

THINK JULIA

COMMENT PENSER COMME UN INFORMATICIEN

AUTEURS : BEN LAUWENS ET ALLEN B. DOWNEY

TRADUCTION FRANÇAISE ET AMÉNAGEMENTS : THIERRY LEPOINT

Version : B5-RV-1.1

Janvier 2021

Table des figures

1.2.1	REPL et invite de saisie (julia >).	3
2.1.1	Exemple d'un diagramme d'état.	12
3.9.1	Diagramme de pile.	29
3.14.1	Grille associée à l'exercice 3.14.3 (partie 1).	35
4.1.1	Déplacement de la tortue en avant (du centre vers la droite).	38
4.1.2	Un angle droit dessiné avec le module Turtle.	39
5.9.1	Diagramme de pile pour countdown appelé avec n = 3 .	60
6.5.1	Diagramme de pile associé à la fonction fact(n) .	76
7.10.1	Tableau produit par la fonction testsquareroot .	93
8.2.1	Exemple d'une table d'indices pour les séquences de caractères.	96
8.6.1	Schématisation du remplacement de lettre dans une chaîne de caractères.	101
10.2.1	Tableau nommé alpha et ses indices.	124
10.2.2	Diagrammes d'état pour 3 tableaux élémentaires.	125
10.10.1	Diagramme d'état et notion d'objets.	133
10.10.2	Diagramme d'état pour 2 tableaux équivalents mais non-identiques.	134
10.11.1	Diagramme d'état pour deux alias.	135
10.12.1	Diagramme de pile dans le cas d'un tableau passé en argument à une fonction.	136
11.1.1	Représentation du dictionnaire eng2fr .	146

11.5.1	Inversion de dictionnaire où les valeurs du dictionnaire inversé sont des tableaux.	154
11.5.2	Diagrammes d'état pour les fonctions hist et inverse	155
11.6.1	Graphe d'appel associé à la fonction fibonacci	156
12.6.1	Diagramme d'état associé au tuple (" Rabelais ", " François ")	173
12.6.2	Diagramme d'état associé au répertoire téléphonique.	174
13.12.1	Graphique « Plots » de la fonction $y = x^2$ pour $0 \leq x \leq 10$	193
15.1.1	Diagramme d'objet pour le type Point avec la valeur de ses attributs.	210
15.4.1	Diagramme d'objet pour l'objet Rectangle	213
16.1.1	Diagramme d'objet associé à MyTime	220
18.9.1	Diagramme de type et relations entre types.	250
19.2.1	Graphique de la fonction anonyme $x \rightarrow x^2 + 2x - 1$ avec Plots.	258
21.3.1	Écran de Visual Studio Codium.	301
21.3.2	Exemple d'une feuille de carnet Jupyter.	303

Liste des tableaux

2.1	Liste des mots-clés de Julia.	13
5.1	Liste des opérateurs relationnels.	55

Table des matières

Droits associés au présent document	xix
Préface de la version française	xxi
Traductions espagnole et portugaise	xxii
Remerciements	xxii
Copyright (version en anglais)	xxiii
Dédicace (version en anglais)	xxv
Préface (version en anglais)	xxviii
Pourquoi choisir Julia ?	xxviii
À qui ce livre s'adresse-t-il ?	xxviii
Conventions typographiques	xxix
Utilisation des exemples de codes (version en anglais)	xxix
Remerciements (version en anglais)	xxx
Liste des contributeurs (version en anglais)	xxxi
1 Mode de fonctionnement d'un programme	1
1.1 Qu'est-ce qu'un programme ?	2
1.2 Exécution de Julia	2
1.3 Un premier programme	3
1.4 Les opérateurs arithmétiques	3
1.5 Valeurs et types	4
1.6 Langages formels et naturels	5
1.7 Débogage	7
1.8 Glossaire	7
1.9 Exercices	8

1.9.1	Exercice	8
1.9.2	Exercice	9
2	Variables, expressions et déclarations	11
2.1	Déclaration d'affectation	11
2.2	Les noms de variables	12
2.3	Expressions et déclarations	13
2.4	Mode « script »	14
2.4.1	Exercice	15
2.5	Priorité des opérateurs	15
2.6	Opérations sur les chaînes de caractères	16
2.7	Commentaires	17
2.8	Débogage	18
2.9	Glossaire	18
2.10	Exercices	20
2.10.1	Exercice	20
2.10.2	Exercice	20
3	Les fonctions	21
3.1	Appel de fonction	21
3.2	Fonctions mathématiques	22
3.3	Composition	23
3.4	Ajout de nouvelles fonctions	24
3.5	Définition et usages	25
3.5.1	Exercice	25
3.6	Flux d'exécution	26
3.7	Paramètres et arguments	26
3.8	Les variables et les paramètres sont locaux	28
3.9	Diagrammes de pile	28
3.10	Fonction avec retour et fonction vide	30
3.11	Pourquoi utiliser des fonctions ?	31
3.12	Débogage	31
3.13	Glossaire	32
3.14	Exercices	33
3.14.1	Exercice	33
3.14.2	Exercice	33
3.14.3	Exercice	34
4	Étude de cas : Conception d'une interface	37

4.1	Turtles	37
4.1.1	Exercice	39
4.2	Répétitions simples	40
4.3	Exercices	41
4.3.1	Exercice	41
4.3.2	Exercice	41
4.3.3	Exercice	41
4.3.4	Exercice	41
4.3.5	Exercice	42
4.3.6	Exercice	42
4.4	Encapsulation	42
4.5	Généralisation	43
4.6	Conception d'une interface	44
4.7	Refonte (ou <i>refactoring</i>)	45
4.8	Plan de développement	47
4.9	Documentation interne	47
4.10	Débogage	48
4.11	Glossaire	49
4.12	Exercices	49
4.12.1	Exercice	49
4.12.2	Exercice	50
4.12.3	Exercice	51
4.12.4	Exercice	51
4.12.5	Exercice	51
5	Les conditions et la récursion	53
5.1	Division euclidienne (entière) et modulo	53
5.2	Expressions booléennes	54
5.3	Opérateurs logiques	55
5.4	Exécution conditionnelle	55
5.5	Exécution alternative	56
5.6	Enchaînement de conditions	56
5.7	Imbrication de conditions	57
5.8	Récursion	58
5.9	Diagrammes de pile pour les fonctions récursives	59
5.9.1	Exercice	60
5.10	Récursion infinie	60
5.11	Saisie au clavier	61
5.12	Débogage	62

5.13	Glossaire	63
5.14	Exercices	64
5.14.1	Exercice	64
5.14.2	Exercice	64
5.14.3	Exercice	65
5.14.4	Exercice	65
5.14.5	Exercice	66
6	Fonctions avec retour	67
6.1	Valeurs retournées	67
6.1.1	Exercice	69
6.2	Développement progressif	69
6.2.1	Exercice	71
6.3	Composition	71
6.4	Fonctions booléennes	72
6.4.1	Exercice	73
6.5	Davantage de récursion	73
6.6	Un acte de confiance	76
6.7	Un exemple supplémentaire	77
6.8	Types et vérification	77
6.9	Débogage	79
6.10	Glossaire	80
6.11	Exercices	80
6.11.1	Exercice	80
6.11.2	Exercice	81
6.11.3	Exercice	81
6.11.4	Exercice	82
6.11.5	Exercice	82
7	Itération	83
7.1	Réaffectation	83
7.2	Mise à jour des variables	84
7.3	while	85
7.3.1	Exercice	87
7.4	break	87
7.5	continue	88
7.6	Racines carrées	88
7.7	Algorithmes	90
7.8	Débogage	91

7.9	Glossaire	92
7.10	Exercices	92
7.10.1	Exercice	92
7.10.2	Exercice	92
7.10.3	Exercice	93
8	Chaînes	95
8.1	Caractères	95
8.2	Une chaîne est une séquence	96
8.3	<code>length</code>	97
8.4	Parcourir une chaîne	98
8.4.1	Exercice	98
8.4.2	Exercice	99
8.5	Segments de chaînes	99
8.5.1	Exercice	100
8.6	Les chaînes sont persistantes	100
8.7	Interpolation des chaînes	101
8.8	Recherche dans les chaînes	102
8.8.1	Exercice	102
8.9	Boucle et compteur	102
8.9.1	Exercice	103
8.10	Bibliothèque des chaînes	103
8.11	L'opérateur <code>∈</code>	104
8.12	Comparaison de chaînes	104
8.13	Débogage	105
8.13.1	Exercice	107
8.14	Glossaire	107
8.15	Exercices	108
8.15.1	Exercice	108
8.15.2	Exercice	108
8.15.3	Exercice	108
8.15.4	Exercice	109
8.15.5	Exercice	110
9	Étude de cas : jeux de mots	113
9.1	Lecture de listes de mots	113
9.2	Exercices	114
9.2.1	Exercice	114
9.2.2	Exercice	114

9.2.3	Exercice	114
9.2.4	Exercice	115
9.2.5	Exercice	115
9.2.6	Exercice	115
9.3	Recherche	115
9.4	Boucles sur les indices	117
9.5	Débogage	119
9.6	Glossaire	120
9.7	Exercices	120
9.7.1	Exercice	120
9.7.2	Exercice	121
9.7.3	Exercice	121
10	Tableaux	123
10.1	Un tableau est une séquence	123
10.2	Les tableaux sont non-persistants	124
10.3	Parcourir un tableau	126
10.4	Segments de tableau	127
10.5	Bibliothèque de fonctions associées aux tableaux	127
10.6	Mise en correspondance (<i>mapping</i>), filtre et réduction	128
10.7	Syntaxe avec <i>dots</i>	130
10.8	Suppression et insertion d'éléments	131
10.9	Tableaux et chaînes	132
10.10	Objets et valeurs	133
10.11	<i>Aliasing</i> ¹	134
10.12	Les tableaux en tant qu'argument	135
10.13	Débogage	138
10.14	Glossaire	139
10.15	Exercices	140
10.15.1	Exercice	140
10.15.2	Exercice	140
10.15.3	Exercice	141
10.15.4	Exercice	141
10.15.5	Exercice	141
10.15.6	Exercice	141
10.15.7	Exercice	142

1. La traduction exacte devrait être pseudonymie. Cependant, alias (c'est-à-dire pseudo ou pseudonyme) et aliasing sont des termes consacrés. En informatique, s'agissant de personnes, il est également fréquent de trouver l'acronyme *aka* pour *also known as*.

10.15.8 Exercice	142
10.15.9 Exercice	142
10.15.10 Exercice	142
10.15.11 Exercice	143
10.15.12 Exercice	143
11 Dictionnaires	145
11.1 Dictionnaire et « mise en correspondance » (<i>mapping</i>)	145
11.2 Les dictionnaires en tant que collections de compteurs	148
11.2.1 Exercice	151
11.3 Boucles et dictionnaires	151
11.4 Recherche inverse	152
11.5 Dictionnaires et tableaux	153
11.6 Mémos	155
11.7 Variables globales	157
11.8 Débogage	159
11.9 Glossaire	160
11.10 Exercices	161
11.10.1 Exercice	161
11.10.2 Exercice	161
11.10.3 Exercice	162
11.10.4 Exercice	162
11.10.5 Exercice	162
11.10.6 Exercice	162
12 Tuples	165
12.1 Les tuples sont persistants	165
12.2 Affectation des tuples	167
12.3 Les tuples en tant que valeurs retournées	168
12.4 Tuples et arguments multiples	169
12.4.1 Exercice	170
12.5 Tableaux et tuples	170
12.6 Dictionnaires et tuples	172
12.7 Séquences de séquences	174
12.8 Débogage	175
12.9 Glossaire	175
12.10 Exercices	176
12.10.1 Exercice	176
12.10.2 Exercice	176

12.10.3	Exercice	177
12.10.4	Exercice	177
13	Étude de cas : structures de données – choix	179
13.1	Analyse de la fréquence des mots	179
13.1.1	Exercice	179
13.1.2	Exercice	179
13.1.3	Exercice	180
13.1.4	Exercice	180
13.2	Nombres aléatoires	180
13.2.1	Exercice	181
13.3	Histogramme des mots	181
13.4	Mots les plus communs	183
13.5	Paramètres optionnels	184
13.6	Soustraction de dictionnaires	184
13.6.1	Exercice	185
13.7	Mots aléatoires	186
13.7.1	Exercice	186
13.8	Analyse de Markov	186
13.8.1	Exercice	188
13.9	Structure de données	188
13.10	Débogage	190
13.11	Glossaire	192
13.12	Exercices	192
13.12.1	Exercice	192
14	Fichiers	195
14.1	Persistance	195
14.2	Lire et écrire	196
14.3	Formatage	196
14.4	Noms de fichiers et chemins	197
14.5	Levée des exceptions	199
14.6	Bases de données	200
14.7	Sérialisation	202
14.8	Objets de commande	203
14.9	Modules	204
14.9.1	Exercice	205
14.10	Débogage	205
14.11	Glossaire	206

14.12 Exercices	207
14.12.1 Exercice	207
14.12.2 Exercice	207
14.12.3 Exercice	207
15 Structures et objets	209
15.1 Types composites	209
15.2 Les structures sont persistantes	210
15.3 Structures non-persistantes	211
15.4 Rectangles	212
15.5 Instances en arguments	213
15.5.1 Exercice	214
15.6 Instances en tant que valeurs retournées	215
15.7 Copies	215
15.7.1 Exercice	216
15.8 Débogage	216
15.9 Glossaire	217
15.10 Exercices	217
15.10.1 Exercice	217
15.10.2 Exercice	218
16 Structures et fonctions	219
16.1 Heures, minutes et secondes	219
16.1.1 Exercice	220
16.1.2 Exercice	220
16.2 Fonctions pures	220
16.3 Modificateurs	222
16.3.1 Exercice	223
16.3.2 Exercice	223
16.4 Prototypage ou planification ?	223
16.4.1 Exercice	224
16.5 Débogage	225
16.6 Glossaire	226
16.7 Exercices	226
16.7.1 Exercice	226
16.7.2 Exercice	226
17 Dispatch multiple	229
17.1 Déclarations de types	229

17.2	Méthodes	230
17.2.1	Exercice	231
17.3	Exemples supplémentaires	231
17.4	Constructeurs	233
17.5	show	234
17.5.1	Exercice	235
17.6	Surcharge d'opérateurs	235
17.7	Dispatch multiple	235
17.7.1	Exercice	236
17.8	Programmation générique (généricité)	237
17.9	Interface et implémentation	238
17.10	Débogage	239
17.11	Glossaire	239
17.12	Exercices	240
17.12.1	Exercice	240
17.12.2	Exercice	240
18	Sous-typage	241
18.1	Cartes	241
18.2	Variables globales	242
18.3	Comparaison de cartes	243
18.3.1	Exercice	244
18.4	Tests unitaires	244
18.5	Paquets de cartes	244
18.6	Ajouter, supprimer, mélanger et trier	245
18.6.1	Exercice	246
18.7	Types abstraits et sous-typage	246
18.8	Types abstraits et fonctions	248
18.9	Diagrammes de types	249
18.10	Débogage	250
18.11	Encapsulation de données	251
18.11.1	Exercice	253
18.12	Glossaire	253
18.13	Exercices	254
18.13.1	Exercice	254
18.13.2	Exercice	254
18.13.3	Exercice	254
19	Bonus : À propos de la syntaxe	257

19.1	Tuples nommés	257
19.2	Fonctions	258
19.2.1	Fonctions anonymes	258
19.2.2	Arguments nommés	259
19.2.3	Fermetures	259
19.3	Blocs	259
19.3.1	Bloc let	260
19.3.2	Blocs do	260
19.4	Structure de contrôle	261
19.4.1	Opérateur ternaire	261
19.4.2	Évaluation en court-circuit	261
19.4.3	Tâches ou co-routines	262
19.5	Types	263
19.5.1	Types primitifs	263
19.5.2	Types paramétriques	263
19.5.3	Unions de types	264
19.6	Méthodes	264
19.6.1	Méthodes paramétriques	264
19.6.2	Objets foncteurs	265
19.7	Constructeurs	265
19.8	Conversions et promotions	266
19.8.1	Conversion	266
19.8.2	Promotion	267
19.9	Méta-programmation	267
19.9.1	Expressions	267
19.9.2	Macros	268
19.9.3	Fonctions générées	269
19.10	Valeurs manquantes	269
19.11	Appel du code C et Fortran	270
19.12	Glossaire	270
20	Bonus : Bibliothèque de base et standard	273
20.1	Mesures de performance	273
20.2	Collections et structures de données	274
20.2.1	Exercice	276
20.3	Mathématiques	276
20.4	Chaînes	277
20.5	Tableaux	278
20.6	Interfaces	279

20.7	Utilitaires interactifs	281
20.8	Débogage	282
20.9	Glossaire	283
21	Débogage	285
21.1	Erreurs de syntaxe	286
21.1.1	Je continue à faire des changements mais sans effet	287
21.2	Erreurs d'exécution	287
21.2.1	Mon programme ne fait absolument rien	288
21.2.2	Mon programme est « suspendu »	288
21.2.3	Quand j'exécute mon programme, j'obtiens une exception	290
21.2.4	J'ai ajouté beaucoup de déclarations d'affichage ; je suis inondé de sorties	291
21.3	Erreurs sémantiques	291
21.3.1	Mon programme ne fonctionne pas	292
21.3.2	J'ai une expression embroussaillée qui ne fait pas ce que j'attends	293
21.3.3	J'ai une fonction qui ne retourne pas ce que j'attends	293
21.3.4	Je suis vraiment, vraiment coincé et j'ai besoin d'aide	294
21.3.5	Non, j'ai vraiment besoin d'aide	294
Annexe A	: entrées Unicode	297
Annexe B	: Installation de Julia	299
	Situation au 1er janvier 2021	299
	Julia en mode local	299
	Installation	300
	Installation de Visual Studio Codium	300
	Installation de modules	302
	Carnets de travail : Jupyter et Pluto	302
	Jupyter	302
	Pluto	302
	Informatique en nuage	303
	Aide et documentation	303
Annexe C	: Notes de traduction	306
Bibliographie		307
Index des fonctions extrinsèques		308

<i>TABLE DES MATIÈRES</i>	xvii
Index	312
Quatrième de couverture	327

Droits associés à la version française

En vertu de la licence Creative Commons (CC BY-NC-SA 3.0 FR ; *Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 3.0 FR*), vous êtes autorisé à partager (copier, distribuer et communiquer le matériel par tous moyens et sous tous formats) et adapter (remixer, transformer et créer à partir du matériel) selon les conditions suivantes :

- *Attribution* — Vous devez créditer le document, intégrer un lien vers la licence (en l'occurrence : CC BY-NC-SA 3.0 FR) et indiquer si des modifications ont été effectuées au document. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que le présent auteur vous soutient ou soutient la façon dont vous avez utilisé son travail.
- *Pas d'utilisation commerciale* — Vous n'êtes pas autorisé à faire un usage commercial du présent document, tout ou partie du matériel le composant.
- *Partage dans les mêmes conditions* — Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les mêmes conditions, c'est-à-dire avec la même licence avec laquelle le document original a été diffusé.

Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.



Préface de l'édition française

Initialement publié en 2012 par Alan Edelman, Stefan Karpinski, Jeff Bezanson et Viral B. Shah, Julia est un langage de programmation libre (licence MIT) dont les prémisses datent de 2009. Le code est arrivé à maturité en 2018 avec la version 1.0. Cependant, c'est seulement durant la période de confinement associée à la covid-19, au début mars 2020, que j'ai été véritablement attiré par Julia du fait qu'il s'agit d'un langage dont la syntaxe, aisée à comprendre, conduit rapidement, entre autre, à la résolution de problèmes scientifiques (mais aussi économiques et financiers), à l'apprentissage machine, à la manipulation de données de masse et à l'intelligence artificielle.

Au tout début mars, j'ai donc cherché un livre permettant d'apprendre le langage Julia. Parmi plusieurs ouvrages [1–5], l'un m'a séduit par sa qualité, sa rigueur et la progressivité de l'apprentissage sans s'encombrer de détails inutiles : *Think Julia – How to Think Like a Computer Scientist* de Ben Lauwens et Allen B. Downey [1]. Comme Julia demande à être diffusé aussi largement que possible compte tenu de ses qualités d'une part et que de l'autre il n'existe pas de livre en français pour « débutant », j'ai rapidement réalisé que s'imposait une traduction de *Think Julia – How to Think Like a Computer Scientist* – lui-même « portage » de *Think Python – How to Think Like a Computer Scientist* d'Allen B. Downey. Quelques aménagements ont été apportés ; ils sont mentionnés dans l'annexe C (page 305).

Traductions en espagnol et en portugais

Le lecteur trouvera également une traduction en espagnol sous [ce lien](#) réalisée par Pamela Alejandra Bustamante Faúndez (Ph.D. à la Pontificia Universidad Católica de Chile) ainsi qu'une traduction en portugais sous [ce lien](#) réalisée par Abel Soares Siqueira, Gustavo Sarturi, João Okimoto, Kally Chung. Abel Soares Siqueira est professeur à la Federal University of Paraná in Curitiba, PR, Brazil. Gustavo Sarturi est

titulaire d'une licence en mathématiques industrielles de l'Universidade Federal do Paraná (UFPR). João Okimoto est étudiant en informatique à l'UFPR et Kally Chung y est professeur d'ingénierie mécanique.

La coordination des traductions est assurée par Christian Peel.

Remerciements

Ma gratitude s'adresse aux développeurs du langage Julia travaillant inlassablement à l'efficacité et au rayonnement du langage ainsi qu'aux développeurs de la Free Software Foundation qui ont contribué aux logiciels LyX , \LaTeX , TeX , Inkscape et Gimp ainsi qu'aux fontes Times Roman et TeX Gyre DejaVu Math, sans lesquels ce document n'aurait pu être réalisé.

Enfin, mes plus vifs remerciements vont aux relecteurs et correcteurs aussi avisés que scrupuleux : Maryse Debrichy, et Claude Just.

Élouges, le 13 janvier 2021.

Thierry Lepoint
Site web : Chimie et Julia

Copyright (version en anglais)

Copyright © 2018 Allen Downey, Ben Lauwens. Tous droits réservés.

Think Julia est disponible sous la licence Creative Commons Attribution-NonCommercial 3.0 Unported License². Les auteurs maintiennent une version en ligne à l'adresse ThinkJulia.jl/latest/book.html.

Ben Lauwens est professeur de mathématiques à l'École royale militaire - Koninklijke Militaire School (RMA Belgium). Il est titulaire d'un doctorat en ingénierie et d'une maîtrise de la KU Leuven et de la RMA, ainsi que d'une licence de la RMA.

Allen B. Downey est professeur d'informatique à l'Olin College of Engineering. Il a enseigné au Wellesley College, au Colby College et à l'U.C. Berkeley. Il est titulaire d'un doctorat en informatique de l'U.C. Berkeley et d'un master et d'une licence du MIT.

Une version papier de ce livre est publiée par O'Reilly Media et peut être achetée sur Amazon.

2. ATTRIBUTION : Toutes les licences Creative Commons obligent ceux qui utilisent vos œuvres à vous créditer de la manière dont vous le demandez, sans pour autant suggérer que vous approuvez leur utilisation ou leur donner votre aval ou votre soutien.

PAS D'UTILISATION COMMERCIALE : vous autorisez les autres à reproduire, à diffuser et (à moins que vous ne choisissiez 'Pas de Modification') à modifier votre œuvre, pour toute utilisation autre que commerciale, à moins qu'ils obtiennent votre autorisation au préalable.

Dédicace (version en anglais)

Pour Emeline, Arnaud et Tibo.

Préface (version en anglais)

Ben Lauwens : *« En janvier 2018, j'ai commencé la préparation d'un cours de programmation destiné aux étudiants sans expérience en programmation. Je voulais utiliser Julia, mais j'ai découvert qu'il n'existait aucun livre ayant pour but d'apprendre à programmer avec Julia comme premier langage de programmation. Il existe de merveilleux tutoriels qui expliquent les concepts-clés de Julia, mais aucun d'entre eux n'accorde suffisamment d'attention à l'apprentissage concernant la façon de penser comme un programmeur.*

Je connaissais le livre « Think Python » d'Allen Downey, qui contient tous les ingrédients-clés pour apprendre à programmer correctement. Cependant, ce livre était basé sur le langage de programmation Python. Ma première ébauche de notes de cours était un melting-pot de toutes sortes d'ouvrages de référence, mais plus j'y travaillais, plus le contenu commençait à ressembler aux chapitres de « Think Python ». Bientôt, l'idée de développer mes notes de cours comme un portage de ce livre pour Julia s'est concrétisée.

Tout le matériel était disponible sous forme de carnets Jupyter dans un dépôt GitHub. Après avoir posté un message sur le site [Julia Discourses](#) concernant les progrès de mon cours, les réactions se sont avérées très positives. Un livre sur les concepts de base de la programmation avec Julia comme premier langage de programmation était apparemment un chaînon manquant dans l'univers de Julia. J'ai contacté Allen pour lui demander si je pouvais lancer un portage officiel de « Think Python » vers Julia, et sa réponse a été immédiate : « Allez-y ! » Il m'a mis en contact avec son éditeur chez O'Reilly Media, et un an plus tard, je mettais la dernière main à ce livre.

C'était un parcours semé d'embûches. En août 2018, Julia v1.0 est sortie et, comme tous mes collègues programmeurs de Julia, j'ai dû procéder à une migration du code. Tous les exemples du livre ont été testés lors de la conversion des fichiers sources en fichiers AsciiDoc compatibles avec O'Reilly. La chaîne d'outils et le code des exemples ont dû être rendus conformes à la version 1.0 de Julia. Heureusement, il n'y a pas de conférences à donner en août...

J'espère que ce livre vous plaira et qu'il vous aidera à apprendre à programmer et à penser comme un informaticien, au moins un peu. »

Pourquoi choisir Julia ?

Julia a été initialement publié en 2012 par Alan Edelman, Stefan Karpinski, Jeff Bezanson et Viral B. Shah. Il s'agit d'un langage de programmation libre et open source (sous licence MIT).

Le choix d'un langage de programmation est toujours subjectif. Les caractéristiques suivantes de Julia sont déterminantes :

- Julia est développé comme un langage de programmation performant,
- Julia utilise le *dispatch* multiple, ce qui permet au programmeur de choisir parmi différents modèles de programmation adaptés à l'application,
- Julia est un langage à typage dynamique qui peut facilement être utilisé de manière interactive,
- Julia a une jolie syntaxe de haut niveau, facile à apprendre,
- Julia est un langage de programmation à typage optionnel dont les types de données (définis par l'utilisateur) rendent le code plus clair et plus robuste,
- Julia dispose d'une bibliothèque standard étendue et de nombreux paquets tiers sont disponibles.

Julia est un langage de programmation unique car il résout le problème dit des « deux langages ». Aucun autre langage de programmation n'est nécessaire pour écrire un code performant. Cela ne signifie pas que cela se fait automatiquement. Il incombe au programmeur d'optimiser le code qui provoque un ralentissement à l'exécution, mais cela peut se faire dans Julia même.

À qui ce livre s'adresse-t-il ?


Ce livre est destiné à tous ceux qui veulent apprendre à programmer. Aucune connaissance préalable formelle n'est requise.

De nouveaux concepts sont introduits progressivement et des sujets plus avancés sont décrits dans les chapitres suivants.

Think Julia peut être utilisé pour un cours d'un semestre au niveau lycée ou université.

Conventions typographiques

italique les nouveaux termes ou des notions-clés sont indiqués en *italique*. Lorsqu'un terme anglais est repris, il est indiqué en *italique*.³

renvois les renvois à des chapitres, sections, sous-sections (voire sous-sous-sections), les renvois aux notes de bas de page et à la bibliographie ainsi que les liens hypertextes sont en gris et cliquables (sur ordinateur à l'aide de la souris (symbole )).

mots chasse fixe petit fonte utilisée pour le code des programmes.

mots chasse fixe bleu fonte faisant référence dans le texte aux éléments des programmes tels que les noms de variables ou de fonctions, les bases de données, les types de données, les variables d'environnement, les déclarations et les mots-clés.

Plusieurs éléments sont indiqués pour attirer l'attention du lecteur :

Conseil. Cette indication donne un conseil ou une suggestion.

Note. Cette mention apporte une information générale.

Avertissement. Cette information signale une mise en garde ou une situation où la prudence s'impose.

Utilisation des exemples de code (version en anglais)

Tout le code utilisé dans ce livre est disponible dans un dépôt Git sur GitHub : [BenLauwens/ThinkJulia.jl](https://github.com/BenLauwens/ThinkJulia.jl). Git est un système de contrôle de version qui permet de garder une trace des fichiers qui composent un projet. Une collection de fichiers sous le contrôle de Git est appelée un « dépôt ». GitHub est un service d'hébergement qui fournit un stockage pour les dépôts Git et une interface web pratique.⁴

Compte tenu de modifications récentes (postérieures à la publication de Think Julia [1]) concernant directement l'environnement de développement intégré de Julia (Atom/Juno) et de l'abandon de JuliaBox, le traducteur se permet de renvoyer le lecteur à l'annexe 21.3.5 pour l'installation de Julia, de ses modules, de Visual Studio Codium et l'usage de carnets (*notebooks*) d'utilisation sur le web ou en mode local.

3. Dans la traduction française, les locutions latines sont en italique.

4. Pour créer un compte GitHub, référez-vous à Comment créer un compte GitHub. Pour installer Git (MS-Windows, MacOS et GNU/LINUX), consulter Installing Git – The easy way.

Remerciements (version en anglais)

Je tiens à remercier Allen B. Downey pour avoir rédigé *Think Python*, *How to Think Like a Computer Scientist* et de m'avoir permis d'effectuer un portage de son livre en Julia. Son enthousiasme est contagieux !

J'aimerais également remercier les relecteurs/correcteurs de ce livre, qui ont fait de nombreuses suggestions utiles : Tim Besard, Bart Janssens et David P. Sanders.

Merci à Melissa Potter de O'Reilly Media, qui a contribué à améliorer ce livre. Vous m'avez forcé à faire les choses correctement et à rendre ce livre aussi original que possible.

Merci à Matt Hacker de O'Reilly Media, qui m'a aidé avec la chaîne d'outils Atlas et qui a résolu quelques problèmes de syntaxe.

Merci à tous les étudiants ayant travaillé avec une première version de ce livre et à tous les contributeurs (cités ci-dessous) qui ont envoyé des corrections et des suggestions.

Liste des contributeurs (version en anglais)

Si vous avez une suggestion ou une correction, veuillez envoyer un courriel à Ben Lauwens ou ouvrir un numéro sur GitHub. Si je fais une modification sur la base de vos commentaires, je vous ajouterai à la liste des contributeurs (sauf si vous demandez à être omis).

Faites-moi savoir avec quelle version du livre vous travaillez et avec quel format. Si vous incluez au moins une partie de la phrase dans laquelle l'erreur apparaît, cela facilitera la recherche. Les numéros de page et de section sont également utiles quoique peu aisés à utiliser.

- Scott Jones a signalé le changement de nom de `Void` en `Nothing`, ce qui a lancé la migration vers Julia v1.0.
- Robin Deits a trouvé quelques fautes de frappe dans le chapitre 2 (Variables, Expressions et Déclarations).
- Mark Schmitz a suggéré d'activer la mise en évidence syntaxique,
- Zigu Zhao a détecté des bogues dans le chapitre 8 (Chaînes),
- Oleg Soloviev a détecté une erreur dans l'URL d'ajout du paquet `ThinkJulia` (chapitre 4),
- Aaron Ang a trouvé des problèmes de rendu et de dénomination,
- Sergey Volkov a détecté un lien brisé dans le chapitre 7 (Itération),
- Sean McAllister a suggéré de mentionner l'excellent paquet `BenchmarkTools`,
- Carlos Bolech a envoyé une longue liste de corrections et de suggestions,

— Krishna Kumar a corrigé l'exemple de Markov dans le chapitre 18 (Sous-typage).

Chapitre 1

Mode de fonctionnement d'un programme

Le but de ce livre est d'apprendre au lecteur à penser comme un informaticien. Cette façon de penser combine certaines des meilleures caractéristiques des mathématiques, de l'ingénierie et des sciences naturelles. Comme les mathématiciens, les informaticiens utilisent des langages formels pour expliciter des idées (en particulier les calculs). Tout comme les ingénieurs, ils conçoivent des structures, assemblent des composants dans divers systèmes et évaluent le meilleur compromis entre différentes possibilités. Comme les scientifiques, ils observent le comportement de systèmes complexes, formulent des hypothèses et vérifient leurs prédictions.

