

INFO0947: Compléments de Programmation

Benoit Donnet, Simon Liénardy

Table des matières

1	Raisonnement Mathématique	7
1.1	Rappels sur la Logique et Quantificateurs	7
1.1.1	Formalisation d'Énoncé	7
1.1.2	Formalisation de Tableaux	7
1.1.3	Expressions Logiques avec Ensembles	7
1.2	Preuve par Récurrence	8
1.2.1	Preuves de Base	8
1.2.2	Exercice Complet	8
1.3	Relation d'Ordre	8
1.4	Induction	9
1.4.1	Notations	9
1.4.2	Définitions et exemples	9
1.4.3	Exercice	9
2	Construction de Programme	11
2.1	Assertions	11
2.1.1	Évaluation	11
2.2	Expression Schématique d'Assertions	11
2.3	Spécifications Formelles	12
2.3.1	Tableaux	12
2.3.2	Procédures/Fonctions	12
2.4	Assertions intermédiaires	13
2.5	Invariant	15
2.6	Construction de Programme	15
2.6.1	Problèmes Élémentaires	15
2.6.2	Tri par Insertion	16
3	Fichiers Séquentiels	17
3.1	Algorithmes sur un Seul Fichier	17
3.2	Algorithmes sur Plusieurs Fichiers	17
4	Récursivité	19
4.1	Compréhension	19
4.2	Construction de Fonctions Récursives	20
4.2.1	Exercices Simples	20
4.2.2	Suite de Lucas	20
4.2.3	Algorithmes sur des Tableaux	21
4.3	Calcul de Complexité	21

5	Types Abstraits	23
5.1	Définition de Types Abstraits	23
5.1.1	TAD Date	23
5.1.2	Les Entiers Naturels	23
5.1.3	Ensemble d'entiers positifs	23
5.2	Exercice Complet	24
6	Listes	25
6.1	Listes Simplement Chainées	25
6.1.1	Exercices Simples	25
6.1.2	Récurtivité	25
6.1.3	Gestion de Personnes	26
6.2	Listes Doublement Chainées	26
6.2.1	Exercices Simples	26
6.2.2	Ligne de Métro	27
7	Piles	29
7.1	Manipulation d'une Pile	29
7.2	Expressions	29
7.2.1	Écriture d'Expressions	29
7.2.2	Manipulation d'Expressions	30
7.2.3	Transformation d'Expressions	31
8	Files	33
8.1	Manipulation d'une File	33
8.2	Utilisation des Files	33
8.3	Implémentation des Files	34
9	Élimination de la Récurtivité	35
9.1	Chaines de Caractères	35
9.1.1	Opération belong_to	35
9.1.2	Opération occurrence	36
9.1.3	Opération reverse	36

Introduction

Le cours INFO0947 (Compléments de Programmation) fait suite au cours INFO0946 (Introduction à la Programmation). Les concepts vus, dès lors, dans le cours INFO0946 sont considérés comme connus et maîtrisés.

L'objectif du cours INFO0947 est de permettre d'étendre les connaissances en programmation sur deux plans :

1. Algorithmique. L'idée est d'améliorer les compétences de l'étudiant dans la compréhension et la mise en place d'algorithmes de complexité moyenne. A cette fin, l'approche constructive est mise en avant. L'idée est de se baser sur la spécification et l'invariant formels d'un problème et sur des assertions intermédiaires pour construire le code. La notion de récursivité est aussi introduite, ainsi que la complexité des fonctions récursives.
2. Structures de Données. Les structures de données vues jusqu'à présent sont assez limitées. Durant le cours, nous allons augmenter nos connaissances en structures de données. En particulier, la notion de type abstrait sera mise en avant. Les structures de données abordées sont des structures linéaires (listes, piles, files).

Ce document se veut être un recueil d'exercices permettant à l'étudiant de pratiquer les concepts vus durant le cours théorique. La pratique est un aspect important (si pas le plus important) du cours INFO0947. C'est dans cette optique que ce syllabus a été rédigé. Les compétences désirées à l'issue du cours ne peuvent s'obtenir sans une pratique intense. La compréhension et l'écoute lors du cours théorique sont des conditions nécessaires mais nullement suffisantes. Il faut s'exercer encore et encore de façon à acquérir l'expérience et l'aisance nécessaires à la réussite de l'examen.

En plus des séances d'exercices, les étudiants devront réaliser, durant le quadrimestre, plusieurs travaux.

Chapitre 1

Raisonnement Mathématique

1.1 Rappels sur la Logique et Quantificateurs

1.1.1 Formalisation d'Énoncé

Si $p(x)$ signifie “ x est un slug”, $q(x)$ signifie “ x aime le chocolat”, $r(x)$ signifie “ x n’est pas un chat” et $k(x, y)$ signifie “ x est plus intéressant que y ”, comment exprimer les propositions suivantes sous forme d’expressions logiques ?

Exercice 1 Xavier est un slug.

Exercice 2 Aucun slug n’aime le chocolat.

Exercice 3 Si tous les chats aiment le chocolat, alors aucun chat n’est un slug.

Exercice 4 Xavier est plus intéressant que tous les autres slugs.

1.1.2 Formalisation de Tableaux

Soient n, j et $k \in \mathbb{N}$ et $a[0 \dots n - 1]$ un tableau d’entiers. Dans les exercices qui suivent, il vous est demandé d’exprimer les propositions sous forme d’expressions logiques.

Exercice 1 Tous les éléments de $a[j \dots k]$ sont nuls.

Exercice 2 Tous les éléments pairs de $a[j \dots k]$ sont inférieur à 10.

Exercice 3 Tous les nombres pairs de $a[j \dots k]$ se trouvent à des emplacements pairs.

1.1.3 Expressions Logiques avec Ensembles

Soient n et $m \in \mathbb{N}$ et deux tableaux d’entiers $a[0 \dots n - 1]$ et $b[0 \dots m - 1]$. Dans les exercices qui suivent, il vous est demandé d’exprimer les propositions sous forme d’expressions logiques. Vous pouvez évidemment utiliser des notations “ensemblistes”

Exercice 1 x est le minimum de a .

