

[1] `conjugué`$(z) =$ `écrire_im`$(-$`lire_im`$(z), z)$

Exercice 15 *Pour chacune des opérations suivantes, donnez son profil, un exemple (si cela vous est utile) et un jeu d'axiome(s) :*

- *L'opération calculant le module d'un nombre complexe (rappel* $|a + ib| = \sqrt{a^2 + b^2}$*);*
- *L'opération calculant la somme de deux nombres complexes (rappel :* $(a_1 + ib_1) + (a_2 + ib_2) = (a_1 + a_2) + i(b_1 + b_2)$*);*
- *L'opération calculant le produit de deux nombres complexes (rappel :* $(a_1 + ib_1)(a_2 + ib_2) = (a_1a_2 - b_1b_2) + i(a_1b_2 + a_2b_1)$*);*
- *L'opération donnant l'inverse d'un nombre complexe (rappel : si* z *est le complexe nul, cela doit donner une erreur, sinon* $\frac{1}{z} = \frac{\overline{z}}{|z|^2}$*);*
- *L'opération calculant le rapport de deux nombres complexes (qui doit donner une erreur si le dénominateur est nul, rappel :* $\frac{z_1}{z_2} = z_1 \frac{1}{z_2}$*);*
- *L'opération qui aux réel positif* ϱ *et au réel* θ *associe le nombre complexe de module* ϱ *et d'argument* θ *(rappel : sa partie réelle est* $\varrho \cos\theta$*, sa partie imaginaire est* $\varrho \sin\theta$*).*

(les rappels ci-dessus sont donnés pour $a, a_1, a_2, b, b_1, b_2 \in \mathbb{R}$ *et* $z, z_1, z_2 \in \mathbb{C}$*).*

5.2 Manipulation algorithmique d'enregistrements

En AP1, la manipulation d'un enregistrement se faisait uniquement via la notation `x . c` où `x` est une valeur d'un type enregistrement dont `c` est un champ. Dans l'exemple des complexes, on aurait par exemple `z . re` pour la partie réelle du complexe `z`.

En AP2, on utilisera le moins possible cette notation. En fait, on l'utilisera uniquement pour définir (algorithmiquement ou en C) les opérations de lecture et d'écriture :

Types
  `complexe` : **enregistrement**
    `re` : `réel`
    `im` : `réel`
Début
  Fonction `écrire_re` $(a$: `réel`$, z$: `complexe`$)$: `complexe`
  Début
    z`.re` $\leftarrow a$
    retourner z
  Fin

  Fonction `écrire_im` $(b$: `réel`$, z$: `complexe`$)$: `complexe`
  Début
    z`.im` $\leftarrow b$
    retourner z
  Fin

  Fonction `lire_re` $(z$: `complexe`$)$: `réel`
  Début
    retourner z`.re`
  Fin

  Fonction `lire_im` $(z$: `complexe`$)$: `réel`
  Début
    retourner z`.im`
  Fin

Une fois ces opérations primitives définies, la notation avec un point sera à éviter le plus possible (application du principe d'encapsulation qui sera décrit et justifié à la section 5.4). Ainsi, si on veut décrire un algorithme de la fonction `conjugué` on peut écrire :

Fonction `conjugué` $(z$: `complexe`$)$: `complexe`
Variables
  c : `complexe`
Début
  $c \leftarrow$ `zéro` /* *Initialisation de c* */
  $c \leftarrow$ `écrire_re`$($`lire_re`$(z), c)$ /* *La partie réelle de c est la partie réelle de z* */
  $c \leftarrow$ `écrire_im`$(-$`lire_im`$(z), c)$ /* *La partie imaginaire de c est l'opposé de la partie imaginaire de z* */
  retourner c
Fin

Exercice 16 *Donnez un algorithme pour chacune des opérations non primitives introduites dans l'exercice précédent.*

5.3 Implantation des enregistrements en C

Le fichier `complexe.c` reproduit ci-dessous est une implantation du type `complexe` avec l'implantation de l'opération non primitive `conjugué` et avec une procédure pour afficher une valeur de ce type :

```
/************************/
/* complexe.c           */
/* auteur : Jean Lieber */
/* date : 07/02/14      */
/************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define PARTIE_REELLE_PAR_DEFAUT 0.
#define PARTIE_IMAGINAIRE_PAR_DEFAUT 0.

struct Complexe
{
  double re ;
  double im ;
} ; // Ne pas oublier ce point-virgule !

typedef struct Complexe complexe ;

/* SIGNATURES DES OPERATIONS PRIMITIVES */
/* Constructeurs et */
complexe zero () ;
complexe ecrire_re (double a, complexe z) ;
complexe ecrire_im (double b, complexe z) ;
/* Acces */
double lire_re (complexe z) ;
double lire_im (complexe z) ;

/* IMPLANTATION DES OPERATIONS PRIMITIVES */
/* Constructeurs */

complexe ecrire_re (double a, complexe z)
{
  z.re = a ;
  return z ;
}

complexe ecrire_im (double b, complexe z)
{
  z.im = b ;
  return z ;
}

complexe zero ()
{
  complexe z ;
  z = ecrire_re (PARTIE_REELLE_PAR_DEFAUT, z) ;
  z = ecrire_im (PARTIE_IMAGINAIRE_PAR_DEFAUT, z) ;
  return z ;
}

/* Accès */
double lire_re (complexe z)
{
  return z.re ;
}

double lire_im (complexe z)
{
  return z.im ;
}

/* OPERATIONS NON PRIMITIVES */

complexe conjugue (complexe z)
{
  complexe c ;
  c = zero() ;
  c = ecrire_re (lire_re (z), c) ;
  c = ecrire_im (-lire_im(z), c) ;
  return c ;
}

void afficher_complexe (complexe z)
{
  double a, b ;
  a = lire_re(z) ;
  b = lire_im (z) ;
  printf ("%f %s %fi",
  a,
  (b < 0. ? "-" : "+"),
  fabs(b)) ; // fabs : valeur absolue
}
```

Le fichier `test_complexe.c` reproduit ci-dessous permet de tester ce type :

```
/************************/
/* test_complexe.c      */
/* auteur : Jean Lieber */
/* date : 07/02/14      */
/************************/

#include "complexe.c"

void test_afficher_complexe_param (complexe z)
{
  printf ("z = ") ;
  afficher_complexe (z) ;
  printf ("\n") ;
}
```

```
void test_afficher_complexe ()
{
  test_afficher_complexe_param (zero()) ;
  test_afficher_complexe_param (ecrire_re(2., zero())) ;
  test_afficher_complexe_param (ecrire_im(-4., zero())) ;
  test_afficher_complexe_param (ecrire_re(2., ecrire_im(-4., zero()))) ;
  test_afficher_complexe_param (ecrire_re(-2., ecrire_im(4., zero()))) ;
}

void test_conjugue_param (complexe z)
{
  printf ("Le conjugué de ") ;
  afficher_complexe (z) ;
  printf (" est ") ;
  afficher_complexe (conjugue (z)) ;
  printf (".\n") ;
}

void test_conjugue ()
{
  test_conjugue_param (zero()) ;
  test_conjugue_param (ecrire_re(2., zero())) ;
  test_conjugue_param (ecrire_im(-4., zero())) ;
  test_conjugue_param (ecrire_re(2., ecrire_im(-4., zero()))) ;
  test_conjugue_param (ecrire_re(-2., ecrire_im(4., zero()))) ;
}

int main ()
{
  test_afficher_complexe () ;
  printf("\n====================\n\n") ;
  test_conjugue () ;
  return EXIT_SUCCESS ;
}
```

5.4 Le principe d'encapsulation appliqué aux enregistrements

L'encapsulation est un principe de programmation issu de la programmation par objets mais qui peut s'appliquer à des langages de programmation qui ne sont pas conçus à l'origine pour ce type de programmation, tel que C. Appliqué à un type enregistrement, il consiste à masquer, pour l'*utilisateur* de ce type, la façon dont le *développeur* de ce type l'a encodé. Si on suit ce principe, cela permet de modifier la façon dont un type est représenté sans que son utilisation, dans un autre programme, par exemple, soit modifiée.

Voyons comment ce principe s'applique au type `complexe`. Le développeur du type a décrit dans un premier temps ce type comme dans les sections ci-dessus : un complexe est donné par un enregistrement à deux champs représentant les parties réelle et imaginaire. Un utilisateur de ce type respectant le principe d'encapsulation n'accèdera à ce type qu'à travers les fonctions d'écriture et de lecture (correspondant, dans le type abstrait, aux constructeurs et aux accès). Il pourra, par exemple, effectuer ainsi une implantation de l'opération `conjugué` (comme présenté ci-dessus) et plein d'autres opérations. Supposons à présent que le développeur du type décide de changer l'implantation. Par exemple, il décidera d'encoder désormais un nombre complexe par un enregistrement à deux champs : le module et l'argument.

Une fois cette modification faite, il redéfinira les opérations `zéro`, `lire_re`, `écrire_re`, `lire_im` et `écrire_im` (qui ne seront plus des opérations primitives) sur la base des nouvelles opérations primitives, par exemple :

```
double lire_re (complexe z)
{
  return lire_module (z) * cos(lire_argument (z)) ;
}
```

Si le principe d'encapsulation a été respecté, il n'y aura pas de changement pour l'utilisateur du type. En revanche, si à un moment, l'utilisateur n'a pas respecté le principe d'encapsulation et a écrit `z.re`, alors son code ne fonctionnera plus.

Exercice 17 *On veut utiliser les enregistrements pour représenter les types suivants :*

- *Type* `point2D` *des points du plan, un point étant donné par une abscisse (un réel), une ordonnée (un réel) et le « point par défaut » étant l'origine (le point d'abscisse et d'ordonnée nulles).*

— *Le type* `triangle` *des triangles du plan, un triangle étant donné par trois sommets, de type* `point2D`. *Les sommets du triangle par défaut sont tous l'origine (on a donc un triangle réduit à un point).*
— *Le type* `appartement` *des appartements, un appartement étant donné par son aire en mètres carrés (un réel), un nombre de pièces (un entier naturel), un étage (un entier relatif — pour considérer le cas des appartements en sous-sol) et le fait que l'appartement est actuellement occupé ou non (un booléen). L'appartement par défaut fait* $20\,m^2$*, contient* 2 *pièces, est au* $8^{ème}$ *étage et est occupé.*

Pour chacun de ces types, suivez la démarche du cours et donnez :
— *Une description du type abstrait (nom, type(s) abstrait(s) importé(s), profils et axiomes des opérations primitives) ;*
— *Un algorithme pour chacune des opérations primitives ;*
— *Une implantation en C du type, y compris la traduction des opérations primitives et une procédure pour afficher une valuer du type.*

Exercice 18 *On considère les types définis dans l'exercice précédent et on demande de définir les opérations non primitives suivantes (jeux d'axiomes, algorithmes, traduction en C) :*
— `distance` *qui à deux points associe leur distance (on considèrera la distance euclidienne canonique, cette distance est donc* $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$*, où* x_1 *et* x_2 *sont les abscisses des deux points et où* y_1 *et* y_2 *sont leurs ordonnées) ;*
— `égaux_point2D` *qui teste si deux points sont égaux ;*
— `périmètre` *qui donne le périmètre d'un triangle ;*
— `égaux_triangle` *qui teste si deux triangles sont égaux (i.e., leurs ensembles de sommets sont les mêmes, même s'ils ne sont pas représentés dans le même ordre ; par exemple, si* $A = Z$*,* $B = X$ *et* $C = Y$ *alors les triangles* ABC *et* XYZ *sont égaux).*

---

6 Les listes

6.1 Les listes et les tableaux

D'un point de vue mathématique, on ne fait guère la distinction entre une liste et un tableau. La liste $(0.\ 3.\ 2.\ 3.\ 4.)$ et le tableau $[0.\ 3.\ 2.\ 3.\ 4.]$ sont deux représentations du même quintuplet de réels $(0, 3, 2, 3, 4) \in \mathbb{R}^5$.