La compétence la plus importante pour un informaticien est la résolution de problèmes, c'est-à-dire la capacité à formuler ces problèmes, à penser de manière créative aux solutions et à exprimer la meilleure solution de manière claire et précise. Il s'avère que le processus d'apprentissage de la programmation est une excellente occasion de mettre en pratique les compétences nécessaires à la résolution de problèmes. C'est pourquoi ce chapitre s'intitule « Mode de fonctionnement d'un programme ».

À un certain niveau, le lecteur apprendra à programmer, une compétence utile en soi. À un autre niveau, la programmation pourra être exploitée comme un moyen d'atteindre un but. Au fur et à mesure, cette fin deviendra plus claire.

1.1 Qu'est-ce qu'un programme ?

Un programme est une séquence d'instructions qui établit comment effectuer un calcul. Le calcul peut être mathématique, comme la résolution d'un système d'équations ou la recherche des racines d'un polynôme, mais il peut aussi être symbolique, comme la recherche et le remplacement de texte dans un document, ou encore graphique, comme le traitement d'une image ou la lecture d'une vidéo.

Les détails sont différents selon les langages, mais quelques instructions élémentaires apparaissent dans presque chaque langage :

Données en entrée : il s'agit d'incorporer des données en provenance du clavier, d'un fichier, du réseau ou d'un autre appareil,

Sortie : il s'agit d'afficher les données à l'écran, les enregistrer dans un fichier, les envoyer sur le réseau, etc.,

Mathématiques : il s'agit d'effectuer des opérations mathématiques de base comme l'addition, la multiplication, etc.,

Exécution conditionnelle : il s'agit de vérifier certaines conditions et, en conséquence, de lancer le code approprié,

Répétition : il s'agit d'effectuer une action de manière répétée, généralement avec une variation donnée,

C'est à peu près tout. Chaque programme que nous avons utilisé, aussi complexe soit-il, est composé d'instructions qui ressemblent à ce qui est décrit précédemment. Nous considérerons donc la programmation comme le processus de décomposition d'une tâche ample et complexe en sous-tâches de plus en plus petites, jusqu'à ce que les sous-tâches soient suffisamment simples pour être exécutées à l'aide de ces instructions de base.

1.2 Exécution de Julia

Après avoir lancé Visual Studio Codium (voir l'annexe 21.3.5), nous arrivons sur une page telle que celle de la figure 1.2.1.

Le REPL (*Read-Eval-Print Loop*) de Julia est un programme qui lit et exécute le code Julia. Le REPL est exécutable en ouvrant un terminal sur JuliaBox et en saisissant julia sur la ligne de commande. Quand le REPL démarre, nous devrions voir l'information telle que reprise dans figure 1.2.1.

Les premières lignes contiennent des informations sur le REPL, il se peut donc que ce soit un peu différent. Il faut vérifier que le numéro de version est au moins 1.0.0.

```
Press Enter to start Julia.
[ Info: Precompiling OhMyREPL [5fb14364-9ced-5910-84b2-373655c76a03]
Starting Julia...
[ Info: Precompiling Atom [c52e3926-4ff0-5f6e-af25-54175e0327b1]

Documentation: https://docs.julialang.org
Type "?" for help, "]" for Pkg help.
Version 1.5.3 (2020-11-09)
Official https://julialang.org/ release

julia> 
```

FIGURE 1.2.1 – REPL et invite de saisie (julia>).

La dernière ligne est une invite qui indique que le REPL est prêt pour que nous puissions saisir du code.

Si nous saisissons une ligne de code et appuyons sur ENTER, le REPL affiche le résultat :

```
julia> 1 + 1
2
```

1.3 Un premier programme

Traditionnellement, le premier programme à écrire dans un nouveau langage informatique s'appelle « Hello, World ! » bien qu'il n'affiche que l'expression Hello, World !. Avec Julia, le code se présente comme ceci :

```
julia> println("Hello, World!")
Hello, World!
```

Il s'agit d'un exemple d'instruction permettant un affichage (à l'écran) avec retour à la ligne. Les guillemets droits marquent le début et la fin du texte à afficher ; ils n'apparaissent pas dans le résultat. Les parenthèses indiquent que `println` est une fonction (voir le chapitre 3).

1.4 Les opérateurs arithmétiques

L'étape suivante concerne l'arithmétique. Julia fournit des opérateurs de multiplication, division, addition, soustraction. Les opérateurs `+`, `-`, et `*` effectuent des additions, des soustractions et des multiplications, comme dans les exemples suivants :

```
julia> 40 + 2
42
```

```
julia> 43 - 1
42
julia> 6 * 7
42
```

L'opérateur `/` effectue la division :

```
julia> 84 / 2
42.0
```

Pourquoi le résultat est-il 42.0 au lieu de 42 ? Ceci est expliqué dans la section 1.5. Enfin, l'opérateur `^` élève un nombre à une puissance donnée :

```
julia> 6^2 + 6
42
```

1.5 Valeurs et types

Une valeur est un des éléments de base avec lesquels un programme fonctionne, comme une lettre ou un nombre. Certaines des valeurs que nous avons vues jusqu'ici sont `2`, `42.0` et `"Hello, World!"`.

Ces valeurs appartiennent à différents types : `2` est un nombre entier, `42.0` un nombre à virgule flottante et `"Hello, World!"` une chaîne de caractères.

S'il y a un doute sur le *type* d'une valeur, le REPL peut fournir des renseignements :

```
julia> typeof(2)
Int64
julia> typeof(42.0)
Float64
julia> typeof("Hello, World!")
String
```

Les nombres entiers appartiennent au type `Int64`, les chaînes de caractères au type `String`, et les nombres à virgule flottante au type `Float64`.

Qu'en est-il des valeurs comme `"2"` et `"42.0"` ? Elles ressemblent à des nombres, mais comme elles sont entre guillemets, il s'agit de chaînes de caractères :

```
julia> typeof("2")
String
julia> typeof("42.0")
String
```

Lorsque des nombres constitués de nombreux chiffres sont saisis, il peut être tentant d'utiliser des virgules entre les groupes de chiffres, comme pour exprimer un million en notation anglo-saxonne : 1,000,000. Ceci n'est pas un nombre *entier* autorisé en langage Julia, bien que cette représentation soit licite :

```
julia> 1,000,000
(1, 0, 0)
```

Évidemment, ce n'est pas du tout le résultat espéré. Julia analyse 1,000,000 comme une suite d'entiers, à savoir 1, 0 et 0, séparés par des virgules. Ce type de séquences sera analysé plus tard. Il est cependant possible d'obtenir le résultat recherché en utilisant 1_000_000.

1.6 Langages formels et naturels

Les langages naturels sont ceux que les êtres humains utilisent quotidiennement, comme l'anglais, l'espagnol, le français, etc. Ils n'ont pas été conçus d'un bloc par leurs utilisateurs parce qu'ils ont évolué naturellement et progressivement.

A contrario, les langages formels sont conçus pour des applications spécifiques. Par exemple, la notation utilisée par les mathématiciens est un langage formel particulièrement efficace pour établir les relations entre les nombres et les symboles. Les chimistes utilisent un langage formel pour représenter la structure chimique des molécules et les réactions. Les langages de programmation sont des langages formels qui ont été conçus pour effectuer des calculs.

Le cœur des langages formels est constitué de règles syntaxiques strictes qui organisent la structure des énoncés. Par exemple, en mathématiques, l'énoncé $3+3=6$ a une syntaxe correcte, mais $3+=3\$6$ n'a pas de sens. En chimie, H_2O est une formule syntaxiquement correcte, cependant que ${}_2Zz$ ne l'est pas.

Les règles syntaxiques se déclinent en deux types, relatives aux signes d'une part et à la structure, de l'autre. Les signes sont les éléments de base de tout langage, tels que les mots, les nombres et les éléments chimiques. Un des problèmes avec $3+=3\$6$ est que $\$$ n'est en principe pas un signe autorisé en mathématiques. De même, ${}_2Zz$ n'est pas licite car il n'y a pas d'élément de symbole Zz . Le deuxième type de règle syntaxique concerne la manière dont les signes sont combinés. L'équation $3+=3$ est incorrecte car, même si $+$ et $=$ sont des signes autorisés, leur succession n'est pas définie en mathématiques et n'a dès lors pas de sens. De même, dans une formule chimique, l'indice vient éventuellement après le nom d'un élément, jamais avant.

Voici un exemple curieux : l@ phrase de françai\$ bien structurée avec des s*ignes mal utilisés. Tous les signes de cette phrase sont valides, mais la structure de la phrase

est erronée.

La lecture d'une phrase en français ou une déclaration dans une langue formelle suppose d'en comprendre la structure (bien que dans une langue naturelle, tout lecteur procède ainsi, inconsciemment). Ce processus s'appelle l'*analyse syntaxique*. Bien que les langues formelles et naturelles aient de nombreuses caractéristiques en commun – les signes, la structure et la syntaxe – il existe quelques différences notoires :

Ambiguïté : les langues naturelles sont chargées d'ambiguïté, que les humains gèrent en utilisant des indices contextuels ainsi que d'autres informations. Les langues formelles sont conçues pour être presque ou totalement dépourvues d'ambiguïté, ce qui signifie que toute déclaration a une –et une seule– signification, quel que soit le contexte,

Redondance : pour pallier l'ambiguïté et réduire les malentendus, les langues naturelles sont caractérisées par de nombreuses redondances. De ce fait, elles sont souvent verbeuses. Les langues formelles sont moins redondantes et plus concises,

Littéralité : les langues naturelles sont remplies d'expressions idiomatiques et de métaphores. Dans l'expression : « le franc est tombé », il y a fort à parier qu'il n'y a pas de franc en jeu et que rien ne tombe (cette expression idiomatique signifie que quelqu'un a compris quelque chose après une période de confusion). Les langues formelles signifient exactement ce qu'elles disent. Comme nous grandissons tous en parlant des langues naturelles, il est parfois difficile de s'adapter aux langues formelles. La différence entre le langage formel et le langage naturel s'apparente à la différence entre la poésie et la prose, mais en plus :

Poésie : les mots sont utilisés pour leurs sons ainsi que pour leur signification. L'ensemble du poème crée un effet ou une réponse émotionnelle. L'ambiguïté est non seulement courante, mais souvent délibérée,

Prose : le sens littéral des mots est important et la structure apporte du sens. La prose se prête davantage à l'analyse que la poésie, mais reste néanmoins souvent ambiguë,

Programmes : la signification d'un programme informatique est sans ambiguïté et littérale, et peut être entièrement comprise par l'analyse des signes et de la structure.

Les langages formels sont plus denses que les langues naturelles, ce qui nécessite plus de temps et de la concentration pour les interpréter. De plus, la structure est fondamentale. Il n'est donc pas toujours judicieux de lire de haut en bas et de gauche à droite. Au lieu de cela, le lecteur apprend à analyser le programme mentalement, à identifier les

signes et à interpréter la structure. Enfin, les détails sont importants. De petites erreurs d'orthographe ou de ponctuation, dont il est possible de s'affranchir dans les langues naturelles, peuvent entraîner des incompréhensions dans un langage formel.

1.7 Débogage

Tous les programmeurs commettent des erreurs. Pour des raisons quelque peu fantaisistes, les erreurs de programmation sont appelées « bugs » et le processus pour les déboguer est appelé « débogage » (*debugging*). La programmation, et surtout le débogage, font parfois surgir des émotions fortes. Confronté à un bogue difficile, un programmeur peu aguerri peut se sentir en colère, découragé ou frustré. Il est prouvé que de très nombreuses personnes réagissent naturellement face aux ordinateurs comme si ces derniers étaient des personnes. Lorsque les ordinateurs fonctionnent bien, nous les considérons comme des coéquipiers mais, lorsqu'ils se montrent obstinés, nous leur répondons de la même manière que si nous répondions à des personnes obstinées.

Se préparer à ces réactions aide à les surmonter. Une approche possible consiste à considérer l'ordinateur comme un employé doté de certains atouts tels la rapidité et la précision, mais aussi de faiblesses particulières, comme le manque d'empathie et l'incapacité à saisir la situation dans son ensemble. Le travail d'un programmeur consiste à être un bon gestionnaire : trouver des moyens de tirer parti des points forts et d'atténuer les points faibles. Trouver des moyens d'effacer ses émotions pour s'attaquer au problème, sans laisser ses réactions interférer avec sa capacité à travailler efficacement.

Apprendre à déboguer peut être frustrant, mais un bon débogueur possède une compétence précieuse très utile pour de nombreuses activités au-delà même de la programmation. À la fin de chaque chapitre, existe une section comme celle-ci, comportant diverses suggestions supposées utiles pour le débogage.

1.8 Glossaire

résolution des problèmes : processus consistant à formuler un problème, à trouver une solution et à l'exprimer,

programme : séquence d'instructions qui explicite un calcul,

REPL : programme qui lit les données d'entrée, les exécute et produit des résultats de manière répétée,

invite : caractères affichés par le REPL pour indiquer qu'il est prêt à recevoir les données de l'utilisateur,

déclaration : instruction qui amène le REPL à afficher une valeur à l'écran,

opérateur : symbole qui permet l'exécution d'un calcul simple comme l'addition, la multiplication de nombres ou la concaténation de chaînes de caractères,

valeur : une des unités de base des données, comme un nombre ou une chaîne de caractères, qu'un programme manipule,

type : une catégorie de valeurs. Les types vus jusqu'à présent sont les nombres entiers ([Int64](#)), les nombres à virgule flottante ([Float64](#)) et les chaînes de caractères ([String](#)),

Int : type qui représente des nombres entiers,

Float : type qui représente les nombres avec un point décimal,

Char : type qui représente des caractères uniques,

langage naturel : toute langue que parlent les Hommes et qui a évolué naturellement,

langage formel : tout langage que les humains ont conçu à des fins spécifiques, comme la représentation de principes mathématiques ou de programmes informatiques. Tous les langages de programmation sont des langages formels,

syntaxe : règles qui organisent la structure d'un programme,

signe : élément de base de la structure syntaxique d'un programme, analogue à un mot dans une langue naturelle,

structure : façon dont les signes sont combinés,

parse (analyse) : examen d'un programme et analyse de la structure syntaxique,

bug (ou bogue) : erreur dans un programme,

débogage : processus de recherche et de correction des bugs.

1.9 Exercices

Conseil. Lire ce contenu devant un ordinateur afin d'essayer les exemples au fur et à mesure constitue une bonne idée.

1.9.1 Exercice

Chaque fois qu'une nouvelle fonctionnalité est expérimentée, il est indiqué de commettre volontairement des erreurs. Par exemple, dans le programme « Hello,

World! », que se passe-t-il en cas d'omission d'un des guillemets ? Et quand les deux sont omis ? Et s'il y avait des fautes d'orthographe ?

Ce genre d'expérience aide à se souvenir de ce qui est lu, ainsi que lors de l'exercice de programmation. En effet, c'est de cette manière qu'un programmeur apprend à connaître la signification des messages d'erreur. Il est préférable de faire des erreurs maintenant et volontairement plutôt que plus tard et accidentellement.

1. Dans une déclaration d'affichage, que se passe-t-il en cas d'omission d'une des parenthèses, ou des deux ?
2. À l'affichage d'une chaîne de caractères, que se passe-t-il si un des guillemets est omis, ou les deux ?
3. Un signe moins peut être utilisé pour obtenir un nombre négatif comme -2. Que se passe-t-il si un signe plus précède un nombre ? Qu'en est-il de 2++2 ?
4. En notation mathématique, les zéros de tête sont acceptés, comme dans 02. Que se passe-t-il lorsqu'une telle notation est utilisée en Julia ?
5. Que se passe-t-il lorsqu'il n'y a pas d'opérateur entre deux valeurs ?

1.9.2 Exercice

Lançons le REPL Julia et utilisons-le comme calculatrice :

1. Combien de secondes dans 42 minutes 42 secondes ?
2. Sachant qu'1 mile terrestre = 1,61 km, combien de miles dans 10 kilomètres ?
3. Si 10 kilomètres sont parcourus en 37 minutes 48 secondes, quel est le rythme moyen (temps par kilomètre en minutes et secondes) ? Quelle est la vitesse moyenne en miles terrestres par heure ?

Chapitre 2

Variables, expressions et déclarations

La possibilité de manipuler des variables constitue une des grandes forces des langages de programmation. Une variable est un nom (lettre ou mot) qui fait référence à une valeur.

2.1 Déclaration d'affectation

Une *déclaration d'affectation* crée une nouvelle variable et lui attribue une valeur :

```
julia> message = "À présent, quelque chose de nouveau"
"À présent, quelque chose de nouveau"
julia> n = 17
17
julia> π_val = 3.141592653589793
3.141592653589793
```

Dans cet exemple, la première affectation associe une chaîne à une nouvelle variable nommée `message`. La deuxième associe l'entier 17 à `n`. La troisième apparie la valeur approximative de π à la variable `π_val` (pour obtenir le symbole π , il convient de saisir `\pi` et de presser la touche `TAB`).

Une manière courante de représenter les variables sur papier consiste à écrire le nom avec une flèche pointant vers sa valeur. Ce type de figure est appelé un *diagramme*

```

message  —→ "À présent, quelque chose de nouveau"
n        —→ 17
π_val    —→ 3.141592653589793

```

FIGURE 2.1.1 – Exemple d'un diagramme d'état.

d'état car il montre dans quel état se trouve chacune des variables. La figure 2.1.1 représente le diagramme d'état de l'encadré précédent.

2.2 Les noms de variables

Généralement, les programmeurs choisissent pour leurs variables des noms qui sont expressifs et, au besoin, ils associent un élément de documentation à la variable afin d'améliorer la compréhension du code.

La longueur des noms des variables n'est pas limitée. Ces noms peuvent contenir presque tous les caractères Unicode (voir la section 8.1), mais ils ne peuvent pas commencer par un nombre. Les lettres majuscules sont autorisées quoiqu'il soit conventionnel de n'utiliser que des minuscules pour les noms de variables. Les caractères Unicode peuvent être saisis en complétant les abréviations de type \LaTeX dans le REPL et ce, par des tabulations.

Le caractère de soulignement `_` peut apparaître dans un nom. Il est souvent utilisé dans les noms comportant plusieurs mots.

Si un nom inapproprié est attribué à une variable, Julia retourne une erreur de syntaxe :

```

julia> 76trombones = ("Belle fanfare")
ERROR: syntax: "76" is not a valid function argument name
julia> more@ = 1000000
ERROR: syntax: extra token "@" after end of expression
julia> struct = "Advanced Theoretical Zimurgy"
ERROR: syntax: unexpected "="

```

`76trombones` est incorrect parce que la variable commence par un chiffre. `more@` est erroné parce cette formulation contient un caractère non-autorisé, `@`. Mais qu'est-ce qui ne va pas avec la troisième déclaration ? Il s'avère que `struct` est un des mots-clés de Julia. Le REPL utilise des mots-clés pour reconnaître la structure d'un programme. Or, ceux-ci ne peuvent pas être utilisés comme noms de variables.

Julia comporte les mots-clés suivants repris dans le tableau 2.1.

TABLE 2.1 – Liste des mots-clés de Julia.

abstract type	baremodule	begin	break	catch
const	continue	do	else	elseif
end	export	finally	for	function
global	if	import	importall	in
let	local	macro	module	mutable struct
primitive type	quote	return	struct	try
using	where	while		

Il est inutile de mémoriser cette liste. Dans la plupart des environnements de développement, les mots-clés sont affichés dans une couleur déterminée. Si un mot de cette liste est attribué à une variable, un message d’alerte/d’erreur apparaîtra.

2.3 Expressions et déclarations

Une expression est une combinaison de valeurs, de variables et d’opérateurs. Une valeur en soi est considérée comme une expression, de même qu’une variable, de sorte que les expressions suivantes sont toutes autorisées :

```
julia> 42
42
julia> n
17
julia> n + 25
42
```

Lorsqu’une expression est saisie à l’invite, le REPL l’évalue pour trouver sa valeur. Dans cet exemple, `n` a la valeur 17 et `n + 25` prend la valeur 42.

Une déclaration (ou une instruction) est une unité de code qui produit un effet comme la création d’une variable ou l’affichage d’une valeur.

```
julia> n = 17
17
julia> println(n)
17
```

La première ligne est une déclaration d’affectation qui associe une valeur à `n`. La deuxième ligne est une instruction qui conduit à l’affichage de la valeur de `n`.

Lorsqu’une instruction est saisie, le REPL l’exécute, ce qui signifie qu’il fait tout ce que contient l’instruction.

2.4 Mode « script »

Jusqu’à présent, nous avons pratiqué Julia en mode interactif, ce qui signifie que nous interagissons directement avec le REPL. Le mode interactif est un bon moyen de démarrer, mais lors d’un usage intensif, cette technique devient rapidement fastidieuse.

L’autre possibilité consiste à enregistrer le code dans un fichier. On se réfère à ce dernier en disant qu’il s’agit d’un *script*. Ensuite, l’utilisateur lance Julia en mode *script* pour exécuter le code qu’il contient. Par convention, les scripts en Julia portent des noms se terminant par `.jl`.

Si nous savons comment créer et exécuter un script, nous sommes prêts (sinon, il faudra utiliser en local Jupyter ou Pluto, par exemple ; voir l’annexe B, page 299). À ce stade, il existe 2 méthodes :

- soit nous ouvrons un fichier texte (avec Vim, Emacs, etc.), nous y écrivons et nous l’enregistrons avec une extension `.jl`. Le script peut être exécuté dans un terminal bash (ou analogue) pour autant que nous nous trouvions dans le répertoire contenant le script à exécuter :

```
name@computer:~$ julia nom_du_programme.jl
```

- soit nous recourons à un environnement de développement intégré comme VS-Codium (voir l’annexe B, page 299). Dans ce cas, nous supposons que le script est enregistré dans un répertoire dénommé `Prgm_Julia`. Dans VS-Codium, il suffit de cliquer sur File → OpenFile et de sélectionner le script. Celui-ci apparaît, colorisé, la partie supérieure de Codium. Il convient de cliquer sur une des lignes du code puis de cliquer sur l’icône ▷ de la barre de commandes supérieure.

Quoiqu’il en soit, parce que Julia propose les deux modes (REPL et exécution de script), il est possible de tester des bouts de code en mode interactif avant de les insérer dans un script. Cependant, il existe des différences entre le mode interactif et le mode script qui peuvent être surprenantes. Par exemple, utilisons Julia comme calculatrice et saisissons :

```
julia> miles = 26.2
26.2
julia> miles * 1.61
42.182
```


La première ligne attribue une valeur à `miles` et affiche la valeur. La deuxième ligne est une expression. Donc le REPL l'évalue et affiche le résultat ¹.

Néanmoins, l'exécution du même code dans un script ne produit aucun résultat. En mode script, une expression, à elle seule, n'a aucun effet visible. Julia évalue bel et bien l'expression, mais n'affiche pas la valeur sauf à lui demander de le faire :

```
miles = 26.2
println(miles * 1.61)
```

Ce comportement peut être déroutant au début.

Un script contient généralement une séquence d'instructions. S'il y a plus d'une déclaration, les résultats apparaissent un à un, à mesure de l'exécution des déclarations.

Par exemple, le code suivant :

```
println(1)
x = 2
println(x)
```

retourne :

```
1
2
```

Les déclarations d'affectation ne produisent aucun résultat.

2.4.1 Exercice

Vérifiez que ceci est bien compris en saisissant les déclarations suivantes dans le REPL de Julia et observez le résultat :

```
5
x + 5
x + 1
```

Maintenant, écrivez les mêmes déclarations dans un script et exécutez-le. Quel est le résultat ? Modifiez le script en transformant chaque expression en une instruction d'affichage, puis exécutez-le à nouveau.

2.5 Priorité des opérateurs

Lorsqu'une expression contient plus d'un opérateur, l'ordre d'évaluation dépend de la présence des opérateurs entre eux. Pour les opérateurs mathématiques, Julia suit

1. Il s'avère qu'un marathon couvre environ 42 kilomètres.

la convention mathématique. L'acronyme *PEMDAS* constitue un moyen mnémotechnique :

1. les **P**arenthèses ont la plus haute priorité et peuvent être utilisées pour forcer une expression à être évaluée dans l'ordre souhaité. Comme les expressions entre parenthèses sont évaluées en premier, $2*(3-1)$ vaut 4, et $(1+1)^(5-2)$ vaut 8. L'usage de parenthèses contribue à rendre une expression plus facile à lire, comme dans $(\text{minute} * 100) / 60$, même si cela ne change pas le résultat,
2. l'**E**xponentiation a la priorité suivante, donc $1+2^3$ est 9 (et non 27), et $2*3^2$ donne 18 (et non 36),
3. la **M**ultiplication et la **D**ivision ont la préséance sur l'**A**ddition et la **S**oustraction. Ainsi, $2*3-1$ est 5 (et non 4) et, $6+4/2$ est 8 (et non 5),
4. les opérateurs ayant la même priorité sont évalués de gauche à droite (sauf l'exponentiation). Ainsi, dans l'expression $\text{degrés} / 2 * \pi$, la division se fait en premier et le résultat est multiplié par π . Pour diviser par 2π , il est possible d'utiliser des parenthèses $\text{degrés} / (2 * \pi)$, d'écrire $\text{degrés} / 2 / \pi$ ou encore $\text{degrés} / 2\pi$.

Conseil. Il n'est pas simple de se souvenir de la préséance des opérateurs et une expression complexe peut rapidement devenir un casse-tête, source d'erreurs. L'usage de parenthèses relève de la prudence.

2.6 Opérations sur les chaînes de caractères

En général, les opérations mathématiques sur des chaînes de caractères ne sont pas permises, même si ces dernières ressemblent à des nombres. Les opérations suivantes sont donc illicites :

"2" - "1" "un œuf" / "le plat" "top" + "charm"

Cependant, il existe deux exceptions $*$ et $^$.

L'opérateur $*$ effectue la concaténation des chaînes de caractères, ce qui signifie qu'il joint les chaînes en les reliant bout à bout. Par exemple :

```
julia> first_str = "paruline "
"paruline "
julia> second_str = "à "
"à "
```

```
julia> third_str = "gorge "
"gorge "
julia> fourth_str = "jaune"
"jaune"
julia> first_str * second_str * third_str * fourth_str
"paruline à gorge jaune"
```

L'opérateur `^` procède également sur des chaînes de caractères. Il effectue des répétitions. Par exemple, `"Spam"^3` retourne `"SpamSpamSpam"`. Si une des valeurs est une chaîne, l'autre doit être un nombre entier.

Cette utilisation de `*` et `^` est logique par analogie avec la multiplication et l'exponentiation. Tout comme 4^3 est équivalent à $4*4*4$, nous nous attendons à ce que `"Spam"^3` soit identique à `"Spam " * "Spam " * "Spam"`. Ce qui est bien le cas.

2.7 Commentaires

Plus les programmes deviennent volumineux et complexes, plus ils sont difficiles à lire. Les langages formels sont denses et il est souvent difficile de regarder un morceau de code et de comprendre ce qu'il représente.

C'est pourquoi il est bon d'ajouter des notes à nos programmes pour expliquer en langage naturel ce que réalise le code. Ces notes sont appelées des *commentaires* et elles commencent par le symbole `#` :

```
# calcule le pourcentage d'heure écoulée
pourcentage = (minute * 100) / 60
```

Dans ce cas, le commentaire apparaît sur une ligne séparée. Cependant, les commentaires peuvent se trouver à la fin d'une ligne :

```
pourcentage = (minute * 100) / 60 # calcule le pourcentage d'heure écoulée
```

Tout ce qui suit le caractère `#` jusqu'à la fin de la ligne est ignoré. Un commentaire n'altère pas l'exécution d'un programme.

Les commentaires sont toujours utiles lorsqu'ils documentent des caractéristiques non évidentes du code. Il est raisonnable de supposer que le lecteur peut comprendre ce que fait le code. Il est plus utile d'expliquer pourquoi.

Par exemple, ce commentaire est redondant avec le code et, par conséquent, inutile :

```
v = 5 # attribue 5 à v
```

Cependant, le commentaire suivant contient une information réellement pertinente :

```
v = 5 # vitesse exprimée en m/s
```

Avertissement. Des noms de variables *ad hoc* peuvent réduire le besoin d'écrire des commentaires. Les noms longs, généralement explicites, peuvent rendre des expressions complexes difficiles à lire. Tout est une question de compromis.

2.8 Débogage

Trois types d'erreurs se produisent dans un programme : des erreurs de syntaxe, des erreurs d'exécution et des erreurs sémantiques. Ce point sera approfondi au chapitre 21. Il est utile de les distinguer afin de les repérer le plus rapidement possible.

Erreur de syntaxe La « syntaxe » fait référence à la structure d'un programme et aux règles relatives à cette structure. Par exemple, les parenthèses doivent venir par paires, donc `(1 + 2)` est licite, mais `8)` constitue une erreur de syntaxe. S'il y a une erreur de syntaxe dans un programme, Julia affiche un message d'erreur et arrête l'exécution. Au cours des premières semaines d'étude, il est fréquent de passer du temps à rechercher les erreurs de syntaxe. Avec l'expérience, ce type d'erreur se manifesterait plus rarement et leur détection sera de plus en plus rapide.

Erreur d'exécution Le deuxième type d'erreur concerne l'exécution. On dit « erreur d'exécution » parce qu'elle n'apparaît qu'après le démarrage du programme. Ces erreurs sont également appelées *exceptions* parce qu'elles indiquent généralement que quelque chose d'exceptionnel (et de malencontreux) s'est produit. Les erreurs d'exécution sont rares dans les programmes simples associés aux premiers chapitres. Il s'écoulera un certain temps avant d'en rencontrer.

Erreur sémantique Le troisième type d'erreur concerne la « sémantique ». Il s'agit d'un problème de sens. Si un programme contient une erreur sémantique, il s'exécute sans émettre le moindre message d'erreur. Cependant, le programme produit un résultat autre que celui attendu bien qu'il accomplisse exactement ce qu'il lui est demandé.

L'identification des erreurs sémantiques peut être délicate car elle oblige à travailler à rebours en examinant la sortie d'un programme et en essayant de comprendre comment il procède.

2.9 Glossaire

variable nom qui fait référence à une valeur,

affectation déclaration qui attribue une valeur à une variable,

diagramme d'état représentation graphique d'un ensemble de variables et des valeurs auxquelles elles se réfèrent,

mot-clé mot réservé utilisé par un langage de programmation. Les mots-clés tels que `if`, `function`, `while`, etc., ne peuvent pas être employés comme noms de variables,

opérande une des valeurs sur lesquelles un opérateur exerce une action,

expression combinaison de variables, d'opérateurs et de valeurs qui représente un seul résultat,

évaluer simplifier une expression en effectuant les opérations afin d'obtenir une valeur unique,

déclaration section de code qui représente un ordre ou une action. Jusqu'à présent, les déclarations que nous avons vues opèrent des affectations (*assignments* en anglais) et des déclarations d'affichage,

exécuter lire une déclaration et faire ce qu'elle exprime,

mode interactif façon d'utiliser le REPL de Julia en saisissant un code à l'invite,

mode script façon d'utiliser Julia pour lire le code d'un script et l'exécuter,

script programme enregistré dans un fichier,

priorité d'un opérateur règles de préséance relative à l'ordre dans lequel les expressions impliquant des opérateurs mathématiques et des opérandes multiples sont évaluées,

concaténer joindre deux éléments bout à bout,

commentaire informations contenues dans un programme qui sont destinées à d'autres programmeurs (ou à toute personne lisant le code source) et qui n'ont aucun effet sur l'exécution de ce programme,

erreur de syntaxe erreur dans un programme qui rend impossible l'analyse (et donc l'interprétation),

erreur d'exécution (appelée également **exception**) erreur qui est détectée pendant que le programme est en cours d'exécution,

sémantique signification ou « sens » d'un programme,

erreur sémantique erreur dans un programme qui l'amène à faire autre chose que ce pour quoi il a été conçu.