Exercice 2 Si une valeur x apparaît dans a , elle ne peut pas apparaître dans b .

Exercice 3 Si une valeur x apparaît deux fois dans b , alors elle ne peut appartenir à a .

Exercice 4 Si une valeur paire apparaît dans a , alors elle apparaît à la même place dans b .

1.2 Preuve par Récurrence

1.2.1 Preuves de Base

Exercice 1 Prouver, par récurrence sur n , l'égalité suivante :

$$\sum_{i=1}^n i^2 = 1^2 + 2^2 + \dots + n^2 = \frac{n \times (n+1) \times (2n+1)}{6}, \forall n \in \mathbb{N}$$

Exercice 2 Prouver, par récurrence sur n , l'égalité suivante :

$$\sum_{i=0}^n x^i = \frac{x^{n+1} - 1}{x - 1}, \forall x \neq 1, \forall n \in \mathbb{N}$$

1.2.2 Exercice Complet

Si les formules du type p_i sont à valeurs dans $V = \{vrai, faux\}$, la formule $p_1 \Rightarrow p_2$ prend la valeur *vrai* pour trois éléments de V^2 correspondant au couple (p_1, p_2) .

De même, la formule $(p_1 \Rightarrow p_2) \Rightarrow p_3$ prend la valeur *vrai* pour 5 éléments de V^3 correspondant au triplet (p_1, p_2, p_3) .

Adoptons la notation Q_n pour désigner la formule $(\dots (p_1 \Rightarrow p_2) \Rightarrow \dots) \Rightarrow p_n$.

Il est demandé de prouver que, si T_n désigne le nombre d'éléments de V^n correspondant au n -uplet (p_1, \dots, p_n) qui donnent à la formule Q_n la valeur *vrai*, on obtient l'égalité suivante :

$$T_{n+1} + T_n = 2^{n+1}, \forall n \geq 2$$

et que, par conséquent, on a si n est pair :

$$T_N = \frac{2^{n+1} + 1}{3}$$

et si n est impair :

$$T_n = \frac{2^{n+1} - 1}{3}$$

1.3 Relation d'Ordre

Les espaces suivants sont-ils bien fondés ou non ? On demande, dans les cas négatifs, de fournir un exemple de suite "infinie décroissante".

Exercice 1 $(\mathbb{N}, <)$ où $<$ représente l'ordre strict usuel sur \mathbb{N} .

Exercice 2 (\mathbb{R}^+, \leq)

Exercice 3 $(\mathbb{L}, \mathcal{R})$ où \mathbb{L} désigne l'ensemble des listes finies de réels et \mathcal{R} la relation binaire définie par

$$\forall l, l' \in \mathbb{L}, l \mathcal{R} l' \Leftrightarrow (\text{longueur}(l) < \text{longueur}(l'))$$

1.4 Induction

1.4.1 Notations

On considère des suites finies de naturels. On note ces suites par des lettres majuscules éventuellement indicées. On note, par simple juxtaposition les opérations de concaténation des suites et d'ajout d'un élément en tête d'une suite.

Par exemple, si $n = 3$, $S_1 = (0, 7, 4)$ et $S_2 = (3, 9)$, on a

$$\begin{cases} nS_1 = (3, 0, 7, 4) \\ S_1S_2 = (0, 7, 4, 3, 9) \\ nS_2S_1 = (3, 3, 9, 0, 7, 4) \end{cases}$$

On dit que les suites finies S_1, \dots, S_n (prises dans cet ordre) constituent une décomposition de la suite finie S si et seulement si $S = S_1 \dots S_n$.

1.4.2 Définitions et exemples

On appelle *suite équilibrée* toute suite construite par applications répétées de la règle suivante. Si n est un naturel et si S_1, \dots, S_n sont des suites équilibrées déjà construites, la suite $nS_1 \dots S_n$ est équilibrée.

En particulier, la suite (0) est équilibrée car on l'obtient en appliquant la règle $n = 0$. Exemples de suites équilibrées :

$(2, 0, 0),$

$(1, 1, 0),$

$(5, 0, 1, 0, 2, 0, 0, 3, 1, 0, 0, 0, 0)$

1.4.3 Exercice

On demande de prouver les trois propriétés suivantes :

- La longueur d'une suite équilibrée est égale à la somme de ses éléments augmentée de 1. Autrement dit, si $S = (s_1, \dots, s_n)$ est équilibrée, on a $n = s_1 + s_2 + \dots + s_n + 1$.
- Soient $n \in \mathbb{N}, m \in \mathbb{N}_0$ et S et S' deux suites finies quelconques telles que $S = nS'$. Alors, S est décomposable en m suites équilibrées si et seulement si S' est décomposable en $m + n - 1$ suites équilibrées.
- Une suite finie de nombres naturels admet au plus une décomposition en suites équilibrées.

Chapitre 2

Construction de Programme

2.1 Assertions

2.1.1 Évaluation

Évaluez les assertions A_1 à A_6 par rapport aux situations concrètes $S_j^1 \wedge S_k^2 \wedge S^3$, avec $1 \leq j \leq 5$ et $1 \leq k \leq 2$, $a[0 \dots 5]$ étant un tableau d'entiers.

$$\begin{array}{lll} S_1^1 : i : \boxed{?} & S_1^2 : x : \boxed{?} & S^3 : a : \boxed{5} \boxed{-3} \boxed{2} \boxed{5} \boxed{-3} \boxed{2} \text{ (n=6)} \\ S_2^1 : i : \boxed{-1} & S_2^2 : x : \boxed{4} & \\ S_3^1 : i : \boxed{3} & & \\ S_4^1 : i : \boxed{5} & & \\ S_5^1 : i : \boxed{6} & & \end{array}$$

$A_1 : x$ initialisée

$A_2 : a[i]$ initialisé

$A_3 : (0 \leq i \leq n-1) \wedge (a[i] == x)$

$A_4 : (-1 \leq i \leq n) \wedge (a[i] == x)$

$A_5 : (-1 \leq i \leq n-1) \wedge (\forall j : i < j \leq n-1 : a[j] \neq x)$

$A_6 : (-1 \leq i \leq n-1) \wedge (\forall j : i \leq j \leq n-1 : a[j] \neq x)$

À chaque fois, on demande de fournir trois réponses : **T** (pour TTrue), **F** (pour FFalse) ou **?** (pour Undefined).