En revanche, d'un point de vue algorithmique, les tableaux et les listes se manipulent très différemment :
— L'accès au $i^{\text{ème}}$ élément d'un tableau se fait très rapidement, en un temps qui est indépendant de i (« temps d'accès en $O(1)$ »). On parle d'accès *direct* (en anglais : *random access*). En revanche, l'accès au $i^{\text{ème}}$ élément d'une liste se fait de façon séquentielle, en parcourant la liste à partir du premier élément (« temps d'accès en $O(i)$ »). On parle d'accès *séquentiel*.
— La structure d'un tableau est « rigide ». Quand on crée un tableau de 100 éléments, on ne peut pas ajouter un élément en tête sans recréer un nouveau tableau de 101 éléments dans lequel on copiera à l'indice 0 l'élément à insérer et dans lequel on copiera tous les éléments du premier tableau. En revanche, la structure d'une liste est « souple » : on peut facilement et très rapidement (en $O(1)$) insérer un élément en tête.

À noter qu'en Python, la structure de donnée `list` est un peu une combinaison des deux types (même si elle a été étudiée en AP1 pour représenter les tableaux).

6.2 Type abstrait `liste`

On considère le type `typélt` (type des éléments). Sur la base de ce type, on définit le type abstrait des listes d'éléments de type `typélt` de la façon suivante :

Nom : `liste`;
Types abstraits importés `typélt`, `booléen`;
opérations primitives
— Profils :
  — Constructeurs :
    `l_vide` : $\longrightarrow$ `liste`
    `cons` : `typélt` $\times$ `liste` $\longrightarrow$ `liste`
  — accès :
    `est_vide` : `liste` $\longrightarrow$ `booléen`
    `prem` : `liste` $\longrightarrow$ `typélt`
    `reste` : `liste` $\longrightarrow$ `liste`
— Axiomes :
  [1] `est_vide(l_vide) = vrai`
  [2] `est_vide(cons(`x, L`)) = faux`
  [3] `prem(l_vide) = Erreur_typélt`
  [4] `prem(cons(`x, L`))` $= x$
  [5] `reste(l_vide) = l_vide` [1]
  [6] `reste(cons(`x, L`))` $= L$

1. Cet axiome est parfois remplacé par l'axiome « `reste(l_vide) = Erreur_liste` ». Néanmoins, celui qui est proposé a des avantages pratiques (qu'on pourra découvrir dans certains exercices) et nous nous y tiendrons dans ce cours.

6.3 Implantation des listes par des enregistrements : les listes chaînées

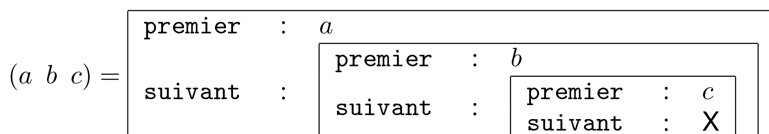
6.3.1 Principe des listes chaînées

Considérons la liste $(a\ \ b\ \ c)$. Le constructeur cons suggère qu'on la voit comme étant composée de deux parties : a et $(b\ \ c)$. Il en va de même pour $(b\ \ c)$ qui est constitué de b et de (c). Enfin, (c) se décompose en c et $()$ (la liste vide). L'idée est donc d'utiliser un enregistrement à deux champs :

— Le champ `premier`, de type `typélt` ;
— Le champ `suivant`, de type `liste`.

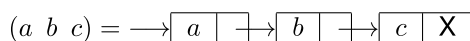
Cette définition du type liste est une définition récursive : le type `liste` est défini en s'appelant lui-même. Pour que cette définition soit fondée, il faut définir une constante pour la liste vide.

En s'appuyant sur cette définition du type `liste` par un enregistrement, on arrive à :



où X représente la liste vide.

Pour visualiser plus simplement une telle liste, on utilisera la convention graphique suivante : $\longrightarrow$ [x | L] pour [premier : x ; suivant : L] (i.e., pour cons(x, L)). Ainsi :



6.3.2 Les listes chaînées en C

Dans cette section, on décrit une implantation en C des listes chaînes d'entiers (`typélt` = `int`).

La difficulté principale tient au fait que si `t` est un type enregistrement en C (défini par `struct`), alors toute valeur de type `t` prendra la même place mémoire, ce qui n'est pas cohérent avec le type `liste`. L'idée est alors de considérer que le champ `suivant` est un pointeur. Plus précisément, on va définir deux types (l'un s'appuyant sur l'autre et vice-versa) :

— Un type `struct Liste`, enregistrement à deux champs : `premier` de type `int` et `suivant` de type `liste` ;
— Un type `liste` : type des pointeurs sur des `struct Liste`.

Ainsi, si L est de type `liste`, pour accéder au champ `premier`, on pourra écrire `(*L).premier` : L est de type `liste`, `*L` — le *contenu* de L — est de type `struct Liste` et on accède au champ `premier` de `*L`.

Comme, de plus, en C, `(*x).c` peut s'écrire `x->c` on aboutit à l'implantation suivante des listes en C :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Liste *liste ;

struct Liste
{
  int premier ;
  liste suivant ;
} ;

/* SIGNATURES DES OPERATIONS PRIMITIVES */
liste l_vide () ;
liste cons (int x, liste L) ;
bool est_vide (liste L) ;
int prem (liste L) ;
liste reste (liste L) ;

/* IMPLANTATION DES OPERATIONS PRIMITIVES */
/* Constructeurs */
liste l_vide ()
{
  return NULL ;
}

liste cons (int x, liste L)
{
  liste M ;
  /* Réservation de la place mémoire nécessaire */
  M = malloc (sizeof (*M)) ;
  M->premier = x ;
  M->suivant = L ;
  return M ;
}

/* accès */
bool est_vide (liste L)
{
  return L == NULL ;
}

int prem (liste L)
{
  if (est_vide (L))
    {
      printf ("Calcul de prem sur liste vide !\n") ;
      exit (0) ;
    }
  return L->premier ;
}

liste reste (liste L)
{
  return L->suivant ;
}
```

Quand une liste a été créée et qu'elle n'est plus référencée, l'espace alloué pour elle doit être libéré. La procédure C suivante permet de faire cela :

```
void liberer_liste (liste L)
{
  if (est_vide (L))
    {
      return ;
    }
  liberer_liste (reste (L)) ;
  free(L) ;
}
```

6.4 Exercice

Exercice 19 *On considère les opérations non primitives suivantes :*

- $*$ `longueur` *qui à une liste associe le nombre d'éléments qu'elle contient. Exemple :* `longueur`$((a\ \ b\ \ c)) = 3$.
- $*$ `appartient` *qui teste si une valeur appartient à une liste. Exemple :* `appartient`$(b, (a\ \ b\ \ c)) =$ `vrai`.
- $*$ `nème_élément` *qui à un entier naturel n et une liste L associe le $n^{ème}$ élément de L. Exemples :* `nème_élément`$(0, (a\ \ b\ \ c)) =$ `Erreur_typélt`, `nème_élément`$(1, (a\ \ b\ \ c)) = a$, `nème_élément`$(2, (a\ \ b\ \ c)) = b$, `nème_élément`$(3, (a\ \ b\ \ c)) = c$, `nème_élément`$(4, (a\ \ b\ \ c)) =$ `Erreur_typélt`.
- $*$ `nème_reste` *qui à un entier naturel n et une liste L associe la liste obtenue en appliquant n fois l'opération* `reste` *sur L. Exemples :* `nème_reste`$(0, (a\ \ b\ \ c)) = (a\ \ b\ \ c)$, `nème_reste`$(1, (a\ \ b\ \ c)) = (b\ \ c)$, `nème_reste`$(2, (a\ \ b\ \ c)) = (c)$, `nème_reste`$(8, (a\ \ b\ \ c)) = ()$.
- $**$ `insérer_élément` *qui au* `typélt` *x, à l'entier naturel n et à la liste L associe la liste obtenue en insérant x à la $n^{ème}$ place de L. Exemples :* `insérer_élément`$(e, 0, (a\ \ b\ \ c)) =$ `Erreur_liste`, `insérer_élément`$(e, 1, (a\ \ b\ \ c)) = (e\ \ a\ \ b\ \ c)$, `insérer_élément`$(e, 2, (a\ \ b\ \ c)) = (a\ \ e\ \ b\ \ c)$, `insérer_élément`$(e, 4, (a\ \ b\ \ c)) = (a\ \ b\ \ c\ \ e)$, `insérer_élément`$(e, 5, (a\ \ b\ \ c)) =$ `Erreur_liste`.
- $**$ `supprimer_élément` *qui au* `typélt` *x et à la liste L associe la liste obtenue en supprimant toutes les occurrences de x dans L. Exemples :* `supprimer_élément`$(a, (a\ \ b\ \ a\ \ c\ \ d\ \ a\ \ a)) = (b\ \ c\ \ d)$, `supprimer_élément`$(a, (b\ \ f)) = (b\ \ f)$.
- $*$ `nb_occurrences` *qui au* `typélt` *x et à la liste L associe le nombre d'occurrences de x dans L. Exemple :* `nb_occurrences`$(a, (a\ \ b\ \ a\ \ c\ \ d\ \ a\ \ a)) = 4$.
- $*$ `égales` *qui teste si deux listes sont égales. Exemples :* `égales`$((a\ \ b\ \ c), (a\ \ b\ \ c)) =$ `vrai`, `égales`$((a\ \ b\ \ c), (a\ \ c\ \ b)) =$ `faux`, `égales`$((a\ \ b\ \ c), (a\ \ b\ \ c\ \ c)) =$ `faux`.
- $*$ `dernier` *qui à une liste non vide L associe son dernier élément. Exemples :* `dernier`$((a\ \ b\ \ c)) = c$, `dernier`$(()) =$ `Erreur_typélt`.
- $***$ `renverser` *qui à une liste L associe la liste obtenue en inversant l'ordre des éléments de L. Exemple :* `renverser`$((a\ \ b\ \ c)) = (c\ \ b\ \ a)$.
- $**$ `snoc` *est la fonction qui à une liste L et à un* `typélt` *x associe la liste obtenue en ajoutant x à la fin de L. Exemple :* `snoc`$((a\ \ b\ \ c), e) = (a\ \ b\ \ c\ \ e)$.
- $**$ `concaténer` *est la fonction qui à deux listes L_1 et L_2 associe la liste obtenue en prenant d'abord les éléments de L_1 puis ceux de L_2. Exemple :* `concaténer`$((a\ \ b\ \ c), (d\ \ e)) = (a\ \ b\ \ c\ \ d\ \ e)$.
- $**$ `supprimer_répétitions` *est la fonction qui à une liste L associe la liste obtenue en supprimant les répétitions entre éléments voisins de L. Exemple :* `supprimer_répétitions`$((a\ \ a\ \ b\ \ a\ \ c\ \ c\ \ c\ \ b\ \ b)) = (a\ \ b\ \ a\ \ c\ \ b)$.
- $*$ `sous_séquence_de` *est la fonction qui teste qu'une liste L_1 est une sous-séquence d'une liste L_2 : tous les éléments de L_1 se trouvent dans L_2 dans le même ordre, mais il peut y avoir des éléments supplémentaires dans L_2 à tout endroit. Exemples :* `sous_séquence_de`$((a\ \ b\ \ c), (w\ \ a\ \ x\ \ y\ \ b\ \ c\ \ z)) =$ `vrai`, `sous_séquence_de`$((a\ \ b\ \ c), (w\ \ a\ \ x\ \ y\ \ c\ \ b\ \ z)) =$ `faux`.
- $****$ `sous_liste_de` *est la fonction qui teste qu'une liste L_1 est une sous-liste d'une liste L_2 : autrement écrit, il existe deux listes L et M telles que L_2 est égal à la concaténation de L, de L_1 et de M. Exemples :* `sous_liste_de`$((a\ b\ c), (a\ b\ x\ c\ a\ b\ c\ y\ z)) =$ `vrai`, `sous_liste_de`$((a\ \ b\ \ c), (w\ \ a\ \ x\ \ y\ \ b\ \ c\ \ z)) =$ `faux`.

Pour chacune d'elle, il est demandé de suivre les huit points de la démarche en 8 points abordée au chapitre 2.

Pour certaines opérations, on pourra supposer que l'opération `égaux`$_{\texttt{typélt}}$ *qui teste l'égalité de deux éléments de type* `typélt` *est définie et implantée.*

Pour les opérations `renverser` *et* `sous_liste_de` *il sera utile d'introduire une fonction auxiliaire pour les jeux d'axiomes, les algorithmes récursifs et les programmes récursifs.*

Pour l'implantation en C ou en Python de ces opérations, on supposera qu'on a des listes d'entiers (`int` *à la place de* `typélt`*).*

Le nombre d'étoiles est caractéristique de la difficulté de l'exercice.

6.5 Opérations non destructrices et opérations destructrices

Considérons par exemple l'opération `répéter_éléments` qui à un `typélt` a et une liste L associe la liste obtenue en répétant dans L les éléments. Par exemple : `répéter_éléments`$((a\ \ b\ \ c)) = (a\ \ a\ \ b\ \ b\ \ c\ \ c)$.

En suivant la démarche algorithmique, on pourrait arriver à la fonction récursive C suivant[2] :