2.10 Exercices

2.10.1 Exercice

Comme souligné dans le chapitre 1, chaque fois que vous apprenez une nouvelle fonction, vous devez l’essayer en mode interactif (REPL) et commettre des erreurs pour évaluer ce qui ne fonctionne pas.

1. Nous avons vu que $n = 42$ est licite. Qu’en est-il de $42 = n$?
2. Que se passe-t-il avec $x = y = 1$?
3. Dans certaines langages, chaque énoncé se termine par un point-virgule (;). Que se passe-t-il si vous mettez un point-virgule à la fin d’une déclaration en Julia ?
4. Que se passe-t-il si vous mettez un point à la fin d’une déclaration ?
5. En notation mathématique, vous pouvez multiplier x et y comme ceci $x y$. Que se passe-t-il si vous essayez cela en Julia ? Quid avec $5x$?

2.10.2 Exercice

Entraînez-vous à utiliser le REPL de Julia comme calculatrice :

1. Le volume d’une sphère de rayon r donné par la formule : $V_{sp} = \frac{4}{3}\pi r^3$. Quel est le volume d’une sphère de rayon $r = 5$?
2. Supposons que le prix de couverture d’un livre soit de 24.95 € mais que les librairies obtiennent une remise de 40 %. Les frais d’expédition sont de 3 € pour le premier exemplaire et de 75 centimes pour chaque exemplaire supplémentaire. Quel est le prix de gros total pour 60 exemplaires ?
3. Si quelqu’un quitte son habitation à 6h52 du matin et qu’il court 1 km au rythme de 8 min 15s par km, puis 3 km au rythme de 7 min 12 par km et encore 1 km à un rythme de 8 min 15s par km, à quelle heure rentrera-t-il pour le petit-déjeuner ?

Chapitre 3

Les fonctions

Dans le contexte de la programmation, une fonction est une séquence d'instructions effectuant un calcul. Une fonction définie porte un nom et contient une suite d'instructions. La fonction étant définie, il y est fait appel en utilisant son nom.

3.1 Appel de fonction

Nous avons déjà rencontré un appel de fonction :

```
julia> println("Hello, World!")  
Hello, World !
```

Le nom de la fonction (interne¹ à Julia) est `println`. L'expression entre parenthèses s'appelle l'*argument* de la fonction.

Il est courant de dire qu'une fonction « prend » un argument et « retourne » un résultat. Le résultat est également appelé *valeur de retour*.

Julia fournit des fonctions qui convertissent des valeurs d'un type à un autre. La fonction `parse` prend une chaîne de caractères et la convertit en n'importe quel type de nombre, si elle le peut. Sinon, Julia retourne un message d'erreur :

1. Le terme « intégrée » est équivalent dans ce contexte.

```
julia> parse{Int64, "32"}
32
julia> parse{Float64, "3.14159"}
3.14159
julia> parse{Int64, "Hello"}
ERROR: ArgumentError: invalid base 10 digit 'H' in "Hello"
```

La fonction `trunc` peut convertir des nombres à virgule flottante en entiers mais sans arrondi :

```
julia> trunc{Int64, 3.9999}
3
julia> trunc{Int64, -2.3}
-2
```

La fonction `float` convertit les nombres entiers en nombres à virgule flottante :

```
julia> float(32)
32.0
```

La fonction `string` convertit son argument en chaîne de caractères :

```
julia> string(32)
"32"
julia> string(3.14159)
"3.14159"
```

3.2 Fonctions mathématiques

Naturellement, avec Julia, la plupart des fonctions mathématiques usuelles sont directement disponibles :

```
ratio = signal_power / noise_power
decibels = 10 * log10(ratio)
```

Ce premier exemple utilise `log10` pour calculer un rapport signal/bruit en décibels (en supposant que la puissance du signal et la puissance du bruit soient définies).

La fonction `log` calcule les logarithmes naturels (ou népériens, aussi appelés hyperboliques – \ln).

À présent, voyons un autre exemple :

```
radians = 0.7
height = sin(radians)
```


Ce deuxième cas avec la fonction `sin` illustre la manière de procéder pour le sinus d'angles exprimés en radians. Un argument passé tel quel aux fonctions trigonométriques (`sin`, `cos`, `tan`, etc.) s'exprime en radians.

Pour convertir des degrés en radians, il faut diviser par 180 et multiplier par π :

```
julia> degrees = 45
45
julia> radians = (degrees / 180) *  $\pi$ 
0.7853981633974483
julia> sin(radians)
0.7071067811865475
```

La valeur de la variable π est une approximation en virgule flottante du nombre π , avec une précision à 16 chiffres.

En recourant à la trigonométrie, on vérifie que le résultat précédent correspond à la racine carrée de 2 divisée par 2 :

```
julia> sqrt(2) / 2
0.7071067811865475
```

3.3 Composition

Jusqu'à présent, nous avons examiné les éléments d'un programme –variables, expressions et déclarations– isolément, sans évoquer la manière de les combiner.

Une autre caractéristique des plus utiles des langages de programmation provient de leur capacité à prendre de petits blocs de construction et à les combiner (les associer). Par exemple, l'argument d'une fonction peut être tout type d'expression, incluant des opérateurs arithmétiques :

```
x = sin((degrees / 360) * 2 *  $\pi$ )
```

Cela fonctionne également avec des appels de fonction :

```
x = exp(log(x+1))
```

Presque partout où une valeur peut être injectée, il est possible de poser une expression arbitraire. À une contrainte près : le membre de gauche d'une déclaration d'affectation doit être un nom de variable. Toute autre expression à gauche conduit une erreur de syntaxe (nous verrons les exceptions à cette règle plus tard).

```
julia> minutes = heures * 60    # licite
45
julia> heures * 60 = minutes    # illicite !
ERROR: syntax: "60" is not a valid function argument name
```

3.4 Ajout de nouvelles fonctions

Jusqu'à présent, nous avons uniquement employé des fonctions internes à Julia. Cependant, ajouter de nouvelles fonctions est une des bases de la programmation. Une *définition de fonction* spécifie le nom d'une nouvelle fonction et la séquence d'instructions qui s'exécutent lorsque la fonction est appelée. Voici un exemple :

```
function printlyrics()
    println("Ses fluctuat nec mergitur,")
    println("C'était pas d'la littérature")
end
```

Le terme `function` est un mot-clé qui indique qu'une nouvelle fonction est définie. Le nom de la fonction est `printlyrics()`. Les règles pour les noms de fonctions sont les mêmes que pour les noms de variables : ils peuvent contenir presque tous les caractères Unicode (voir la section 8.1), si ce n'est que le premier caractère ne peut pas être un nombre. Par ailleurs, il n'est pas permis d'utiliser un mot-clé comme nom de fonction. En outre, il faut éviter qu'une variable et une fonction porte le même nom. Les parenthèses vides après le nom indiquent que cette fonction ne prend pas d'arguments.

La première ligne s'appelle l'*en-tête* ; le reste s'appelle le *corps*. Le corps se termine par le mot-clé `end` et peut contenir un nombre quelconque d'instructions. Pour une meilleure lisibilité, le corps de la fonction devrait être indenté.

Les guillemets doivent être des "guillemets droits" (idem pour les ' guillemets' simples). Les guillemets de ce type " " ou ' ' ne sont pas admis en Julia.

Si une définition de fonction est saisie en mode interactif, le REPL s'indente automatiquement pour indiquer que la définition n'est pas complète :

```
julia> function printlyrics()
        println("Ses fluctuat nec mergitur,")
```

La terminaison d'une fonction est indiquée par le mot-clé `end`.

La syntaxe pour appeler la nouvelle fonction est la même que pour les fonctions internes à Julia :

```
julia> printlyrics()
Ses fluctuat nec mergitur,
C'était pas d'la littérature
```

Dès qu'une fonction est définie, elle peut être utilisée au sein d'une autre fonction. Par exemple, pour répéter les vers précédents, nous pouvons écrire une fonction appelée `repeatlyrics` :

```
function repeatlyrics()
    printlyrics()
    printlyrics()
end
```

Lorsque `repeatlyrics` est appelée, nous obtenons :

```
julia> repeatlyrics()
Ses fluctuat nec mergitur,
C'était pas d'la littérature
Ses fluctuat nec mergitur,
C'était pas d'la littérature
```

3.5 Définition et usages

En rassemblant les fragments de code, l'ensemble du programme ressemble à ceci :

```
function printlyrics()
    println("Ses fluctuat nec mergitur,")
    println("C'était pas d'la littérature")
end

function repeatlyrics()
    printlyrics()
    printlyrics()
end

repeatlyrics()
```

Ce programme contient deux définitions de fonctions : `printlyrics` et `repeatlyrics`. Les définitions de fonction sont exécutées comme les autres instructions, mais elles ont pour effet de créer des objets de fonction. Les instructions à l'intérieur de la fonction ne s'exécutent pas avant que la fonction ne soit appelée. La définition de la fonction ne produit aucune sortie.

Comme on peut s'y attendre, il faut créer une fonction avant de pouvoir l'exécuter. En d'autres termes, la définition de la fonction doit être exécutée avant que la fonction ne soit appelée.

3.5.1 Exercice

Déplacez la dernière ligne de ce programme vers le haut, afin que l'appel de fonction apparaisse avant les définitions. Lancez le programme et voyez quel message d'er-

reur est retourné.

Ceci fait, déplacez l'appel de fonction vers le bas et déplacez la définition de `print-lyrics` après la définition de `repeatlyrics`. Que se passe-t-il lors de l'exécution de ce programme ?

3.6 Flux d'exécution

Pour s'assurer qu'une fonction est définie avant sa première utilisation, il faut connaître le *flux d'exécution*.

L'exécution commence toujours à la première instruction d'un programme. Les instructions sont exécutées séquentiellement, dans l'ordre de haut en bas. Les définitions de fonction ne modifient pas le flux d'exécution du programme, mais il faut se rappeler que les instructions à l'intérieur de la fonction ne s'exécutent pas avant que la fonction ne soit appelée. Un appel de fonction équivaut à un détour dans le flux d'exécution. Au lieu de passer à l'instruction suivante, le flux effectue un saut jusqu'au corps de la fonction appelée, y exécute les instructions, puis revient pour reprendre là où il s'était arrêté.

Cela semble assez simple, jusqu'à ce qu'on se souvienne qu'une fonction peut en appeler une autre. Au milieu d'une fonction, un programme peut être conduit à exécuter les instructions se trouvant dans une autre fonction. Puis, pendant l'exécution de cette nouvelle fonction, le programme peut à nouveau devoir exécuter une autre fonction. Heureusement, Julia est doué pour conserver la trace de l'endroit où reprendre. Ainsi, chaque fois qu'une fonction se termine, le programme reprend où il s'était arrêté au sein de la fonction qu'il a appelée.

En résumé, la lecture d'un programme ne se fait pas toujours linéairement de haut en bas. Généralement, il s'avère préférable de suivre le déroulement de l'exécution.

3.7 Paramètres et arguments

Certaines des fonctions que nous avons vues nécessitent des arguments. C'est le cas de la fonction `sin` qui requiert un nombre en argument. D'autres fonctions en nécessitent plusieurs. Par exemple, `parse` en prend deux : un *type* de nombre et une chaîne de caractères.

À l'intérieur d'une fonction, les arguments sont affectés à des variables appelées *paramètres*. Le petit programme suivant décrit une fonction qui prend un argument pour calculer son carré (ceci à titre d'exemple numérique) :

```
julia> function eleve_au_carre(t)
           carre = t * t
           println(carre)
       end

julia> eleve_au_carre(5)
25
```

En l'occurrence, l'appel de la fonction (c'est-à-dire l'instruction `eleve_au_carre(5)`) passe son argument (`5`) à `t` (de la `function eleve_au_carre(t)`). `t` est ensuite transféré comme *paramètre* à l'intérieur de la fonction `eleve_au_carre(t)`. Le résultat de la multiplication (`t * t`) est affecté à la variable `carre`. Celle-ci est passée en argument à la fonction `println` et affichée.

L'intérêt de ce type de pratique est de pouvoir réutiliser la fonction ultérieurement. Par exemple, en vertu de ce qui a été discuté en section 2.6, `eleve_au_carre("lu")` permet d'obtenir le mot `lulu`.

Il est également possible d'utiliser une variable en tant qu'argument :

```
julia> name = "lu"
"lu"
julia> eleve_au_carre(name)
lulu
```

Le nom de la variable que nous passons en argument (`name`) n'a rien à voir avec le nom du paramètre (`t`). Peu importe comment la valeur a été nommée dans l'appel. Nous appelons `t` tout qui arrive en argument dans la fonction `eleve_au_carre`.

Voici un autre exemple de fonction simple :

```
function printtwice(eluard)
    println(eluard)
    println(eluard)
end
```

Avec cet appel :

```
printtwice(π)
```

Julia retourne :

```
π
π
```

3.8 Les variables et les paramètres sont locaux

Une variable créée à l'intérieur d'une fonction est locale. Ceci signifie qu'elle n'existe qu'à l'intérieur de la fonction. Par exemple :

```
function cattwice(part1, part2)
    concat = part1 * part2
    printtwice(concat)
end
```

Cette fonction prend deux arguments, les concatène et affiche le résultat. Voici un exemple d'utilisation :

```
julia> line1 = "Facile est beau "
"Facile est beau "
julia> line2 = "sous tes paupières"
"sous tes paupières"
julia> cattwice(line1, line2)
Facile est beau sous tes paupières
Facile est beau sous tes paupières
```

Lorsque `cattwice` se termine, la variable `concat` est détruite. Si nous essayons de l'afficher, nous obtenons une exception :

```
julia> println(concat)
ERROR: UndefVarError: concat not defined
```

Ceci illustre le fait que les paramètres sont locaux. Par exemple (voir la section 3.7), hors de la fonction `elevé_au_carre`, `t` n'existe pas.

3.9 Diagrammes de pile

Pour savoir quelles variables utiliser et à quel endroit, il est parfois utile de dessiner un diagramme de pile. Comme les diagrammes d'état, les diagrammes de pile montrent la valeur de chaque variable. En outre, ils montrent la fonction à laquelle chaque variable appartient. Chaque fonction est représentée par un cadre. Un cadre est un espace fermé avec le nom d'une fonction en vis-à-vis et les paramètres et variables de la fonction à l'intérieur.

Le diagramme de pile de l'exemple précédent (avec `cattwice`) est présenté dans la figure 3.9.1.

Les cadres sont disposés dans une pile qui indique quelle fonction est appelée. Dans cet exemple, `printtwice` a été appelé par `cattwice` et, `cattwice` l'a été par `Main`. `Main` est

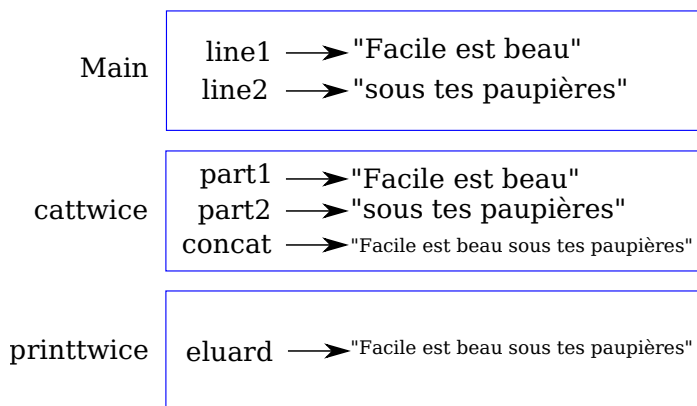


FIGURE 3.9.1 – Diagramme de pile.

un nom spécial désignant le cadre supérieur. Lorsqu'une variable est créée en dehors de toute fonction, elle appartient à **Main**. Chaque paramètre fait référence à la même valeur que son argument correspondant. Ainsi, **part1** a la même valeur que **line1**, **part2** a la même valeur que **line2**, et **eluard** a la même valeur que **concat**.

Si une erreur se produit pendant un appel de fonction, Julia imprime le nom de la fonction, le nom de la fonction qui l'a appelée et le nom de la fonction qui a appelé tout cela jusqu'à **Main**.

Par exemple, si nous essayions d'accéder à **concat** à partir de **printtwice**, nous obtiendrions une erreur **UndefVarError** :

```

ERROR: UndefVarError: concat not defined
Stacktrace:
 [1] printtwice at ./REPL[1]:2 [inlined]
 [2] cattwice (::String, ::String) at ./REPL[2]:]
  
```

Cette liste de fonctions est appelée **trace de pile** ou **stacktrace**. Elle indique dans quel fichier du programme l'erreur s'est produite ainsi que sur quelle ligne. Elle indique quelles fonctions étaient en cours d'exécution à ce moment-là et, également, la ligne de code qui a causé l'erreur.

L'ordre des fonctions dans la trace de pile est l'inverse de celui des cadres dans le diagramme de pile. La fonction en cours d'exécution se trouve au sommet de la pile.

3.10 Fonction avec retour et fonction vide

Certaines des fonctions que nous avons utilisées, comme les fonctions mathématiques, retournent des résultats. Appelons-les « fonctions avec retour ». D'autres fonctions effectuent une action mais ne retournent pas de valeur. On les appelle des fonctions nulles ou vides (*void functions*). Dans l'exemple suivant :

```
function printtwice(brassens)
    println(brassens)
    println(brassens)
end
```

la fonction `printtwice` vue précédemment (section 3.7) ne retourne aucune valeur bien qu'elle effectue deux actions.

En appelant une fonction avec retour, l'intention est d'exploiter le résultat obtenu. Par exemple, il est tout-à-fait possible d'affecter le résultat à une variable ou l'utiliser comme partie d'une expression :

```
x = cos(radians)
golden = (sqrt(5) + 1) / 2
```

Lorsqu'une fonction est appelée en mode interactif, Julia affiche le résultat :

```
sqrt(5)
2.23606797749979
```

Cependant, dans un script contenant une instruction telle que :

```
sqrt(5)
```

si une fonction avec retour est appelée seule, la valeur de retour est définitivement perdue. Ce script calcule bien la racine carrée de 5, mais comme il n'enregistre ni n'affiche le résultat, il n'est guère utile.

Les fonctions vides ou nulles peuvent afficher un message à l'écran ou avoir quel-qu'autre effet, mais elles n'ont pas de valeur de retour. Si le résultat est attribué à une variable, nous obtenons une valeur spéciale appelée *nothing*.

```
julia> resultat = eleve_au_carre(5)
25
julia> show(resultat)
nothing
```

Pour imprimer la valeur `nothing`, nous devons utiliser la fonction `show` qui agit comme `println` mais en gérant la valeur `nothing`.

La valeur `nothing` n'est pas de la même nature que la chaîne "nothing". C'est une valeur spéciale qui est caractérisée par son propre type :

```
julia> typeof(nothing)
Nothing
```

Nous commencerons à rédiger des fonctions avec retour au chapitre 6 et suivants.

3.11 Pourquoi utiliser des fonctions ?

Il ne semble peut-être pas évident de comprendre pourquoi il est astucieux de diviser un programme en fonctions. Il existe plusieurs raisons à cela :

- la création d'une nouvelle fonction offre la possibilité de nommer un groupe d'instructions, ce qui rend tout programme plus facile à lire et à déboguer,
- les fonctions rendent un programme plus concis en éliminant le code répétitif. Plus tard, s'il apporte une modification, le programmeur ne doit l'introduire qu'à un seul endroit.
- diviser un programme de grande taille en fonctions permet de déboguer chaque partie une à une et les assembler ensuite en un tout opérationnel.
- des fonctions bien conçues sont souvent utiles pour de nombreux programmes. Une fois écrite et déboguée, une fonction peut être réutilisée,
- en Julia, les fonctions améliorent considérablement les performances.

3.12 Débogage

Le débogage est une des compétences les plus importantes qu'un programmeur doit absolument acquérir. Bien qu'il puisse être frustrant, le débogage est une des parties les plus riches intellectuellement, les plus stimulantes et les plus intéressantes de la programmation. D'une certaine manière, le débogage ressemble au travail de détective. Confronté à des indices, le programmeur doit déduire les processus et les événements qui ont conduit aux résultats erronés observés.

Le débogage s'avère également similaire à une science expérimentale. Lorsque se présente une idée de ce qui coince, il convient de modifier le programme et l'essayer à nouveau. Si l'hypothèse est correcte, le résultat de la modification peut être prédit et, graduellement, on s'approche d'un programme opérationnel. Si l'hypothèse est fausse, il faut en formuler une nouvelle. Comme l'a fait remarquer Sherlock Holmes :

When you have eliminated the impossible, whatever remains, however improbable, must be the truth. (A. Conan Doyle – The Sign of Four)

Pour certains, la programmation et le débogage sont une seule et même notion : la programmation est le processus qui consiste à déboguer progressivement un programme jusqu'à ce qu'il exécute ce à quoi il est destiné. L'idée est de commencer avec un programme qui fonctionne et d'y apporter de petites modifications, en les déboguant au fur et à mesure.

Par exemple, LINUX est un système d'exploitation qui contient des millions de lignes de code. Pourtant, il a commencé comme un simple programme que Linus Torvalds a utilisé pour explorer la puce Intel 80386. Selon Larry Greenfield dans *The Linux User's Guide* : « Un des premiers projets de Linus était un programme qui permettait de passer de l'impression "AAAA" à "BBBB". Ce programme a ensuite évolué vers le noyau LINUX ».

3.13 Glossaire

fonction séquence nommée d'instructions qui effectue une opération utile. Les fonctions peuvent ou non prendre des arguments et peuvent ou non produire un résultat,

définition de fonction déclaration qui crée une nouvelle fonction, en précisant son nom, ses paramètres et les instructions qu'elle contient,

objet de fonction valeur créée par la définition d'une fonction. Le nom de la fonction est une variable qui fait référence à un objet de la fonction,

en-tête première ligne de la définition d'une fonction,

corps séquence d'instructions à l'intérieur d'une fonction,

paramètre nom utilisé à l'intérieur d'une fonction pour désigner la valeur passée en argument,

appel de fonction déclaration qui dirige l'exécution vers une fonction. Elle se compose du nom de la fonction suivi d'une liste d'arguments entre parenthèses.

argument valeur fournie à une fonction lorsque celle-ci est appelée. Cette valeur est attribuée au paramètre correspondant dans la fonction,

variable locale variable définie à l'intérieur d'une fonction. Une variable locale ne peut être utilisée qu'à l'intérieur de la fonction où elle est déclarée,

valeur de retour résultat du traitement d'une fonction. Si un appel de fonction est utilisé comme une expression, la valeur de retour est la valeur de l'expression,

fonction avec retour fonction qui retourne une valeur à l'appelant,

fonction void (fonction nulle ou vide) fonction qui retourne toujours [nothing](#),

- nothing** valeur spéciale renvoyée par les fonctions vides (dites nulles),
- composition** (ou combinaison) utilisation d’une expression (ou d’une déclaration) dans le cadre d’une expression (ou d’une déclaration) plus large,
- flux d’exécution** ordre d’exécution des instructions,
- diagramme de piles** représentation graphique d’une pile de fonctions, de leurs variables et des valeurs auxquelles elles se réfèrent,
- cadre** rectangle dans un diagramme de pile qui représente un appel de fonction. Un cadre contient les variables locales et les paramètres de la fonction,
- trace d’appel (ou *stacktrace*)** liste des fonctions qui sont en cours d’exécution, imprimée lorsqu’une exception se produit. Ce terme a pour synonyme *trace de pile*, *stack trace*, *stack backtrace* ou encore *stack traceback*.

3.14 Exercices

Conseil. Ces exercices doivent être effectués en utilisant uniquement les déclarations et autres notions telles qu’étudiées jusqu’à présent.

3.14.1 Exercice

Écrivez une fonction nommée `rightjustify` qui prend une chaîne de caractères nommée `s` comme paramètre et affiche la chaîne avec suffisamment d’espaces de tête pour que la dernière lettre de la chaîne² se trouve dans la colonne 70 de l’affichage.

```
julia> rightjustify(Julieta)
```

```
Julieta
```

Conseil. Utilisez la concaténation et la répétition des chaînes de caractères. De plus, Julia fournit une fonction intégrée appelée `length` qui retourne la longueur d’une chaîne de caractères, de sorte que la valeur de `length("Julieta")` est 7.

3.14.2 Exercice

Un objet de fonction est une valeur que vous pouvez attribuer à une variable ou passer en argument. Par exemple, `dotwice` est une fonction qui prend un objet de fonction comme argument et l’appelle deux fois :

2. Dans l’exemple, *Julieta* est une allusion au film de Pedro Almodovar.

```
function dotwice(f)
  f()
  f()
end
```

Voici un exemple qui utilise `dotwice` pour appeler deux fois une fonction intitulée `printspam`.

```
function printspam()
  println("spam")
end

dotwice(printspam)
```

Saisissez cet exemple dans un script et testez-le.

1. Modifiez la fonction `dotwice` pour qu'elle prenne deux arguments, un objet fonction ainsi qu'une valeur, et qu'elle appelle la fonction deux fois en passant la valeur comme argument.
2. Dans votre script, copiez la définition de `printtwice` donnée plus haut dans ce chapitre (voir la section 3.7).
3. Utilisez la version modifiée de `dotwice` pour appeler `printtwice` deux fois, en passant `"spam"` comme argument.
4. Définissez une nouvelle fonction appelée `dofour` qui prend un objet de fonction et une valeur, puis qui appelle la fonction quatre fois en passant la valeur comme paramètre. Il ne doit y avoir que deux énoncés dans le corps de cette fonction et non quatre.

3.14.3 Exercice

1. Rédigez une fonction `grid` qui affiche une grille telle que représentée dans la figure 3.14.1.
2. Écrivez une fonction qui dessine une grille similaire avec quatre lignes et quatre colonnes (ce problème est basé sur un exercice de Steve Oualline, *Practical C Programming* [6]).

Conseil. Pour imprimer plus d'une valeur sur une ligne, vous pouvez imprimer une séquence de valeurs séparées par des virgules :

```
println("+", "-")
```

La fonction `print` ne passe pas à la ligne suivante :

```
print("+")  
println("-")
```

Le résultat de ces deux dernières déclarations est "+ -" sur la même ligne. La sortie de la déclaration suivante commencerait à la ligne suivante.

```
+ - - - - + - - - - +  
|           |           |  
|           |           |  
|           |           |  
|           |           |  
+ - - - - + - - - - +  
|           |           |  
|           |           |  
|           |           |  
+ - - - - + - - - - +
```

FIGURE 3.14.1 – Grille associée à l'exercice 3.14.3 (partie 1).

Chapitre 4

Étude de cas : Conception d'une interface

Ce chapitre présente une étude de cas illustrant la conception de fonctions travaillant collaborativement.

Il présente des graphiques « tortue » (Turtle), un moyen de créer des dessins à l'aide de programmes. Les graphiques « tortue » ne sont pas inclus dans la bibliothèque standard, donc le module ThinkJulia doit être ajouté à l'installation de Julia (avant de poursuivre, consultez l'annexe B (page 299), en particulier la sous-section 21.3.5 qui traite de l'installation des modules Julia).

4.1 Turtles

Un module est un fichier qui contient un ensemble de fonctions connexes. Julia fournit certains modules dans sa bibliothèque standard. Des fonctionnalités supplémentaires peuvent être ajoutées à partir d'une collection croissante de paquets. Le lecteur se référera à *Julia Observer* .

Les paquets peuvent être installés à partir du REPL en entrant dans le mode Pkg du REPL à l'aide de la touche `]`.

```
(@v1.5) pkg> add https://github.com/BenLauwens/ThinkJulia.jl  
➡ pour la version française :  
(@v1.5) pkg> add https://github.com/aquarelleX332/ThinkJuliaFR.jl
```

L'installation peut prendre du temps.

Avant de pouvoir utiliser les fonctions d'un module, nous devons importer ce dernier avec une déclaration d'utilisation :

```
using ThinkJulia

➡ pour la version française :
using ThinkJuliaFR

🐢 = Turtle()
Turtle(0.0, 0.0, true, 0.0, (0.0, 0.0, 0.0))
```

Le module ThinkJulia (ou ThinkJuliaFR) fournit une fonction appelée [Turtle](#) qui crée un objet nommé [Luxor.Turtle](#), que nous attribuons à une variable nommée [🐢](#) ([\:turtle: TAB](#)). Une fois que nous avons créé une tortue, nous pouvons appeler une fonction pour la déplacer sur une feuille dessin. Par exemple, pour faire avancer la tortue :

```
@svg begin
    forward(🐢, 100)
end
width: 600.0
height: 600.0
filename: luxor-drawing-165611_608.svg
type: svg
```

En conséquence de quoi, un graphique –intitulé, en l'occurrence, [luxor-drawing-165611_608.svg](#)– s'affiche dans la partie Plots du REPL. Nous pouvons effectuer une recherche pour déterminer dans quel répertoire se trouve ce fichier (sous GNU/LINUX : [sudo updatedb](#) et, ensuite en mode utilisateur, [locate](#) indiquera l'emplacement du fichier .svg). Le résultat devrait ressembler au dessin de la figure 4.1.1.

FIGURE 4.1.1 – Déplacement de la tortue en avant (du centre vers la droite).

Le mot-clé `@svg` permet à une macro de fonctionner, qui dessine une image SVG. Les macros sont une fonctionnalité importante mais avancée de Julia. Les arguments de marche sont 🐢 et une distance exprimées en pixels. De ce fait, la taille réelle dépend de l’affichage de l’utilisateur.

`turn` est une autre fonction pouvant être sollicitée avec 🐢 comme premier argument et qui permet de réorienter la tortue. Le deuxième argument que prend `turn` est un angle exprimé en degrés.

De plus, chaque tortue tient un stylo, qui est soit vers le bas soit vers le haut. Si le stylo est vers le bas, la tortue laisse une trace lorsqu’elle se déplace. *A contrario*, si le stylo est relevé, la tortue peut avancer mais sans déposer de trace. Les fonctions `penup` et `pendown` signifient respectivement stylo baissé et stylo relevé.

Pour dessiner un angle droit, modifions l’appel de la macro :

```
@svg begin
  forward(🐢, 100)
  turn(🐢, -90)
  forward(🐢, 100)
end
```

Nous devrions observer un dessin tel que représenté dans la figure 4.1.2.



FIGURE 4.1.2 – Un angle droit dessiné avec le module Turtle.

4.1.1 Exercice

Modifiez la macro pour dessiner un carré. Ne continuez pas tant que vous n’arrivez pas au bon résultat.

4.2 Répétitions simples

Il y a beaucoup de chances que vous ayez écrit ceci :

```
@svg begin
  forward(🐢, 100)
  turn(🐢, -90)
  forward(🐢, 100)
  turn(🐢, -90)
  forward(🐢, 100)
  turn(🐢, -90)
  forward(🐢, 100)
end
```

Nous pouvons faire la même chose de manière plus concise avec une déclaration **for** :

```
julia> for i in 1:4
    println("Hello!")
end
Hello!
Hello!
Hello!
Hello!
```

C'est l'utilisation la plus simple de la déclaration **for**. Nous en verrons davantage plus tard. Cependant, cela devrait suffire pour permettre de reformuler le programme de dessin permettant d'obtenir un carré. Ne continuez pas tant que vous n'aurez pas réussi.

Voici une déclaration **for** qui dessine un carré :

```
🐢 = Turtle()
@svg begin
  for i in 1:4
    forward(🐢, 100)
    turn(🐢, -90)
  end
end
```

La syntaxe d'une déclaration **for** est similaire à celle d'une définition de fonction. Elle comporte un en-tête et un corps qui se termine par le mot-clé **end**. Le corps peut contenir un nombre quelconque d'instructions. Une instruction **for** est également appelée boucle car le flux d'exécution passe par le corps et revient ensuite en boucle vers le haut. Dans le cas présent, le corps est exécuté quatre fois.

Cette version est en fait un peu différente du précédent code de dessin de carré parce qu'elle effectue une rotation supplémentaire après avoir dessiné le dernier côté du carré. Cette rotation supplémentaire allonge le temps d'exécution, mais elle simplifie le code parce que le programme exécute la même chose dans chaque boucle. Cette version a également pour effet de laisser la tortue dans la position de départ, tournée dans la direction initiale.

4.3 Exercices

Voici une série d'exercices utilisant `Turtle`. Ils sont censés être amusants, mais ils ont aussi un but. Pendant que vous travaillez sur ces exercices, réfléchissez à l'intérêt qu'ils présentent.