Pour ce faire, on complètera un tableau du même style que le tableau 2.1 (les couples en ordonnées représentent les différentes valeurs pour les paramètres j et k).

2.2 Expression Schématique d'Assertions

Exercice 1 Soit $a[0 \dots n-1]$ un tableau d'entiers. Exprimez les prédicats suivants à l'aide d'un schéma :

1. $0 \leq i \leq n-1 \wedge \forall j : 0 \leq j \leq i : a[j] \neq x$
2. $(0 \leq p \leq q \leq n) \wedge (\forall i : 0 \leq i \leq p-1 : a[i] \leq x) \wedge (\forall i : q < i \leq n-1 : a[i] > x)$
3. $(0 \leq i \leq j < n) \wedge (\forall k : (0 \leq k < i) \vee (j < k < n-1) : a[k] \leq a[k+1])$

	A_1 ... A_6
(1, 1)	
(1, 2)	
(2, 1)	
\vdots	
(5, 1)	

TABLE 2.1 – Évaluation d’Assertion

Exercice 2 Exprimez le schéma suivant sous forme de prédicat.

a :	0	i	$n-1$
	trié (croissant)	vide	

2.3 Spécifications Formelles

2.3.1 Tableaux

On se situe dans un programme contenant les déclarations suivantes :

```

1 const int N = ...;
2 const int M = ...;
3
4 int a[N];
5 int b[M];

```

Spécifiez complètement des procédures (ou fonctions) C correspondant aux problèmes suivants. Il vous appartient justement de les éliminer. Si nécessaire, n’hésitez pas à introduire des notations. Adoptez l’approche la plus formelle possible.

Exercice 1 Rechercher l’emplacement d’une valeur donnée dans $a[0 \dots N-1]$.

Exercice 2 Recherche l’emplacement d’une valeur donnée dans une portion du tableau a.

Exercice 3 Déplacer une portion du tableau a d’un nombre de pas donné vers la droite.

Exercice 4 Construire un tableau d’entiers strictement trié correspondant à l’union de deux tableaux d’entiers strictement triés.

2.3.2 Procédures/Fonctions

Spécifiez complètement des fonctions (ou procédures) C correspondant aux problèmes suivants. Si des définitions s’avèrent nécessaires ou simplement pertinentes, n’hésitez pas à introduire de « nouvelles notions ».

Exercice 1 Trouver la plus longue portion triée d’un tableau d’entiers.

Exercice 2 Trouver la plus longue sous-séquence d’un tableau présente dans un autre tableau.

2.4 Assertions intermédiaires

Voici une série d'extraits de code. En supposant que les assertions intermédiaires indiquées sont vraies, complétez les assertions manquantes (indiquées par des $\{\dots\}$). Si ce n'est pas précisé, les variables sont entières.

1.

```
1 // {a = ??}  
2 a = 1;  
3 // {...}
```

2.

```
1 // {a < 3}  
2 a += 1;  
3 // {...}
```

3.

```
1 // {x > 4}  
2 x /= 2;  
3 // {...}
```

4.

```
1 // {2 ≤ b < 7}  
2 b *= 3;  
3 // {...}
```

5.

```
1 // {Predicat(i)}  
2 ++i;  
3 // {...}
```

6.

```
1 // {∀k, 0 ≤ k < d, vect[k] = 0}  
2 vect[d] = 0;  
3 // {...}
```

7.

```
1 // {N > 0, tab[0, N - 1] init}  
2 for (int j = 0; j < N; ++j) {  
3     tab[j] = N;  
4 }  
5 // {...}
```

8.

```
1 // {m = max_{0 ≤ k < i} t[k]}
2 if (t[i] > m) {
3     // {...}
4     m = t[i]
5     // {...}
6 } else {
7     // {...}
8 }
9 // {...}
10 ++i;
11 // {...}
```

9.

```
1 // {N > 0, t[0, N-1] init}
2 int i = 0; j = N-1
3 while (j >= i) {
4     // {(∀k, 0 ≤ k < i, t[k] = k mod 2) ∧ (∀k, j < k < N, t[k] = k mod 2)}
5     t[j] = j % 2; t[i] = i % 2;
6     // {...}
7     j--; i++;
8     // {...}
9 }
10 // {...}
```

10.

```
1 int sum = 0, i = 0;
2
3 while (i <= N) {
4     // {sum = ∑_{j=0}^{i-1} j}
5     sum += i;
6     // {...}
7     ++i;
8     // {...}
9 }
10 // {...}
```

2.5 Invariant

Dans les exercices qui suivent, soyez le plus formel possible.

Exercice 1 Soit le bout de code suivant :

Pré : a initialisé

```
1 int i = -1;
2 int ou = 0;
3 while (!ou && i < n - 1) {
4     ou = a[i + 1];
5     i++;
6 }
```

Post : a inchangé; $ou = \bigvee_{i=0}^{n-1} a[i]$

Remarque : la notation $\bigvee_{i=0}^{n-1} a[i]$ est équivalente à $a[0] \vee a[1] \vee \dots \vee a[n-1]$. En particulier, on

a $\bigvee_{i=0}^{-1} a[i] = \text{False}$.

Donnez un invariant pour le bout de code ci-dessus.

Exercice 2 Soit $b[0 \dots m-1]$, un tableau d'entiers.

Pré : b et x initialisés

```
1 int i = 0, nbre = 0;
2 while (i < m) {
3     if (b[i] == x)
4         nbre++;
5     i++;
6 }
```

Post : b, x inchangés; $nbre = \#\{k | 0 \leq k \leq m-1; b[k] = x\}$

Donnez un invariant pour le bout de code ci-dessus.