```
liste repeter_elements (liste L)
{
  int x ;
  if (est_vide (L))
    {
      return l_vide () ;
    }
  x = prem (L) ;
  return cons (x, cons (x, repeter_elements (reste(L)))) ;
}
```

Cette fonction crée deux listes en mémoire complètement disjointes en mémoire :

si L ⟶ [1 |] ⟶ [2 | X] alors `répéter_éléments`(L) ⟶ [1 |] ⟶ [1 |] ⟶ [2 |] ⟶ [2 | X]

(il y a 6 valeurs distinctes de l'enregistrement `struct Liste`).

On pourrait, notamment pour des raisons d'économie de la mémoire, vouloir une implantation de l'opération `répéter_éléments` qui modifie son argument. Comme une liste dans notre implantation C est un pointeur, un élément de type `struct Liste` est passé par référence quand on passe une liste en paramètre par valeur. On parlera d'opération destructrice, puisqu'elle modifie son argument. Cette opération peut s'implanter à l'aide d'une procédure, qu'on appellera `repeter_elements_D` et telle que, si `L` est de type `liste`, alors l'instruction `repeter_elements_D (L)` modifiera `L`.

Par exemple, si on le bout de code suivant (où `afficher_liste` est une procédure d'affichage d'une liste) :

```
  liste L ;
  L = cons (1, cons (2, cons (3, l_vide ()))) ;
  printf ("Avant : L = ") ;
  afficher_liste (L) ;
  repeter_elements_D (L) ;
  printf ("\nAprès : L = ") ;
  afficher_liste (L) ;
  printf ("\n") ;
```

alors, on attend le résultat suivant :

```
Avant : L = (1 2 3)
Après : L = (1 1 2 2 3 3)
```

Pour implanter la procédure `repeter_elements` on a besoin d'opérations modifiant une valeur de type `struct Liste` : ce seront les opérations définies comme suit :

```
void ecrire_prem (int x, liste L)
{
  L->premier = x ;
}
```

```
void ecrire_reste (liste R, liste L)
{
  L->suivant = R ;
}
```

On peut alors écrire la procédure :