Conseil. Les sections suivantes présentent des solutions aux exercices. Ne regardez pas avant d'avoir terminé (ou du moins avant d'avoir vraiment essayé).

4.3.1 Exercice

Écrivez une fonction appelée `square` qui prend un paramètre appelé `t`, qui soit une tortue. Elle doit utiliser `Turtle` pour dessiner un carré.

4.3.2 Exercice

Écrivez un appel de fonction qui passe `t` comme argument à `square`, puis exécutez à nouveau la macro.

4.3.3 Exercice

Ajoutez à `square` un autre paramètre, nommé `len`. Modifiez le corps de manière à ce que la longueur des côtés soit `len`, puis modifiez l'appel de fonction pour fournir un deuxième argument. Exécutez à nouveau la macro. Testez le programme avec une plage de valeurs pour `len`.

4.3.4 Exercice

Faites une copie de `square` et changez le nom en `polygon`. Ajoutez un autre paramètre nommé `n` et modifiez le corps pour qu'il dessine un polygone régulier à `n` côtés.

Conseil. L'angle au centre des polygones réguliers à `n`-côtés vaut $360/n$ degrés. Par exemple, pour un dodécagone régulier, cet angle vaut 30° .

4.3.5 Exercice

Écrivez une fonction appelée `circle` incluant comme paramètres une tortue `t` ainsi qu'un rayon `r` et qui dessine un cercle approximatif en appelant `polygon` avec une longueur et un nombre de côtés appropriés. Testez votre fonction avec une gamme de valeurs de `r`.

Conseil. Calculez la circonférence du cercle et assurez-vous que `len * n == circumference`.

4.3.6 Exercice

Développez une version plus générale de `circle` appelée `arc` qui prend un paramètre d'angle supplémentaire et qui détermine la fraction d'un cercle à dessiner. `Angle` s'exprime en degrés. En conséquence, quand `angle = 360`, `arc` doit dessiner un cercle complet.



4.4 Encapsulation

Le premier exercice demande de placer le code de dessin `square` dans une définition de fonction et d'appeler la fonction en passant la tortue comme paramètre. Voici une solution :

```
function square(t)
  for i in 1:4
    forward(t, 100)
    turn(t, -90)
  end
end

t = Turtle()
@svg begin
  square(t)
end
```

Les déclarations `forward` et `turn` sont indentées deux fois pour montrer qu'elles se trouvent à l'intérieur de la boucle `for` figurant elle-même à l'intérieur de la définition de la fonction `square`.

À l'intérieur de la fonction `square`, `t` fait référence à la tortue , donc `turn(t, -90)` a le même effet que `turn(, -90)`. Dans ce cas, pourquoi ne pas appeler le paramètre

🐢 ? L'idée est que `t` peut être n'importe quelle tortue, pas seulement 🐢. Donc un autre animal qui effectue le même travail que 🐢 peut être créé et passé comme argument à `square` :

```
🐘 = Turtle()
@svg begin
    square(🐘)
end
```

Le procédé consistant à incorporer un morceau de code dans une fonction s'appelle l'*encapsulation*. Le premier avantage de l'encapsulation est d'associer un nom au code, qui sert en quelque sorte de documentation. Un autre avantage résulte du fait que lors de la réutilisation ultérieure du code, il devient plus concis d'appeler une fonction deux fois que d'en « copier-coller » le corps.

4.5 Généralisation

L'étape suivante consiste à ajouter un paramètre `len` à `square`. Voici une solution :

```
function square(t, len)
    for i in 1:4
        forward(t, len)
        turn(t, -90)
    end
end

🐢 = Turtle()
@svg begin
    square(🐢, 100)
end
```

L'ajout d'un paramètre à une fonction s'appelle une *généralisation*. Dans la version précédente, le carré dessiné présente toujours la même taille. Dans la dernière version, sa taille peut varier.

L'étape suivante est également une *généralisation*. Au lieu de dessiner des carrés, `polygon` dessine des polygones réguliers avec un nombre quelconque de côtés. Voici une solution :

```
function polygon(t, n, len)
    angle = 360 / n
```

```

    for i in 1:4
        forward(t, len)
        turn(t, -angle)
    end
end

t = Turtle()
@svg begin
    polygon(t, 7, 70)
end

```

Ce code permet de dessiner un heptagone, chacun des côtés ayant une longueur de 70 pixels.

4.6 Conception d'une interface

L'étape suivante consiste à écrire `circle`, qui prend un rayon, `r`, comme paramètre. Voici une solution simple qui utilise `polygon` pour dessiner un polygone à 50 côtés (pentacontagone) :

```

function circle(t, r)
    circumference = 2 * π * r
    n = 50
    len = circumference / n
    polygon(t, n, len)
end

```

La première ligne calcule la circonférence d'un cercle de rayon `r` en utilisant la formule $2\pi r$. Le paramètre `n` est le nombre de segments dans notre approximation d'un cercle, donc `len` est la longueur de chaque segment. Ainsi, la fonction `polygon` dessine-t-elle un pentacontagone qui se rapproche d'un cercle de rayon `r`.

Une des limites de cette solution vient de ce que `n` est une constante, ce qui signifie que pour les très grands cercles, les segments de droite sont trop longs. En revanche, pour les petits cercles, nous perdons du temps à dessiner de très petits segments. Une solution consisterait à généraliser la fonction en prenant `n` comme paramètre. Cela donnerait à l'utilisateur (celui qui appelle `circle`) plus de contrôle, mais l'interface serait moins propre.

L'interface d'une fonction est un résumé de son utilisation : quels sont les paramètres ? Que fait la fonction ? Et quelle est la valeur retournée ? Une interface est « propre » si elle permet à l'utilisateur de faire ce qu'il souhaite sans avoir à s'occuper de détails inutiles.

Dans cet exemple, `r` est un membre de l'interface car ce paramètre conditionne le cercle à dessiner. Le paramètre `n` est moins approprié dans la mesure où il concerne les détails relatifs à la façon dont le cercle doit être rendu.

Plutôt que d'encombrer l'interface, il est préférable de choisir une valeur appropriée de `n` en fonction de la circonférence :

```
function circle(t, r)
  circumference = 2 *  $\pi$  * r
  n = trunc(circumference / 3) + 3
  len = circumference / n
  polygon(t, n, len)
end
```

Désormais, le nombre de segments est un nombre entier proche de la valeur de `circumference / 3` et la longueur de chaque segment vaut 3. Cette valeur est un bon compromis : assez petite pour que les cercles présentent une allure convenable, tout en étant assez grande pour être efficace, et qui plus est, acceptable pour toute taille de cercle. L'addition de 3 à `n` garantit que le polygone ait au moins 3 côtés.

4.7 Refonte (ou *refactoring*)

Quand nous avons écrit `circle`, il a été possible de réutiliser `polygon` car un polygone à grand nombre de côtés est une bonne approximation d'un cercle. Si nous voulions tracer un arc, pourrait-on utiliser `polygon` ou `circle`? Cela demande un peu de travail.

Une possibilité est de commencer avec une copie de `polygon` et de la transformer en `arc`. Le résultat ressemblerait à ceci :

```
function arc(t, r, angle)
  arc_length = 2 *  $\pi$  * r * (angle / 360)
  n = trunc(arc_len / 3) + 1
  step_len = arc_len / n
  step_angle = angle / n
  for i in 1:n
    forward(t, step_len)
    turn(t, -step_angle)
  end
end
```

La deuxième moitié de cette fonction ressemble à `polygon`. Malheureusement, on ne peut pas réutiliser `polygon` sans en changer l'interface. Nous pourrions généraliser `polygon` pour prendre un angle comme troisième argument. Cependant, `polygon` ne

serait plus un nom approprié. En conséquence, renommons de manière plus générale cette fonction en `polyline` :

```
function polyline(t, n, len, angle)
  for i in 1:n
    forward(t, len)
    turn(t, -angle)
  end
end
```

À présent, nous pouvons réécrire `polygon` et `arc` pour tirer parti de `polyline` :

```
function polygon(t, n, len)
  angle = 360 / n
  polyline(t, n, len, angle)
end

function arc(t, r, angle)
  arc_length = 2 *  $\pi$  * r * (angle / 360)
  n = trunc(arc_len / 3) + 1
  step_len = arc_len / n
  step_angle = angle / n
  polyline(t, n, step_len, step_angle)
end
```

Enfin, nous pouvons réécrire `circle` afin d'utiliser `arc` :

```
function circle(t, r)
  arc(t, r, 360)
end
```

Ce processus, qui consiste à réorganiser un programme pour améliorer les interfaces et faciliter la réutilisation du code, s'appelle la *refonte* (ou *refactoring*). Dans le cas présent, nous avons remarqué un code similaire dans `arc` et `polygon`, nous l'avons donc « remanié » (refondu) en `polyline`.

Si nous avions planifié la programmation, nous aurions peut-être écrit `polyline` au premier jet et évité la refonte. Souvent au début d'un projet, on n'en connaît pas assez pour concevoir les interfaces optimales. Lorsqu'on entame le codage, le problème est mieux cerné. Parfois, le *refactoring* est le signe d'une compréhension approfondie d'un problème.

4.8 Plan de développement

Un *plan de développement* est un processus de rédaction de programmes. Celui que nous avons utilisé dans cette étude de cas est l'encapsulation et la généralisation.

Les étapes de ce processus sont les suivantes :

1. commencer par écrire un petit programme sans définition de fonction,
2. une fois le programme opérationnel, identifier une partie cohérente de celui-ci. L'encapsuler dans une fonction et le nommer,
3. généraliser la fonction en ajoutant les paramètres appropriés,
4. répéter les étapes 1 à 3 jusqu'à obtenir un ensemble de fonctions opérationnelles. Copier et coller le code de travail (pour éviter des saisies et déboguer à nouveau),
5. chercher des possibilités d'améliorer le programme en le remaniant (processus de refonte). Par exemple, avec un code similaire à plusieurs endroits, envisager de le refondre dans une fonction générale idoine.

Cette manière de procéder présente quelques inconvénients –nous verrons les alternatives plus tard– mais il s'avère utile si le programmeur ignore comment diviser le programme en fonctions. Cette approche permet de concevoir un programme de manière progressive.

4.9 Documentation interne

Une documentation interne brève (*docstring*) est un bloc de commentaires situé avant une fonction pour en expliciter l'interface :

```
"""
Draws n line segments with the given length and
angle (in degrees) between them. t is a turtle.
"""

function polyline(t, n, len, angle)
    for i in 1:n
        forward(t, len)
        turn(t, -angle)
    end
end
```

On peut accéder à la documentation dans le REPL en saisissant le caractère ? suivi du nom d'une fonction ou d'une macro, et en appuyant sur ENTER :

```
help ?> polyline
search

polyline(t, n, len, angle)
  Draws n line segments with the given length and angle (in degrees) between them. t
  is a turtle.
```

La documentation brève consiste souvent en des chaînes précédées et suivies d'un triple `"`, également appelées chaînes multilignes car les triples guillemets permettent à la chaîne de s'étendre sur plus d'une ligne.

Conseil. Une documentation brève contient les informations essentielles qu'un utilisateur peut requérir pour utiliser correctement une fonction. Elle explique de manière concise ce que fait la fonction, sans toutefois entrer dans les détails techniques. Elle explique l'effet de chaque paramètre sur le comportement de la fonction et le type de chaque paramètre (quand cela n'est pas évident).

4.10 Débogage

Une interface s'apparente à un « contrat » entre une fonction et un appelant. L'appelant accepte de fournir certains paramètres et la fonction consent à effectuer certains travaux.

Par exemple, `polyline` nécessite quatre arguments : `t` doit être de « type » `Turtle`, `n` un nombre entier, `len` un nombre positif et `angle` un nombre exprimé en degrés.

Ces exigences sont appelées conditions préalables (ou *a priori*) car elles sont supposées être vraies avant que la fonction ne commence à s'exécuter. *A contrario*, les conditions résultant de l'exécution de la fonction sont dites *a posteriori*. Ces conditions *a posteriori* comprennent l'effet prévu de la fonction (comme les segments de ligne de dessin) et tout effet secondaire (comme le déplacement de la tortue ou d'autres modifications). Les conditions préalables sont de la responsabilité de l'appelant. Si l'appelant viole une condition préalable (correctement documentée) et que la fonction ne donne pas le résultat attendu, le bogue se trouve chez l'appelant et non dans la fonction. Si les conditions préalables sont satisfaites et que les conditions *a posteriori* ne le sont pas, le bogue se trouve dans la fonction. Si les conditions préalables et les conditions *a posteriori* sont claires, elles peuvent aider au débogage.

4.11 Glossaire

module fichier qui contient une collection de fonctions connexes et diverses définitions,

paquet bibliothèque externe avec des fonctionnalités supplémentaires,

déclaration d'utilisation déclaration qui lit un fichier de module et crée un objet de module,

boucle partie d'un programme qui peut être exécutée de manière répétitive,

encapsulation processus de transformation d'une séquence d'instructions en une définition de fonction,

généralisation processus consistant à remplacer un élément inutilement spécifique (comme un nombre) par élément suffisamment général (comme une variable ou un paramètre),

interface description de la manière d'utiliser une fonction, y compris le nom ainsi que la description des arguments et de la valeur de retour,

refonte (*refactoring*) processus de modification d'un programme pour améliorer les interfaces des fonctions et d'autres qualités du code,

plan de développement processus de conception et de rédaction de programmes,

documentation courte (*docstring*) chaîne qui apparaît juste avant la définition d'une fonction pour documenter l'interface de cette dernière,


condition préalable (ou *a priori*) exigence qui doit être satisfaite par l'appelant avant le début d'une fonction,

condition *a posteriori* exigence qui doit être satisfaite par la fonction avant qu'elle ne prenne fin.

4.12 Exercices

4.12.1 Exercice

À moins d'utiliser des scripts, inscrivez le code de ce chapitre dans un carnet Jupyter ou dans Pluto.

1. Dessinez un diagramme de pile qui montre l'état du programme pendant l'exécution de `circle(, radius)`. Vous pouvez faire l'arithmétique à la main ou ajouter des commentaires au code.

2. La version d'`arc` dans la section 4.7 n'est pas très précise car l'approximation linéaire du cercle se fait toujours en dehors du cercle vrai. Par conséquent, la tortue se retrouve à quelques pixels de sa destination correcte. Ci-dessous, une solution qui illustre un moyen de réduire l'effet de cette erreur. Lisez le code et voyez si cela vous semble logique. Si vous dessinez un diagramme de pile, vous appréhendez très probablement le fonctionnement.

```

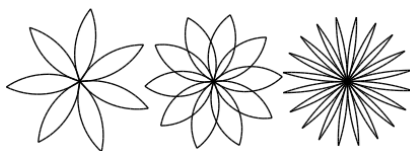
"""
Trace un arc d'un rayon et d'un angle donnés :
t : turtle
r : rayon
angle : angle sous-tendu par l'arc, en degrés
"""
function arc(t, r, angle)
    arc_len = 2 * π * r * abs(angle) / 360
    n = trunc(arc_len / 4) + 3
    step_len = arc_len / n
    step_angle = angle / n

    # faire un léger virage à gauche avant de démarrer
    # réduit l'erreur causée par l'approximation linéaire de l'arc
    turn(t, -step_angle/2)
    polyline(t, n, step_len, step_angle)
    turn(t, step_angle/2)
end

```

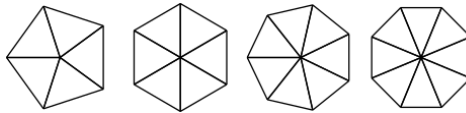
4.12.2 Exercice

Rédigez un ensemble de fonctions générales appropriées pour dessiner des fleurs telles que celles-ci :



4.12.3 Exercice

Rédigez un ensemble de fonctions générales appropriées pour dessiner des formes :



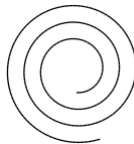
4.12.4 Exercice

Les lettres de l'alphabet peuvent être construites à partir d'un nombre restreint d'éléments de base, comme des lignes verticales et horizontales ainsi que quelques courbes. Concevez un alphabet qui peut être dessiné avec un nombre minimal d'éléments de base et écrivez ensuite des fonctions qui dessinent les lettres.

Vous devez écrire une fonction pour chaque lettre, avec les noms `draw_a`, `draw_b`, etc., et mettre vos fonctions dans un fichier nommé `letters.jl`.

4.12.5 Exercice

Consultez cette [page Wikipédia](#) afin d'apprendre des informations sur les spirales. Ensuite, écrivez un programme qui dessine une spirale d'Archimède telle que ceci :



Chapitre 5

Les conditions et la récursion

Le sujet principal de ce chapitre est la déclaration `if`, qui exécute un code différent selon l'état du programme. Auparavant, considérons deux nouveaux opérateurs : la division euclidienne `÷` (`\div TAB`)¹ et le modulo.

5.1 Division euclidienne (entière) et modulo

L'opérateur de division euclidienne ou entière `÷` divise deux nombres et arrondit au quotient c'est-à-dire au nombre entier inférieur. Par exemple, supposons que la durée d'un film soit de 105 minutes. Combien de temps en heures cela représente-t-il ? La division conventionnelle `/` donne un nombre à virgule flottante :

```
julia> minutes = 105
105
julia> minutes / 60
1.75
```

Or, nous n'écrivons normalement pas les heures avec des points décimaux. La division entière `÷` retourne le quotient :

```
julia> heures = minutes ÷ 60
1
```

Pour obtenir le reste, nous pourrions pratiquer ainsi :

1. Le signe `÷` est un obèle.

```
julia> reste = minutes - heures * 60
45
```

Une autre solution consiste à utiliser l'opérateur modulo, `%`, qui divise deux nombres et retourne le reste.

```
julia> reste = minutes % 60
45
```

Conseil. L'opérateur modulo est plus utile qu'il n'y paraît. Par exemple, dans le cas où il faut déterminer si un nombre est divisible par un autre. Si `x % y` vaut zéro, alors `x` est divisible par `y`. L'opérateur modulo est commode pour extraire le ou les chiffres les plus à droite d'un nombre. Par exemple, en base 10, `x % 10` donne le chiffre le plus à droite d'un nombre entier `x`. De même, `x % 100` donne les deux derniers chiffres.

5.2 Expressions booléennes

Une *expression booléenne* est une expression qui est soit vraie, soit fausse. Les exemples suivants utilisent l'opérateur `==`, qui compare deux opérandes et retourne `true` s'ils sont égaux et `false` sinon :

```
julia> 5 == 5
true
julia> 5 == 6
false
```

`true` et `false` sont des valeurs spéciales qui appartiennent au type `Bool`. Il ne s'agit en rien de chaînes de caractères :

```
julia> typeof(true)
Bool
julia> typeof(false)
Bool
```

L'opérateur `==` fait partie des opérateurs relationnels (voir tableau 5.1).

Avertissement. Bien que ces opérations soient probablement familières, les symboles de Julia sont différents des symboles mathématiques. Une erreur courante consiste à utiliser un signe égal simple (`=`) au lieu d'un signe égal double (`==`). Il faut garder à l'esprit que `=` est un opérateur d'affectation et que `==` est un opérateur relationnel. Il n'existe pas de `=<` ou de `=>`.

TABLE 5.1 – Liste des opérateurs relationnels.

Opérateurs	Signification
$x == y$	# x est égal à y
$x != y$	# x n'est pas égal à y
$x \neq y$	# (\ne TAB)
$x > y$	# x est plus grand que y
$x < y$	# x est plus petit que y
$x \geq y$	# x est plus grand ou égal à y
$x \geq y$	# (\ge TAB)
$x \leq y$	# x est plus petit ou égal à y
$x \leq y$	# (\le TAB)

5.3 Opérateurs logiques

Il y a trois opérateurs logiques : **&&** (et), **||** (ou), et **!** (non). La signification de ces opérateurs est la même que dans le langage courant. Par exemple, $x > 0 \ \&\& \ x < 10$ n'est vrai que si x appartient au segment fermé [1,9].

L'expression $n \% 2 == 0 \ || \ n \% 3 == 0$ est vraie si l'une ou l'autre des conditions (ou les deux) est vraie, c'est-à-dire si le nombre est divisible par 2 ou par 3.

Les deux **&&** et **||** s'associent à droite, mais **&&** a une priorité plus élevée que **||**.

Enfin, l'opérateur **!** annule une expression booléenne, donc $!(x > y)$ est vrai si $x > y$ est faux, c'est-à-dire si x est inférieur ou égal à y .

5.4 Exécution conditionnelle

Afin d'écrire des programmes utiles, il est presque toujours nécessaire de pouvoir vérifier les conditions et modifier le comportement du programme en conséquence. Les déclarations conditionnelles nous offrent cette capacité. La forme la plus simple est la déclaration **if** :

```
if x > 0
    println("x est positif")
end
```

L'expression booléenne après **if** est appelée une *condition*. Si elle est vraie, les instructions du bloc indenté sont exécutées. Sinon, rien ne se passe.

Les instructions `if` ont la même structure que les définitions de fonctions : un en-tête suivi d'un corps terminé par le mot-clé `end`. Les instructions de ce type sont dites composées.

Il n'y a pas de limite au nombre d'instructions formant le corps. Parfois, il est utile d'avoir un corps sans déclaration (généralement pour réserver la place pour un code non-encore écrit).

```
if x < 0
  # À FAIRE gérer les valeurs négatives !
end
```

5.5 Exécution alternative

Une deuxième forme de la déclaration `if` consiste en une « exécution alternative » avec deux possibilités. La condition détermine quelle branche de l'alternative doit être exécutée. La syntaxe est :

```
if x % 2 == 0
  println("x est pair")
else
  println("x est impair")
end
```

Lorsque `x` est divisible par 2, le programme affiche `x est pair`. Si la condition est fausse, la deuxième série d'instructions est exécutée. Comme la condition doit être vraie ou fausse, l'une des branches de l'alternative est toujours exécutée.

5.6 Enchaînement de conditions

Il arrive souvent qu'un programme requière plus de deux branches. Une manière de procéder consiste à exploiter les enchaînements de conditions avec le mot-clé `elseif` :

```
if x < y
  println("x est inférieur à y")
elseif x > y
  println("x est supérieur à y")
else
  println("x est égal à y")
end
```

À nouveau, seule une branche fonctionnera. Il n'y a pas de limite au nombre de déclarations `elseif`. S'il y a une clause `else`, elle doit se trouver à la fin, mais n'est pas nécessaire.

```
if choice == "a"  
  draw_a  
elseif choice == "b"  
  draw_b  
elseif choice == "c"  
  draw_c  
end
```

Chaque condition est examinée dans l'ordre. Si la première est fausse, la suivante est vérifiée, et ainsi de suite. Si l'une d'entre elles est vraie, la branche correspondante est parcourue et la déclaration se termine. Même si plus d'une condition est vraie, seule la première branche vraie est exécutée.

5.7 Imbrication de conditions

Une condition peut être imbriquée dans une autre. L'exemple de la section précédente peut s'écrire ainsi :

```
if x == y  
  println("x est égal à y")  
else  
  if x < y  
    println("x est inférieur à y")  
  else  
    println("x est égal à y")  
  end  
end
```

La condition « extérieure » contient deux branches. La première branche contient une seule instruction. La deuxième branche renferme un autre bloc `if`, qui lui-même comprend deux branches. Ces deux branches contiennent des instructions simples, bien qu'elles puissent également être des instructions conditionnelles à leur tour.

Même lorsque l'indentation est scrupuleusement observée, les conditions imbriquées deviennent très rapidement difficiles à lire. Il est bon de les éviter quand c'est possible.

Les opérateurs logiques permettent souvent de simplifier les déclarations conditionnelles imbriquées. Par exemple, le code suivant peut être reformulé en utilisant une seule condition :

```

if 0 < x
    if x < 10
        println("x est un nombre positif")
    end
end

```

L'instruction d'affichage ne fonctionne que si les deux conditions sont passées. Le même résultat peut être obtenu avec l'opérateur `&&` :

```

if 0 < x && x < 10
    println("x est un nombre positif")
end

```

5.8 Récursion

Il est permis qu'une fonction en appelle une autre et aussi qu'elle s'auto-appelle. C'est un des aspects les plus « magiques » de la programmation. Considérons, par exemple, la fonction suivante :

```

function countdown(n)
    if n ≤ 0
        println("mise à feu !")
    else
        print(n, " ")
        countdown(n-1)
    end
end

```

Si `n` vaut 0 ou est négatif, le programme affiche "mise à feu !". Sinon, il affiche `n` suivi d'un espace. Ensuite, le programme appelle une fonction appelée `countdown` (compte à rebours) se passant à elle-même `n-1` comme argument.

Que se passe-t-il si nous appelons cette fonction comme ceci ?

```

julia> countdown(3)
3 2 1 "mise à feu !"

```

L'exécution de `countdown` commence avec `n = 3`, et comme `n` est supérieur à 0, le programme passe à l'intérieur du `else` et affiche la valeur 3 suivie d'un espace. L'exécution auto-appelle `countdown` qui se poursuit avec `n = 2`. Puisque `n` est supérieur à 0, le programme passe à nouveau dans la partie `else` ; il affiche la valeur 2 suivie d'un espace. Le programme continue avec un auto-appel de `countdown` avec `n = 1`. Puisque `n` est supérieur à 0, le programme passe à nouveau dans la partie `else` ; il affiche la valeur 1 suivie d'un espace. Le cycle se prolonge avec un auto-appel de `countdown` qui se

poursuit avec $n = 0$. Cette fois, comme n n'est pas supérieur à 0, le programme affiche l'expression "mise à feu !".

Une fonction qui s'appelle elle-même est récursive ; le processus qui y est associé se nomme une *récursion*.

Voici un autre exemple d'une fonction qui affiche n fois une chaîne de caractères à l'écran :

```
function printn(s, n)
  if n ≤ 0
    return
  end
  println(s)
  printn(s, n-1)
end
```

Si $n \leq 0$, la déclaration `return` permet de quitter la fonction. Le flux d'exécution revient immédiatement à l'instruction appelante. Donc, le programme s'arrête et les autres lignes de la fonction ne sont pas exécutées. Par exemple, imaginons la fonction appelante suivante :

```
printn("bonjour", 2)
bonjour
bonjour
```

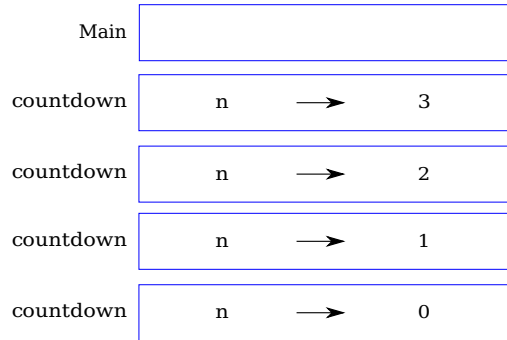
Du fait que $n = 2$, le programme affiche une première fois `bonjour`. Puis, il appelle `printn` avec $n = 1$. Cet auto-appel correspond à `printn("bonjour", 1)`. Le programme affiche une fois encore `bonjour`. Puis, il appelle `printn` avec $n = 0$. En rentrant une troisième fois dans la fonction, le programme constate que la condition $n \leq 0$ est remplie : il effectue un `return`.

Pour des exemples simples comme celui-ci, il est probablement plus facile d'utiliser une boucle `for`. Cependant, nous verrons plus tard des exemples difficiles à écrire avec une boucle `for` et beaucoup plus faciles à rédiger en invoquant une récursion.

5.9 Diagrammes de pile pour les fonctions récursives

Dans la section 3.9, nous avons utilisé un diagramme de pile pour représenter l'état d'un programme lors d'un appel de fonction. Le même type de diagramme peut aider à interpréter une fonction récursive.

Chaque fois qu'une fonction est appelée, Julia crée un cadre pour contenir les variables et paramètres locaux de la fonction. Pour une fonction récursive, il peut y avoir

FIGURE 5.9.1 – Diagramme de pile pour `countdown` appelé avec `n = 3`.

plus d'un cadre sur la pile en même temps. La figure 5.9.1 montre un diagramme de pile pour la fonction `countdown` (compte à rebours) appelée avec `n = 3`.

Comme d'habitude, le haut de la pile est le cadre de `Main`. Il est vide parce qu'aucune variable n'a été instanciée dans `Main` et qu'aucun argument ne lui a été passé. Les quatre cadres de `countdown` ont des valeurs différentes pour le paramètre `n`. Le bas de la pile, où `n = 0`, est appelé le cas de base. Aucun appel récursif n'a lieu pour `n = 0`, donc il n'y a plus de cadres en-dessous.

5.9.1 Exercice

En guise d'exercice, dessinez un diagramme de pile pour une fonction `printn` appelée avec `s = "bonjour"` et `n = 2`. Ensuite, écrivez une fonction appelée `do_n` qui prend un objet de fonction et un nombre `n` comme argument, et qui appelle la fonction `n` fois.

5.10 Récursion infinie

Si une récursion n'atteint jamais le cas de base, elle continue à faire des appels récursifs indéfiniment : le programme ne se termine jamais. Il s'agit là d'une *récursion infinie*, ce qui constitue un mauvais cas de figure. Voici un programme minimal avec une récursion infinie :

```
function recurse()
  recurse()
end
```

Dans la plupart des environnements de programmation, un programme à récursion infinie ne fonctionne pas indéfiniment. Julia envoie un message d'erreur lorsque la profondeur de récursion maximale est atteinte :

```
julia> recurse()  
ERROR: StackOverflowError:  
Stacktrace:  
 [1] recurse() at ./REPL[1]:2 (repeats 80000 times)
```

La trace d'appels (appelée encore trace de pile ou *stacktrace*) est un peu plus grande que celle que nous avons vue dans le chapitre 3. Lorsque l'erreur se produit, il y a 80.000 images récurrentes dans la pile.

Lorsqu'une récursion infinie est rencontrée accidentellement, la fonction incriminée doit être analysée pour confirmer qu'un cas de base ne fait pas d'appel récursif. Si un cas de base existe, il est nécessaire de vérifier que son accès est garanti.

5.11 Saisie au clavier

Les programmes que nous avons écrits jusqu'ici n'acceptent aucune entrée de la part de l'utilisateur. Leur exécution produit toujours le même résultat.

Julia fournit une fonction interne appelée `readline` qui arrête le programme et attend que l'utilisateur saisisse une information. Lorsque l'utilisateur presse la touche ENTER, le programme reprend et `readline` transmet l'information que l'utilisateur a saisie sous forme de chaîne de caractères.

```
julia> texte = readline()  
Qu'attendez-vous ?  
"Qu'attendez-vous ?"
```

Avant d'obtenir la contribution de l'utilisateur, il est conseillé d'afficher un message lui indiquant ce qu'il doit saisir :

```
julia> print("Quel est votre prénom ? "); readline()  
Quel est votre prénom ? Thierry  
"Thierry"
```

Un point-virgule `;` permet de placer plusieurs déclarations sur la même ligne. Dans le REPL, seul le dernier énoncé retourne sa valeur.

S'il est attendu de l'utilisateur qu'il saisisse un entier, il convient de convertir la valeur transmise en `Int64` :

```
julia> print("Quelle est la célérité du son dans l'air au niveau de la mer (m/s)"); c =
readline()
Quelle est la célérité du son dans l'air au niveau de la mer (m/s) ? 340
"340"
julia> parse{Int64, c}
340
```

Toutefois, si l'utilisateur saisit autre chose qu'une chaîne de chiffres, une erreur apparaît à la conversion :

```
julia> print("Quelle est la célérité du son dans l'air au niveau de la mer (m/s)"); c =
readline()
Quelle est la célérité du son dans l'air au niveau de la mer (m/s) ? Que signifie
célérité ?
"Que signifie célérité ?"
julia> parse{Int64, c}
ERROR: ArgumentError invalid base 10 digit 'C' in "Célérité? "
[...]
```

Ultérieurement, nous verrons comment gérer ce genre d'erreur.

5.12 Débogage

Lorsqu'une erreur de syntaxe ou d'exécution se produit, le message d'erreur contient beaucoup d'informations, mais il peut être surchargé. Les parties les plus utiles sont généralement :

- quel type d'erreur s'est-il produit ?
- à quel endroit du programme l'erreur s'est-elle produite ?