2.6 Construction de Programme

2.6.1 Problèmes Élémentaires

Spécifiez et construisez, en suivant la méthode constructive vue au cours, des procédures (ou des fonctions) C permettant de résoudre les problèmes ci-après. Pour chacun de ces problèmes, on demande de trouver une résolution de **complexité linéaire**, i.e., une résolution ne parcourant qu'une seule fois chaque tableau.

Les variables a , b , et c sont de type `tab` qui est déclaré comme suit :

```
1 #define N 1000
2 typedef int tab[N];
```

Remarque : si de nouvelles notations ou définitions s'avèrent utiles, n'hésitez pas à les introduire.

Exercice 1 Trouver le maximum d'un préfixe donné du tableau a .

Exercice 2 Trouver l'emplacement de la plus grande valeur strictement inférieure à une valeur donnée dans un préfixe de a trié par ordre croissant.

Exercice 3 Soient un préfixe de a et un préfixe de b , tous deux strictement triés (par ordre croissant). Construire un tableau correspondant à « l'intersection » de ces deux préfixes (i.e., contenant les valeurs communes aux deux préfixes et uniquement elles).

Exercice 4 Tester si un préfixe donné de b constitue une sous-suite d'un préfixe donné de a (pour rappel si $s = (s_1, s_2, \dots, s_n)$, on appelle sous-suite de s toute suite de forme $(s_{i_1}, \dots, s_{i_m})$ avec $0 \leq m \leq n$ et $1 \leq i_1 < \dots < i_m \leq n$).

Exercice 5 Déterminer un sous-tableau de somme maximale dans un préfixe donné de a .

Exemple, pour le préfixe de a suivant :

0								9		N-1
-3	1	4	-2	-1	5	-7	0	2	-1	

le sous-tableau $a[1 \dots 5]$ convient avec une somme de 7.

2.6.2 Tri par Insertion

Il s'agit de construire un programme implémentant le *tri par insertion* pour un tableau d'entiers. Ce tri fonctionne comme suit : à chaque étape le tableau est divisé en deux parties ; la seconde partie est inchangée depuis le début tandis que la première est triée. Il s'agit alors « tout simplement » d'insérer le premier élément de la seconde partie « à sa place » dans la première partie.

Par exemple, considérons le tableau (de taille 5) suivant :

0				4
3	2	1	1	0

On obtiendra successivement les résultats suivants :

		On obtiendra successivement les tableaux suivants :				
— 1ère itération		0 4				
		3	2	1	1	0
— 2ème itération		0 4				
		2	3	1	1	0
— 3ème itération		0 4				
		1	2	3	1	0
— 4ème itération		0 4				
		1	1	2	3	0
— 5ème itération		0 4				
		0	1	1	2	3

Dans cet exercice, on demande de

1. spécifier précisément le problème ;
2. décomposer le problème en sous-problèmes ;
3. spécifier précisément chaque sous-problèmes ;
4. construire le morceau de programme correspondant à chaque sous-problème en utilisant l'approche constructive vue au cours ;
5. rédiger le morceau de programme final.

Chapitre 3

Fichiers Séquentiels

3.1 Algorithmes sur un Seul Fichier

Spécifiez et construisez, en suivant la méthode constructive vue au cours, des procédures (ou des fonctions) permettant de résoudre les problèmes ci-après. Vous devez utiliser le pseudo-langage vu durant le cours.

Exercice 1 Écrivez une fonction qui retourne le nombre d'éléments d'un fichier.

Exercice 2 Écrivez une fonction qui délivre la somme des éléments de rang pair d'un fichier de nombres entiers (i.e., $x_2 + x_4 + \dots$).

Exercice 3 Écrivez une fonction qui vérifie qu'un fichier possède au moins n éléments.

Exercice 4 Écrivez une fonction qui calcule le nombre d'occurrences de la valeur `val` entre le i^{e} et le j^{e} élément.

Exercice 5 Écrivez une fonction qui vérifie que le fichier f est trié dans l'ordre croissant, sans répétition.

3.2 Algorithmes sur Plusieurs Fichiers

Spécifiez et construisez, en suivant la méthode constructive vue au cours, des procédures (ou des fonctions) permettant de résoudre les problèmes ci-après. Vous devez utiliser le pseudo-langage vu durant le cours.

Exercice 1 Écrivez une fonction qui retourne la concaténation de deux fichiers.

Exercice 2 Écrivez une fonction qui éclate un fichier f en deux fichiers, le premier contenant les éléments de rang pair de f , l'autre les éléments de rang impair.

Par exemple, si $f = \langle 2, 3, 1, 6, 8, 5, 4 \rangle$, alors $f_{\text{pair}} = \langle 2, 6, 8, 4 \rangle$ et $f_{\text{impair}} = \langle 3, 1, 5 \rangle$.

Exercice 3 On appelle *monotonie croissante* d'un fichier $f = \langle x_1, x_2, \dots, x_n \rangle$ tout sous-fichier non vide trié par ordre croissant f_i^j tel que :

$$x_j > x_{j+1} \text{ pour } j < n$$

$$x_i < x_{i-1} \text{ pour } i > 1$$

Par analogie, on peut définir une *monotonie décroissante* d'un fichier $f = \langle x_1, x_2, \dots, x_n \rangle$ tout sous-fichier non vide trié par ordre décroissant f_i^j tel que

$$x_j < x_{j+1} \text{ pour } j < n$$

$$x_i > x_{i-1} \text{ pour } i > 1$$

Par exemple, le fichier $f = \langle 2, 10, -5, 18, 22, 14, 7, 16 \rangle$ contient les monotonies croissantes suivantes : (2, 10), (-5, 18, 22), (14) et (7, 16). Il contient les monotonies décroissantes suivantes : (2), (10, -5), (18), (22, 14, 7) et (16).

A partir de maintenant, on appellera *monotonie* une monotonie croissante.