```
void repeter_elements_D (liste L)
{
  int x ;
  if (est_vide (L))
    {
      return ; /* quitter la procédure */
    }
  x = prem (L) ;
  ecrire_reste (cons (x, reste (L)), L) ;
  repeter_elements_D (reste (reste (L))) ;
}
```

Exercice 20 *Pour chacune des opérations de l'exercice 19 qui retournent une liste, écrire une version destructrice du programme C*[3].

2. Le raisonnement qui suit s'appliquerait aussi à une fonction itérative.
3. Dans certains cas, on pourra changer légèrement la spécification pour simplifier. Par exemple, on pourra supposer que l'opération `supprimer_élément`, dans sa version destructrice, ne supprime jamais le premier élément de la liste.

Exercice 21 *On considère l'opération* `clone` *qui à une liste L associe une liste L_2 telle que* $\texttt{égaux}(L, L_2) = \texttt{vrai}$. *De plus, on suppose que L_2 est une « reconstruction » de L, autrement écrit, L et L_2 ne partagent pas d'espace mémoire.*

Écrivez une fonction C implantant `clone`.

Étant donné une procédure C destructrice (par exemple `repeter_elements_D`*) comment peut-on définir une fonction C non destructrice à partir de cette procédure et de la fonction* `clone` *?*

---

7 Piles, files et autres structures linéaires

Ce chapitre présente plusieurs structures de données proches des listes : les piles, les files, les ensembles finis, les multiensembles finis, les listes circulaires, les listes bidirectionnelles et les listes hétérogènes. Le niveau de détail de la description sera variable.

7.1 Les piles

Les piles sont des structures de données obéissant au « paradigme *LIFO* » : « paradigme » peut être compris comme « modèle » et « *LIFO* » est l'acronyme de *Last In First Out* (dernier arrivé, premier sorti). La pile d'assiettes est une bonne façon de visualiser les piles. Et, en effet, la dernière assiette ajoutée à la pile est aussi la première qu'on enlèvera.

7.1.1 Type abstrait `pile`

Nom : `pile`;
Types abstraits importés `typélt`, `booléen`;
opérations primitives
- Profils :
  - Constructeurs :
    `pile_vide` : $\longrightarrow$ `pile`
    `empiler` : `typélt` $\times$ `pile` $\longrightarrow$ `pile`
  - Accès :
    `est_vide` : `pile` $\longrightarrow$ `booléen`
    `sommet` : `pile` $\longrightarrow$ `typélt`
    `dépiler` : `pile` $\longrightarrow$ `pile`
- Axiomes :
  [1] $\texttt{est_vide}(\texttt{pile_vide}) = \texttt{vrai}$
  [2] $\texttt{est_vide}(\texttt{empiler}(x, P)) = \texttt{faux}$
  [3] $\texttt{sommet}(\texttt{pile_vide}) = \texttt{Erreur_typélt}$
  [4] $\texttt{sommet}(\texttt{empiler}(x, P)) = x$
  [5] $\texttt{dépiler}(\texttt{pile_vide}) = \texttt{pile_vide}$
  [6] $\texttt{dépiler}(\texttt{empiler}(x, P)) = L$

On pourra noter que le type abstrait des piles est le même que celui des listes, à quelques renommages près. La différence réside davantage dans l'implantation. En particulier, alors qu'on peut aisément insérer un élément au « milieu » d'une liste chaînée, cette opération est coûteuse pour une pile (il faut dépiler plusieurs fois, empiler l'élément à insérer et empiler les éléments qui ont été dépilés).

7.1.2 Implantation

Une façon d'implanter une pile est d'utiliser une liste.

Une autre façon (plus courante) consiste à utiliser un enregistrement dont les trois champs sont :
- `tab` de type tableau de N `typélt`,
- `h` : un entier naturel prenant ses valeurs dans $\{0, 1, \ldots, N-1\}$

où N est un entier naturel considéré comme suffisamment grand. L'idée est que la pile suivante

a
b
c

$= \texttt{empiler}(a, \texttt{empiler}(b, \texttt{empiler}(c, \texttt{pile_vide})))$

soit représentée par $\texttt{h} = 3$ (`h` est la hauteur de la pile : son nombre d'éléments) et `tab` tel que $\texttt{tab}[0] = c$, $\texttt{tab}[1] = b$ et $\texttt{tab}[2] = a$. La manipulation de la liste se fait de la façon suivante :
- `pile_vide` se fait en créant une valeur de ce type enregistrement et en initialisant `tab` par un tableau de N valeurs quelconques et `h` à 0 ;
- $P \leftarrow \texttt{empiler}(x, P)$ se fait en faisant $\texttt{tab[h]} \leftarrow x$ et $\texttt{h} \leftarrow \texttt{h} + 1$ (sauf si la pile est déjà pleine...) ;
- $\texttt{est_vide}(P)$ revient au test $\texttt{h} = 0$.
- $\texttt{sommet}(P)$ se calcule par $\texttt{tab}[\texttt{h} - 1]$ (qui déclenchera une erreur si `h` est nul.
- $P \leftarrow \texttt{dépiler}(P)$, se fait comme suit. Si $\texttt{h} = 0$, aucune action n'est effectuée. Si $\texttt{h} > 0$, alors l'instruction à effectuer sera $\texttt{h} \leftarrow \texttt{h} - 1$.

7.1.3 Applications

Test du bon parenthésage d'une chaîne de caractères. Considérons la chaîne de caractère suivante : `"ab{c[de(fg)](())}h"`. Cette chaîne est bien parenthésée : chaque caractère de signe de parenthésage ouvert ({, [et () correspond à un signe de parenthésage fermé (},] et)) et les correspondances ne se croisent pas. À l'inverse, les chaînes de caractères suivantes sont mal parenthésées : `"{c[d]"`, `"{(})"`, `"[ab)"`.

On peut se servir d'une pile P de caractères pour tester si une chaîne de caractères est bien ou mal parenthésée. P est initialisée à `pile_vide`. On parcourt de gauche à droite la chaîne de caractères. À chaque fois qu'on rencontre un signe de parenthésage ouvrant, on l'empile dans P. À chaque fois qu'on rencontre un signe de parenthésage fermant, on teste si le sommet de la pile correspond (p. ex., { et } correspondent). Dans l'affirmative, on dépile P. Dans la négative, la chaîne est mal parenthésée (et on quitte la fonction). À la fin du parcours de la chaîne, la pile est vide si et seulement si la chaîne était bien parenthésée.

Gestion des contextes lors d'appels à des fonctions et procédures. Dans un langage interprété, comme Python, tout comme dans un langage compilé, comme C, l'appel à des fonctions utilise une pile, la pile des contextes. Un contexte décrit des informations sur les variables et paramètres tels qu'ils doivent être considérés à un instant donné (en particulier, leurs valeurs). Quand on fait appel à une fonction ou à une procédure, on crée un nouveau contexte dans lequel le contexte précédent est inaccessible.

Par exemple, si la fonction f a comme paramètre a de type `entier` et x de type `réel` et fait appel à la fonction g qui a comme paramètre a de type `réel` et une variable x de type `entier`, dans le contexte de l'exécution des instructions de g, a est un réel et x, un entier. L'exécution d'une fonction récursive se fait à l'aide de cette pile.

7.1.4 Exercices

Exercice 22 *Simulez sur un exemple le test de bon parenthésage d'une chaîne de caractères.*

Exercice 23 *On considère un algorithme récursif de la fonction factorielle (à écrire). Simulez le comportement de la pile des contextes pour l'appel récursif à la factorielle avec le paramètre* 4.

7.2 Les files

Les files sont des structures de données obéissant au « paradigme *FIFO* » : « *FIFO* » est l'acronyme de *First In First Out* (premier arrivé, premier sorti). La file d'attente est une bonne façon de se représenter les files : le premier arrivé est le premier servi et donc le premier à partir.

Le terme anglais pour « file » est *queue*, le terme anglais *file* signifie « fichier » (ce qui n'a guère à voir).

7.2.1 Type abstrait `file`

Nom : `file`;
Types abstraits importés `typélt`, `booléen`;
opérations primitives
- Profils :
  - Constructeurs :
    `file_vide` : $\longrightarrow$ `file`
    `enfiler` : `typélt` $\times$ `file` $\longrightarrow$ `file`
  - Accès :
    `est_vide` : `file` $\longrightarrow$ `booléen`
    `premier` : `file` $\longrightarrow$ `typélt`
    `défiler` : `file` $\longrightarrow$ `file`
- Axiomes :
  [1] `est_vide(file_vide)` = `vrai`
  [2] `est_vide(enfiler(`x, F`))` = `faux`
  [3] `premier(file_vide)` = `Erreur_typélt`
  [4] `premier(enfiler(`x, F`))` = **Si** `est_vide(`F`)` **Alors** | x **Sinon** | `premier(`F`)` **Finsi**
  [5] `défiler(file_vide)` = `file_vide`
  [6] `défiler(enfiler(`x, F`))` = **Si** `est_vide(`F`)` **Alors** | `file_vide` **Sinon** | `enfiler(`x`, défiler(`F`))` **Finsi**

7.2.2 Implantation

Une façon d'implanter une file est d'utiliser une liste. Par exemple, l'opération `enfiler` se fait par l'opération `snoc` et l'opération `défiler` par l'opération primitive `reste`. Un inconvénient de cette implantation est la complexité de l'implantation de l'opération `enfiler` qui ne prendra pas un temps constant (en $O(1)$) mais un temps linéaire ($O(n)$ où n est le nombre d'éléments de la liste).

Une autre façon de faire consiste à utiliser un enregistrement dont les trois champs sont :

— `tab` de type tableau de N `typélt` ;
— `début` et `fin` : deux entiers naturels prenant leurs valeurs dans $\{0, 1, \ldots, N-1\}$

où N est un entier naturel considéré comme suffisamment grand. L'idée est que :
— Si `début` $\leqslant$ `fin` alors les éléments de la file sont `tab`[`début`], `tab`[`début` + 1], ..., `tab`[`fin`] ;
— Si `début` $>$ `fin` alors les éléments de la file sont `tab`[`début`], `tab`[`début` + 1], ..., `tab`[$N-1$], `tab`[0], `tab`[1], ..., `tab`[`fin`].

Par exemple, la file suivante :

→	d	c	b	a	→

$= \texttt{enfiler}(d, \texttt{enfiler}(c, \texttt{enfiler}(b, \texttt{enfiler}(a, \texttt{file_vide}))))$

puisse être représentée, pour $N = 10$, par exemple par :

$$\texttt{début} = 2 \qquad \texttt{fin} = 5 \qquad \texttt{tab} = [x, x, a, b, c, d, x, x, x, x]$$

Les valeurs x sont quelconques. Ils remplissent la *zone de débordement* (i.e., la partie du tableau qui n'est pas dans la zone utile). La même file peut être représentée par :

$$\texttt{début} = 8 \qquad \texttt{fin} = 1 \qquad \texttt{tab} = [c, d, x, x, x, x, x, x, a, b]$$

La manipulation de la file se fait de la façon suivante :
— `file_vide` se fait en créant une valeur de ce type enregistrement et en initialisant `tab` par un tableau de N valeurs quelconques, `début` à 1 et `fin` à 0 : on se donne la convention qui dit que la file est vide si `début` = `fin` + 1 ;
— $F \leftarrow$ `enfiler`(x, F) se fait en faisant `fin` = (`fin` + 1) `mod` N et `tab`[`fin`] $\leftarrow x$;
— `est_vide`(F) revient au test `début` = `fin` + 1 ;
— `premier`(F) se calcule de la façon suivante. Si F est vide, alors le résultat est `Erreur_typélt`. Sinon, c'est `tab`[`début`].
— $F \leftarrow$ `défiler`(F), se fait comme suit. Si F est vide, alors le résultat est `file_vide`. Sinon, l'instruction à effectuer sera `début` $\leftarrow$ (`début` + 1) `mod` N.

7.2.3 Applications

Les files peuvent être utilisées pour gérer une ressource partagée sur un réseau, par exemple, une imprimante : les requêtes d'impressions sont placées dans une file, de façon à ce qu'une requête ancienne soit privilégiée par rapport à une requête plus récente.

Les files sont aussi utilisés pour certains algorithmes de parcours en largeur d'arbres (voir AP3).

7.2.4 Exercice

Exercice 24 *Pour chacune des opérations suivantes, suivez les points (1) à (5) de la démarche algorithmique décrite dans le chapitre 2 (profil, exemple(s), jeu d'axiome(s), algorithme récursif, algorithme itératif) :*

* `longueur` *: qui à une file associe le nombre d'éléments qu'elle contient. Exemple :* `longueur` (→ | d | c | b | a | →) = 4.
* `copier` *: qui à une file associe un clone de cette file. On supposera que les valeurs du type* `typélt` *sont également clonables.*

7.3 Les ensembles finis

On peut avoir besoin de représenter les ensembles finis quand on veut considérer des valeurs sans considérer aucun ordre entre ces valeurs et quand on ne considère pas les occurrences multiples. Par exemple, on a les égalités suivantes entre ensembles :

$$\{a, b, c\} = \{c, a, b\} = \{a, a, c, a, b, c\}$$

7.3.1 Type abstrait `ensemble`

Nom : `file` ;

Types abstraits importés `typélt`, `booléen` ;

opérations primitives

— Profils :
  — Constructeurs :
    `ensemble_vide` : $\longrightarrow$ `ensemble`
    `ajouter` : `typélt` $\times$ `ensemble` $\longrightarrow$ `ensemble`
  — Accès :
    `est_vide` : `ensemble` $\longrightarrow$ `booléen`
    `soit` : `ensemble` $\longrightarrow$ `typélt`
    `supprimer` : `typélt` $\times$ `ensemble` $\longrightarrow$ `ensemble`

— Axiomes :

[1] `est_vide`(`ensemble_vide`) = `vrai`

[2] `est_vide`(`ajouter`(x, E)) = `faux`

[3] `soit`(`ensemble_vide`) = `Erreur_typélt`

[4] `soit`(`ajouter`$(x,$ `ensemble_vide`$)$) = x

[5] `soit`(`ajouter`$(x,$ `ajouter`$(y, E)))$
$\in \{x,$ `soit`(`ajouter`$(y, E))\}$

[6] `supprimer`$(x,$ `ensemble_vide`$)$ = `ensemble_vide`

[7] $\texttt{supprimer}(x, \texttt{ajouter}(y, E))$
    Si $x = y$ **Alors**
        $\texttt{supprimer}(x, E)$
$=$ **Sinon**
        $\texttt{ajouter}(y, \texttt{supprimer}(x, E))$
    Finsi

On notera que l'opération `soit` n'est pas caractérisée aussi précisément que les opérations `prem`, `sommet` et `premier`, des types `liste`, `pile` et `file`. Ce qu'on sait simplement est que, si E est un ensemble non vide, $\texttt{soit}(E)$ est un élément de E. En fait, l'instruction $x \leftarrow \texttt{soit}(E)$ peut se lire « Soit $x \in E$. » On ne considérera pas que `soit` est une opération déterministe. Autrement écrit, si E est un ensemble non vide donné, on ne suppose pas que $\texttt{soit}(E)$ donne toujours le même résultat [1].

7.3.2 Parcours d'un ensemble

Supposons qu'on ait un ensemble d'entiers ($\texttt{typélt} = \texttt{entier}$) et qu'on veut écrire une fonction calculant la somme des éléments de cet ensemble. Le profil de cette opération est $\texttt{somme_éléments} : \texttt{ensemble} \longrightarrow \texttt{entier}$. On peut le définir par le jeu d'axiomes suivant :

[1] $\texttt{somme_éléments}(\texttt{ensemble_vide}) = 0$

[2] $\texttt{somme_éléments}(\texttt{ajouter}(x, E)) = x + \texttt{somme_éléments}(\texttt{supprimer}(x, E))$

Un algorithme récursif traduisant ce jeu d'axiomes serait :

```
Fonction somme_éléments (E : ensemble) : entier
Variables
  | x : entier
Début
  | Si est_vide(E) Alors
  |   | retourner 0
  | Finsi
  | x ← soit(E)
  | retourner x + somme_éléments(supprimer(x, E))
Fin
```

Notez qu'on introduit x et qu'on ne pourrait pas le remplacer, dans les deux occurrences de la dernière instruction par $\texttt{soit}(E)$ (puisqu'on ne suppose pas que $\texttt{soit}(E)$ donne toujours la même valeur).

Un algorithme itératif pour cette opération est le suivant :

```
Fonction somme_éléments (E : ensemble) : entier
Variables
  | x, S : entier
  | F : ensemble
Début
  | F ← E
  | S ← 0
  | Tant que non est_vide(F) Faire
  |   | x ← soit(F)
  |   | S ← S + x
  |   | F ← supprimer(x, F)
  | Fintantque
  | retourner S
Fin
```

On peut réécrire cet algorithme en utilisant la structure **Pour** $v \in E$ **Faire** où v est une variable de type `typélt` et E est un ensemble fini de `typélt` :