Les erreurs de syntaxe sont généralement faciles à trouver, mais quelques astuces existent. En général, les messages d'erreur indiquent l'endroit où le problème a été découvert, mais l'erreur réelle peut se situer en amont dans le code, parfois sur la ligne précédente.

Il en va de même pour les erreurs d'exécution. Supposons que nous tentions de calculer un rapport signal/bruit en décibels (Signal-to-Noise Ratio, SNR). La formule est la suivante :

$$SNR_{dB} = 10 \log_{10} \left(\frac{P_{signal}}{P_{bruit}} \right)$$

où P représente l'amplitude. En Julia, il est possible d'écrire ce code :


```

signal_power = 9
noise_power = 10
ratio = signal_power ÷ noise_power
decibels = 10 * log10(ratio)
print(decibels)

```

Cependant, Julia retourne :

```
-Inf
```

Ce n'est pas le résultat attendu. Pour trouver l'erreur, il pourrait être utile d'afficher la valeur de `ratio`, qui s'avère valoir `0`. Le problème se situe à la ligne 3, qui utilise la division euclidienne `÷` au lieu de la division décimale `/`.

Avertissement. Il est nécessaire de prendre le temps de lire attentivement les messages d'erreur bien qu'ils ne soient pas forcément explicites.

5.13 Glossaire

division euclidienne opération, notée `÷`, qui divise deux nombres et les arrondit au nombre entier inférieur (à la différence de la division décimale qui retourne un quotient à virgule flottante),

opérateur modulo opérateur, désigné par le signe pourcentage `(%)`, qui fonctionne sur des nombres entiers et retourne le reste lorsqu'un nombre est divisé par un autre,

expression booléenne expression dont la valeur est soit vraie (`true`), soit fausse (`false`),

opérateur relationnel un des opérateurs qui compare deux opérandes : `==`, `≠` (`!=`), `>`, `<`, `≥` (`>=`) et `≤` (`<=`),

opérateur logique un des opérateurs qui combine les expressions booléennes : `&&` (et), `||` (ou) et `!` (non),

déclaration conditionnelle déclaration qui contrôle le flux d'exécution en fonction de certaines conditions,

condition expression booléenne dans une déclaration conditionnelle qui détermine quelle branche emprunter,

instruction composée instruction complexe qui se compose d'un en-tête et d'un corps. Le corps est terminé par le mot-clé `end`,

branche une des options d'instructions dans une déclaration conditionnelle,

chaîne conditionnelle déclaration conditionnelle avec une série de branches alternatives,

imbrication conditionnelle déclaration conditionnelle qui apparaît dans une des branches d'une autre déclaration conditionnelle,

instruction de retour instruction qui fait qu'une fonction s'arrête immédiatement et revient à l'appelant,

réursion processus d'auto-appel d'une fonction en cours d'exécution,

cas de base branche conditionnelle dans une fonction réursive qui n'effectue pas d'appel récursif,

réursion infinie réursion qui n'a pas de cas de base ou qui ne l'atteint jamais. Finalement, une réursion infinie provoque une erreur d'exécution.

5.14 Exercices

5.14.1 Exercice

La fonction `time` retourne le temps actualisé, au méridien de Greenwich en secondes depuis une date de référence arbitraire qui, sur les systèmes UNIX, est le 1^{er} janvier 1970.

```
julia> time()
1.602969109299345e9
```

Écrivez un script qui lit l'heure actuelle puis la convertit en jours, heures, minutes et secondes depuis la date de référence.

5.14.2 Exercice

Selon le dernier théorème de Fermat, il n'existe pas d'entiers positifs a , b et c tel que pour toute valeur de $n > 2$:

$$a^n + b^n = c^n$$

1. Écrivez une fonction appelée `checkfermat` qui prend quatre paramètres (a , b , c et n) et qui vérifie si le théorème de Fermat est valide. Si n est supérieur à 2 et que `a^n + b^n == c^n`, le programme doit afficher « Diantre, Fermat avait tort ! ». Sinon, le programme doit afficher « Non, ça ne fonctionne pas... ».

2. Écrivez une fonction qui invite l'utilisateur à entrer des valeurs pour **a**, **b**, **c** et **n**, les convertit en nombres entiers et utilise `checkfermat` pour vérifier s'ils violent le théorème de Fermat.

5.14.3 Exercice

Avec trois bâtonnets, vous pouvez éventuellement construire un triangle. Par exemple, si l'un des bâtonnets mesure 12 cm de long et les deux autres 1 cm de long, vous ne pourrez pas faire se rencontrer les bâtonnets courts. Pour trois longueurs quelconques, il existe un test simple permettant de déterminer s'il est possible de former un triangle :

Conseil. Si une des trois longueurs est supérieure à la somme des deux autres, un triangle sera impossible à former. Par ailleurs, si la somme de deux longueurs est égale à la troisième, nous obtenons un triangle dégénéré.

1. Écrivez une fonction appelée `istriangle` qui prend trois entiers comme arguments, et qui imprime soit "Oui" soit "Non", selon que vous pouvez ou non former un triangle à partir de bâtonnets ayant les longueurs entrées.
2. Écrivez une fonction qui invite l'utilisateur à entrer trois longueurs de bâtonnets, les convertit en nombres entiers et utilise `istriangle` pour vérifier si les bâtonnets ayant les longueurs données peuvent former un triangle.

5.14.4 Exercice

Quel est le résultat du programme suivant ? Dessinez un diagramme de pile qui montre l'état du programme lorsqu'il affiche le résultat.

```
function recurse(n, s)
  if n == 0
    println(s)
  else
    recurse(n-1, n+s)
  end
end
recurse(3, 0)
```

Que se passerait-il si l'appel de fonction devenait : `recurse(-1, 0)` ?

Écrivez une chaîne de caractères qui explique tout ce que quelqu'un doit savoir pour utiliser cette fonction (et rien d'autre).

Les exercices suivants utilisent le module `ThinkJulia` (ou `ThinkJuliaFR`), décrit dans le chapitre 4.

5.14.5 Exercice

Lisez la fonction suivante. Pouvez-vous déterminer ce qu'elle effectue (voir les exemples dans le chapitre 4) ? Ensuite, exécutez-la pour voir si vous avez bien compris.

```
function draw(t, length, n)
  if n == 0
    return
  end
  angle = 50
  forward(t, length*n)
  turn(t, -angle)
  draw(t, length, n-1)
  turn(t, 2*angle)
  draw(t, length, n-1)
  turn(t, -angle)
  forward(t, -length*n)
end
```

Chapitre 6

Fonctions avec retour

De nombreuses fonctions de Julia, telles que les fonctions mathématiques (dont nous avons vu un échantillon) produisent des valeurs de retour. Cependant, toutes les fonctions que nous avons écrites sont vides (ou nulles) : elles ont un effet comme l’affichage d’une valeur ou le déplacement d’une tortue, mais elles ne retournent rien. Ce chapitre aborde les fonctions avec retour.

6.1 Valeurs retournées

Dans ce cas, l’appel d’une fonction retourne une valeur que nous attribuons généralement à une variable ou que nous utilisons dans le cadre d’une expression. Par exemple :

```
e = exp(1.0)
height = radius * sin(radians)
```

Le premier exemple d’une fonction avec retour est [area](#), qui retourne l’aire d’un cercle de rayon donné :

```
function area(radius)
    a = π * radius^2
    return a
end
```

Nous avons déjà vu la déclaration [return](#). Cependant, dans une fonction avec retour, l’instruction [return](#) comprend une expression. Cette instruction signifie littéralement :

« Quitter immédiatement la fonction et utiliser l'expression qui suit `return` comme valeur de retour ». L'expression peut être plus ou moins compliquée. Ainsi, la fonction `area` aurait pu être plus concise :

```
function area(radius)
     $\pi$  * radius^2
end
```

En effet, la valeur retournée par une fonction est la valeur de la dernière expression évaluée qui, par défaut, est la dernière expression figurant dans le corps de la définition de la fonction.

Cependant, des variables temporaires comme `a` (deuxième cadre de cette section) et des déclarations de retour explicites facilitent grandement le débogage. Il est parfois même utile d'avoir plusieurs déclarations de retour, une dans chaque branche conditionnelle :

```
function absvalue(x)
    if x < 0
        return -x
    else
        return x
    end
end
```

Comme ces instructions `return` sont dans un branche conditionnelle alternative, une seule des options est empruntée. Dès qu'une déclaration `return` est exécutée, la fonction se termine sans exécuter d'instructions ultérieures. Le code qui apparaît après une instruction `return` ou après tout autre partie que le flux d'exécution ne peut jamais atteindre, est appelé un *code mort*.

Dans une fonction avec retour, il est bon de s'assurer que tous les chemins possibles du programme aboutissent à une instruction `return`. Par exemple :

```
function absvalue(x)
    if x < 0
        return -x
    end
    if x > 0
        return x
    end
end
```

Cette fonction est incorrecte car si `x` vaut 0, aucune des deux conditions n'est avérée si bien que la fonction se termine sans atteindre une instruction `return`. Si le flux

d'exécution arrive ainsi à la fin d'une fonction, la valeur retournée est `nothing`, ce qui ne correspond pas à la valeur absolue de 0. Ceci est vérifiable :

```
julia> show(absvalue(0))
nothing
```

Conseil. Julia fournit une fonction interne `abs` qui calcule les valeurs absolues.

6.1.1 Exercice

Écrivez une fonction `compare` qui prend deux valeurs, `x` et `y`, et retourne `1` si `x > y`, `0` si `x == y`, et `-1` si `x < y`.

6.2 Développement progressif

À mesure que les fonctions deviennent de plus en plus volumineuses, le temps de débogage s'allonge. Pour faire face à des programmes de plus en plus complexes, il est astucieux de tirer parti d'un procédé appelé *développement progressif* ou *incrémental*. L'objectif est d'éviter de longues sessions de débogage en ajoutant et en testant individuellement un code de petite taille.

Par exemple, supposons que vous vouliez trouver la distance entre deux points, donnée par les coordonnées (x_1, y_1) et (x_2, y_2) . Selon le théorème de Pythagore, la distance est donnée par :

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

La première étape consiste à examiner à quoi devrait ressembler une fonction `distance` en Julia. En d'autres termes, quelles sont les entrées (paramètres) et quelle est la sortie (valeur de retour) ?

Dans ce cas, les entrées sont deux points, que vous pouvez représenter à l'aide de quatre nombres. La valeur de retour est la distance représentée par une valeur en virgule flottante.

Dans l'immédiat, écrivons un aperçu de la fonction :

```
function distance(x1, x2, y1, y2)
    0.0
end
```

Évidemment, cette version ne calcule pas les distances ; elle retourne toujours zéro. Cependant, elle est syntaxiquement correcte. Elle fonctionne, ce qui signifie que vous

pouvez la tester avant de la complexifier. Les numéros d'indice sont disponibles dans le codage des caractères Unicode ([_1 TAB](#), [_2 TAB](#), etc.).

Pour tester la nouvelle fonction, appelons-la avec des exemples d'arguments :

```
distance(1, 2, 4, 6)
```

Ces valeurs ont été choisies pour que la distance horizontale soit 3 et la distance verticale 4. De cette façon, le résultat est 5 : c'est-à-dire l'hypoténuse d'un triangle 3-4-5. Lorsqu'on teste une fonction, il est utile de connaître la bonne réponse.

À ce stade, il est acquis que la fonction est syntaxiquement correcte. Dès lors, du code peut être ajouté dans le corps. La prochaine étape raisonnable consiste à trouver les différences $(x_2 - x_1)$ et $(y_2 - y_1)$. La version suivante enregistre ces valeurs dans des variables temporaires et les affiche avec la macro [@show](#).

```
function distance(x1, x2, y1, y2)
  dx = x2 - x1
  dy = y2 - y1
  @show dx dy
  0.0
end
```

Pour un appel [distance\(1, 2, 4, 6\)](#), la fonction doit afficher [dx = 3](#) et [dy = 4](#). Si c'est le cas, cela signifie que la fonction reçoit les bons arguments et effectue correctement le premier calcul (sinon, il n'y a que quelques lignes à vérifier).

Ensuite, la somme des carrés de [dx](#) et [dy](#) peut être incorporée :

```
function distance(x1, y1, x2, y2)
  dx = x2 - x1
  dy = y2 - y1
  d2 = dx^2 + dy^2
  @show d2
  0.0
end
```

Là encore, il convient de lancer le programme à ce stade et de vérifier le résultat ($d^2 = 25$). Les puissances sont également disponibles ([^2 TAB](#)). Enfin, la fonction [sqrt](#) est introduite afin de calculer et de retourner le résultat :

```
function distance(x1, y1, x2, y2)
  dx = x2 - x1
  dy = y2 - y1
  d2 = dx^2 + dy^2
  sqrt(d2)
end
```


Si tout fonctionne correctement, nous en avons terminé. Sinon, il sera nécessaire d'afficher la valeur de `sqrt(d2)` avant `end` à l'aide de `@show`.

La version finale de la fonction n'affiche rien lorsqu'elle est exécutée. Elle ne fait que retourner une valeur. Les instructions d'affichage que nous avons écrites sont utiles pour le débogage, mais une fois la fonction opérationnelle, il convient de les supprimer. Un tel code est appelé « canevas » (ou *scaffolding*) parce qu'il est utile pour construire le programme tout en n'apparaissant pas dans le produit final.

Au début, il est judicieux de n'ajouter qu'une ou deux lignes de code à la fois. Avec une expérience s'affinant progressivement, il devient possible d'écrire et de déboguer des parties un peu plus longues. Quoi qu'il en soit, le développement incrémental permet de gagner beaucoup de temps de débogage.

Les principaux aspects du processus sont les suivants :

1. commencer par un programme opérationnel et apporter de petites modifications progressives. À tout moment, s'il y a une erreur, il est essentiel d'avoir une bonne idée de l'endroit où elle se trouve,
2. utiliser des variables pour maintenir des valeurs intermédiaires afin de pouvoir les afficher et les vérifier,
3. une fois que le programme complété fonctionne, supprimer une partie du canevas ou consolider plusieurs instructions en expressions composées (voir la section 6.3), mais à la condition que cela ne rende pas le programme difficile à lire.

6.2.1 Exercice

Utilisez le développement incrémental pour écrire une fonction appelée `hypotenuse` qui retourne la longueur de l'hypoténuse d'un triangle rectangle en fonction de la longueur des deux autres côtés comme arguments. Enregistrez chaque étape du processus de développement au fur et à mesure.

6.3 Composition

Bien entendu, une fonction peut être appelée depuis une autre. Par exemple, nous allons écrire une fonction qui prend deux points, le centre d'un cercle et un point sur la circonférence. Ceci nous permet de calculer le rayon et partant, la surface de ce cercle.

Supposons que les coordonnées du centre soient enregistrées dans les variables x_c et y_c , et que les coordonnées du point de la circonférence le soient dans x_p et y_p . La

première étape consiste à trouver le rayon du cercle. Dans la section 6.2, nous avons écrit une fonction `distance`. En transposant pour le rayon :

```
radius = distance(xc, yc, xp, yp)
```

L'étape suivante consiste à trouver l'aire d'un cercle de ce rayon (ce que nous avons aussi écrit dans la section 6.1), ce qui fait que nous pouvons appeler cette fonction :

```
result = area(radius)
```

Par encapsulation, nous obtenons :

```
function circlearea(xc, yc, xp, yp)
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
end
```

Les variables temporaires `radius` et `result` sont utiles pour le développement et le débogage. Lorsque qu'il est opérationnel, le programme peut être rédigé de manière plus concise en composant les appels de fonction :

```
function circlearea(xc, yc, xp, yp)
    area(distance(xc, yc, xp, yp))
end
```

6.4 Fonctions booléennes

Les fonctions peuvent renvoyer des booléens, ce qui est souvent pratique pour cacher des tests complexes à l'intérieur de celles-ci. Par exemple :

```
function isdivisible(x, y)
    if x % y == 0
        return true
    else
        return false
    end
end
```

Il est courant de donner aux fonctions booléennes des noms qui ressemblent à des questions dichotomiques¹. La fonction `isdivisible` retourne soit `true`, soit `false` pour indiquer si `x` est divisible, ou non, par `y`.

1. Questions qui reposent sur une division binaire.

Voici deux exemples :

```
julia> isdivisible(6, 3)
true
julia> isdivisible(6, 4)
false
```

Le résultat de l'opérateur `==` étant un booléen, la fonction peut être reformulée de manière plus concise :

```
function isdivisible(x, y)
    x % y == 0
end
```

Les fonctions booléennes sont souvent utilisées dans les déclarations conditionnelles :

```
if isdivisible(x, y)
    println("x est divisible par y")
end
```

Il pourrait être tentant d'écrire quelque chose comme ceci :

```
if isdivisible(x, y) == true
    println("x est divisible par y")
end
```

Cependant, la comparaison supplémentaire avec `true` est inutile.

6.4.1 Exercice

Écrivez une fonction `isbetween(x, y, z)` qui retourne `true` si $x \leq y \leq z$ ou `false` dans le cas contraire.

6.5 Davantage de récursion

Nous n'avons couvert qu'un petit sous-ensemble de Julia. Cependant, Julia est un langage de programmation complet, ce qui signifie que tout ce qui est calculable peut être exprimé dans ce langage. Tout programme écrit à ce jour pourrait être retranscrit en utilisant uniquement les caractéristiques de Julia telles que présentées jusqu'à présent. En principe, il n'y a plus que quelques commandes à apprendre pour contrôler des périphériques comme la souris, les disques, etc.

Prouver cette affirmation est un exercice complexe réalisé pour la première fois par Alan Turing, un des pionniers de l'informatique² et mathématicien à l'origine. C'est pourquoi elle est connue sous le nom de thèse de Turing³.

Pour se persuader de ce qui est réalisable avec les outils étudiés jusqu'à présent, nous allons évaluer quelques fonctions mathématiques définies de manière récursive. Une définition récursive est analogue à une définition circulaire, dans le sens où la définition contient une référence à ce qui y est défini (ou à son exact contraire). Une définition vraiment circulaire n'est pas très utile⁴. Dans le dictionnaire anglais, voyons la définition de

vorpai an adjective used to describe something that is vorpai⁵.

Nous pourrions imaginer en français :

gauche inverse de la droite (avec droite : inverse de la gauche).

De telles définitions rendent perplexe. En revanche, si on cherche la définition de la fonction factorielle, désignée par le symbole $!$, on obtiendra probablement ceci :

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

Cette définition établit que le factoriel de 0 est 1, et que le factoriel de toute autre valeur n , s'exprime comme n multiplié par le factoriel de $n-1$. Donc $3!$ est 3 fois $2!$ avec $2!$ qui est 2 fois $1!$ avec $1!$ qui est 1 fois $0!$. En combinant le tout, $3!$ est égal à $3 * 2 * 1 * 1$, ce qui fait 6.

Si un programmeur peut rédiger une définition récursive, il peut développer un programme Julia pour l'évaluer. La première étape consiste à décider des paramètres. Dans ce cas, il doit être clair que le factoriel prend un entier :

```
function fact(n)
end
```

S'il advenait que l'argument vaille 0, la fonction devrait retourner 1 :

2. Depuis 1966, un [prix Turing](#) est annuellement décerné à une personne sélectionnée pour sa contribution de nature technique vis-à-vis de la communauté informatique. Les contributions doivent être d'une importance technique majeure et durable dans le domaine informatique.

3. Pour une discussion plus complète et plus précise de la Thèse de Turing, consultez le livre de Michael Sipser, *Introduction to the Theory of Computation* [7].

4. Les truismes, les tautologies, les lapalissades sont de cet ordre.

5. Ce mot a été inventé par Lewis Carroll dans le poème Jabberwocky et semble désigner le tranchant d'une épée. Voir la traduction française de ce poème [sous ce lien](#).

```
function fact(n)
  if n == 0
    return 1
  end
end
```

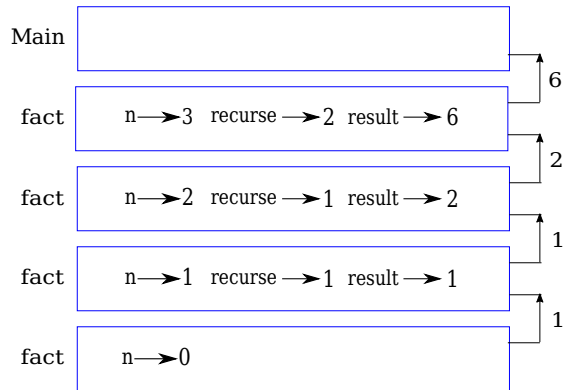
Sinon –et c’est la partie intéressante, nous devons faire un appel récursif pour trouver le factoriel de $n-1$ et ensuite la multiplier par n :

```
function fact(n)
  if n == 0
    return 1
  else
    recurse = fact(n-1)
    result = n * recurse
    return result
  end
end
```

Le flux d’exécution de ce programme est similaire au flux de `countdown` vu dans la section 5.8. Si nous appelons `fact` avec $n = 3$:

- vu que 3 n’est pas égal à 0, nous prenons la deuxième branche et calculons le factoriel de $n-1$...
 - Puisque 2 n’est pas égal à 0, nous prenons la deuxième branche et calculons le factoriel de $n-1$...
 - Puisque 1 n’est pas égal à 0, nous prenons la deuxième branche et calculons le factoriel de $n-1$...
 - Puisque nous arrivons au cas 0, nous prenons la première branche et nous retournons 1 sans faire d’appels récursifs supplémentaires.
 - La valeur de retour, 1, est multipliée par n , qui est 1, et le résultat est retourné.
 - La valeur de retour, 1, est multipliée par n , qui vaut 2, et le résultat est retourné.
- La valeur de retour 2 est multipliée par n (qui vaut 3) et le résultat, c’est-à-dire 6, devient la valeur de retour de l’appel de fonction qui a lancé l’ensemble du processus.

Le diagramme de pile (figure 6.5.1) illustre cette séquence d’appels de fonction. Les valeurs de retour sont indiquées en remontant dans la pile. Dans chaque cadre, la valeur de retour est celle de `result`, qui est le produit de n et de `recurse`. Dans le dernier cadre, les variables locales `recurse` et `result` n’existent pas, car la branche qui les crée n’est pas exécutée.

FIGURE 6.5.1 – Diagramme de pile associé à la fonction `fact(n)`.

Conseil. Julia fournit une fonction interne `factorial` qui calcule le factoriel d'un entier.

6.6 Un acte de confiance

Suivre le déroulement de l'exécution est une manière de lire les programmes, bien que cela puisse s'avérer ardu. Souvent, nous agissons en confiance. Ainsi, lorsque nous arrivons à un appel de fonction, au lieu de suivre le flux d'exécution, nous supposons que la fonction se comporte correctement et retourne le bon résultat.

En fait, ceci est déjà vrai, par exemple, lorsque des fonctions internes sont utilisées. Lorsque `cos()` ou `exp()` sont appelées, leur utilisateur n'examine pas le corps de ces fonctions. Il suppose assez naturellement qu'elles sont opérationnelles parce que les personnes qui les ont écrites sont de bons programmeurs.

Il en va de même lorsque nous appelons une de nos propres fonctions. Par exemple, dans les fonctions booléennes (section 6.4), nous avons écrit `isdivisible` qui détermine si un nombre est divisible par un autre. Une fois que nous nous sommes convaincus que cette fonction est correcte –en examinant le code et en faisant des tests, nous pouvons utiliser la fonction sans avoir à analyser à nouveau le corps.

Il en va de même pour les programmes récursifs. Arrivé à un appel récursif, au lieu de suivre le déroulement de l'exécution, il est supposé que l'appel récursif fonctionne (c'est-à-dire qu'il retourne le résultat correct). Ensuite, survient la question : « En supposant que je puisse trouver le factoriel de $n - 1$, puis-je calculer la factorielle de n ? ». Il est clair que cela peut se faire en multipliant par `n`.

Il est un peu étrange d'accepter que la fonction se comporte correctement alors même qu'on n'a pas fini de l'écrire. C'est pour cela que l'expression « acte de confiance » peut être invoquée.

6.7 Un exemple supplémentaire

Après le factoriel, l'exemple le plus courant d'une fonction mathématique définie de manière récursive est la suite de Fibonacci.

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

Voici un programme Julia représentant cette suite :

```
function fib(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
    end
end
```

Suivre le déroulement de l'exécution, même pour des valeurs assez faibles de **n**, conduit rapidement à la saturation. Néanmoins, selon le principe de l'acte de confiance, il suffit de constater que deux appels récursifs fonctionnent correctement, pour admettre que le bon résultat est obtenu par addition.

6.8 Types et vérification

Que se passe-t-il si **fact** est appelée avec 1.5 comme argument ?

```
julia> fact(1.5)
ERROR: UndefVarError: fact not defined
Stacktrace:
 [1] fact(::Float64) at ./REPL[3]:2
```

Cela ressemble à une récursion infinie. Au premier abord, c'est surprenant car la fonction possède un cas de base lorsque `n == 0`. Cependant, si `n` n'est pas un entier, le cas de base peut ne pas être atteint. En conséquence, la récursion opère à l'infini.

Dans le premier appel récursif, la valeur de `n` est `0.5`. Dans le suivant, elle est `-0.5`. À partir de là, cette valeur devient inférieure à 0 sans jamais passer par 0 toutefois.

Pour résorber ce problème, nous avons deux options : (i) soit essayer de généraliser la fonction factorielle pour travailler avec des nombres à virgule flottante, (ii) soit faire vérifier le type de son argument à la fonction `fact`. La première option recourt à la fonction `gamma` mais cela dépasse le cadre de ce livre. La seconde option est à notre portée.

L'opérateur interne `isa` permet de vérifier le type de l'argument. Par ailleurs, pourquoi ne pas s'assurer que l'argument est positif ?

```
function fact(n)
    if !(n isa Int64)
        error("Factoriel est seulement défini pour les entiers.")
    elseif n < 0
        error("Factoriel n'est pas défini pour les entiers négatifs.")
    elseif n == 0
        return 1
    else
        return n * fact(n-1)
    end
end
```

Le premier cas de base traite des non-entiers, le second des entiers négatifs. Dans les deux cas, le programme affiche un message d'erreur et ne retourne rien pour indiquer que quelque chose s'est mal passé :

```
julia> fact("Knuth")
"Factoriel est seulement défini pour les entiers."
julia> fact(-2)
ERROR: Factoriel n'est pas défini pour les entiers négatifs.
```

Si les deux contrôles sont correctement franchis, cela signifie que `n` est positif ou nul. En conséquence, la preuve est faite que la récursion prend fin.

Ce programme démontre un schéma parfois appelé « sentinelle ». Les deux premières conditions agissent comme des sentinelles, protégeant le code qui suit des valeurs qui pourraient causer une erreur. Les sentinelles permettent de prouver l'exactitude du code.

Dans la section 14.5, nous verrons une option plus souple à l'affichage d'un message d'erreur : la levée d'une exception.

6.9 Débogage

La subdivision d'un programme de grande taille en petites fonctions crée des points de contrôle naturels pour le débogage. Si une fonction n'opère pas correctement, trois possibilités sont à envisager :

1. un problème avec les arguments que la fonction reçoit : une condition *a priori* est violée,
2. un problème avec la fonction elle-même : une condition *a posteriori* est violée,
3. un problème avec la valeur de retour ou la façon dont elle est utilisée.

Pour écarter la première possibilité, il convient d'ajouter une instruction d'affichage au début de la fonction et d'y indiquer les valeurs des paramètres (et peut-être leur type). Une option consiste à écrire un code qui vérifie explicitement les conditions préalables.

Si les paramètres semblent bons, il convient d'ajouter une instruction d'affichage avant chaque instruction de retour et afficher la valeur de retour. Lorsque c'est possible, il est conseillé de vérifier le résultat à la main. Exécuter des appels de la fonction avec des valeurs qui facilitent la vérification du résultat aide beaucoup au débogage (voir la section 6.2 qui traite du développement incrémental).

Si la fonction semble opérationnelle, l'appel de la fonction doit être examiné pour s'assurer que la valeur de retour est utilisée correctement.

L'ajout d'instructions d'affichage au début et à la fin d'une fonction peut contribuer à rendre le flux d'exécution plus compréhensible. Par exemple, voici une version de `fact` avec des instructions d'affichage :

```
function fact(n)
  space = " " ^ (4 * n)
  println(space, "factorial ", n)
  if n == 0
    println(space, "returning 1")
    return 1
  else
    recurse = fact(n-1)
    result = n * recurse
    println(space, "returning ", result)
    return result
  end
end
```

`space` est une chaîne de caractères d'espacement qui contrôle l'indentation de la sortie :

```
julia> fact(4)
      factorial 4
      factorial 3
      factorial 2
      factorial 1
      factorial 0
      returning 1
      returning 1
      returning 2
      returning 6
      returning 24
24
```

Ce type de résultat peut être utile quand l'exécution d'un programme se passe de manière déroutante. Mettre au point un canevas efficace prend du temps, mais cette technique en épargnera beaucoup lors du débogage.

6.10 Glossaire

variable temporaire variable utilisée pour stocker une valeur intermédiaire dans un calcul complexe,

code mort partie d'un programme qui ne peut jamais fonctionner, souvent parce qu'il apparaît après une instruction de retour,

développement progressif (ou incrémental) plan de développement de programmes destiné à limiter le débogage en ajoutant et en testant seulement de petits blocs d'instructions, un à un,

canevas (*scaffolding*) code utilisé pendant le développement du programme mais qui ne fait pas partie de la version finale,

sentinelle modèle de programmation qui utilise une déclaration conditionnelle pour vérifier et gérer les circonstances susceptible de causer une erreur.

6.11 Exercices

6.11.1 Exercice

Dessinez un diagramme de pile pour le programme suivant. Qu'affiche le programme ?

```

function b(z)
    prod = a(z, z)
    println(z, " ", prod)
    prod
end

function a(x, y)
    x = x + 1
    x * y
end

function c(x, y, z)
    total = x + y + z
    square = b(total)^2
    square
end

x = 1
y = x + 1
println(c(x, y+3, x+y))

```

6.11.2 Exercice

La fonction d'Ackermann, $A(m, n)$, est définie comme :

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Rédigez une fonction appelée `ack` qui évalue la fonction d'Ackermann. Utilisez votre fonction pour évaluer `ack(3, 4)`. Le résultat devrait être 125. Que se passe-t-il pour les valeurs plus grandes de `m` et `n` ?

6.11.3 Exercice

Un palindrome est un mot qui s'écrit de la même façon à l'envers et à l'endroit, comme « ada »⁶, « kayak » ou « ressasser ». Récursivement, un mot est un palindrome si la première et la dernière lettre sont identiques et si le milieu est un palindrome.

Voici des fonctions qui prennent un argument de chaîne de caractères et retournent la première, la dernière et le milieu des lettres :

6. Un langage de programmation (voir Wikipédia).

```

function first(word)
    first = firstindex(word)
    word[first]
end

function last(word)
    last = lastindex(word)
    word[last]
end

function middel(word)
    first = firstindex(word)
    last = lastindex(word)
    word[nextind(word, first) : prevind(word, last)]
end

```

Nous verrons comment cela fonctionne dans le chapitre 8.

1. Testez ces fonctions. Que se passe-t-il si vous appelez le milieu avec une chaîne de deux lettres ? Une lettre ? Qu'en est-il de la chaîne vide, qui s'écrit " " et ne contient pas de lettres ?
2. Écrivez une fonction appelée `ispalindrome` qui prend un argument sous la forme d'une chaîne de caractères et soit retourne `true` si c'est un palindrome, soit `false` dans le cas contraire. N'oubliez pas que vous pouvez utiliser la fonction intégrée `length` pour vérifier la longueur d'une chaîne de caractères.

6.11.4 Exercice

Un nombre `a` est une puissance de `b` s'il est divisible d'une part par `b` et que de l'autre, `ab` est une puissance de `b`. Écrivez une fonction appelée `ispuissance` qui prend les paramètres `a` et `b` et qui retourne `true` si `a` est une puissance de `b`.

Conseil. Rappelez-vous les caractéristiques d'un cas de base.

6.11.5 Exercice

Le plus grand commun diviseur (PGCD) de deux entiers `a` et `b` est le plus grand nombre qui divise les deux entiers avec un reste nul.

Il existe une manière simple de trouver le PGCD de deux nombres : si `r` est le reste de la division `a / b`, alors `PGCD(a, b) = PGCD(b, r)`. Comme cas de base, nous pouvons utiliser `PGCD(a, 0) = a`.