Écrivez une fonction qui éclate le fichier f en deux fichiers $f1$ et $f2$ contenant respectivement les monotonies de rang impair de f et les monotonies de rang pair de f .

Par exemple, si $f = \langle 2, 10, -5, 18, 22, 14, 7, 16 \rangle$, on obtient $f1 = \langle 2, 10, 14 \rangle$ et $f2 = \langle -5, 18, 22, 7, 16 \rangle$.

Chapitre 4

Récurtivité

4.1 Compréhension

Exercice 1 Soit la fonction suivante

```
1 int f(int n){  
2     if(n == 0)  
3         return 1;  
4     else  
5         return f(n + 1);  
6 }//fin f()
```

La fonction $f()$ décrite ci-dessus est exécutée avec une valeur $n \geq 0$. Exécutez la fonction sur un exemple. Justifiez si cette fonction se termine, en général, ou non. Si oui, déterminez ce qu'elle fait ou calcule.

Exercice 2 Soit la fonction suivante

```
1 int f(int n){  
2     if(n == 0)  
3         return 0;  
4     else{  
5         int result = f(n - 1);  
6         result += n;  
7  
8         return result;  
9     }  
10 }//fin f()
```

La fonction $f()$ décrite ci-dessus est exécutée avec une valeur $n < 0$. Exécutez la fonction sur un exemple. Justifiez si cette fonction se termine, en général, ou non. Si oui, déterminez ce qu'elle fait ou calcule.

Exercice 3 Soit la fonction suivante :

```
1 int f(int n){  
2     if(n <= 1)  
3         return 1;  
4     else  
5         return 1 + f(n - 2);  
6 }//fin f()
```

La fonction $f()$ décrite ci-dessus est exécutée avec une valeur $n \geq 0$. Exécutez la fonction sur un exemple. Justifiez si cette fonction se termine, en général, ou non. Si oui, déterminez ce qu'elle fait ou calcule.

4.2 Construction de Fonctions Récursives

4.2.1 Exercices Simples

Pour tous les exercices qui suivent, vous devez appliquer, au moment de l'écriture du code, l'approche constructive telle que vue au cours.

Exercice 1 Exprimez de manière récursive la somme des nombres de 1 à n , avec $n > 0$. Spécifiez et construisez ensuite la fonction C correspondante.

Exercice 2 Soit a , un entier et n un entier positif ou nul. On se propose de calculer a^n en utilisant les 3 propriétés suivantes :

$$1. a^n = \begin{cases} a \times a^{n-1} & \text{si } n > 0, \\ 1 & \text{si } n = 0. \end{cases}$$

$$2. a^n = \begin{cases} (a^2)^{\lfloor n/2 \rfloor} & \text{si } n > 0 \text{ est pair,} \\ a \times (a^2)^{\lfloor n/2 \rfloor} & \text{si } n \text{ est impair,} \\ 1 & \text{si } n = 0. \end{cases}$$

$$3. a^n = \begin{cases} (a^{\lfloor n/2 \rfloor})^2 & \text{si } n > 0 \text{ est pair,} \\ a \times (a^{\lfloor n/2 \rfloor})^2 & \text{si } n \text{ est impair,} \\ 1 & \text{si } n = 0. \end{cases}$$

Spécifiez et construisez, pour chacune de ces trois propriétés, une fonction récursive qui retourne a^n .

Exercice 3 Spécifiez et construisez une fonction récursive qui calcule l'addition de deux entiers x et y en utilisant uniquement les opérations $+1$ et -1 . On rappelle que

$$x + y = \begin{cases} x & \text{si } y = 0, \\ (x + 1) + (y - 1) & \text{si } y > 0. \end{cases}$$

Écrivez ensuite un programme qui calcule et affiche la somme de 4 et 3. Enfin, représentez la pile d'appels à l'exécution du programme.

4.2.2 Suite de Lucas

On considère la suite de Lucas définie par $L_0 = 2, L_1 = 1$ et, pour $n > 0, L_{n+2} = L_{n+1} + L_n$. Spécifiez et écrivez une fonction récursive qui calcule L_n en fonction de n . On admet que

$$L_n = \left(\frac{1 + \sqrt{5}}{2} \right)^n + \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

4.2.3 Algorithmes sur des Tableaux

Pour tous les exercices qui suivent, vous devez appliquer, au moment de l'écriture du code, l'approche constructive telle que vue au cours.

Exercice 1 Spécifiez et construisez une fonction récursive qui retourne la valeur minimum d'un tableau d'entiers t .

Exercice 2 Spécifiez et construisez une fonction récursive renvoyant la position d'un entier donné dans un tableau t d'entiers donné (-1 si le nombre ne s'y trouve pas).

Exercice 3 Spécifiez et écrivez une fonction récursive qui détermine si une matrice carrée (d'entiers) donnée est symétrique, en vous basant sur le fait qu'une matrice carrée est symétrique si sa première ligne est égale à sa première colonne et si le reste de la matrice est symétrique.

4.3 Calcul de Complexité

Exercice 1 Soit la fonction récursive suivante :

```
1 int fonction(int n){
2     if (n == 0)
3         return 2;
4     else
5         return 2* fonction(n - 1);
6 }//fin fonction()
```

Déterminez le plus formellement possible la complexité de cette fonction.

Exercice 2 Soit la fonction récursive suivante :

```
1 void fonction(float x, float y, float z){
2     y = 2 * z;
3
4     if (x > x / (y - z)){
5         x -= 2;
6         y /= 4;
7         z /= 5;
8
9         fonction(x, y, z);
10    }
11 }//fin fonction()
```

Déterminez le plus formellement possible la complexité de cette fonction (Envisagez d'abord le cas où x , y et z sont positifs. Envisagez par la suite les autres cas).

Exercice 3 Soit la fonction récursive suivante :

```
1 int fonction(int n){
2     if(n == 0)
3         return 2;
4     else if (n == 1)
5         return 1;
6     else
7         return fonction(n - 1) + fonction(n - 2);
8 }//fin fonction()
```

Déterminez le plus formellement possible la complexité de cette fonction.