```
Fonction somme_éléments (E : ensemble) : entier
Variables
  | x, S : entier
Début
  | S ← 0
  | Pour x ∈ E Faire
  |   | S ← S + x
  | Finpour
  | retourner S
Fin
```

1. Cela n'implique *pas* qu'on suppose qu'il donne parfois des résultats différents. En fait, on ne fait pas l'hypothèse du déterminisme de `soit`, ce qui n'équivaut pas à faire l'hypothèse de son indéterminisme.

Notons que le comportement du `soit` fait qu'on ne peut pas supposer l'ordre dans lequel seront faites les additions. Dans la mesure où l'addition est associative et commutative, cet ordre n'a pas d'importance (il en serait de même pour les calculs du produit, du maximum et du minimum d'un ensemble d'entiers), mais si on remplaçait l'addition par une opération non commutative (par exemple, l'élévation à la puissance) on ne pourrait pas déterminer le résultat (par exemple, avec l'ensemble $\{1, 2, 3\}$, $(2^3)^1 \neq (3^1)^2$), ce qui signifie que la structure d'ensemble ne serait pas appropriée pour cela.

7.3.3 Implantations

A priori, on peut représenter un ensemble fini par une liste, une pile, une file ou un tableau. Nous allons Considérer l'implantation d'un ensemble fini par une liste. Cela signifie qu'un ensemble fini donné pourra généralement être représenté par plusieurs listes possibles. Par exemple, l'ensemble $\{a, b, c\}$ pourra être représenté par $(a\ \ b\ \ c)$, par $(b\ \ a\ \ c)$, etc.

Deux conventions sont possibles :

- Un élément d'un ensemble représenté par une liste peut apparaître plusieurs fois dans la liste. Dans ce cas, l'opération `ajouter` peut se faire simplement par `cons`. L'inconvénient est que la liste peut être inutilement trop grande.
- Un élément d'un ensemble représenté par une liste apparaîtra exactement une fois dans la liste. Dans ce cas, l'opération `ajouter` se fera en testant d'abord l'appartenance de l'élément à la liste.

Exercice 25 *Pour chacune de ces deux implantations d'ensembles finis par des listes, expliquez comment les opérations primitives des ensembles finis sont implantées.*

Il existe d'autres implantations de ce type, notamment une implantation efficace s'appuyant sur le type des arbres (voir fin du cours d'AP2 et, surtout, cours d'AP3).

7.3.4 Exercice

Exercice 26 *On Considére les opérations non primitives suivantes :*

- $*$ `appartient` *: qui teste l'appartenance d'une valeur à un ensemble ($x \in E$);*
- $**$ `cardinalité` *: qui compte le nombre d'éléments d'un ensemble ($\mathrm{card}(E)$);*
- $*$ `est_inclus_dans` *: qui teste l'inclusion d'un ensemble dans un autre ($E \subseteq F$);*
- $*$ `intersection` *: qui calcule l'intersection de deux ensembles ($E \cap F$);*
- $*$ `union` *: qui calcule l'union de deux ensembles ($E \cup F$);*
- $*$ `différence` *: qui calcule la différence de deux ensembles ($E \setminus F = \{x \in E \mid x \notin F\}$);*
- $*$ `différence_symétrique` *: qui calcule la différence symétrique de deux ensembles ($E \bigtriangleup F = (E \setminus F) \cup (F \setminus E)$).*

Pour chacune d'elle, il est demandé de suivre les huit points de la démarche en 8 points abordée au chapitre 2.

Pour certaines opérations, on pourra supposer que l'opération $\texttt{égaux}_{\texttt{typélt}}$ *qui teste l'égalité de deux éléments de type* `typélt` *est définie et implantée.*

7.4 Les multiensembles finis

Un multiensemble est une structure de données dans laquelle l'ordre des éléments est sans importance (comme pour les ensembles) mais le nombre d'occurrences d'un élément en a (comme pour les listes). Par exemple :

$$\{\!\{a, a, b, c, c, c\}\!\} = \{\!\{a, b, c, a, c, c\}\!\} = \{\!\{c, c, b, a, c, a\}\!\} \qquad \text{et} \qquad \{\!\{a, a, b, c, c, c\}\!\} \neq \{\!\{a, b, c, c\}\!\}$$

où $\{\!\{a, a, b, c, c, c\}\!\}$ est le multiensemble contenant 2 fois a, 1 fois b, 3 fois c et 0 fois tout autre valeur.

Le cardinal d'un multiensemble M est la somme de ses éléments comptée avec les nombre d'occurrences. Ainsi, le cardinal de $\{\!\{a, a, b, c, c, c\}\!\}$ est 6. Un multiensemble est fini si son cardinal est fini.

On trouve parfois le terme « sac » comme synonyme de « multiensemble » (en anglais, on trouve les termes

---

Exercice 27 *Donnez un type abstrait pour le type* `multiensemble` *des multiensembles dont les éléments sont de type* `typélt`.

Proposez des implantations de ce type abstrait.

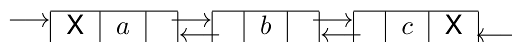
7.5 Les listes circulaires

Une liste circulaire est une liste chaînée dont le dernier élément point sur le premier (au lieu de pointer sur la liste vide).

Exercice 28 *Proposez des implantations du type des listes circulaires.*

7.6 Les listes bidirectionnelles

Les listes chaînées décrites dans le chapitre précédent correspondent à un chaînage monodirectionnel : du premier élément vers le dernier. Il arrive qu'on ait besoin de chaînage bidirectionnel, pour remonter à l'élément précédent ou aller vers l'élément suivant. On parle alors de liste bidirectionnelle. Par exemple, on peut avoir la liste bidirectionnelle suivante :



Une application des listes bidirectionnelles est l'implantation de files.

Exercice 29 *Proposez des implantations du type des listes bidirectionnelles.*

7.7 Les listes hétérogènes

Les listes vues au chapitre précédent sont homogènes : les éléments d'une telle liste sont tous du même type, on a des listes d'entiers naturels, des listes de booléens, des listes de réels, etc.

Une liste hétérogène est une structure de données dont les éléments sont potentiellement de différents types : `booléen`, `entier_nat`, `réel`, `pile`, `liste`, etc. Considérons la liste hétérogène suivante :

$$L = (\texttt{vrai}\ \ 12\ \ \texttt{"abc"}\ \ (4\ \ ()\ \ 12.14))$$

son troisième élément est une chaîne de caractères et son dernier élément est une liste hétérogène dont le deuxième élément est la liste vide.

LISP est un langage de programmation s'appuyant sur la structure de listes hétérogènes.

8 Les tables

8.1 Notion de table

Considérons un dictionnaire français-anglais. Ce dictionnaire peut être considéré comme une application qui à un mot en français associe un mot en anglais :

$$\texttt{"chat"} \mapsto \texttt{"cat"} \qquad \texttt{"chien"} \mapsto \texttt{"dog"} \qquad \texttt{"baleine"} \mapsto \texttt{"whale"}$$

De façon générale, une table est une représentation informatique d'une application finie $f : A \to B$. Le type des éléments de A est le type des entrées (noté `entrée`). Le type des éléments de B est le type des sorties (noté `sortie`).

On considérera également le type `donnée` : un type enregistrement dont les deux champs sont `ent` et `sor`. On pourra considérer qu'une table est un ensemble de valeurs de type `donnée`.

On suppose que les opérations dont les profils sont les suivants sont définies pour les types `entrée`, `sortie` et `donnée` :

- `Échec_sortie` : $\longrightarrow$ `sortie` (constante indiquant que la donnée correspondant à l'élément cherché n'existe pas).
- $=$: `entrée` $\times$ `entrée` $\longrightarrow$ `booléen` (égalité sur le type `entrée`).
- $=$: `sortie` $\times$ `sortie` $\longrightarrow$ `booléen` (égalité sur le type `sortie`).
- `lire_ent` : `donnée` $\longrightarrow$ `entrée` (accès en lecture au champ `ent` de `donnée`).
- `écrire_ent` : `entrée` $\times$ `donnée` $\longrightarrow$ `donnée` (accès en écriture au champ `ent` de `donnée`).
- `lire_sor` : `donnée` $\longrightarrow$ `sortie` (accès en lecture au champ `sor` de `donnée`).
- `écrire_sor` : `entrée` $\times$ `donnée` $\longrightarrow$ `donnée` (accès en écriture au champ `sor` de `donnée`).

Exercice 30 *On considère les objets suivants : un dictionnaire anglais-français, un dictionnaire français, un annuaire téléphonique, une table de logarithmes, un emploi du temps. Chacun de ces objets peut être représenté par une table. Précisez quel sont les types* `entrée` *et* `sortie` *pour chacune de ces tables.*

Ce chapitre décrit le type des tables et trois implantations de ce type.

8.2 Type abstrait

Le type abstrait des tables dont les entrées sont de type `entrée` et les sorties de type `sortie` est le suivant :

Nom : `table`;

Types abstraits importés `entrée`, `sortie`, `booléen`;

Opérations primitives

- Profils :
  - Constructeurs :
    `table_vide` : $\longrightarrow$ `pile`
    `ajouter` : `entrée` $\times$ `sortie` $\times$ `table` $\longrightarrow$ `table`
  - Accès :
    `est_vide` : `table` $\longrightarrow$ `booléen`
    `accès` : `entrée` $\times$ `table` $\longrightarrow$ `sortie`
    `supprimer` : `entrée` $\times$ `table` $\longrightarrow$ `table`
- Axiomes :

  [1] $\texttt{est_vide}(\texttt{table_vide}) = \texttt{vrai}$

  [2] $\texttt{est_vide}(\texttt{ajouter}(x, y, T)) = \texttt{faux}$

  [3] $\texttt{accès}(e, \texttt{table_vide}) = \texttt{Échec_sortie}$

  [4] $\texttt{accès}(e, \texttt{ajouter}(x, y, T))$
  = **Si** $e = x$ **Alors**
  | y
  Sinon
  | $\texttt{accès}(e, T)$
  Finsi

  [5] $\texttt{supprimer}(e, \texttt{table_vide}) = \texttt{table_vide}$

  [6] $\texttt{supprimer}(e, \texttt{ajouter}(x, y, T))$
  = **Si** $e = x$ **Alors**
  | $\texttt{supprimer}(e, T)$
  Sinon
  | $\texttt{ajouter}(x, y, \texttt{supprimer}(e, T))$
  Finsi

Par exemple, la table

$$\texttt{ajouter("baleine","whale",ajouter("chien","dog",ajouter("chat","cat",table_vide)))} \quad (8.1)$$

correspond au dictionnaire français-anglais donné en début de chapitre (section 8.1).

La suite du chapitre présente plusieurs implantations classiques des tables.

8.3 Implantation par une liste d'association (« en vrac »)

Le plus simple est sans doute de représenter une table par un ensemble fini de données. On parlera de « liste d'association ». Par exemple, si on note une donnée d par un couple (e, s) (les types respectifs de d, e et s étant `donnée`, `entrée` et `sortie`), la table (8.1) pourra être représentée par l'ensemble

$$\{(\texttt{"chat"}, \texttt{"cat"}), (\texttt{"chien"}, \texttt{"dog"}), (\texttt{"baleine"}, \texttt{"whale"})\}$$

Exercice 31 *En supposant donnée une implantation du type* `ensemble` *des ensembles finis de valeurs de type* `donnée`*, décrivez l'implantation du type table par une « liste d'association » (algorithmes des différentes opérations primitives).*

8.4 Implantation par un tableau trié

Une autre façon de représenter une table consiste à la représenter par une enregistrement à deux champs :
- Un champ `tab` de type tableau de N éléments de type `donnée`;
- Un champ `nb_ent` de type entier naturel (codant le nombre d'entrées dans la table).

Par ailleurs, `tab` doit être trié sur les `entrée` en fonction d'un ordre total noté $\leqslant$ pour les éléments d'indices inférieurs à n :

$$\text{pour tout } i \in \{0, 1, \ldots, n-2\}, \quad \texttt{lire_ent}(\texttt{tab}[i]) \leqslant \texttt{lire_ent}(\texttt{tab}[i+1])$$

Par exemple, la table (8.1) pourra être représentée par

`tab` =

("baleine", "whale")	("chat", "cat")	("chien", "dog")	X	X	X	X	X	X	X

`nb_ent` $= 3$

avec $N = 10$ et X représente une donnée quelconque (i.e., sa la valeur n'a pas d'importance). L'ordre $\leqslant$ pour cet exemple est l'ordre lexicographique sur les chaînes de caractères (i.e., l'ordre des mots du dictionnaire : `"baleine"` $\leqslant$ `"chat"` $\leqslant$ `"chien"`).

L'accès à un élément de la table se fait alors par une recherche *dichotomique*. Le principe est le suivant, pour le calcul de $\texttt{accès}(e, T)$:
- Soit a, b et m trois variables de type entier naturel et représentant des indices du tableau.