Écrivez une fonction appelée `pgcd` qui prend les paramètres `a` et `b` et retourne leur plus grand diviseur commun.

Chapitre 7

Itération

Ce chapitre traite de l'itération, c'est-à-dire de la capacité à exécuter un bloc d'instructions de manière répétée. Une sorte d'itération a déjà été rencontrée lors de l'utilisation de la récursion ([chapitre 5](#)). Une autre l'a été avec les boucles, dans la section 4.2. Dans le présent chapitre, nous utilisons l'instruction `while` qui constitue un autre type d'itération.

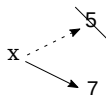
Auparavant, il est utile de revenir sur l'affectation des variables.

7.1 Réaffectation

Comme le lecteur l'a certainement observé, il est permis de faire plus d'une affectation à la même variable. Une nouvelle affectation fait en sorte qu'une variable existante se réfère à une nouvelle valeur et, en conséquence, cesse de se référer à l'ancienne valeur.

```
julia> x = 5
5
julia> x = 7
7
```

La première fois que nous affichons `x`, sa valeur est de `5` ; la deuxième fois, sa valeur est de `7`. La réaffectation peut se représenter dans un diagramme d'état comme suit :



À ce stade, il est temps d'aborder une source commune de confusion. Du fait que Julia utilise le signe égal `=` pour l'affectation, il est tentant d'interpréter une déclaration telle que `a = b` comme une proposition mathématique d'égalité, c'est-à-dire l'affirmation que `a` et `b` sont égaux. *Cette interprétation est erronée.*

Premièrement, l'égalité est une relation symétrique. L'affectation ne l'est pas. Par exemple, en mathématiques, si `a = 7` il est permis d'écrire `7 = a`. Dans Julia, l'énoncé `a = 7` est licite ; `7 = a` ne l'est pas.

En outre, en mathématiques, une proposition d'égalité est soit vraie soit fausse *ad vitam*. Si `a = b` maintenant, alors `a` sera toujours égal à `b`. Dans Julia, une déclaration d'affectation peut rendre deux variables égales, mais elles ne doivent pas nécessairement le rester éternellement.

```
julia> a = 5
5
julia> b = a      # a et b pointent vers la même valeur 5
5
julia> a = 3      # a ne pointe plus vers 5 mais vers 3
3
julia> b          # b pointe toujours vers 5
5
```

La troisième ligne modifie la valeur de `a` mais ne change pas la valeur de `b`, de sorte qu'elles ne sont plus égales.

Avertissement. La réaffectation des variables est souvent utile, mais il faut l'utiliser avec prudence. Si les valeurs des variables changent fréquemment, cela peut rendre le code difficile à lire et à déboguer.

Il n'est pas permis de définir une fonction qui porte le même nom qu'une variable définie précédemment.

7.2 Mise à jour des variables

Il est courant de réaffecter une variable *via* la mise à jour de son ancienne valeur :

```
julia> x = x + 1
8
```

Cela signifie « obtenir la valeur actuelle de `x` (à savoir 7), y ajouter 1 et ensuite affecter 8 à `x` ».

Lors d'une tentative de mise à jour d'une variable non déclarée, Julia retourne une erreur :

```
julia> y = y + 1
ERROR: UndefVarError: y not defined
```

Avant de pouvoir mettre à jour une variable, il est impératif de l'initialiser, généralement grâce à une simple affectation :

```
julia> y = 0
0
julia> y = y + 1
1
```

7.3 while

Les ordinateurs sont souvent utilisés pour automatiser les tâches répétitives. Répéter des tâches identiques ou similaires sans faire d'erreurs constitue un point fort des ordinateurs, contrairement aux humains. Dans un programme informatique, la répétition est également appelée *itération*.

Nous avons déjà vu deux fonctions, `countdown` et `println`, qui utilisent la récursion (section 5.8). Comme l'itération est très courante, Julia propose des fonctions pour la rendre plus facile d'utilisation. L'une d'entre elles est la déclaration `for` que nous avons vue dans la section 4.2. Nous y reviendrons ultérieurement.

`while` constitue une autre déclaration. Voici une version de `countdown` exploitant la déclaration `while` :

```
function countdown(n)
    while n > 0
        print(n, " ")
        n = n - 1
    end
    println("mise à feu !")
end
```

Cette déclaration `while` peut quasiment être lue comme si elle était formulée en langage naturel : « Si n est supérieur à 0, affichez la valeur de n , puis décrémente n . Lorsque vous arrivez à 0, affichez l'expression "mise à feu !" ».

De manière un peu plus formelle, voici le déroulement de l'instruction `while` :

1. déterminer si la condition est vraie ou fausse,
2. si elle est fausse, quitter la déclaration `while` et continuer l'exécution à l'instruction suivante,
3. si la condition est vraie, exécuter le corps et revenir ensuite à l'étape 1.

Ce type de flux est appelé une boucle car la troisième étape revient en boucle vers le haut. Il est fondamental que le corps de la boucle modifie la valeur d'une ou plusieurs variables de sorte que la condition devienne fausse à terme et que la boucle se termine. Sinon, celle-ci se répète indéfiniment (*boucle infinie*).¹

Dans le cas de **countdown**, on peut prouver que la boucle se termine : si **n** est nul ou négatif, la boucle ne se répète jamais. Sinon, **n** est décrémenté à chaque passage dans la boucle, de sorte que **n** finit par valoir 0. Pour certaines autres boucles, ce n'est pas si facile à identifier. Par exemple :

```
function seq(n)
  while n != 1
    println(n)
    if n%2 == 0    # n est pair
      n = n / 2
    else          # n est impair
      n = n*3 + 1
    end
  end
end
```

La condition pour cette boucle est **n != 1**, donc la boucle continue jusqu'à ce que **n** vaille 1 (ce qui rend la condition fausse). À chaque fois que la boucle se poursuit, le programme fournit la valeur de **n** et vérifie ensuite si elle est paire ou impaire. S'il est pair, **n** est divisé par 2 tandis que s'il est impair, la valeur de **n** est remplacée par **n*3 + 1**. Par exemple, si l'argument passé à la séquence vaut 3, les valeurs résultantes de **n** sont 3, 10, 5, 16, 8, 4, 2, 1.

Comme **n** augmente parfois et qu'il diminue parfois, il n'y a pas de preuve évidente que **n** atteindra 1, ou que le programme se termine. Pour certaines valeurs particulières de **n**, nous pouvons prouver la fin du programme. Par exemple, si la valeur de départ est une puissance de deux, **n** sera pair à chaque passage dans la boucle jusqu'à ce qu'il atteigne 1. L'exemple précédent se termine par une telle séquence, en commençant par 16.

La question véritablement difficile est de savoir s'il peut être prouvé que ce programme se termine pour toutes les valeurs positives de **n**. Jusqu'à présent, personne n'a été capable ni de prouver ni de réfuter. Il s'agit de la *conjecture de Collatz*.

1. Une source d'amusement pour les informaticiens provient des instructions sur les bouteilles de sham-ping : « Faire mousser, rincer, répéter », ce qui constitue une boucle infinie.

7.3.1 Exercice

Réécrivez la fonction `printn` (voir la section 5.8) en tirant parti de l'itération plutôt que de la récursion.

7.4 **break**

Parfois, on ne sait pas qu'il est temps de terminer une boucle avant d'avoir parcouru la moitié du corps. Dans ce cas, l'*instruction break* peut être utilisée pour sortir de la boucle. Supposons, par exemple, qu'il faille capter les données d'un utilisateur jusqu'à ce que celui-ci saisisse « done ». Voici un exemple :

```
while true
  print("> ")
  line = readline()
  if line == "done"
    break
  end
  println(line)
end
println("Done !")
```

La boucle conditionnelle est toujours vraie. Donc, la boucle se déroule jusqu'à ce qu'elle atteigne l'instruction `break`.

À chaque passage, la boucle présente une invite `>` à l'utilisateur. Si l'utilisateur saisit le terme `done`, l'instruction `break` force la sortie de la boucle. Sinon, le programme retourne la saisie de l'utilisateur et, ensuite, il revient au début de la boucle. Voici un exemple d'exécution :

```
> not done
not done
> done
Done !
```

Cette manière d'écrire des boucles `while` est courante car on peut vérifier la condition n'importe où dans la boucle (pas seulement en haut) et la condition d'arrêt peut être exprimée affirmativement (« arrêtez quand cela se produit ») plutôt que négativement (« continuez tant que cela ne se produit pas »).

7.5 continue

L'instruction `break` force la sortie d'une boucle. Lorsque l'instruction `continue` est rencontrée à l'intérieur d'une boucle, l'exécution saute au début de la boucle afin d'entamer l'itération suivante. Donc, l'exécution des instructions à l'intérieur du corps de la boucle est contournée. Par exemple :

```
for i in 1:10
    if i % 3 == 0
        continue
    end
    print(i, " ")
end
```

Julia retourne :

```
1 2 4 5 7 8 10
```

Si `i` est divisible par 3, l'instruction `continue` arrête l'itération en cours et, l'itération suivante commence. Seuls les nombres compris entre 1 et 10 non divisibles par 3 sont affichés.

7.6 Racines carrées

Les boucles sont souvent utilisées dans les programmes qui traitent des résultats numériques en commençant par une réponse approximative et en l'améliorant de manière itérative pour converger vers la solution.

Par exemple, la méthode de Newton permet de calculer les racines carrées. Supposons que nous souhaitons connaître la racine carrée d'un nombre α . En commençant avec presque n'importe quelle estimation `x`, il est possible de calculer une meilleure estimation avec la formule suivante :

$$y = \frac{1}{2} \left(x + \frac{\alpha}{x} \right)$$

Si $\alpha = 4$ et `x` = 3 (α s'obtient par `\alpha` TAB et ainsi de suite pour tout l'alphabet grec) :

```
julia> α = 4
4
```

```
julia> x = 3
3
julia> y = (x + α/x) / 2
2.1666666666666665
```

Par rapport à la valeur initiale introduite ($x = 3$), le résultat s'approche davantage de $\sqrt{4} = 2$. Après une répétition, le calcul converge vers la solution :

```
julia> x = y
2.1666666666666665
julia> y = (x + α/x) / 2
2.0064102564102564
```

Après quelques itérations manuelles, l'estimation devient presque exacte :

```
julia> x = y
2.0064102564102564
julia> y = (x + α/x) / 2
2.0000102400262145
julia> x = y
2.0000102400262145
julia> y = (x + α/x) / 2
2.0000000000262146
```

En général, le nombre d'itérations nécessaires pour obtenir la bonne réponse est *a priori* inconnu. En revanche, lorsque la valeur de l'estimation ne change plus, le calcul peut être arrêté :

```
julia> x = y
2.0000000000262146
julia> y = (x + α/x) / 2
2.0
julia> x = y
2.0
julia> y = (x + α/x) / 2
2.0
```

C'est le cas quand $y == x$. Voici une boucle qui commence avec une estimation initiale, x , et qui améliore celle-ci jusqu'à ce que le résultat cesse de changer :

```
while true
  println(x)
  y = (x + α/x) / 2
  if y == x
    break
  end
  x = y
end
```

Pour la plupart des valeurs, ce code fonctionne bien. Cependant, en général, il est dangereux de tester l'égalité de nombres à virgules flottantes. Les valeurs à virgule flottante ne sont qu'approximativement correctes : la plupart des nombres rationnels, comme $\frac{1}{3}$, et les nombres irrationnels, comme $\sqrt{2}$, ne peuvent pas être représentés exactement par un `Float64`.

Plutôt que de vérifier si `x` et `y` sont rigoureusement égaux, il est plus sûr d'utiliser la fonction interne `abs` pour calculer la valeur absolue de leur différence :

```
if abs(x - y) < ε
  break
end
```

Dans ce code, `ε` (`\varepsilon` TAB) a une valeur telle que 0.0000001 qui détermine la proximité au résultat réel.

7.7 Algorithmes

La méthode de Newton est un exemple d'*algorithme*. Il s'agit d'un processus mécanique pour résoudre une catégorie de problèmes (dans ce cas, le calcul de racines carrées).

Pour comprendre ce qu'est un algorithme, il peut être utile de commencer par un procédé qui ne relève pas d'un algorithme. Pour apprendre la multiplication des nombres à un chiffre, il est nécessaire de mémoriser les tables de multiplication avec 100 solutions spécifiques. Ce type de connaissance n'est pas algorithmique.

Toutefois, un individu « paresseux » aurait peut-être appris quelques astuces. Par exemple, pour trouver le produit de `n` et 9, il est possible d'écrire `n - 1` comme premier chiffre et `10 - n` comme deuxième chiffre. Cette astuce est une solution générale pour multiplier tout nombre à un chiffre par 9 : il s'agit d'un algorithme.

De même, les techniques apprises à l'école primaire pour l'addition avec report, la soustraction par emprunt (ou celle par compensation) et la division longue (méthode de

la potence) sont toutes de nature algorithmiques. Les algorithmes ont pour caractéristiques communes de ne nécessiter aucune intelligence pour être exécutés. Ce sont des processus mécaniques où chaque étape suit la précédente selon un ensemble de règles simples.

Exécuter des algorithmes est ennuyeux, mais les concevoir est intéressant, intellectuellement stimulant et constitue une part centrale de l'informatique.

Certains actes naturellement exécutés par les humains, sans difficulté ni pensée consciente, sont les plus difficiles à exprimer par des algorithmes. La compréhension du langage naturel en est un bon exemple. Nous agissons tous ainsi, mais jusqu'à présent, personne n'a pu expliquer comment nous pratiquons, du moins pas sous la forme d'un algorithme.

7.8 Débogage

Plus la taille des programmes augmente, plus de temps de débogage s'allonge. Davantage de code signifie davantage de risques d'erreur et plus d'endroits où les bogues peuvent se camoufler.

Le « débogage par bisection » est une des méthodes permettant de réduire le temps de débogage. Par exemple, s'il y a 100 lignes dans un programme et qu'elles sont vérifiées une à une, il en résultera 100 étapes. Il est plus astucieux de diviser le problème en deux en regardant vers le milieu du programme pour capter une valeur intermédiaire vérifiable. Il suffit d'ajouter une instruction d'affichage (ou quelque technique vérifiable) et lancer le programme. Si la vérification intermédiaire est incorrecte, le problème se situe dans la première moitié du programme. Si la vérification est correcte, le problème se situe dans la seconde moitié.

Chaque fois qu'un contrôle de ce type est utilisé, le nombre de lignes à analyser est divisé par deux. Pour 100 lignes de codes et après seulement six étapes, il n'y a plus qu'une ou deux lignes de code à vérifier, du moins en théorie.

Dans la pratique, il n'est pas toujours aisé de déterminer le « milieu du programme » et une vérification n'est pas toujours réalisable. Il n'est pas pertinent de compter les lignes et de trouver l'exact milieu. Mieux vaut reprérer les endroits du programme où des erreurs pourraient apparaître et où il est facile de pratiquer une vérification. Ensuite, il est judicieux de choisir un endroit où les chances de débusquer le bogue avant ou après la vérification sont approximativement les mêmes.

7.9 Glossaire

réaffectation attribution d'une nouvelle valeur à une variable qui existe déjà,

mise à jour affectation où la nouvelle valeur de la variable dépend de l'ancienne,

initialisation affectation qui donne une valeur initiale à une variable qui sera mise à jour ultérieurement,

incrémentation mise à jour qui augmente la valeur d'une variable (typiquement de 1),

décrémentation mise à jour qui diminue la valeur d'une variable (typiquement de 1),

itération exécution répétée d'un ensemble d'instructions en utilisant soit un appel de fonction récursif, soit une boucle,

while instruction qui permet des itérations contrôlées par une condition,

break instruction permettant de sortir d'une boucle,

continue instruction à l'intérieur d'une boucle qui provoque un saut au début de la boucle pour l'itération suivante,

boucle infinie boucle dans laquelle la condition de fin n'est jamais remplie,

algorithme processus général pour résoudre une catégorie de problèmes.

7.10 Exercices

7.10.1 Exercice

Copiez la boucle de la section 7.6 et encapsulez-la dans une fonction appelée `mysqrt` qui prend `a` comme paramètre, choisit une valeur raisonnable de `x` et retourne une estimation de la racine carrée de `a`.

Pour tester `mysqrt`, écrivez une fonction appelée `testsquareroot` qui affiche un tableau tel que représenté en figure 7.10.1.

7.10.2 Exercice

La fonction interne `Meta.parse` prend une chaîne de caractères et la transforme en une expression. Cette expression peut être évaluée dans Julia avec la fonction `Core.eval`. Par exemple :

a	mysqrt	sqrt	diff
-	-----	----	----
1.0	1.0	1.0	0.0
2.0	1.414213562373095	1.4142135623730951	2.220446049250313e-16
3.0	1.7320508075688772	1.7320508075688772	0.0
4.0	2.0	2.0	0.0
5.0	2.23606797749979	2.23606797749979	0.0
6.0	2.449489742783178	2.449489742783178	0.0
7.0	2.6457513110645907	2.6457513110645907	0.0
8.0	2.82842712474619	2.8284271247461903	4.440892098500626e-16
9.0	3.0	3.0	0.0

FIGURE 7.10.1 – Tableau produit par la fonction `testsquareroot`.

```
julia> expr = Meta.parse("1+2*3")
:(1 + 2 * 3)
julia> eval(expr)
7
julia> expr = Meta.parse("sqrt(π)")
:(sqrt(π))
julia> eval(expr)
1.7724538509055159
```

Écrivez une fonction appelée `evalloop` qui invite l'utilisateur de manière itérative, prend les données résultantes et les évalue à l'aide de la fonction `eval`, puis affiche le résultat. `eval` doit continuer jusqu'à ce que l'utilisateur saisisse `done`, puis retourner la valeur de la dernière expression évaluée.

7.10.3 Exercice

Le mathématicien Srinivasa Ramanujan a trouvé une série infinie qui peut être utilisée pour générer une approximation numérique de $1/\pi$:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Écrivez une fonction appelée `estimatepi` qui utilise cette formule pour calculer et retourner une estimation de π . Elle doit utiliser une boucle `while` pour calculer les termes de la somme jusqu'à ce que le dernier terme soit inférieur à `1e-15` (ce qui est la notation Julia pour 10^{-15}). Vous pouvez vérifier le résultat en le comparant à la valeur de π fournie par Julia.

Chapitre 8

Chaînes

Les chaînes sont des entités différentes des entiers, des flottants et des booléens. Elles font partie des structures de données (comme les tableaux (chapitre 10), les dictionnaires (chapitre 11), les tuples (chapitre 12) et les structures composites (chapitre 16 et suivants)). Une chaîne est une séquence, ce qui signifie qu'elle forme un ensemble ordonné de valeurs. Dans ce chapitre, nous voyons comment accéder aux caractères qui composent une chaîne, et nous découvrirons certaines fonctions d'aide fournies par Julia afin de manipuler des chaînes.

8.1 Caractères

Les locuteurs de langue anglaise sont familiarisés avec les caractères tels que les lettres de l'alphabet (A, B, C, ...), les chiffres et la ponctuation courante. Ces caractères sont normalisés et mis en correspondance avec des valeurs entières comprises entre 0 et 127 par la norme ASCII (*American Standard Code for Information Interchange*).

Il existe bien sûr de nombreux autres caractères utilisés dans d'autres langues que l'anglais, y compris des variantes des caractères ASCII avec des accents et d'autres modifications, des écritures connexes telles que le cyrillique et le grec, et des écritures sans aucun rapport avec l'ASCII et l'anglais, notamment l'arabe, le chinois, l'hébreu, l'hindi, le japonais et le coréen.

La norme Unicode traite les difficultés associées à la définition exacte d'un caractère et elle est généralement acceptée comme la norme définitive pour résoudre ces problèmes. Elle fournit un numéro unique pour chaque caractère à l'échelle mondiale.

Une valeur `Char` représente un seul caractère et elle est entourée de guillemets

droits simples :

```
julia> 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
julia> '🍌'
'🍌': Unicode U+1F34C (category So: Symbol, other)
julia> typeof('x')
Char
```

Même les emojis ¹ font partie de la norme Unicode (exemple `\:banana: TAB`)

8.2 Une chaîne est une séquence

Une chaîne est une séquence de caractères. Il nous est loisible d'accéder aux caractères un à un avec l'opérateur double crochets `[]` :

```
julia> fruit = "banane"
"banane"
julia> letter = fruit[1]
'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
```

La deuxième déclaration sélectionne le caractère numéro 1 de la variable `fruit` et l'attribue à la variable `letter`.

L'expression entre parenthèses contient un indice qui désigne le caractère traité de la séquence.

En Julia, toute séquence indicée commence à 1 : le premier élément de tout objet indicé sur la base de nombres entiers se trouve à l'indice 1 et le dernier élément correspond à `end` (voir la figure 8.2.1).

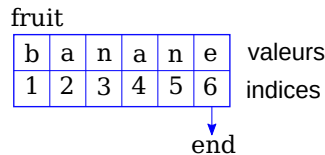


FIGURE 8.2.1 – Exemple d'une table d'indices pour les séquences de caractères.

Par exemple,

1. C'est-à-dire les pictogrammes.

```
julia> fruit[end]
'e': ASCII/Unicode U+0065 (category Ll: Letter, lowercase)
```

Comme pour tout indice, une expression qui contient des variables et des opérateurs peut être employée :

```
julia> i = 1
1
julia> fruit[i+1]
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
julia> fruit[end-1]
'n': ASCII/Unicode U+006E (category Ll: Letter, lowercase)
```

Naturellement, la valeur d'un indice doit être un entier, à défaut de quoi Julia retourne une erreur :

```
julia> fruit[1.5]
ERROR: MethodError: no method matching getindex(::String, ::Float64)
```

8.3 length

`length` est une fonction interne qui retourne le nombre de caractères d'une chaîne :

```
julia> fruit = "🍌🍎🍇"
"🍌🍎🍇"
julia> length(fruit)
5
```

Pour obtenir la dernière lettre d'une chaîne, il est tentant d'essayer ceci :

```
julia> last = length(fruit)
": ASCII/Unicode U+0020 (category Zs: Separator, space)
```

Le résultat est quelque peu inattendu. Les chaînes sont encodées en utilisant le codage UTF-8. Or, l'UTF-8 est un codage à largeur variable, ce qui signifie que tous les caractères ne sont pas codés avec le même nombre d'octets.

La fonction `sizeof` retourne le nombre d'octets d'une chaîne :

```
julia> sizeof("🍌")
4
```

Étant donné qu'un pictogramme est codé sur 4 octets et que l'indexation des chaînes est basée sur les octets, le 5^{ème} élément de `fruit` est un ESPACE. Cela signifie également que l'indice de chaque octet dans une chaîne UTF-8 n'est pas nécessairement

l'indice valide pour un caractère. Si un tel indice d'octet non-valide était utilisé dans une chaîne, Julia afficherait une erreur :

```
julia> fruit[2]
ERROR: StringIndexError("🍌🍏🍐", 2)
```

Dans le cas de `fruit`, le pictogramme 🍌 s'étend sur quatre octets. Les indices 2, 3 et 4 ne sont donc pas valables et l'indice du caractère suivant est 5. Cet indice valable suivant peut être calculé par `nextind(fruits, 1)`. L'indice suivant peut être obtenu par `nextind(fruits, 5)` et ainsi de suite.

8.4 Parcourir une chaîne

De nombreux programmes requièrent le traitement en série d'une ou plusieurs chaînes de caractères. Souvent, ces traitements commencent au tout début. Ils sélectionnent chaque caractère à tour de rôle, effectuent une ou plusieurs modifications puis, continuent jusqu'à la fin de la chaîne. Ce type de traitement correspond à une traversée ou un parcours de chaîne. Une façon d'effectuer un tel traitement consiste à recourir à une boucle `while` :

```
index = firstindex(fruits)
while index <= sizeof(fruits)
    letter = fruits[index]
    println(letter)
    global index = nextind(fruits, index)
end
```

Cette boucle parcourt la chaîne et affiche chaque élément ligne après ligne. La condition de la boucle est `index <= sizeof(fruit)`. Donc, quand l'indice est plus grand que le nombre d'octets dans la chaîne, la condition devient fausse et le corps de la boucle n'est plus exécuté.

La fonction `firstindex` retourne le premier indice d'octet valide. Le mot-clé `global` précédant `index` indique que nous voulons réaffecter l'indice de la variable définie dans `Main` (voir la section 11.7).

8.4.1 Exercice

Écrivez une fonction qui prend une chaîne de caractères comme argument et affiche les lettres à l'envers, une par ligne.

La lecture intégrale d'une chaîne peut être menée à bien avec une boucle `for` :

```
for letter in fruits
  println(letter)
end
```

À chaque passage dans la boucle, le caractère suivant de la chaîne est attribué à la variable `letter`. La boucle se poursuit jusqu'à ce qu'il n'y ait plus de caractère à lire.

L'exemple suivant montre comment utiliser la concaténation (mise bout à bout de plusieurs chaînes de caractères) et une boucle `for` pour produire une série alphabétiquement ordonnée. Dans le célèbre livre américain pour enfants de Robert McCloskey intitulé *Make Way for Ducklings* (dont la traduction en français a pour titre : *Laissez passer les canards*), les noms des canetons sont Jack, Kack, Lack, Mack, Nack, Ouack, Pack et Quack. La boucle `for` ci-dessous produit ces noms dans l'ordre :

```
prefixes = "JKLMNOPQ"
suffix = "ack"
for letter in prefixes
  println(letter * suffix)
end
```

Le programme retourne :

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Quack
```

Ce retour est toutefois incorrect puisque Ouack et Quack ne sont pas correctement écrits.

8.4.2 Exercice

Modifiez le programme précédent pour corriger cette erreur.

8.5 Segments de chaînes

Un segment de chaîne est appelé *slice* en anglais. La sélection d'un segment est similaire à la sélection d'un caractère :

```
julia> str = "Jules César";  
julia> str[1:5]  
"Jules"
```

L'opérateur `[n:m]` retourne la partie de la chaîne du `n`-ième au `m`-ième octet. Il faut donc faire preuve de la même prudence que pour lors de la manipulation d'indices simples.

Le mot-clé `end` peut être utilisé pour indiquer le dernier octet de la chaîne :

```
julia> str[7:end]  
"César"
```

Si le premier indice est supérieur au second, le résultat est une chaîne vide entourée de deux guillemets doubles :

```
julia> str[78:7]  
""
```

Une chaîne vide ne contient aucun caractère et a une longueur égale à 0. Mis à part cela, ses caractéristiques sont identiques à toute autre chaîne.

8.5.1 Exercice

Pour poursuivre cet exemple, que pensez-vous que `str[:]` signifie ? Essayez.

8.6 Les chaînes sont persistantes

Il est tentant d'utiliser l'opérateur `[]` sur le côté gauche d'une affectation, avec l'intention de changer un caractère dans une chaîne. Par exemple :

```
julia> salut = "Hello, world !"  
"Hello, world !"  
julia> salut[1] = 'J'  
ERROR: MethodError: no method matching setindex!{::String, ::Char, ::Int64}
```

La raison de cette erreur provient de ce que les chaînes de caractères sont persistantes ou inchangeables (*immutable* en anglais). Cela signifie qu'une chaîne existante ne peut pas être modifiée. Le mieux qu'on puisse faire est de créer une nouvelle chaîne qui soit une variation de l'originale :

```
julia> salut = J * salut[2:end]  
"Jello, world !"
```

Cet exemple concatène une nouvelle première lettre avec un segment de `salut`. Il n'a aucun effet sur la chaîne de caractères originale. Ceci est explicité dans la figure 8.6.1.

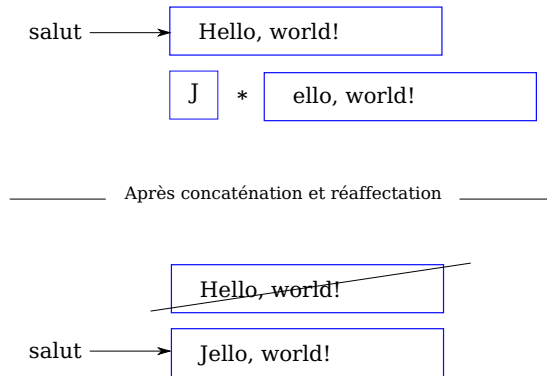


FIGURE 8.6.1 – Schématisation du remplacement de lettre dans une chaîne de caractères.

8.7 Interpolation des chaînes

Construire des chaînes de caractères en utilisant la concaténation peut devenir rapidement ardu. Pour réduire la nécessité de ces appels verbeux à des multiplications de chaînes ou répétées, Julia autorise l'*interpolation de chaînes* à l'aide de `$` :

```
julia> salut1 = "Hello"
"Hello !"
julia> a_qui = "World"
"World"
julia> "$salut1, $(a_qui)!"
"Hello, World!"
```

Cette formulation est plus lisible et plus pratique que la concaténation de chaînes : `salut * ", " * a_qui * " !"`

L'expression complète la plus courte après le `$` est prise comme l'expression dont la valeur doit être interpolée. Ainsi, toute expression dans une chaîne peut être interpolée en utilisant des parenthèses² :

2. similairement à ce qui se pratique dans le shell `bash` par exemple.

```
julia> greet = "1 + 2 = $(1 + 2)"  
"1 + 2 = 3"
```

8.8 Recherche dans les chaînes

Que fait la fonction suivante ?

```
function find(word, letter)  
    index = firstindex(word)  
    while index <= sizeof(word)  
        if word[index] == letter  
            return index  
        end  
        index = nextind(word, index)  
    end  
    -1  
end
```

Dans un sens, cette fonction `find` est l'inverse de l'opérateur `[]`. Au lieu de prendre un indice et d'extraire le caractère correspondant, elle prend un caractère et trouve l'indice où ce caractère apparaît. Si le caractère n'est pas trouvé, la fonction retourne `-1`.

C'est le premier exemple que nous voyons d'une déclaration de retour à l'intérieur d'une boucle. Si `word[index] == letter`, la fonction sort de la boucle et retourne immédiatement la position de la lettre.

Si le caractère n'apparaît pas dans la chaîne de caractères, le programme sort normalement de la boucle en retournant `-1`.

Ce schéma de calcul s'appelle une *recherche*.

8.8.1 Exercice

Modifiez `find` pour qu'elle contienne un troisième paramètre, en l'occurrence l'indice dans `word` où la recherche doit commencer.

8.9 Boucle et compteur

Le programme suivant compte le nombre d'occurrences de la lettre `a` dans une chaîne :


```
word = "banane"
counter = 0
for letter in word
    if letter == 'a'
        global counter = counter + 1
    end
end
println(counter)
```

Ce programme illustre un mode de calcul avec emploi d'un *compteur*. Le compteur est une variable initialisée à 0 puis incrémentée (+1) à chaque fois que la lettre **a** est détectée. Lorsque la boucle se termine, le compteur contient la somme des occurrences de **a**.

8.9.1 Exercice

Encapsulez le code précédent dans une fonction appelée **count**, et généralisez-le pour qu'il accepte la chaîne et la lettre comme arguments.

Ensuite, réécrivez la fonction de sorte qu'au lieu de parcourir la chaîne de caractères, **count** utilise la version à trois paramètres de **find** évoquée dans la section 8.8.

8.10 Bibliothèque des chaînes

Julia fournit des fonctions qui permettent d'effectuer diverses opérations utiles sur les chaînes de caractères. Par exemple, la fonction **uppercase** prend une chaîne de caractères et retourne une nouvelle chaîne, toutes ses lettres étant converties en majuscules.