Exercice 4 Donnez la complexité pour toutes les fonctions récursives que vous avez définies à la Sec. 4.2.

Chapitre 5

Types Abstraits

5.1 Définition de Types Abstraits

5.1.1 TAD Date

Une date est un triplet de nombres ($\in \mathbb{N}$) où le premier indique le jour, le deuxième le mois, et le troisième l'année.

Exercice 1 Spécifiez complètement le TAD `Date`. On doit pouvoir, sur base d'une date, obtenir le jour, le moi, l'année, et, enfin, la date du lendemain.

Exercice 2 Proposez une structure de données concrètes permettant d'implémenter une `Date`.

5.1.2 Les Entiers Naturels

On désire représenter les entiers naturels. La suite d'entiers est infinie et dénombrable.

Exercice 1 Spécifiez complètement le TAD `Natural` qui représente les entiers naturels. On doit pouvoir créer un entier naturel, déterminer son successeur, son prédécesseur, additionner deux entiers et, enfin, déterminer si un entier naturel est égal à 0.

5.1.3 Ensemble d'entiers positifs

On souhaite représenter un *ensemble d'entiers positif* (i.e., nombre naturel). Ce type devra disposer des opérations suivantes :

- créer un ensemble vide ;
- déterminer si un ensemble est vide ;
- déterminer le cardinal (i.e., le nombre d'éléments) de l'ensemble ;
- déterminer si un ensemble est un sous-ensemble d'un autre ;
- déterminer si un élément appartient ou non à un ensemble ;
- déterminer si deux ensembles sont égaux ;
- déterminer l'union de deux ensembles d'entiers ;
- déterminer l'intersection de deux ensembles d'entiers ;
- ajouter un entier à l'ensemble ;
- retirer un entier de l'ensemble ;

À titre d'exercice supplémentaire Vous pouvez ajouter d'autres opérations sur les ensembles.

5.2 Exercice Complet

On désire définir un type abstrait permettant de manipuler un *nombre complexe*. Ce type abstrait sera appelé `Complex`.

Le type défini devra comporter des opérations pour additionner, soustraire, multiplier, et diviser deux nombres complexes, pour calculer le module et le conjugué d'un nombre complexe, pour indiquer la partie réelle et la partie imaginaire d'un nombre complexe ainsi qu'un constructeur donnant, à partir de deux réels a et b , le nombre complexe $a + bi$.

Exercice 1 Spécifiez complètement le type abstrait `Complex`.

Exercice 2 Proposez une structure de données permettant de représenter le type abstrait `Complex`.

Exercice 3 Écrivez le *header* C pour le type abstrait `Complex`. N'oubliez pas de spécifier, formellement, chaque fonction/procédure.

Exercice 4 Écrivez le module C implémentant le header défini à l'exercice 3.

Exercice 5 Écrivez un programme utilisant le type abstrait `Complex`.

Chapitre 6

Listes

6.1 Listes Simplement Chainées

Pour chacun des exercices de cette section, il est bien entendu demander d'appliquer la démarche constructive vue au cours (spécifications formelles et construction du code basée sur l'invariant). Si nécessaire, n'hésitez pas à introduire des notations.

6.1.1 Exercices Simples

Exercice 1 Écrivez un programme qui lit une suite de nombres au clavier et crée la liste chaînée correspondante. L'ordre des éléments dans la liste doit traduire l'ordre de lecture des nombres.

Exercice 2 On considère une liste représentée par son pointeur de début L . Écrivez une fonction qui prend une valeur X et L pour argument et retourne l'adresse de la cellule contenant X .

Exercice 3 On considère une liste, donnée par son pointeur de début L . Écrivez une fonction qui recopie cette liste et retourne le pointeur de début de cette nouvelle liste.

Exercice 4 On considère une liste, donnée par son pointeur de début L . Écrivez une fonction qui recopie cette liste en inversant l'ordre de ses éléments. La fonction retourne le pointeur de début de cette nouvelle liste.

6.1.2 Récursivité

Exercice 1 Spécifiez et construisez (de manière récursive) une fonction qui affiche à l'écran le contenu d'une liste simplement chaînée d'entiers.

Exercice 2 Spécifiez et construisez (de manière récursive) une fonction qui teste la présence d'une valeur donnée ($\in \mathbb{Z}$) dans une liste simplement chaînée d'entiers.

Exercice 3 Spécifiez et construisez (de manière récursive) une fonction qui détermine le nombre d'occurrences d'une valeur donnée ($\in \mathbb{Z}$) dans une liste simplement chaînée d'entiers.

6.1.3 Gestion de Personnes

On considère une personne comme ayant un nom (maximum 20 caractères) et un âge. On demande de créer une structure de données permettant de manipuler une liste simplement chaînée, triée par ordre croissant d'âge, de personnes.

Exercice 1 Écrire une fonction qui insère une personne dans une liste chaînée en fonction de son âge.

Exercice 2 Écrire une fonction qui détermine si une personne particulière est présente ou non dans la liste de personnes.

Exercice 3 Écrire une fonction qui enlève une personne particulière de la liste. La fonction retourne 1 en cas de succès, -1 si la personne n'est pas présente dans la liste.

Exercice 4 Écrire une fonction qui va retourner une liste de personnes correspondant à la fusion de deux listes données en argument. La liste résultante conserve, bien entendu, la propriété de tri en fonction de l'âge.

Exercice 5 Écrire une fonction qui imprime, dans un fichier, le contenu de la liste de personnes. Le fichier doit avoir la forme suivante :

NOM	AGE
Laurence	19
Catherine	20
Sébastien	21
Yvette	35

6.2 Listes Doublement Chainées

Pour chacun des exercices de cette section, il est bien entendu demander d'appliquer la démarche constructive vue au cours (spécifications formelles et construction du code basée sur l'invariant). Si nécessaire, n'hésitez pas à introduire des notations.

6.2.1 Exercices Simples

Exercice 1 Donnez les définitions de type nécessaire pour représenter une liste doublement chaînée de caractères.