- à tout moment, l'élément cherché, s'il est dans le tableau, doit être à un indice i appartenant à l'intervalle $[a, b]$.
- Initialement, $a = 0$ et $b = \texttt{nb_ent} - 1$.
- On répète les opérations suivantes tant que $a < b$:
  - m prend la valeur de $(a + b)$ // 2 (m est le « milieu » de l'intervalle $[a, b]$).
  - f prend la valeur $\texttt{lire_ent}(\texttt{tab}[m])$ (la valeur de l'entrée à l'indice m du tableau).
  - Si $f = e$ alors l'élément est trouvé et la valeur retrouvée est $\texttt{lire_sor}(\texttt{tab}[m])$ (on quitte la fonction avec cette valeur).
  - Si $f < e$, alors cela signifie que e n'est pas dans la partie $[a, f]$ du tableau et donc, la partie restant à explorer est la partie $[f + 1, b]$. Cela conduit à faire l'affectation de a par $f + 1$.

— Si $f > e$, alors cela signifie que e n'est pas dans la partie $[f, b]$ du tableau et donc, la partie restant à explorer est la partie $[a, f-1]$. Cela conduit à faire l'affectation de b par $f-1$.

— En sortie de boucle, si $f = e$ alors l'élément est trouvé et retourné. Sinon, la fonction retournera la valeur `Échec_sortie`.

On peut montrer que le nombre de tests à effectuer est logarithmique. Autrement écrit : si on double la taille de la table (deux fois plus de données) alors le temps de calcul moyen passe de t à $t + K$ (où K est une constante).

Exercice 32 *Décrivez l'implantation du type table à l'aide d'un tableau trié (algorithmes des différentes opérations primitives).*

8.5 Implantation par une table de hachage (*hash table*)

Soit `ensemble`, le type des ensembles dont les éléments sont de type `donnée` (comme pour l'implantation en vrac). On considère un tableau `tab` de N éléments (où N « est grand »), chaque élément étant de type `ensemble`. L'ensemble des données de la table est l'union des `tab`$[i]$ ($0 \leqslant i \leqslant N$). Ainsi, la table vide est représenté par `tab` tel que `tab`$[i] =$ `ensemble_vide` pour tout i.

On se donne une fonction h, appelée *fonction de hachage*, qui à e de type `entrée` associe un indice $h(e) \in \{0, 1, \ldots, N-1\}$.

Le principe des tables de hachage est simplement le fait que la donnée $d = (s, e)$ de la table sera un élément de l'ensemble `tab`$[h(e)]$.

L'efficacité (rapidité du calcul) de la table de hachage dépend à la fois de N et de la fonction de hachage choisie. L'idée est que, pour deux entrées e_1, e_2 différentes, la situation $h(e_1) = h(e_2)$ soit rare [1]. Si tel est le cas et que le nombre de données de la table est inférieur à N alors, la plupart des ensembles `tab`$[i]$ contiendront peu d'éléments (mettons 0, 1 ou 2) et la recherche dans cet ensemble sera donc très rapide.

Il reste à choisir une bonne fonction de hachage. Si `entrée` est le type des chaînes de caractères, on choisit souvent la somme de leurs codes modulo N.

8.6 Autres implantations

Il existe d'autres implantations des tables.

L'une d'entre elle fait appel à la notion d'arbre binaire (cf. le cours d'AP3) et s'appelle « arbre binaire de recherche ».

8.7 Quelle implantation choisir ?

Le choix de l'implantation doit se faire en fonction :

— De la simplicité d'implantation (par exemple, pour une application utilisant de petites tables, une table en vrac est suffisante, cela dit, le plus simple est souvent l'utilisation d'une bibliothèque existante et efficace) ;

— Du besoin de rapidité (la table de hachage est une bonne solution mais peut nécessiter beaucoup de mémoire) ;

— Du mode d'utilisation (le tableau trié est une implantation efficace pour l'accès aux données, mais peut efficace pour l'ajout ou la suppression de données).

Une autre option consiste à interfacer le programme avec un SGBD (système de gestion de bases de données) qui stockera de façon efficace les données.

9 Calcul numérique sur les réels et les flottants

9.1 Les flottants et les réels

Dans les algorithmes, on peut manipuler des nombres réels. Dans les programmes C, les nombres réels sont représentés par des valeurs approchées : les « flottants », qui sont des nombres décimaux avec un nombre de chiffres significatifs fixé.

9.2 Les flottants en C

En C, deux types de base sont prédéfinis pour les flottants : `float` et `double`, avec une précision deux fois plus grande pour le deuxième que pour le premier : on utilisera de préférence `double`. Les opérations de base sur les flottants sont notées comme pour les entiers : `+`, `-`, `*`, `/`, `<`, `<=`, `>=`, `>`.

Attention : `/`, s'il est calculé entre entiers donne le quotient de la division euclidienne, alors que s'il est calculé entre flottants donne le quotient des deux flottants. Par exemple, `17/5` donne `3`, alors que `17./5.` donne `3.4`. Si `a` et `b` sont deux variables de type `int`, si on

1. Imposer que cette situation soit impossible revient à imposer que h soit injective, ce qui implique que le nombre de valeurs du type `entrée` soit inférieur ou égal à N. Si on considère le type `entrée` des mots sur l'alphabet $\{a, b, \ldots, z\}$ de longueurs inférieures ou égales à 10, le nombre de valeurs possible est $\displaystyle\sum_{i=0}^{i=10} 26^i = \frac{1-26^{11}}{1-26} \simeq 1,47 \cdot 10^{14}$ qui constitue une taille de tableau bien trop grande pour les machines actuelles !

veut calculer le quotient de `a` et `b`, considérés comme des réels, on peut procéder de la façon suivante : `((double)a)/((double)b)`. En effet, `(double)n` est la traduction de `n` en `double` (on parle de transtypage ou, en anglais, de *cast*).

Notons une précaution à prendre à cause des arrondis : pour tester l'égalité de deux flottants `a` et `b`, on teste si la valeur absolue de leur différence est suffisamment petite : `fabs(b - a) <= epsilon`, où `fabs` est la valeur absolue (définie dans `math.h`) et où `epsilon` est une constante représentant un « petit » réel (p. ex., `1e-10`).

9.3 Exercices

Les exercices de ce chapitre sont tous construits sur le même modèle. Ils consistent à rappeler (ou donner) un résultat de mathématiques appliquées et à demander de concevoir un algorithme et d'implanter un programme sur la base de ce résultat mathématiques.

Exercice 33 *Pour $x \in \mathbb{R}$, soit $S_n(x) = \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$. La suite $(S_n(x))_n$ converge vers $\sin(x)$, autrement dit $\sin x = \lim_{n\to+\infty} S_n(x)$.*
Écrire un algorithme de la fonction qui à l'entier naturel n et au réel x associe $S_n(x)$. Traduire cet algorithme en C.

On peut montrer, en s'appuyant sur la formule de Taylor-Lagrange, l'inégalité $|\sin x - S_n(x)| \leqslant \left|\frac{x^{2n+2}}{(2n+2)!}\right|$. Utiliser cette inégalité pour concevoir un algorithme qui à un réel x et à un réel strictement positif ε associe une valeur approchée de $\sin x$, à ε près.

Traduire cet algorithme en C.

***Indication.** On pourra reprendre l'algorithme précédent et changer la condition d'arrêt de la boucle.*

Comment tester ce dernier programme ?

Exercice 34 *Soit f une fonction dérivable de $[a;b]$, intervalle de $\mathbb{R}$, dans $\mathbb{R}$ telle que $|f'(x)| \leqslant K$ pour tout $x \in [a;b]$, où $0 < K < 1$. Par exemple : $f : x \in [0;1] \mapsto 1 - \frac{x^2}{4}$. On peut montrer qu'il existe un seul point fixe de f, i.e., une seule solution de l'équation $f(x) = x$. Soit à présent la suite de réels (u_n) définie par $u_{n+1} = f(u_n)$ et $u_0 = a$ (en fait, n'importe quelle valeur de $[a;b]$ pour u_0 aurait convenu). On peut montrer que (u_n) converge, que sa limite ℓ est le point fixe de f et que, pour tout n, $|\ell - u_n| \leqslant K^n(b-a)$.*

Écrire un algorithme qui calcule le point fixe de f avec une erreur majorée par $\varepsilon > 0$, paramètre de l'algorithme.

Traduire cet algorithme en C.

Comment tester ce programme ?

Exercice 35 *Soit f une fonction continue sur un intervalle $[a;b]$ de $\mathbb{R}$ ($a < b$) et à valeurs réelles telle que $f(a) < 0 < f(b)$. On cherche une solution approchée de l'équation $f(x) = 0$ (qui en a forcément une : cf. théorème des valeurs intermédiaires). La dichotomie est une méthode pour ce faire qui consiste à construire une suite d'intervalles $[a_n;b_n]$ dont la longueur $b_n - a_n$ diminue de moitié à chaque itération et qui contient une solution de l'équation. Dès que $b_n - a_n$ est inférieur à 2ε où $\varepsilon > 0$ est donné, le milieu de l'intervalle, $(a_n + b_n)/2$ est une solution à ε près de l'équation.*

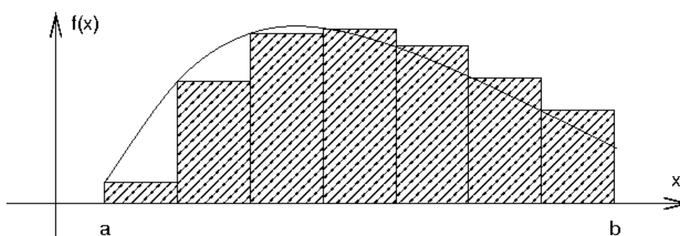
Pour construire les $[a_n;b_n]$, on procède de la façon suivante :

- *Pour $n = 0$, on pose $a_0 = a$ et $b_0 = b$.*
- *étant donné $[a_n;b_n]$, soit $M = \frac{a_n+b_n}{2}$. On définit $[a_{n+1};b_{n+1}]$ ainsi :*
  - *Si $f(M) < 0$, alors $[a_{n+1};b_{n+1}] = [M;b_n]$.*
  - *Si $f(M) > 0$, alors $[a_{n+1};b_{n+1}] = [a_n;M]$.*
  - *Si $f(M) = 0$ (cas rare !), alors M est une solution exacte de l'équation.*

Écrire un algorithme de résolution de $f(x) = 0$ par dichotomie. Traduire en C. Comment tester ce programme ?

Exercice 36 *Soit f une fonction intégrable (p. ex., continue) sur un intervalle $[a;b]$ de $\mathbb{R}$. Le calcul de $\int_a^b f(x)dx$ par la méthode des rectangles consiste*

1. *À découper l'intervalle $[a;b]$ en $n > 0$ intervalles $[a_i;a_{i+1}]$ avec $a_0 = a$, $a_1 = a + h$, $a_k = a + k \cdot h$ où $h = \frac{b-a}{n}$ (et donc, $a_n = b$).*
2. *À approcher f sur chaque intervalle $[a_i;a_{i+1}]$ par la valeur de $f(a_i)$. Notons $\widehat{f}$ la fonction en escalier ainsi obtenue.*
3. *À calculer $\int_a^b \widehat{f}(x)dx$, en faisant la somme des aires des rectangles de sommets de coordonnées respectives $(a_i, 0)$, $(a_{i+1}, 0)$, $(a_i, \widehat{f}(a_i))$ et $(a_{i+1}, \widehat{f}(a_i))$ (cf. représentation graphique, ci-dessous).*



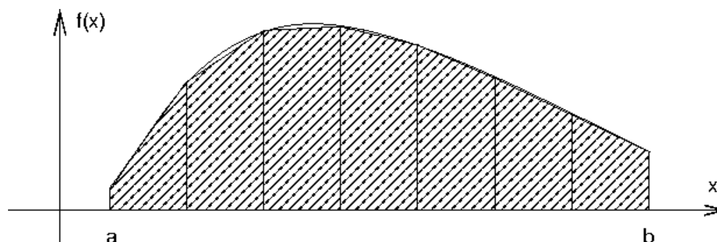
4. *À proposer cette valeur comme valeur approchée de l'intégrale de f entre a et b.*

Écrire un algorithme et un programme C effectuant ce calcul, la fonction f étant supposée donnée et le nombre d'intervalles n étant un paramètre de la fonction.

Si f est dérivable, on peut majorer l'erreur de la méthode des rectangles par $\frac{M_1(b-a)^2}{2n}$ où M_1 est un majorant de $\sup_{x\in[a;b]} |f'(x)|$. écrire un algorithme et un programme C qui effectue ce calcul, la fonction f et le majorant M étant supposés donnés et le paramètre étant $\varepsilon > 0$: un majorant de l'erreur admise (n n'est pas *un paramètre).*

Mettre au point un test de ce programme.

Exercice 37 *Le calcul de $\int_a^b f(x)dx$ par la méthode des trapèzes est similaire à celui suivant la méthode des rectangles (cf. exercice précédent), à la différence qu'on approche f par une fonction $\widehat{f}$ affine par morceaux et continue, chaque morceau correspondant à un intervalle $[a_i; a_{i+1}]$. Autrement dit, pour $i \in \{0; 1; \ldots; n-1\}$ et pour $x \in [a_i; a_{i+1}]$, $\widehat{f}(x) = f(a_i) + \frac{f(a_{i+1}) - f(a_i)}{h}(x - a_i)$ (cf. représentation graphique ci-dessous). En particulier, $\widehat{f}(a_i) = f(a_i)$.*



Écrire un algorithme et un programme C effectuant ce calcul, la fonction f étant supposée donnée et le nombre d'intervalles n étant un paramètre de la fonction. On rappelle que l'aire d'un trapèze de hauteur h et de bases B et b est $\frac{B+b}{2} \cdot h$.