```
julia> uppercase("Hello, World!")
"HELLO, WORLD!"
```

Il s'avère qu'il existe une fonction appelée **findfirst** qui est remarquablement similaire à la fonction **find** que nous avons écrite :

```
julia> findfirst("a", "banane")
2:2
```

En fait, la fonction **findfirst** est plus générale que notre fonction **find**. En plus des caractères, elle peut trouver des sous-chaînes :

```
julia> findfirst("ane", "banane")
4:6
```

Par défaut, `findfirst` commence au début de la chaîne de caractères, mais la fonction `findnext` prend un troisième argument, l'indice où elle doit commencer :

```
julia> findnext("ne", "banane", 3)
5:6
```

8.11 L'opérateur `∈`

L'opérateur `∈` (`in` TAB) est un booléen³ qui prend un caractère et une chaîne. Il retourne `true` si le caractère apparaît dans la chaîne :

```
julia> 'a' ∈ "banane"
true
```

Par exemple, la fonction suivante permet d'afficher toutes les lettres de `word1` qui apparaissent également dans `word2` :

```
function inboth(word1, word2)
    for letter in word1
        if letter ∈ word2
            print(letter, " ")
        end
    end
end
```

Avec des noms de variables bien choisis, Julia se lit parfois comme du français. Ainsi, cette boucle peut être lue comme ceci : « pour (chaque) lettre du (premier) mot, si (la) lettre est un élément du (deuxième) mot, affichez (la-dite) lettre ». Voici ce qui se passe quand on « compare » des pommes et des poires.

```
julia> inboth("pommes", "poires")
p o e s
```

8.12 Comparaison de chaînes

Les opérateurs relationnels fonctionnent aussi avec les chaînes. Pour voir si deux chaînes sont égales :

3. `∈` signifie « appartient à » ou « est un élément de »

```
word = "Prigogine"
if word == "thermodynamique"
  println("Génial, la thermodynamique.")
end
```

D'autres opérations relationnelles sont utiles pour mettre les mots en ordre alphabétique :

```
word = "Prigogine"
if word < "thermodynamique"
  println("Le mot, $word, vient avant thermodynamique.")
elseif word > "thermodynamique"
  println("Le mot, $word, vient après thermodynamique.")
else
  println("Génial, la thermodynamique.")
end
```

Julia ne gère pas les lettres majuscules et minuscules comme nous le faisons. Toutes les lettres majuscules passent avant les minuscules, donc :

```
Le mot, Prigogine, vient avant thermodynamique.
```

Conseil. Une manière courante de résoudre ce problème consiste à convertir les chaînes de caractères en un format standard, par exemple toutes les minuscules, avant d'effectuer la comparaison.

8.13 Débogage

Lorsque des indices sont exploités pour parcourir les valeurs d'une chaîne séquentiellement, il est délicat d'obtenir le début et la fin exacts de la traversée de chaîne. Voici une fonction qui est censée comparer deux mots et retourner **true** si l'un des mots est l'inverse de l'autre. Cependant, elle contient deux erreurs :

```
function isreverse(word1, word2)
  if length(word1) != length(word2)
    return false
  end
  i = firstindex(word1)
  j = lastindex(word2)
  while j >= 0
    j = prevind(word2, j)
```

```

        if word1[i] != word2[j]
            return false
        end
        i = nextind(word1, i)
    end
    true
end

```

La première déclaration vérifie si les mots sont de la même longueur. Si ce n'est pas le cas, nous pouvons immédiatement retourner `false`. Sinon, pour le reste de la fonction, nous supposons que les mots sont de la même longueur. C'est un exemple avec une sentinelle.

`i` et `j` sont des indices : `i` parcourt `word1` vers l'avant tandis que `j` parcourt `word2` à rebours. Si nous trouvons deux lettres qui ne correspondent pas, nous pouvons retourner `false` immédiatement. Si toute la boucle est parcourue et que toutes les lettres correspondent, le programme retourne `true`.

La fonction `lastindex` retourne le dernier indice d'octet valide de la chaîne de caractères et `prevind` l'indice valide du caractère précédent.

Si nous testons cette fonction avec les mots « pots » et « stop », nous nous attendons à ce que la valeur de retour soit `true`. ce n'est pourtant pas le cas :

```

julia> isreverse("pots", "stop")
false

```

Pour déboguer ce genre d'erreur, la première démarche consiste à imprimer la valeur des indices dans la séquence :

```

while j >= 0
    j = prevind(word2, j)
    @show i j
    if word1[i] != word2[j]
    end
end

```

En relançant le programme, nous obtenons de précieuses informations :

```

julia> isreverse("pots", "stop")
i = 1
j = 3
false

```

Lors du premier passage dans la boucle, `j` vaut 3. Or, ce devrait être 4. Cette valeur peut être fixée en déplaçant la ligne `j = prevind(word2, j)` à la fin de la boucle `while`.

Cela accompli, le programme est relancé, ce qui conduit au résultat :

```
julia> isreverse("pots", "stop")
```

```
i = 1  
j = 4  
i = 2  
j = 3  
i = 3  
j = 2  
i = 4  
j = 1  
i = 5  
j = 0
```

```
ERROR: BoundsError: attempt to access "pots" at index [5]
```

Cette fois, une **BoundsError** a été émise. La valeur de **i** est de 5, ce qui est en dehors de la plage pour la chaîne « pots ».

8.13.1 Exercice

Exécutez le programme sur papier, en changeant les valeurs de **i** et **j** à chaque itération. Trouvez et corrigez la deuxième erreur dans cette fonction.

8.14 Glossaire

séquence collection ordonnée de valeurs où chaque valeur est identifiée par un indice entier,

norme ASCII norme de codage des caractères pour la communication électronique spécifiant 128 caractères,

norme Unicode norme de l'industrie informatique pour le codage, la représentation et le traitement cohérents des textes exprimés dans la plupart des systèmes d'écriture du monde,

indice valeur entière utilisée pour sélectionner un élément dans une séquence, tel qu'un caractère dans une chaîne de caractères. En Julia, les indices commencent à 1,

encodage UTF-8 encodage de caractères à largeur variable capable de coder tous les 1112064 points de code valides en Unicode en utilisant un à quatre octets de 8 bits (abréviation de l'anglais *Universal Character Set Transformation Format – 8 bits*),

traversée lecture des éléments dans une séquence, en effectuant une opération similaire sur chacun d'eux,

segment (*slice*) partie d'une chaîne de caractères délimitée par une série d'indices,

chaîne vide chaîne sans caractères et de longueur 0, représentée par deux guillemets droits,

persistance (ou *immutability*) propriété d'une séquence dont les éléments ne peuvent pas être modifiés,

interpolation de chaîne processus d'évaluation d'une chaîne contenant un ou plusieurs caractères, donnant un résultat dans lequel les caractères sont remplacés par leurs valeurs correspondantes,

recherche modèle de traversée d'une séquence qui s'arrête lorsqu'il trouve ce qu'il cherche,

compteur variable généralement initialisée à zéro puis incrémentée d'une unité à chaque passage dans une boucle.

8.15 Exercices

8.15.1 Exercice

Le but de cet exercice est de consulter la [documentation Julia](#) associées aux chaînes. Vous voudrez peut-être essayer certaines fonctions pour vous assurer que vous comprenez leur fonctionnement. Les méthodes [strip](#) et [replace](#) sont particulièrement utiles.

La documentation utilise une syntaxe qui peut être déroutante. Par exemple, dans [search\(string::AbstractString, chars::Chars, \[start::Integer\]\)](#), les crochets indiquent des arguments optionnels. Ainsi, [string](#) et [chars](#) sont obligatoires tandis que [start](#) est optionnel.

8.15.2 Exercice

Il existe une fonction interne appelée [count](#) similaire à la fonction [count](#) de la section 8.9 (voir en particulier l'exercice 8.9.1 et la section 4.9). Lisez la [documentation de count](#) et utilisez-la pour compter le nombre de [a](#) dans « abracadabra ».

8.15.3 Exercice

Un segment de chaîne peut prendre un troisième indice. Le premier indique le début, le troisième la fin et le deuxième la « taille du pas », c'est-à-dire le nombre d'espaces entre les caractères successifs. Un pas de 2 signifie un caractère sur deux, un pas de 3 signifie un caractère sur trois, etc.

```
julia> mot = "servovalve"  
"servovalve"  
julia> mot[1:2:6]  
"sro"
```

Un pas de -1 traverse le mot à l'envers, de sorte que `[end:-1:1]` produit une chaîne inversée. Utilisez cette technique pour écrire une version d'une ligne de la fonction `ispalindrome` (exercice de la sous-section 6.11.3).

8.15.4 Exercice

Les fonctions suivantes sont toutes destinées à vérifier si une chaîne de caractères contient des minuscules, mais certaines d'entre elles sont erronées. Pour chaque fonction, décrivez ce que la fonction fait réellement (en supposant que le paramètre est une chaîne de caractères).

```
function anylowercase1(s)  
    for c in s  
        if islowercase(c)  
            return true  
        else  
            return false  
        end  
    end  
end  
  
function anylowercase2(s)  
    for c in s  
        if islowercase('c')  
            return "true"  
        else  
            return "false"  
        end  
    end  
end
```

```

        end
    end
end

function anylowercase3(s)
    for c in s
        flag = islowercase(c)
    end
    flag
end

function anylowercase4(s)
    flag = false
    for c in s
        flag = flag || islowercase(c)
    end
    flag
end

function anylowercase5(s)
    for c in s
        if !islowercase(c)
            return false
        end
    end
    true
end

```

8.15.5 Exercice

Un code César est une forme de chiffrement faible qui implique le décalage de chaque lettre par un nombre fixe (le cas échéant, une lettre peut revenir à sa place initiale). Par exemple, 'A' décalé de 3 devient 'D' et 'Z' décalé de 1 devient 'A'.

Pour faire pivoter un mot, il suffit de décaler chaque lettre de la même manière. Par exemple, « oui » décalé de 10 devient « yes » et « lit » décalé de 7 devient « spa », et décalé de -4 devient « hep ». Dans le film *2001 : l'Odyssée de l'espace*, l'ordinateur du vaisseau s'appelle HAL, c'est-à-dire IBM décalé de -1.⁴

Écrivez une fonction appelée `rotateword` qui prend une chaîne de caractères ainsi qu'un entier comme paramètres et retourne une nouvelle chaîne de caractères contenant les lettres de la chaîne originale pivotée de la quantité donnée.

4. Des blagues potentiellement grivoises sur l'internet sont parfois encodées en ROT13, un code César avec un décalage de 13. Si vous n'êtes pas facilement offensé, trouvez et décodez certaines d'entre elles.

Conseil. Vous pouvez utiliser la fonction intégrée `Int`, qui convertit un caractère en un code numérique et `Char`, qui convertit les valeurs numériques en caractères. Les lettres de l'alphabet sont codées dans l'ordre alphabétique, par exemple :

```
julia> Int('c') - Int('a')
```

```
2
```

Cela parce que `c` est la troisième lettre de l'alphabet. Prudence : les codes numériques des lettres majuscules sont différents :

```
julia> Char(Int('A') + 32)
```

```
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```


Chapitre 9

Étude de cas : jeux de mots

Ce chapitre présente la deuxième étude de cas, qui consiste à résoudre des énigmes lexicales en recherchant des mots présentant certaines propriétés. Par exemple, nous trouverons les palindromes les plus longs en anglais et nous chercherons des mots dont les lettres apparaissent dans l'ordre alphabétique. Un autre plan de développement de programmes sera présenté : la réduction à un problème précédemment résolu.

9.1 Lecture de listes de mots

Pour les exercices de ce chapitre, nous utiliserons une liste de mots français. Celle qui convient le mieux à notre objectif est une liste de mots collectée et mise dans le domaine public par Christophe Pallier dans le cadre du projet de dictionnaire Français-GUTenberg de Christophe Pythoud [8]. Il s'agit d'une liste de 336.531 mots encodés en UTF-8. Un navigateur comme Firefox permet d'enregistrer directement ce fichier sous le nom `mots_FR.txt` (par exemple).

Ce fichier est en mode texte brut, tout éditeur de texte peut l'afficher. Cependant, ce fichier est lisible depuis Julia. La fonction intégrée `open` prend le nom du fichier comme paramètre et retourne un flux de fichiers utilisable pour lire le fichier.

```
julia> fin = open("/home/chemin_vers_fichier/mots_FR.txt")  
IOStream(<file /home/chemin_vers_fichier/mots_FR.txt>)
```

`fin` est un flux de fichiers utilisé pour l'entrée et lorsqu'il n'est plus nécessaire, il doit être fermé avec `close(fin)`.

Julia fournit plusieurs fonctions de lecture, dont `readline` qui lit les caractères du fichier jusqu'à arriver à une `NEWLINE` et retourne le résultat sous forme de chaîne :

```
julia> readline(fin)
"a "
```

Le premier mot de cette liste est "a" (conjugaison du verbe « avoir » à la troisième personne au singulier du présent de l'indicatif).

Le flux de fichiers garde une trace de l'endroit où il s'est arrêté. Si bien qu'appeler à nouveau `readline` conduit à l'affichage du mot suivant :

```
julia> readline(fin)
"a "
```

Parce que le fichier peut être parcouru dans une boucle `for` ce programme lit le fichier `mots_FR.txt` et imprime chaque mot, un par ligne :

```
for line in eachline("/chemin_vers_fichier/mots_FR.txt")
    println(line)
end
```

Le dernier mot affiché est « zythums ».

9.2 Exercices

9.2.1 Exercice

Écrivez un programme qui lit `mots_FR.txt` et n'imprime que les mots de plus de 20 caractères (sans compter les espaces).

9.2.2 Exercice

Le roman *La Disparition* est un *lipogramme* écrit par Georges Perec (membre de l'Oulipo) en 1968 et publié en 1969. Il tire son originalité du fait que ses 300 pages (nombre variable selon les éditions) ne comportent pas une seule fois la lettre **e**, pourtant généralement la plus utilisée dans la langue française.

Écrivez une fonction appelée `hasno_e` qui retourne `true` si le mot passé en argument ne contient pas la lettre **e**.

Modifiez votre programme par rapport à la section précédente pour imprimer seulement les mots qui ne contiennent pas de **e** et calculez-en le pourcentage sur la liste.

9.2.3 Exercice

Écrivez une fonction nommée `avoids` qui prend un mot et une chaîne de lettres interdites en arguments, et qui retourne `true` si le mot n'en utilise aucune.

Modifiez votre programme pour demander à l'utilisateur d'entrer une chaîne de lettres interdites, puis imprimez le nombre de mots qui ne contiennent aucune de ces lettres. Pouvez-vous trouver une combinaison de 5 lettres interdites qui exclut le plus petit nombre de mots ?

9.2.4 Exercice

Écrivez une fonction nommée `usesonly` qui prend un mot et une chaîne de lettres, et qui retourne `true` si le mot ne contient que des lettres de cette série. Pouvez-vous faire une phrase (autre que « Hoe alfalfa ») en utilisant uniquement les lettres `acefhlo` ?

9.2.5 Exercice

Écrivez une fonction appelée `usesall` qui prend un mot et une chaîne de lettres obligatoires, et qui retourne `true` si le mot utilise toutes les lettres obligatoires au moins une fois. Combien y a-t-il de mots qui utilisent toutes les voyelles `aeiou` ? Que diriez-vous de `aeiouy` ?

9.2.6 Exercice

Écrivez une fonction appelée `isabecedarian` qui retourne `true` si les lettres d'un mot apparaissent dans l'ordre alphabétique (les lettres doubles sont admises). Combien existe-t-il de mots abécédaires ?

9.3 Recherche

Tous les exercices de la section 9.2 présentent un point commun. Ils peuvent être résolus avec le modèle de recherche dans les chaînes (voir la section 8.8). L'exemple le plus simple est le suivant :

```
function hasno_e(word)
  for letter in word
    if letter == 'e'
      return false
    end
  end
  true
end
```

La boucle `for` traverse les caractères du mot passé en argument. Si la lettre `e` est détectée, le programme retourne immédiatement `false`. Sinon, il passe à la lettre suivante. Si le programme sort de la boucle normalement, cela signifie que la lettre `e` n'a pas été décelée et, par conséquent, le programme retourne `true`.

Cette fonction pourrait être reformulée de manière plus concise en utilisant l'opérateur `notin` (`not in TAB`), mais la version de base illustre bien la logique du schéma de recherche.

La fonction `avoids` est une version plus générale de `hasno_e` bien qu'elle présente la même structure :

```
function avoids(word, forbidden)
  for letter in word
    if letter notin forbidden
      return false
    end
  end
  true
end
```

Le programme peut retourner `false` dès qu'il trouve une lettre interdite passée comme second argument. S'il arrive au bout de la boucle, le programme retourne `true`.

La fonction `usesonly` est analogue, si ce n'est que le sens de la condition est inversé :

```
function usesonly(word, available)
  for letter in word
    if letter notin available
      return false
    end
  end
  true
end
```

Au lieu d'une série de lettres interdites, nous avons une série de lettres permises. Si nous trouvons une lettre qui n'est pas présente dans un mot, le programme retourne `false`.

`useall` est similaire, sauf que nous inversons le rôle du mot et de la série de lettres :

```
function useall(word, required)
  for letter in required
    if letter ∉ word
      return false
    end
  end
  true
end
```

Au lieu de parcourir les lettres du mot, la boucle parcourt les lettres obligatoires. Si l'une des lettres requises n'apparaît pas dans le mot, le programme retourne *false*.

Si vous pensez vraiment comme un informaticien, vous avez identifié que *usesall* est un exemple de problème déjà résolu. Dans ce cas, il est vraisemblable que vous ayez écrit :

```
function useall(word, required)
  usesonly(required, word)
end
```

Il s'agit d'un exemple de plan de développement de programmes appelé *réduction à un problème préalablement résolu*. Cela signifie que le programmeur identifie le problème sur lequel il travaille comme un cas de problème résolu et qu'il applique une solution existante.

9.4 Boucles sur les indices

Les fonctions de la section 9.3 ont été écrites avec des boucles *for* parce que seuls divers caractères des chaînes étaient utiles. Il était donc possible de travailler sans les indices.

Pour *isabecedarian*, nous devons comparer des lettres adjacentes, ce qui est un peu délicat avec une boucle *for* :

```
function isabecedarian(word)
  i = firstindex(word)
  previous = word[i]
  j = nextind(word, i)
  for c in word[j:end]
    if c < previous
      return false
    end
  end
end
```

```

        end
        previous = c
    end
    true
end

```

Une autre possibilité consiste à faire appel à la récursion :

```

function isabecedarian(word)
    if length(word) <= 1
        return true
    end
    i = firstindex(word)
    j = nextind(word, i)
    if word[i] > word[j]
        return false
    end
    isabecedarian(word[j:end])
end

```

Une option supplémentaire revient à exploiter une boucle **while** :

```

function isabecedarian(word)
    i = firstindex(word)
    j = nextind(word, 1)
    while j <= sizeof(word)
        if word[j] < word[i]
            return false
        end
        i = j
        j = nextind(word, i)
    end
    true
end

```

La boucle commence à **i=1** et **j=nextind(word, 1)**. Elle se termine lorsque **j > sizeof(word)**. À chaque passage, la boucle compare le **i^{ème}** caractère (qui peut être considéré comme le caractère courant) au **j^{ème}** caractère (qui est le suivant).

Si le caractère suivant est inférieur en terme d'ordre alphabétique au caractère courant, alors il y a rupture dans le schéma abécédaire. En conséquence, le programme retourne **false**.

Si la fin de la boucle est atteinte sans trouver de rupture, alors le mot passe le test. Pour se convaincre que la boucle se termine correctement, nous pouvons prendre pour exemple « **choux** ».

À présent, voici une version d'`ispalindrome` qui utilise deux indices. L'un commence au début et croît, l'autre commence à la fin et décroît.

```
function ispalindrome(word)
  i = firstindex(word)
  j = lastindex(word)
  while i < j
    if word[i] != word[j]
      return false
    end
    i = nextind(word, i)
    j = prevind(word, j)
  end
  true
end
```

Ce cas peut être réduit à un problème préalablement résolu :

```
function ispalindrome(word)
  isreverse(word, word)
end
```

La fonction `isreverse` a étudiée à la section 8.13.

9.5 Débogage

Tester des programmes est une tâche complexe. Les fonctions présentées dans ce chapitre sont relativement faciles à tester car les résultats sont vérifiables manuellement. Nonobstant cela, il est difficile –voire impossible– de choisir un ensemble de mots qui permettent de tester toutes les erreurs possibles.

Par exemple avec `hasno_e`, deux cas sont évidents à vérifier : les mots qui ont un `e` devraient amener le programme à retourner `false`, et les mots qui n'en ont pas devrait forcer le programme à retourner `true`. Trouver chacun de ces cas est assez facile.

Très souvent, existent des sous-cas moins évidents. Parmi les mots qui contiennent un « e », il convient de tester les mots qui ont un « e » au début, à la fin et quelque part au milieu. Cela implique de tester les mots longs, les mots courts et les mots très courts, comme la chaîne vide. Or, la chaîne vide est un exemple de cas particulier non trivial où des erreurs se camouflent souvent.

En plus des situations de test à traiter manuellement, tester le programme avec une liste de mots comme `mots_FR.txt` est nécessaire. En analysant la sortie, il est peut-être possible de détecter des erreurs. Attention, toutefois : on court toujours le risque de

détecter un type d'erreur (des mots qui ne devraient pas être inclus, mais qui le sont) et inversement (des mots qui devraient être inclus et qui... ne le sont pas).

En général, les tests contribuent à trouver des bogues. En tout état de cause, il n'est guère facile de produire un bon ensemble de tests. Même dans ce cas, personne ne peut être sûr à 100% que le programme est correct. Selon le célèbre informaticien Edsger W. Dijkstra :

*Program testing can be used to show the presence of bugs, but never to show their absence!*¹

9.6 Glossaire

flux de fichiers valeur qui représente un fichier ouvert,

réduction à un problème préalablement résolu procédé destiné à résoudre un problème en l'exprimant comme un cas de figure déjà résolu,

cas particulier cas de test atypique ou non trivial (et susceptible de ne pas être traité correctement).

9.7 Exercices

9.7.1 Exercice

Cet exercice est basée sur un casse-tête qui fut diffusé dans l'émission de radio Car Talk :

« Donnez-moi un mot avec trois paires de lettres consécutives.

Je vous donne deux mots qui se qualifient presque. Par exemple, le mot « committee », c-o-m-m-i-t-t-e-e. Ce serait génial, sauf pour le i qui se faufile au milieu. Ou « Mississippi » : M-i-s-s-i-s-s-i-p-p-i. Si vous pouviez enlever ces i, ce serait parfait. Pourtant il existe un mot qui a trois paires de lettres consécutives et, à ma connaissance, c'est peut-être le seul mot. Bien sûr, il y en a probablement 500 autres, mais je n'en connais qu'un seul. Quel est ce mot ? »

Écrivez un programme qui résout ce problème.

1. Les tests de programmes peuvent être utilisés pour montrer la présence de bogues, mais jamais pour montrer leur absence !

9.7.2 Exercice

Voici un autre casse-tête de l'émission de radio Car Talk :

« L'autre jour, je conduisais sur l'autoroute et, par hasard, j'ai jeté un coup d'oeil à mon compteur kilométrique. Comme la plupart des odomètres, il affiche six chiffres, en kilomètres entiers seulement. Ainsi, si ma voiture avait par exemple fait 300.000 miles, je verrais 3-0-0-0-0-0.

Ce que j'ai vu ce jour-là était très intéressant. J'ai remarqué que les quatre derniers chiffres étaient palindromiques, c'est-à-dire qu'ils lisaient de la même façon en avant et en arrière. Par exemple, 5-4-4-5 est un palindrome. Donc, mon odomètre aurait pu lire 3-1-5-4-4-5.

Un kilomètre plus tard, les 5 derniers chiffres étaient palindromiques. Par exemple, il aurait pu lire 3-6-5-4-5-6. Un kilomètre plus loin, les 4 chiffres du milieu sur 6 étaient palindromiques. Vous êtes prêt ? Un kilomètre plus tard, les 6 chiffres formaient un palindrome !

La question est de savoir ce qui était inscrit sur l'odomètre quand je l'ai regardé la première fois. »

Rédigez un programme Julia qui teste tous les nombres à six chiffres et qui affiche tous les nombres répondant aux exigences du casse-tête.

9.7.3 Exercice

Voici un dernier casse-tête de Car Talk :

« Récemment, j'ai eu la visite de ma mère. Nous avons réalisé que les deux chiffres qui composent mon âge, lorsqu'ils sont inversés, donnent son âge. Par exemple, si elle est âgée de 73 ans, j'ai 37 ans. Nous nous sommes demandées combien de fois cela s'était produit au fil des ans, mais nous nous sommes laissées distraire par d'autres sujets et nous n'avons jamais trouvé de réponse.

En rentrant chez moi, je me suis rendu compte que les chiffres de notre âge ont été réversibles six fois jusqu'à présent. J'ai aussi compris que si nous avons de la chance, cela se reproduirait dans quelques années, et si nous avons vraiment de la chance, cela se reproduirait encore une fois après. En d'autres termes, cela se produirait 8 fois en tout. La question est donc de savoir quel âge j'ai maintenant ? »

Écrivez un programme Julia qui cherche des solutions à cette énigme.

Conseil. La fonction `lpad` pourrait vous être utile.

Chapitre 10

Tableaux

Ce chapitre présente un des types intégrés les plus utiles de Julia : les tableaux. Nous allons apprendre également ce qu'est un objet et ce qui se produit éventuellement lorsque plusieurs noms désignent le même objet.

10.1 Un tableau est une séquence

À l'instar d'une chaîne de caractères, un *tableau* est une séquence de valeurs. Dans une chaîne, les valeurs sont des caractères. Dans un tableau, elles peuvent être de n'importe quel type. Les valeurs d'un tableau sont appelées des *éléments* (ou parfois des *items*).

Il existe plusieurs façons de créer un nouveau tableau. La plus simple consiste à mettre les éléments entre crochets [] :

```
julia> [10, 20, 30, 40]
4-element Array{Int64,1}:
 10
 20
 30
 40
```

```
julia> ["ici", "là", "encore", "déjà"]
julia> ["spam", 2.0, 5, [10, 20]]
```

La deuxième ligne du dernier encadré illustre un cas de tableau *imbriqué*.

Un tableau qui ne contient aucun élément est dit « vide ». Il est possible d'en créer un avec des crochets sans contenu `[]`.

Comme on peut s'y attendre, il est possible d'affecter des tableaux à des variables :

```
julia> fromages = ["Maroilles", "Beaufort", "Ossau-iraty"];
julia> nombres = [42, 123];
julia> vide = [];
julia> print(fromages, " ", nombres, " ", vide)
["Maroilles", "Beaufort", "Ossau-iraty"] [42, 123] Any[]
```

La fonction `typeof` permet de connaître le type d'un tableau :

```
julia> typeof(fromages)
Array{String,1}
julia> typeof(nombres)
Array{Int64,1}
julia> typeof(vide)
Array{Any,1}
```

Entre les accolades, se trouvent le type et un nombre. Ce nombre indique les dimensions du tableau (un tableau simple a une seule dimension ; un tableau bidimensionnel ou matrice, deux, etc.). Le tableau `vide` contient des valeurs de type `Any`, c'est-à-dire qu'il peut contenir des valeurs de tout type.

10.2 Les tableaux sont non-persistants

La syntaxe pour accéder aux éléments d'un tableau est la même que celle permettant d'accéder aux caractères d'une chaîne : il convient d'utiliser l'opérateur `[]`. L'expression à l'intérieur des parenthèses spécifie l'indice. Pour rappel, en Julia, les indices commencent à 1 comme indiqué dans la figure 10.2.1.

alpha

"spam"	2.0	5	<table><tr><td>10</td><td>20</td></tr><tr><td>1</td><td>2</td></tr></table>	10	20	1	2	valeurs
10	20							
1	2							
1	2	3	4	indices				

FIGURE 10.2.1 – Tableau nommé alpha et ses indices.

Par exemple :

```
julia> fromages[1]
"Maroilles"
```

```
julia> alpha = ["spam", 2.0, 5, [10, 20]]
julia> alpha[4][2]
20
```

Contrairement aux chaînes, les tableaux sont non-persistents (c'est-à-dire qu'ils sont modifiables). Lorsque l'opérateur `[]` apparaît à gauche d'une affectation, il identifie l'élément du tableau concerné par cette dernière :

```
julia> nombres[2] = 5
5
julia> print(nombres)
[42, 5]
```

Alors qu'il était auparavant égal à 123, le second élément de `nombres` vaut désormais 5.

La figure 10.2.2 montre les diagrammes d'état pour `fromages`, `nombres` et `vide`.

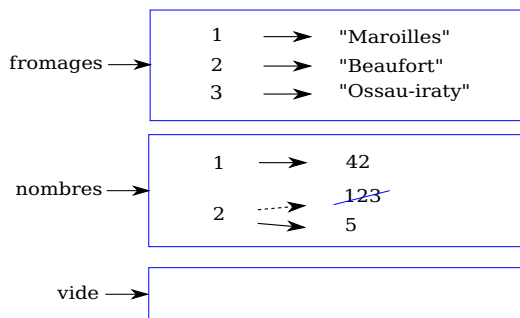


FIGURE 10.2.2 – Diagrammes d'état pour 3 tableaux élémentaires.

Les tableaux sont représentés par des cases contenant les éléments et les indices. `fromages` fait référence à un tableau avec trois éléments indicés 1, 2 et 3. Le tableau `nombres` contient deux éléments. Le diagramme montre que la valeur du deuxième élément a été réaffectée de 123 à 5. Quant à lui, `vide` fait référence à un tableau sans élément.

Les indices de tableau fonctionnent de la même manière que ceux des chaînes (mais sans les mises en garde UTF-8) :

- Toute expression entière peut être utilisée comme un indice. Par exemple, `alpha[1+1]` retourne `2.0`.
- Si vous essayez de lire ou d'écrire un élément qui n'existe pas, vous obtenez une `BoundsError`.
- Le mot-clé `end` pointe vers le dernier indice d'un tableau.

L'opérateur `∈` fonctionne également sur les tableaux :

```
julia> "Ossau-iraty" ∈ fromages
true
julia> "Brie" in fromages
false
```

10.3 Parcourir un tableau

Avant de parcourir un tableau, il est utile de voir comment déterminer le nombre d'éléments qui le constituent en utilisant la fonction `length` :

```
julia> length(nombres)
2
julia> length(alpha)
4
```

Bien qu'un tableau puisse en contenir un autre (comme c'est le cas pour `alpha`), le tableau imbriqué compte toujours comme un élément unique.

La manière la plus courante de parcourir les éléments d'un tableau consiste à utiliser une boucle `for`. La syntaxe est la même que pour les chaînes de caractères :

```
for fromage in fromages
    println(fromage)
end
```

Cela ne fonctionne bien que quand où il suffit de lire les éléments du tableau. S'il est nécessaire d'en ajouter, d'en retirer ou d'en mettre à jour, il faut manipuler les indices. Une façon courante de procéder repose sur l'usage de la fonction interne `eachindex` :

```
for i in eachindex(nombres)
    nombres[i] = nombres[i] * 2
end
```

Cette boucle `for` parcourt le tableau et met à jour chaque élément. Chaque fois que la boucle est parcourue, nous obtenons l'indice de l'élément suivant. L'instruction d'affectation dans le corps utilise `i` pour lire l'ancienne valeur de l'élément et pour affecter la nouvelle valeur.

Une boucle `for` sur un tableau vide ne fait jamais tourner le corps de cette boucle :

```
for x in []  
    println("Ceci n'arrivera pas !")  
end
```

10.4 Segments de tableau

L'opérateur de segmentation fonctionne également avec les tableaux :

```
julia> t = ['a', 'b', 'c', 'd', 'e', 'f'];  
julia> print(t[1:3])  
['a', 'b', 'c']  
julia> print(t[3:end])  
['c', 'd', 'e', 'f']
```

L'opérateur de segmentation `[:]` effectue une copie de l'ensemble du tableau :

```
julia> print(t[:])  
['a', 'b', 'c', 'd', 'e', 'f']
```

Les tableaux étant non persistants, il est souvent utile d'en faire une copie avant d'effectuer des opérations qui les modifient.

Un opérateur de segmentation sur le côté gauche d'une affectation peut mettre à jour plusieurs éléments d'un coup :

```
julia> t[2:3] = ['x', 'y'];  
julia> print(t)  
['a', 'x', 'y', 'd', 'e', 'f']
```

10.5 Bibliothèque de fonctions associées aux tableaux

Plusieurs fonctions internes de Julia sont conçues pour agir sur les tableaux. Par exemple, `push!` ajoute un nouvel élément à la fin d'un tableau :

```
julia> t = ['a', 'b', 'c'];  
julia> push!(t, 'd');  
julia> print(t)  
['a', 'b', 'c', 'd']
```

La fonction `append!` ajoute des éléments d'un second tableau à la fin d'un premier :

```
julia> t1= ['a', 'b', 'c'];  
julia> t2= ['d', 'e'];  
julia> append!(t1, t2);  
julia> print(t1)  
['a', 'b', 'c', 'd', 'e']
```

Il doit être noté que `t2` n'est pas modifié au cours de cette opération.