Exercice 2 Écrivez une fonction qui prend en argument une liste doublement chaînée et qui détermine si cette liste correspond à un début de liste.

Exercice 3 Écrivez une fonction qui prend en argument une liste doublement chaînée et retourne un pointeur vers le début de la liste.

Exercice 4 Écrivez une procédure qui affiche à l'écran tous les éléments d'une liste doublement chaînée.

Exercice 5 Écrivez une fonction qui détermine, pour une variable pointant sur une liste doublement chaînée passée en argument, le nombre de cellules avant cette variable dans la liste. De même, écrivez une autre fonction qui calcule le nombre de cellules dans la liste après cette variable.

Exercice 6 Écrivez une fonction qui ajoute un élément en tête d’une liste doublement chaînée. De même, écrivez une fonction qui ajoute un élément en fin de liste.

6.2.2 Ligne de Métro

Exercice 1 Une station de métro a un nom. Une ligne de métro est constituée :

- d’un numéro (unique) permettant de l’identifier
- d’une liste (bi-directionnelle) de stations

Proposez des structures de données permettant de représenter une station de métro et une ligne de métro. Écrivez aussi toutes les fonctions nécessaires à la création de ces structures.

Exercice 2 Proposez une structure de données permettant de gérer l’ensemble des lignes de métro d’une ville. Pensez à écrire toutes les fonctions nécessaires à la création de la structure.

Exercice 3 Écrivez une fonction qui prend en argument une ligne de métro et une station et ajoute cette station en fin de ligne.

Exercice 4 Écrivez une fonction qui prend en argument une chaîne de caractères correspondant au chemin du fichier contenant une ligne de métro, charge les informations en mémoire et renvoie un pointeur vers la ligne de métro nouvellement créée.

On supposera que le format du fichier est le suivant :

- la première ligne est le numéro de la ligne de métro ;
- la deuxième ligne est le nombre de stations de la ligne ;
- jusqu’à la fin du fichier, un nom de station est écrit par ligne.

Exercice 5 Écrivez une fonction qui prend en entrée une ligne de métro et affiche son numéro et ses terminus.

Exercice 6 Écrivez une fonction qui permet de savoir si une station particulière appartient ou non à une ligne de métro donnée. On pensera à utiliser la fonction `strcmp` définie dans `string.h`.

Exercice 7 Écrivez une fonction qui prend en entrée une station de métro et affiche la liste des lignes de métro auxquelles elle appartient.

Exercice 8 Écrivez une fonction qui prend en argument deux stations de métro et calcule un trajet permettant d’aller de l’une à l’autre.

Chapitre 7

Piles

7.1 Manipulation d'une Pile

Exercice 1 Soit le bout de code suivant manipulant une pile d'entiers :

```
1 Stack *s = empty_stack();
2
3 s = push(s, 1);
4 s = push(s, 7);
5 s = push(s, 5);
6
7 int i = top(s);
8 printf("%d\n", i);
9 s = pop(s);
10
11 i = top(s);
12 printf("%d\n", i);
13 s = pop(s);
14
15 i = top(s);
16 printf("%d\n", i);
17 s = pop(s);
18
19 printf("%d\n", is_empty(s));
```

Donnez les états successifs de la pile *s* dans le bout de code supra.

7.2 Expressions

7.2.1 Écriture d'Expressions

Exercice 1 Écrivez sous forme postfixée l'expression $((((A + B) + D) > H) \wedge (E < F))$.

Exercice 2 Écrivez sous forme préfixée l'expression $((((A + B) + D) > H) \wedge (E < F))$.

7.2.2 Manipulation d'Expressions

Dans les exercices qui suivent, nous supposons que nous disposons d'une pile telle que spécifiée comme suit :

Type:
Stack

Utilise:

Boolean, Element

Opérations:

empty_stack: \rightarrow Stack

is_empty: Stack \rightarrow Boolean

push: Stack \times Element \rightarrow Stack

pop: Stack \rightarrow Stack

top: Stack \rightarrow Element

Préconditions:

$\forall s \in \text{Stack}, \forall e \in \text{Element}$

pop(s) is defined iff is_empty(s) = False

top(s) is defined iff is_empty(s) = False

Axiomes:

$\forall s \in \text{Stack}, \forall e \in \text{Element}$

is_empty(empty_stack) = True

is_empty(push(s, e)) = False

pop(push(s, e)) = s

top(push(s, e)) = e

Pour les exercices qui suivent, vous disposez aussi des fonctions/procédures suivantes :

- `opérateur(α)` qui détermine si α est un opérateur binaire ;
- `unaire(α)` qui détermine si α est un opérateur unaire ;
- `variable(α)` qui détermine si α est une variable ;
- `valeur(α)` qui retourne la valeur associée à la variable α .

Exercice 1 Spécifiez et construisez un algorithme utilisant une pile et permettant d'évaluer une expression écrite sous forme postfixée. Nous considérons que l'expression postfixée se présente comme un tableau de caractères.

Exercice 2 Soit les variables (A, B, C, D) et l'environnement (20, 4, 9, 7). Indiquez la trace de votre algorithme, sur votre pile, pour l'expression $AB \times CD + /$.

Exercice 3 Spécifiez et construisez un algorithme utilisant une pile et permettant d'évaluer une expression complètement parenthésée. Nous considérons que l'expression complètement parenthésée se présente comme un tableau de caractères.

Exercice 4 Spécifiez et construisez un algorithme utilisant une pile et permettant d'évaluer une expression écrite sous forme préfixée. Nous considérons que l'expression préfixée se présente comme un tableau de caractères.

Exercice 5 Donnez la trace de votre algorithme pour l'expression préfixée obtenue à l'Exercice 2 de la Sec. 7.2.1 avec l'environnement (4, 5, 1, 3, 2, 0) pour les variables (A, B, D, E, H, F)

7.2.3 Transformation d'Expressions

Nous avons vu que l'évaluation d'une expression est plus simple à partir d'une représentation postfixée. Cependant, une représentation parenthésée rend l'expression plus lisible.