Si f est deux fois dérivable, on peut majorer l'erreur de la méthode des trapèzes par $\frac{M_2(b-a)^3}{12n^2}$ où M_2 est un majorant de $\sup_{x\in[a;b]} |f''(x)|$. Écrire un algorithme et un programme C qui effectue ce calcul, la fonction f et le majorant M étant supposés donnés et le paramètre étant $\varepsilon > 0$: un majorant de l'erreur admise (n n'est pas *un paramètre).*

Mettre au point un test de ce programme.

10 Conclusion

Ce cours a eu pour objectif principal l'apprentissage de diverses notions d'algorithmique et de programmation mais aussi, et peut-être surtout, d'apprendre le savoir-faire suivant : si je suis capable d'effectuer un calcul à la main de façon systématique, je dois être capable de formaliser ce calcul par un algorithme et de le programmer, c'est-à-dire de l'« expliquer » à une machine.

Bien des notions liées à l'algorithmique et à la programmation doivent encore être étudiées, en particulier l'algorithme sur les arbres et les graphes ainsi que les techniques pour analyser un algorithme (terminaison, complexité, preuve de programmes, etc.).

A Algorithme et C : aide-mémoire

Cet document est destiné aux étudiants suivant le cours d'AP2 à Nancy (ou de quiconque y trouvera une utilité). Son objectif est de rappeler différentes notations pour l'écriture des algorithmes et leurs correspondances en C. On rappelle que certaines correspondances ne sont pas exactes, ainsi un nombre réel sera représenté de façon approximative par un nombre flottant. Ce document ne prétend nullement à l'exhaustivité.

A.1 Les commentaires

Notation algorithmique	C

`# commentaire sur`
`# plusieurs lignes`
	`// commentaire sur`
	`// plusieurs lignes`
	ou
	`/* commentaire sur`
	`   plusieurs lignes */`

A.2 Les variables

Types simples : (souvent, il existe plusieurs types et un choix a été fait) :

Notation algorithmique	**C**
`booléen`	`bool` (faire un `#include <stdbool.h>`)
`caractère`	`char`
`entier naturel`	`unsigned int` `size_t` (pour les indices d'un tableau, p. ex.)
`entier` (entier relatif)	`int`
`réel`	`double`

Déclaration de variables (sur un exemple) :

Notation algorithmique	**C**

Variables
`trouvé` : `booléen`
i, j : `entier`
x, y, z : `réel`
	`bool trouve ;`
	`int i, j ;`
	`double x, y, z ;`

Où faire la déclaration de variables :

Notation algorithmique	**C**
entre la signature de la fonction et le mot-clef **Début**	au début du corps de la fonction (après l'accolade ouvrante)

Affectation de variable (p. ex. : initialisation) :

Notation algorithmique	**C**
$a \leftarrow e$	`a = e`

où a (ou `a`) est un nom de variable et e (ou `e`) est une expression de même type que la variable.

A.3 Les constantes

Quelques constantes :

Notation algorithmique	**C**
`vrai`, `faux`	`true`, `false` (faire un `#include <stdbool.h>`)
π	`M_PI` (faire un `#include <math.h>`)
e $(\exp(1))$	`M_E` (faire un `#include <math.h>`)

Déclaration de constante en C : utiliser `#DEFINE`. Par exemple, pour déclarer que `CONST` vaut `1.234`, on écrit : `#DEFINE CONST 1.234`.

A.4 Les opérations de base sur les booléens, les entiers et les réels

Sur les booléens (voir aussi la section A.7, sur les conditionnelles) :

<table>
<tr><th>Notation algorithmique</th><th>C</th></tr>
<tr><td>et, ou, non</td><td><code>&&</code>, <code>||</code>, <code>!</code></td></tr>
<tr><td>$=, \neq, <, \leqslant, >, \geqslant$</td><td><code>==</code>, <code>!=</code>, <code><</code>, <code><=</code>, <code>></code>, <code>>=</code></td></tr>
</table>

Sur les entiers :

Notation algorithmique	**C**
$+, -, \times,$ `//`, `mod`	`+`, `-`, `*`, `/`, `%`
x, y : deux entiers x^y	`pow (x, y)` (faire un `#include <math.h>`)

Attention : la division est euclidienne (ou division entière, p. ex., 12 `//` 5 vaut 2). En C, si `x` et `y` sont des expressions de type entier (naturel ou relatif), `x / y` est le quotient de la division euclidienne de `x` par `y`.

Sur les réels (et les flottants) :

Notation algorithmique	**C**
$+, -, \times, /$	`+`, `-`, `*`, `/`
x^y	`pow (x, y)`
$\lvert x \rvert$	`fabs(x)`
$x = y$	`fabs(y - x) <= eps` où `eps` est une constante strictement positive mais « petite »
$\sin x$, $\cos x$, $\tan x$, $\arctan x$, etc.	`sin(x)`, `cos(x)`, `tan(x)`, `atan(x)`, etc.
e^x, x^y	`exp(x)`, `pow(x, y)`

x et y sont des expressions de type `réel`. Pour certaines opérations C, il faut faire un `#include <math.h>`.

Attention : en C, pour que `x / y` soit le résultat de la division entre flottants, il faut que `x` et/ou `y` soient des flottants. Ainsi `12. / 5.`, `12 / 5.` et `12. / 5` donnent `2.4` alors que `12 / 5` donne `2` (si `n` est une expression de type entier, `(double)n` est la traduction de `n` en `double`, exemple : `(double)(4 * 6)` donne `24.`).

A.5 La génération pseudo-aléatoire de nombres

Dans les exemples ci-dessous, x et a sont des variables, de types respectifs réel et entier :

Notation algorithmique	C
	(faire un #include <stdlib.h>)
$x \leftarrow$ réel choisi dans $[0.; 1.[$	x = ((double)rand() / (RAND_MAX)) ;
$x \leftarrow$ réel choisi dans $[10.; 100.[$	x = ((double)rand() / (RAND_MAX)) * 90. + 10. ;
$a \leftarrow$ entier choisi dans $\{0, \ldots, 100\}$	a = rand () % 101
$a \leftarrow$ entier choisi dans $\{-20, \ldots, 20\}$	a = rand () % 41 - 20

A.6 Les entrées/sorties

La procédure d'affichage se fait comme dans l'exemple suivant. n est une variable de type entier et $f(x)$ est une expression de type réel. On veut afficher n = <valeur de n> et f(x) = <valeur de f(x)>. Cela peut se faire de la façon suivante :

— Sans retour à la ligne après l'affichage :

Notation algorithmique	C
afficher("n =", n, "f(x) =", $f(x)$)	printf ("n = %d et f(x) = %f", n, f(x))

— Avec retour à la ligne après l'affichage :

Notation algorithmique	C
afficher("n =", n, "f(x) =", $f(x)$) retour_à_la_ligne ()	printf ("n = %d et f(x) = %f\n", n, f(x))

En C, les %<lettre> indiquent l'endroit où doit être inséré une valeur. La lettre dépend du type, par exemple, %c correspond à un caractère (un char), %u correspond à un entier naturel (p. ex., un unsigned int ou un size_t), %d correspond à un entier relatif (p. ex., un int), %f correspond à un flottant (p. ex., un double), %s correspond à une chaîne de caractères (un pointeur sur un char).

En Python 3, end="" indique qu'il ne faut rien afficher de plus en fin d'instruction (par défaut, end est la chaîne de caractères de retour à la ligne). De la même façon, si on veut éviter que les virgules introduisent des espaces supplémentaires, on ajoute sep="".

La fonction pour saisir une valeur sera utilisée *a minima* en AP2. En particulier, on évitera de l'utiliser dans les procédures de test.

Notation algorithmique	C

Saisie d'un entier :
$n \leftarrow$ **saisir**("Entrez n")
	n = 0 ; // initialisation de n
	r = 0 ; // résultat (succès/échec)
	printf ("Entrez n ") ;
	r = scanf ("%d", &n) ;

Saisie d'un réel :
$x \leftarrow$ **saisir**("Entrez x")
	x = 0 ; // initialisation de x
	r = 0 ; // résultat (succès/échec)
	printf ("Entrez x ") ;
	r = scanf ("%f", &x) ;

Saisie d'une chaîne de caractères :
$ch \leftarrow$ **saisir**("Entrez un texte")
	char ch[10] ; // initialisation de ch
	r = 0 ; // résultat (succès/échec)
	printf ("Entrez ch ") ;
	r = scanf ("%s", ch) ;

convertit la chaîne de caractères ch en entier (resp., en flottant, etc.).

A.7 Les conditionnelles

Instructions conditionnelles :

Notation algorithmique	C

Si <condition> **Alors**
│ <instructions>
Finsi
	if (<condition>)
	{
	<instructions1>
	}

Si <condition> **Alors**
│ <instructions1>
Sinon
│ <instructions2>
Finsi
	if (<condition>)
	{
	<instructions1>
	}
	else {
	<instructions2>
	}

où <condition> est une expression à valeur booléenne et <instructions>, <instructions1> et <instructions2> sont des séquences d'instructions.

Expressions conditionnelles :

Notation algorithmique	C

Si <condition> **Alors**
  <valeur1>
Sinon
  <valeur2>
Finsi
	<condition>
	  ? <valeur1>
	  : <valeur2>

où <condition> est une expression à valeur booléenne et <valeur1> et <valeur2> sont des expressions *de même type*. Le type de l'expression conditionnelle est le même type que celui de <valeur1> (et de <valeur2>).

A.8 Les boucles

Boucle tant que :

Notation algorithmique	C

Tant que <condition> **Faire**
  <instructions>
Fintantque
	while (<condition>)
	  {
	    <instructions>
	  }

<condition> est une expression à valeur booléenne, <instructions> est une séquence d'instructions.

Boucle pour :

Notation algorithmique	C

s'applique si $e_1 \leqslant e_2$:
Pour i **allant de** e_1 **à** e_2 **Faire**
  <instructions>
Finpour
	for (i = e1 ; i <= e2 ; i = i+1)
	  {
	    <instructions>
	  }

s'applique si $e_1 \geqslant e_2$:
Pour i **allant de** e_1 **à** e_2 *en descendant* **Faire**
  <instructions>
Finpour
	for (i = e1 ; i >= e2 ; i = i-1)
	  {
	    <instructions>
	  }

où i (ou i) : est une variable de type entier (ou entier naturel), et e_1 et e_2 (ou e1 et e2) sont des expressions à valeurs entières. En Python range(0, e2+1) s'écrit aussi range(e2+1).

A.9 Les fonctions et les procédures

A.9.1 Les fonctions et procédures avec passage de paramètre uniquement par valeur

Signature d'une fonction ou d'une procédure pour lesquels tous les passages de paramètres se font par valeur :

Notation algorithmique	C
Fonction nomf (a1 : t1, ..., an : tn) : t	t nomf (t1 a1, ..., tn an)
Procédure nomp (a1 : t1, ..., an : tn)	void nomp (t1 a1, ..., tn an)

où nomf est le nom de la fonction, nomp est le nom de la procédure, a1, ..., an sont les noms des paramètres, t1, ..., tn sont les types des paramètres et t est le type du résultat de la fonction.

Structure d'une fonction ou d'une procédure :

Notation algorithmique	C

<signature>
Début
  <instructions>
Fin
	<signature>
	{
	  <instructions>
	}

où <signature> est la signature de la fonction et <instructions> est une séquence d'instructions.

A.9.2 Les fonctions et procédures avec passage de paramètre par valeur ou par référence

Les passages de paramètre par référence induisent des effets de bord qu'il faut tenter d'éviter. Néanmoins, dans certains cas, il est plus pratique de passer outre ce principe.

Signature d'une fonction ou d'une procédure pour lesquels le paramètre a1 est passé par valeur et le paramètre a2 est passé par référence :

Notation algorithmique	C
Fonction nomf (a1 : t1, a2 : t2) : t	t nomf (t1 a1, t2 *a2)
Procédure nomp (a1 : t1, a2 : tn)	void nomp (t1 a1, t2 *a2)

Exemples d'appels à nomf et nomp :

Notation algorithmique	C

Variables
x1 : t1
x2 : t2
x : t
Début
x1 ← *une valeur de type* t1
x2 ← *une valeur de type* t2
# Les deux instructions ci-dessous sont 	
# susceptibles de modifier x2 mais pas x1. 	
x ← nomf(x1, x2)
nomp(x1, x2)
...
Fin
	{
	t1 x1 ;
	t2 x2 ;
	t x ;
	/* Les deux instructions ci-dessous sont
	susceptibles de modifier x2 mais pas x1. */
	x = nomf(x1, &x2) ;
	nomp(x1, &x2) ;
	...
	}

A.10 Les enregistrements

Déclaration d'un type enregistrement, sur l'exemple du type représentant les tomates (décrites par deux champs : est_mûre de type booléen et masse_en_g, de type réel) :

Notation algorithmique	C

Types
tomate_t : **enregistrement**
est_mûre : booléen
masse_en_g : réel
	struct TOMATE_T
	{
	bool est_mure ;
	double mass_en_g ;
	} ; // n'oubliez pas le point-virgule !
	
	typedef struct TOMATE_T tomate_t ;

Une fois un type enregistrement défini, une variable sur ce type peut être déclarée de la façon habituelle :

Notation algorithmique	C
Variables ma_tomate : tomate_t	tomate_t ma_tomate ;

En notation algorithmique et en C, la déclaration fait office d'initialisation (la valeur de type ma_tomate est créée). En Python, il faut faire une instanciation de la classe :