Nous nous trouvons, ici, devant un aspect très simplifié du processus de compilation d'un langage de programmation qui ne serait formé que d'expressions complètement parenthésées : le

programme source fourni par l'utilisateur doit être transformé en une chaîne postfixée équivalente (*programme objet*) directement interprétable pour être évaluée.

Exercice 1 Construisez (via l'approche constructive) un algorithme qui permet de transformer une expression complètement parenthésée en une expression postfixée. Les deux expressions (i.e., ECP et EPOST) sont présentées comme des tableaux de caractères. Le signe '`\0`' indique la fin de l'expression.

Nous faisons ici l'hypothèse que l'ECP fournie en entrée est syntaxiquement correcte.

Exercice 2 Modifiez l'algorithme obtenu à l'Exercice 1 afin qu'il détecte les erreurs de syntaxe dans l'ECP fournie en entrée.

Chapitre 8

Files

8.1 Manipulation d'une File

Pour chacun des exercices qui suit, il est demandé d'évaluer l'expression à l'aide du type abstrait Queue tel que vu au cours.

Exercice 2 Soit l'expression

```
is_empty(enqueue(dequeue(enqueue(empty_queue, a)), b))
```

Évaluez cette expression à l'aide des axiomes du type abstrait Queue.

Exercice 2 Soit l'expression

```
head(dequeue(enqueue(enqueue(enqueue(empty_queue, a)), b), c))
```

Évaluez cette expression à l'aide des axiomes du type abstrait Queue.

8.2 Utilisation des Files

Soit le bout de code C suivant manipulant une file.

```
1 int i;  
2 Queue *q = empty_queue();  
3  
4 for (i=1; i<=8; i++)  
5     q = enqueue(q, i);  
6  
7 q = dequeue(q);  
8 q = dequeue(q);  
9  
10 for (i=0; i<3; i++)  
11     q = enqueue(q, i);  
12  
13 q = dequeue(q);  
14  
15 q = enqueue(q, 10);  
16 q = enqueue(q, 12);
```

Exercice 1 Supposons que la file utilisée dans le bout de code indiqué supra soit implémentée comme un tableau contigu à N cases numérotées de 0 à $N - 1$.

Supposons que $N = 10$. Quel sera le contenu de la file q après l'exécution du bout de code ? Indiquez les différentes étapes de l'évolution de q .

Exercice 2 Supposons que la file utilisée dans le bout de code indiqué supra soit implémentée comme un tableau circulaire à N cases numérotées de 0 à $N - 1$.

Supposons que $N = 10$. Quel sera le contenu de la file q après l'exécution du bout de code ? Indiquez les différentes étapes de l'évolution de q .

Exercice 3 Supposons que la file utilisée dans le bout de code indiqué supra soit implémentée de manière dynamique à l'aide de pointeurs (i.e., liste chaînée avec pointeur de début et de fin).

Quel sera le contenu de la file q après l'exécution du bout de code ? Indiquez les différentes étapes de l'évolution de q .

8.3 Implémentation des Files

Exercice 1 Montrez comment implémenter une file à l'aide de deux piles.

Exercice 2 Analysez le temps d'exécution des primitives de la file que vous venez d'implémenter.

Chapitre 9

Élimination de la Récursivité

9.1 Chaines de Caractères

L'objectif de ces exercices est de travailler sur les chaines de caractères. Pour ce faire, nous disposons d'un type abstrait `String` dont voici les opérations de base (nous n'indiquons pas ici les axiomes liés à ces opérations, ce n'est pas important pour les exercices qui suivent) :

Type:

`String`

Utilise:

`Integer, Char, Boolean`

Operations:

`empty_string: → String`

`is_empty: String → Boolean`

`first: String → Char`

`remainder: String → String`

`adj: String × Char → String`

`last: String → Char`

`except_last: String → String`

Préconditions:

$\forall s \in \text{String}$

`first(s)` is defined iff `is_empty(s) = False`

`remainder(s)` is defined iff `is_empty(s) = False`

`last(s)` is defined iff `is_empty(s) = False`

`remainder()` est l'opération qui enlève le premier caractère de la chaine (en paramètre) et retourne les caractères restants. Par exemple, `remainder("toto")` retourne "oto". `adj` est l'opération de concaténation. `last()` est l'opération qui retourne le dernier caractère de la chaine. `except_last()` est l'opération qui retourne la chaine amputée de son dernier caractère.

9.1.1 Opération `belong_to`

Soit l'opération `belong_to` qui retourne `True` si le caractère passé en paramètre appartient à la chaine. La définition est la suivante :

`belong_to: String × Char → Boolean`

Exercice 1 Spécifiez et construisez une implémentation récursive de l'opération `belong_to()`.

Exercice 2 Éliminez la récursivité dans l'algorithme obtenu à l'exercice 1. Attention, il n'est pas question ici de fournir un algorithme itératif mais bien d'éliminer la récursivité comme cela a été vu au cours.

9.1.2 Opération **occurrence**

Soit l'opération `occurrence` qui retourne un entier (positif) correspondant au nombre d'occurrences d'un caractère donné dans une chaîne. La définition est la suivante :

`occurrence: String × Char → Integer`

Exercice 1 Spécifiez et construisez une implémentation récursive de l'opération `occurrence()`.

Exercice 2 Éliminez la récursivité dans l'algorithme obtenu à l'exercice 1. Attention, il n'est pas question ici de fournir un algorithme itératif mais bien d'éliminer la récursivité comme cela a été vu au cours.

9.1.3 Opération **reverse**

Soit l'opération `reverse` qui retourne la chaîne passée en argument mais à l'envers. La définition est la suivante :

`reverse: String → String`

Exercice 1 Spécifiez et construisez une implémentation récursive de l'opération `reverse()`.

Exercice 2 Éliminez la récursivité dans l'algorithme obtenu à l'exercice 1. Attention, il n'est pas question ici de fournir un algorithme itératif mais bien d'éliminer la récursivité comme cela a été vu au cours.