```
ma_tomate = tomate_t()
```

A.10.1 Manipulation directe des enregistrements (à éviter)

Si <x> est une valeur d'un type enregistrement dont <c> est un champ, l'expression « <x>.<c> » donne la valeur de <x> projetée sur le champ <c>. Par exemple, pour connaître la masse en grammes de ma_tomate, on écrit

```
ma_tomate.masse_en_g
```

expression de type réel / double / float.

A.10.2 Manipulation des enregistrements par fonctions de lecture et d'écriture

Principe (issu du principe d'encapsulation) : utilisation minimale de la notation « . ». Pour chaque champ <c>, on définit deux fonctions : lire_c et écrire_c et la notation avec un point est utilisée que dans le corps de ces fonctions et nulle part ailleurs. Exemple, pour le champ masse_en_g de l'enregistrement tomate_t :

Notation algorithmique	C

Fonction lire_masse_en_g (tom : tomate_t) : réel
Début
  retourner tom.masse_en_g
Fin
	`double lire_masse_en_g (tomate_t tom)`
	`{`
	`  return tom.masse_en_g ;`
	`}`

Fonction écrire_masse_en_g (m : réel, tom : tomate_t) : tomate_t
Variables
  $tom2$: tomate_t
Début
  $tom2 \leftarrow tom$
  $tom2$.masse_en_g $\leftarrow m$
  retourner $tom2$
Fin
	`tomate_t ecrire_masse_en_g`
	`          (double m, tomate_t tom)`
	`{`
	`  tom.masse_en_g= m ;`
	`  return tom ;`
	`}`

A.11 Les tableaux

A.11.1 Les tableaux à une entrée

Déclaration d'une variable tab de type tableau de n éléments de type ty, suivi d'un exemple pour la déclaration d'un tableau T de 10 réels :

Notation algorithmique	C
Variables tab : tableau de ty[n]	`ty tab[n];`
Variables T : tableau de réel[10]	`double T[10] ;`

L'initialisation d'un tableau T de 10 réels se fait de la façon suivante :

Notation algorithmique	C

La déclaration est supposée initialiser le tableau.
Cependant, la valeur de T[i] ($0 \leqslant i < 10$)
est inconnue.
Si on veut l'initialiser à 0., on peut faire :
Pour i **allant de** 0 **à** 9 **Faire**
  $T[i] \leftarrow 0.$
Finpour
	La déclaration est aussi une initialisation du tableau.
	Cependant, la valeur de T[i] ($0 \leqslant i < 10$) est inconnue.
	Si on veut l'initialiser à 0., on peut faire :
	`for (i = 0 ; i < 10 ; i = i + 1)`
	`  {`
	`    T[i] = 0. ;`
	`  }`

L'accès en lecture et en écriture au $i^{\text{ème}}$ élément du tableau tab se fait par :

$$tab[i]$$

(en notation algorithmique et en C).

Pour la notation algorithmique, le passage d'un tableau en paramètre d'une fonction ou d'une procédure se fait de façon analogue à la déclaration d'un tableau. En revanche, en C, il faut passer deux paramètres : un paramètre pour la taille du tableau (de type `size_t`) et un paramètre pour le tableau lui-même. Ainsi, la fonction C suivante calcule le produit des éléments d'un tableau `T` de `n` éléments de type `double` :

```
double produit_elements (size_t n, double T[n])
{
  size_t i ;
  double produit ;
  produit = 1. ;
  for (i = 0 ; i < n ; i = i + 1)
    {
      produit = produit + T[i] ;
    }
  return produit ;
}
```

A.11.2 Les tableaux à plusieurs entrées

Les tableaux à deux entrées, que ce soit en notation algorithmique ou en C seront représentés par des tableaux de tableaux. Ainsi, si M est un tableau de m tableaux de n réels, il représente une matrice $m \times n$ dont l'élément à la $i^{\text{ème}}$ ligne et à la $j^{\text{ème}}$ colonne est $M[i][j]$.