La fonction `sort!` organise les éléments du tableau dans l'ordre croissant (ou alphabétique) :

```
julia> t = ['d', 'c', 'e', 'b', 'a'];  
julia> sort!(t);  
julia> print(t)  
['a', 'b', 'c', 'd', 'e']
```

La fonction `sort` retourne une copie des éléments du tableau dans l'ordre :

```
julia> t1 = ['d', 'c', 'e', 'b', 'a'];  
julia> t2 = sort(t1);  
julia> print(t1)  
['d', 'c', 'e', 'b', 'a']  
julia> print(t2)  
['a', 'b', 'c', 'd', 'e']
```

Note. Par convention syntaxique, dans Julia, `!` est associé aux noms des fonctions qui modifient leurs arguments.

10.6 Mise en correspondance (*mapping*), filtre et réduction

Pour additionner tous les nombres d'un tableau, une boucle comme celle-ci est utilisable :

```
function addall(t)
    total = 0
    for x in t
        total += x
    end
    total
end
```

La variable locale `total` est initialisée à 0. À chaque passage dans la boucle `for`, `+=` extrait un élément du tableau et l'ajoute à `total`. L'opérateur `+=` constitue un raccourci pour mettre à jour une variable. Cette *déclaration d'affectation avec incrémentation*,

```
total += x
```

équivalent à :

```
total = total + x
```

À chaque passage dans la boucle, `total` accumule la somme des éléments. Une variable utilisée de cette façon est appelée un *accumulateur*.

L'addition des éléments d'un tableau est une opération si courante que Julia fournit la fonction interne `sum` :

```
julia> t = [1, 2, 3, 4]
sum(t)
10
```

Une opération de ce type qui combine une séquence d'éléments en une seule valeur s'appelle une *opération de réduction*.

Souvent, il faut parcourir un tableau tout en en construisant un autre. Par exemple, la fonction suivante prend un tableau de chaînes de caractères et retourne un nouveau tableau qui contient les chaînes en majuscules :

```
function capitalizeall(t)
    res = []
    for s in t
        push!(res, uppercase(s))
    end
    res
end
```

La variable `res` est initialisé avec un tableau vide. À chaque passage par la boucle, l'élément suivant est ajouté à `res` qui, en conséquence, constitue un autre type d'*accumulateur*.

L'opération qu'effectue `capitalizeall` est parfois appelée *mapping* parce que cette fonction effectue une « mise en correspondance » sur chacun des éléments d'une séquence (en l'occurrence avec une conversion en majuscules).

Une autre opération courante consiste à sélectionner certains des éléments d'un tableau et à retourner un sous-tableau. Par exemple, la fonction suivante prend un tableau de chaînes de caractères et retourne un tableau qui ne contient que les chaînes en majuscules :

```
function onlyupper(t)
    res = []
    for s in t
        if s == uppercase(s)
            push!(res, s)
        end
    end
    res
end
```

Une fonction comme `onlyupper` agit tel un *filtre* puisqu'elle opère sélectivement sur certains éléments.

La plupart des opérations courantes sur les tableaux peuvent être exprimées comme une combinaison de « mise en correspondance » (*mapping*), de filtrage et de réduction.

10.7 Syntaxe avec *dots*

Pour chaque opérateur binaire tel que `^`, il y a un opérateur pointé correspondant `.^` automatiquement défini pour effectuer *distributivement* `^` élément par élément sur les tableaux. Par exemple, `[1, 2, 3] ^ 3` n'est pas défini, mais `[1, 2, 3] .^ 3` (attention à l'espace avant `.^`) est défini comme le calcul du résultat élément par élément `[1^3, 2^3, 3^3]` :

```
julia> print([1, 2, 3] .^ 3)
[1, 8, 27]
```

Dans Julia, toute fonction `f` peut être appliquée de manière distribuée à tout tableau. Par exemple, pour mettre en majuscule un tableau de chaînes de caractères, nous n'avons pas besoin d'une boucle explicite :

```
julia> t = uppercase(["abc", "def", "ghi"]);
julia> print(t)
["ABC", "DEF", "GHI"]
```

C'est une façon élégante de créer un *mapping*. La fonction `capitalizeall` peut être mise en œuvre en une seule ligne :

```
function capitalizeall(t)
    uppercase.(t)
end
```

10.8 Suppression et insertion d'éléments

Il existe plusieurs façons de supprimer des éléments appartenant à un tableau. Si l'indice de l'élément à supprimer est connu, la fonction `splice!` est utilisable :

```
julia> t = ['a', 'b', 'c'];
julia> splice!(t, 2)
'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
julia> print(t)
['a', 'c']
```

`splice!` modifie le tableau et retourne l'élément supprimé.

`pop!` supprime le dernier élément d'un tableau et le retourne :

```
julia> t = ['a', 'b', 'c'];
julia> pop!(t)
'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
julia> print(t)
['a', 'b']
```

`popfirst!` supprime le premier élément et le retourne :

```
julia> t = ['a', 'b', 'c'];
julia> popfirst!(t)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
julia> print(t)
['b', 'c']
```

Les fonctions `pushfirst!` et `push!` insèrent un élément au début et à la fin d'un tableau, respectivement.

Si vous n'avez pas besoin de la valeur supprimée, vous pouvez tirer profit de la fonction `deleteat!` :

```
julia> t = ['a', 'b', 'c'];
julia> print(deleteat!(t, 2))
['a', 'c']
```

La fonction `insert!` insère un élément à un indice donné :

```
julia> t = ['a', 'b', 'c'];
julia> print(insert!(t, 2, 'x'))
['a', 'x', 'b', 'c']
```

10.9 Tableaux et chaînes

Une chaîne est une séquence de caractères alors qu'un tableau est une séquence de valeurs. Ceci dit, un tableau de caractères n'est pas une chaîne. Pour convertir une chaîne de caractères en un tableau de caractères, il convient d'employer la fonction `collect` :

```
julia> t = collect("spam");
julia> print(t)
['s', 'p', 'a', 'm']
```

La fonction `collect` permet de décomposer une chaîne ou une autre séquence en éléments individuels.

Si vous souhaitez décomposer une chaîne de caractères en mots, vous pouvez utiliser la fonction `split` : ¹

```
julia> t = split("Au calme clair de lune triste et beau");
julia> print(t)
SubString{String}["Au", "calme", "clair", "de", "lune", "triste", "et", "beau"]
```

Un *délimiteur* (qui constitue un *argument optionnel*) précise le(s) caractère(s) à utiliser pour détacher les mots les uns des autres. L'exemple suivant utilise un trait d'union comme délimiteur :

```
julia> t = split("spam-spam-spam", '-');
julia> print(t)
SubString{String}["spam", "spam", "spam"]
```

La fonction `join` effectue l'opération inverse de `split`. Elle prend un ensemble de chaînes de caractères et concatène les éléments :

```
julia> t = ["Au", "calme", "clair", "de", "lune", "triste", "et", "beau"];
julia> s = join(t, ' ')
"Au calme clair de lune triste et beau"
```

1. Le vers est extrait du poème « Clair de lune » de Paul Verlaine

Dans ce cas, le délimiteur est un caractère d'espacement. Pour concaténer des chaînes de caractères sans espace(s), il convient de ne pas spécifier de délimiteur.

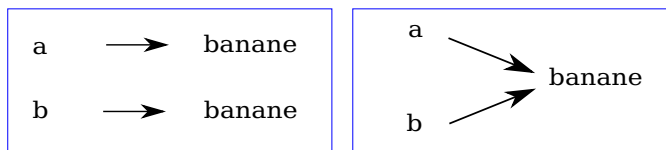
10.10 Objets et valeurs

Un objet est une structure de données à laquelle une variable peut se référer. Jusqu'à présent, vous pouviez utiliser indifféremment les termes « objet » et « valeur », qui constituent des séquences informatiques.

Si les déclarations d'affectation sont effectuées telles que :

```
a = "banane"
b = "banane"
```

il est clair que **a** et **b** font tous deux référence à une chaîne de caractères. Cependant, **a** et **b** font-elles référence à la même chaîne ? Il existe deux états possibles illustrés dans la figure 10.10.1.



(a) a et b pointent vers 2 objets différents. (b) a et b pointent vers le même objet.

FIGURE 10.10.1 – Diagramme d'état et notion d'objets

Pour vérifier si deux variables désignent le même objet, il faut utiliser l'opérateur `≡` (`\equiv TAB`) ou `===`.

```
julia> a = "banane"
"banane"
julia> b = "banane"
"banane"
julia> a ≡ b
true
```

Dans cet exemple, Julia n'a créé qu'un seul objet « chaîne de caractères ». Nous avons donc affaire au cas représenté à la figure 10.10.1b.

A contrario, lorsque deux tableaux sont instanciés, nous sommes confrontés à deux objets :

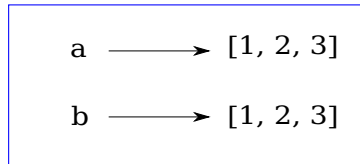


FIGURE 10.10.2 – Diagramme d'état pour 2 tableaux équivalents mais non-identiques.

```

julia> a = [1, 2, 3];
julia> b = [1, 2, 3];
julia> a == b
false

```

Dans ce cas, les deux tableaux sont équivalents parce qu'ils contiennent les mêmes éléments. Toutefois, ils ne sont pas identiques parce qu'ils ne représentent pas le même objet. Ce cas est illustré dans le diagramme d'état de la figure 10.10.2.

Si deux objets sont identiques, ils sont forcément équivalents, mais s'ils sont équivalents, ils ne sont pas nécessairement identiques.

Pour être précis, un objet a une valeur. Si nous évaluons `[1, 2, 3]`, nous obtenons un tableau d'objets dont la valeur est une séquence d'entiers. Si un autre tableau a les mêmes éléments, on dit qu'il a la même valeur, mais il ne s'agit pas du même objet.

10.11 *Aliasing*²

Si `a` se rapporte à un objet et que nous effectuons l'affectation `b = a`, alors les deux variables se rapportent au même objet :

```

julia> a = [1, 2, 3];
julia> b = a;
julia> a == b
true

```

Le diagramme d'état lié à cette situation est donné à la figure 10.11.1.

2. La traduction exacte devrait être pseudonymie. Cependant, alias (c'est-à-dire pseudo ou pseudonyme) et aliasing sont des termes consacrés. En informatique, s'agissant de personnes, il est également fréquent de trouver l'acronyme *aka* pour *also known as*.

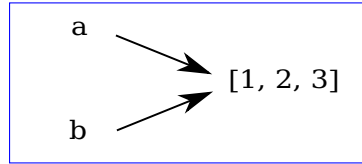


FIGURE 10.11.1 – Diagramme d'état pour deux alias.

L'association d'une variable à un objet est un *référencement*. Dans cet exemple, il y a deux *références* au même objet, symbolisées par les deux flèches.

Un objet ayant plus d'une référence possède plus d'un nom, on dit donc que l'objet est *aliasé*.

Si l'objet *aliasé* est non persistant, les modifications apportées à un alias affectent l'autre :

```
julia> b[1] = 42
42
julia> print(a)
[42, 2, 3]
```

Avertissement. Bien que ce comportement puisse être utile, il est sujet à des erreurs. En général, il est plus sûr d'éviter les alias lorsque sont traités des objets non persistants.

En revanche, pour les objets persistants comme les chaînes de caractères, l'*aliasing* ne constitue pas un véritable problème. Dans cet exemple (qui se réfère à la sous-figure 10.10.1a) :

```
a = "banane"
b = "banane"
```

travailler sur **a** n'affecte pas **b**.

10.12 Les tableaux en tant qu'argument

Lorsqu'un tableau est passé en argument à une fonction, la fonction acquiert une référence au tableau puisqu'elle y accède. Si la fonction modifie le tableau, l'appelant perçoit le changement. Par exemple, [deletehead!](#) supprime le premier élément d'un tableau :

```
function deletehead!(t)
    popfirst!(t)
end
```

Voici un exemple :

```
julia> letters = ['a', 'b', 'c'];
julia> deletehead!(letters);
julia> print(letters)
['b', 'c']
```

Le paramètre `t` et la variable `letters` sont des alias pour le même objet. Le diagramme de pile est donné à la figure 10.12.1.

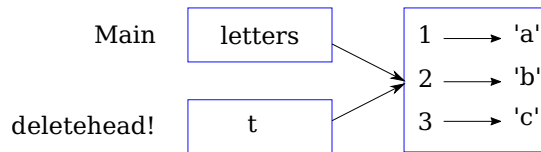


FIGURE 10.12.1 – Diagramme de pile dans le cas d'un tableau passé en argument à une fonction.

Lorsque `letters` est passé à `t`, les 2 variables (l'une globale, `letters`, et l'autre locale, `t`) se réfèrent au même tableau.

Il est important de distinguer les opérations qui modifient les tableaux et celles qui en créent de nouveaux. Par exemple, `push!` modifie un tableau alors que `vcat` en crée un nouveau.

Voici un exemple d'utilisation de `push!` :

```
julia> t1 = [1, 2];
julia> t2 = push!(t1,3);
julia> print(t1)
[1, 2, 3]
```

En l'espèce, `t2` est un alias de `t1`.

À présent, voici un exemple d'utilisation de la fonction `vcat` :

```
julia> t3 = vcat(t1, [4]);
```

```
julia> t3 = vcat(t1, [4]);
julia> print(t1)
['1', '2', '3']
julia> print(t3)
['1', '2', '3', '4']
```

Le résultat de `vcat` est un nouveau tableau. Le tableau original est inchangé.

Cette différence est importante lorsque sont rédigées des fonctions censées modifier des tableaux.

Par exemple, cette fonction `baddeletehead` *ne supprime pas* le premier élément d'un tableau :

```
function baddeletehead(t)
    t = t[2:end] # erroné
end
```

L'opérateur de segmentation crée un nouveau tableau et l'affectation y fait référence, mais cela n'altère pas l'argument de la fonction `baddeletehead` en lui-même.

```
julia> t4 = baddeletehead(t3);
julia> print(t3)
['1', '2', '3', '4']
julia> print(t4)
['2', '3', '4']
```

Au début de `baddeletehead`, `t` et `t3` font référence au même tableau. À la fin, `t` fait référence à un nouveau tableau, mais `t3` pointe toujours vers le tableau original, non modifié.

Une autre manière de procéder consiste à écrire une fonction qui crée et retourne un nouveau tableau. Par exemple, `tail` retourne tout sauf le premier élément d'un tableau :

```
function tail(t)
    t[2:end]
end
```

Cette fonction ne modifie pas le tableau original. Voici comment elle est utilisée :

```
julia> letters = ['a', 'b', 'c'];
julia> rest = tail(letters);
julia> print(rest)
['b', 'c']
```

10.13 Débogage

Une utilisation imprudente des tableaux (et des autres objets non persistants) peut entraîner de longues heures de débogage. Voici quelques pièges courants et les moyens de les éviter :

- La plupart des fonctions agissant sur les tableaux modifient ces derniers. Ceci *a contrario* des fonctions traitant les chaînes de caractères, qui retournent une nouvelle chaîne de caractères et laissent l'original intact. Si vous avez l'habitude d'écrire du code de type chaîne de caractères de cette façon :

```
new_word = strip(word)
```

Il est tentant d'écrire un code de tableau comme celui-ci :

```
t2 = sort!(t1)
```

Parce que `sort!` retourne le tableau original modifié `t1`, `t2` est un alias de `t1`.

Conseil. Avant d'utiliser les fonctions et les opérateurs agissant sur les tableaux, il est vivement recommandé de lire attentivement la documentation et de tester ces fonctions en mode interactif.

- Il est recommandé de choisir une manière de faire et de s'y tenir. Une partie du problème avec les tableaux provient du grand nombre de techniques utilisables. Par exemple, pour supprimer un élément d'un tableau, il est concevable utiliser `pop!`, `popfirst!`, `delete_at`, ou même l'affectation d'un segment extrait du tableau. Pour ajouter un élément, il est possible de tirer parti de `push!`, `pushfirst!`, `insert!` ou `vcats`. En supposant que `t` soit un tableau et `x` un élément de ce tableau, ces méthodes sont correctes :

```
insert!(t, 4, x)
push!(t, x)
append!(t, [x])
```

Alors que celles-ci sont erronées :

```
insert!(t, 4, [x]) # erroné
push!(t, [x])      # erroné
vcats(t, [x])       # erroné
```

- Il est fortement préconisé d'effectuer des copies pour éviter les alias. Lors de l'utilisation d'une fonction comme `sort!` qui modifie l'argument, alors

qu'il s'avère nécessaire de conserver également le tableau original, une copie devrait être créée :

```
julia> t = [3, 1, 2];
julia> t2 = t[:]; # t2 = copy(t)!
julia> sort!(t2);
julia> print(t)
[3, 1, 2]
julia> print(t2)
[1, 2, 3]
```

Dans cet exemple, il est judicieux de tirer avantage de la fonction intégrée `sort`, qui retourne un nouveau tableau trié tout en laissant intact l'original :

```
julia> sort!(t2);
julia> println(t)
[3, 1, 2]
julia> println(t2)
[1, 2, 3]
```

10.14 Glossaire

tableau séquence de valeurs,

élément (ou *item*) une des valeurs d'un tableau (ou de toute autre séquence, comme une matrice, par exemple).

tableau imbriqué tableau constituant un élément d'un autre tableau,

accumulateur variable utilisée dans une boucle pour additionner (c'est-à-dire accumuler) un résultat,

déclaration d'affectation incrémentée déclaration qui met à jour la valeur d'une variable en utilisant un opérateur comme `=` (par exemple, `+=`),

opérateur point (*dot*) opérateur binaire qui est appliqué élément par élément à des tableaux,

syntaxe avec *dot* syntaxe utilisée pour appliquer une fonction par élément à un tableau quelconque,

opération de réduction modèle de traitement qui parcourt un tableau et accumule les éléments en un seul résultat,

mapping (ou mise en correspondance) modèle de traitement qui parcourt un tableau et effectue une opération sur chaque élément,

filtre modèle de traitement qui parcourt un tableau et sélectionne les éléments qui satisfont à un critère donné,

objet composant auquel une variable peut se référer. Tout objet a un type et une valeur,

équivalence propriété qu'ont deux séquences (par exemple des tableaux) à posséder les mêmes éléments de deux objets différents,

identité propriété qu'ont deux séquences (par exemple des tableaux) à désigner le même objet (ce qui implique *a minima* une équivalence),

référence association entre une variable et sa valeur,

aliasing cas de pseudonymie où deux ou plusieurs variables se réfèrent au même objet,

arguments facultatifs arguments figurant dans une fonction mais dont l'usage n'est pas toujours nécessaire,

délimiteur caractère ou chaîne de caractères utilisés pour indiquer l'endroit où une chaîne doit être démarquée du reste de la chaîne.

10.15 Exercices

10.15.1 Exercice

Écrivez une fonction appelée `nestedsum` qui prend un tableau contenant des tableaux d'entiers et additionne les éléments de tous les tableaux imbriqués. Par exemple :

```
julia> t = [[1, 2], [3], [4, 5, 6]];
```

```
julia> nestedsum(t)
```

```
21
```

10.15.2 Exercice

Écrivez une fonction appelée `cumulsum` qui prend un tableau de nombres et retourne la somme cumulée; c'est-à-dire un nouveau tableau où le $i^{\text{ème}}$ élément est la somme des i premiers éléments du tableau original. Par exemple :

```
julia> t = [1, 2, 3];  
julia> print(cumulsum(t))  
2Any[1, 3, 6]
```

10.15.3 Exercice

Écrivez une fonction appelée `interior` qui prend un tableau et retourne un nouveau tableau qui contient tous les éléments sauf le premier et le dernier. Par exemple :

```
julia> t = [1, 2, 3, 4];  
julia> print(interior(t))  
[2, 3]
```

10.15.4 Exercice

Écrivez une fonction appelée `interior!` qui prend un tableau, le modifie en supprimant le premier et le dernier élément, et retourne `nothing`. Par exemple :

```
julia> t = [1, 2, 3, 4];  
julia> interior!(t)  
julia> print(t)  
[2, 3]
```

10.15.5 Exercice

Écrivez une fonction appelée `issort` qui prend un tableau comme paramètre et retourne `true` si le tableau est trié par ordre croissant et `false` dans le cas contraire. Par exemple :

```
julia> issort([1, 2, 2])  
true  
julia> issort(['b', 'a'])  
false
```

10.15.6 Exercice

Deux mots sont des anagrammes si vous pouvez réarranger les lettres de l'un pour écrire l'autre. Écrivez une fonction appelée `isanagramme` qui prend deux chaînes de caractères et qui retourne `true` si ces mots sont des anagrammes.

10.15.7 Exercice

Écrivez une fonction appelée `hasduplicates` qui prend un tableau et retourne `true` s'il y a un élément qui apparaît plus d'une fois. Elle ne doit pas modifier le tableau original.

10.15.8 Exercice

Cet exercice se rapporte au « paradoxe des anniversaires » (voir [ce lien web](#)).

Dans une classe de 23 élèves, quelles sont les chances que deux d'entre eux fêtent leur anniversaire le même jour ? Vous pouvez estimer cette probabilité en générant des échantillons aléatoires de 23 anniversaires et en vérifiant s'il y a des correspondances.

Conseil. Vous pouvez générer des dates aléatoires avec `rand(1:365)`.

10.15.9 Exercice

Écrivez une fonction qui lit le fichier `mots_FR.txt` (voir la section 9.1) et construit un tableau avec un élément par mot. Écrivez deux versions de cette fonction, l'une utilisant `push!` et l'autre utilisant l'expression `t = [t..., x]`. Laquelle prend le plus de temps à exécuter ? Pourquoi ?

10.15.10 Exercice

Pour vérifier si un mot se trouve dans le tableau de mots, vous pourriez utiliser l'opérateur `∈`, mais ce serait lent car ce dernier recherche les mots dans l'ordre.

Comme les mots sont dans l'ordre alphabétique, nous pouvons accélérer la recherche grâce à la bisection (également appelée recherche binaire, voir la section 7.8), qui est similaire à ce que vous faites lorsque vous cherchez un mot dans le dictionnaire. Vous commencez au milieu et vous vérifiez si le mot que vous recherchez vient avant le mot du milieu du tableau. Si c'est le cas, vous recherchez la première moitié du tableau de la même manière. Sinon, vous cherchez dans la deuxième moitié.

Dans les deux cas, vous réduisez de moitié l'espace de recherche restant. Si le tableau contient 336.531 mots, il faudra environ 19 étapes pour trouver le mot ou conclure qu'il n'y est pas.

Écrivez une fonction appelée `inbisect` qui prend un tableau trié et une valeur cible et retourne `true` si le mot est dans le tableau et `false` s'il n'y est pas.

10.15.11 Exercice

Deux mots sont une « paire inversée » si chacun est l'inverse de l'autre. Écrivez un programme [reversepairs](#) qui trouve toutes les paires inversées dans le tableau de mots.

10.15.12 Exercice

Deux mots « s'emboîtent » si on prend des lettres alternées de chacun d'eux pour former un nouveau mot. Par exemple, « cor » et « ria » s'emboîtent pour former « croira ».

1. Écrivez un programme qui trouve toutes les paires de mots qui s'emboîtent.

Remarque. Cet exercice est inspiré d'un exemple décrit sur [ce lien](#).

Conseil. N'énumérez pas toutes les paires.

2. Pouvez-vous trouver des mots qui s'emboîtent les uns dans les autres, c'est-à-dire qu'une lettre sur trois forme un mot, en commençant par la première, la deuxième ou la troisième ?

Chapitre 11

Dictionnaires

Ce chapitre présente un autre type intégré, les dictionnaires.

11.1 Dictionnaire et « mise en correspondance » (*mapping*)

Un dictionnaire constitue la généralisation d'un tableau. Dans un tableau, les indices doivent être des nombres entiers. Dans un dictionnaire, ils peuvent être de (presque) n'importe quel type.

Un dictionnaire contient une collection d'indices, appelés *clés*, et une collection de valeurs. Chaque clé est associée à une valeur unique. L'association d'une clé et d'une valeur est appelée une paire clé-valeur ou parfois un élément.

En langage informatique, un dictionnaire représente une *mise en correspondance* (ou *mapping*) ou association entre des clés et des valeurs, de sorte qu'on peut également dire que chaque clé « correspond » à une valeur (voir la figure 11.1.1). À titre d'exemple, nous allons construire un dictionnaire qui établit une correspondance entre des mots anglais et français, de sorte que les clés et les valeurs sont toutes des chaînes de caractères.

La fonction `Dict` crée un nouveau dictionnaire ne contenant aucun élément. Comme `Dict` est le nom d'une fonction intégrée, il faut éviter de l'utiliser comme nom de variable.

```
julia> eng2fr = Dict()  
Dict{Any,Any} with 0 entries
```

eng2fr

"un"	"deux"	"trois"	...		valeurs
"one"	"two"	"three"			clés

↑
mapping
↓

FIGURE 11.1.1 – Représentation du dictionnaire eng2fr.

Le type de dictionnaire est entouré d'accolades : les clés sont de type [Any](#) et les valeurs également.

Le dictionnaire est vide. Pour y ajouter des éléments, utilisons des crochets :

```
julia> eng2fr["one"] = "un";
```

Cette ligne crée un élément qui relie la clé "one" à la valeur "un". À nouveau affiché, le dictionnaire montre une paire clé-valeur. La clé est connectée à la valeur par une flèche => :

```
julia> eng2fr
Dict{Any,Any} with 1 entry:
  "one" => "un"
```

Ce format de sortie est également un format d'entrée. Par exemple, nous pouvons créer un nouveau dictionnaire comportant trois éléments :

```
julia> eng2fr = Dict{"one" => "un", "two" => "deux", "three" => "trois"}
Dict{String,String} with 3 entries:
  "one" => "un"
  "two" => "deux"
  "three" => "trois"
```

La figure 11.1.1 résume les attributs du dictionnaire [eng2fr](#).

Toutes les clés et valeurs initiales sont des chaînes de caractères, de sorte qu'un [Dict{String,String}](#) est créé.

Avertissement. L'ordre des paires « clé-valeur » n'est peut-être pas le même si vous essayez ces instructions sur votre ordinateur. En général, l'ordre des éléments (c'est-à-dire des couples « clé-valeur ») dans un dictionnaire est imprévisible.

Néanmoins, cela n'a guère d'importance du fait que les éléments d'un dictionnaire ne sont jamais indicés avec des entiers. Ce sont les clés qui sont utilisées pour rechercher les valeurs correspondantes :

```
julia> eng2fr["two"]
"deux"
```

La clé "two" correspond toujours à la valeur "deux", l'ordre des éléments n'a donc pas d'importance.

Si la clé n'est pas dans le dictionnaire, Julia retourne une exception :

```
julia> eng2fr["four"]
ERROR: KeyError: key "four" not found
```

La fonction `length` opère également sur les dictionnaires. Elle retourne le nombre de paires clé-valeur :

```
julia> length(eng2fr)
3
```

La fonction `keys` retourne la collection de clés du dictionnaire :

```
julia> ks = keys(eng2fr);
julia> print(ks)
["two", "one", "three"]
```

À présent, l'opérateur `∈` est utilisable pour déterminer si un terme est une clé du dictionnaire :

```
julia> "one" ∈ ks
true
julia> "un" ∈ ks
false
```

Pour déterminer si une valeur appartient à un dictionnaire, la fonction `values` est exploitable. Elle retourne l'ensemble des valeurs, si bien qu'ensuite il est alors possible de tirer parti de l'opérateur `∈` :

```
julia> vs = values(eng2fr);
julia> "un" ∈ vs
true
```

L'opérateur `∈` utilise des algorithmes différents selon que sont traités des tableaux ou des dictionnaires. Pour les tableaux, il recherche les éléments dans l'ordre, comme dans la section 8.8. Le temps de recherche s'allonge proportionnellement à la taille des tableaux.

Pour les dictionnaires, Julia utilise un algorithme appelé *table de hachage* (ou *hash table*) qui a une propriété remarquable : l'opérateur `∈` prend à peu près le même temps quel que soit le nombre d'éléments du dictionnaire.

11.2 Les dictionnaires en tant que collections de compteurs

Soit une chaîne dont il faut compter l'occurrence¹ de chaque lettre. *A priori*, il y a trois possibilités :

1. créer 26 variables, une pour chaque lettre de l'alphabet, parcourir la chaîne et pour chaque caractère, incrémenter le compteur correspondant, probablement en utilisant des conditions imbriquées.
2. créer un tableau de 26 éléments. Ensuite, convertir chaque caractère en un nombre (en utilisant la fonction interne `Int`), utiliser les nombres comme indices dans le tableau et incrémenter le compteur correspondant.
3. créer un dictionnaire où les clés sont des caractères et où les valeurs correspondantes sont des compteurs. La première fois qu'un caractère est rencontré, le programme ajoute un élément au dictionnaire. Ensuite, la valeur d'un élément existant est incrémentée. Par exemple, pour le terme « abracadabra », la valeur de la clé « a » est 5, la valeur de la clé « b » est 2, la valeur de la clé « r » est 2, la valeur de la clé « c » est 1 et la valeur de la clé « d » est 1.

Chacune de ces options effectue le même calcul, mais avec une mise en œuvre différente.

Une *implémentation* est définie comme la mise en œuvre d'un calcul. Certaines s'avèrent plus performantes que d'autres. Par exemple, un des avantages de l'implémentation par dictionnaire provient du fait qu'il n'est pas nécessaire de connaître préalablement les lettres constituant la chaîne. Nous n'avons qu'une obligation : faire de la place pour insérer dans le dictionnaire les lettres détectées.

Voici à quoi pourrait ressembler le code :

```
function histogram(s)
  d = Dict()
  for c in s
    if c ∉ keys(d)
      d[c] = 1
    else
      d[c] += 1
    end
  end
  d
end
```

1. C'est-à-dire la fréquence d'apparition.

Le nom de la fonction est `histogram`, un terme associé au domaine des statistiques pour désigner une série de fréquences (c'est-à-dire d'occurrences). Comment fonctionne ce code ? Pour être concret, effectuons un appel sous la forme `histogram("brontosau-
re")` :

```
julia> h = histogram("brontosau-  
re")
```

On passe donc la chaîne `"brontosau-
re"` à `s`. Ensuite, un dictionnaire vide `d` est créé.

Dans un premier temps, supposons que la boucle `for` ne contienne pas le code du test conditionnel `if` et qu'à sa place il n'y ait qu'une instruction `println(c)`. Que retournerait le programme ? Il afficherait chaque lettre du mot « brontosau-
re » avec, chaque fois, un retour à la ligne. En effet, au premier passage, la variable `c` vaut `'b'`, au deuxième passage, elle vaut `'r'` et ainsi de suite jusqu'à valoir `'e'`. La boucle `for` parcourt donc entièrement la chaîne de caractères.

Dans un deuxième temps, on peut s'assurer de ce que contient `keys(d)`. Pour ce faire, il suffit de remplacer l'instruction `println(c)` par `@show keys(d)`. Nous devons nous attendre à 11 retours `keys(d) = Any{ }`.

Dans un troisième temps, nous allons remplacer le test conditionnel `if` par ce bloc :

```
if c ∉ keys(d)
    d[c] = 1
    print(d[c])
else
    print("ω")
end
```

De sorte que la fonction prenne transitoirement cette forme :

```
function histogram(s)
    d = Dict{ }
    for c in s
        if c ∉ keys(d)
            d[c] = 1
            println(d[c])
        else
            println("ω")
        end
    end
end
```

Que retourne la fonction `histogram("brontosau-
re")` ? Au premier passage dans la boucle `for`, la variable `c` contient `'b'`. Le code teste si `'b'` se trouve parmi les clés du dictionnaire. Le dictionnaire étant vide, `'b'` ne s'y trouve pas. À la ligne suivante, `d['b']`

effectue l'entrée de 'b' comme clé du dictionnaire et l'entier 1 est affecté à la valeur `d['b']` correspondante. Le code remonte à la boucle `for` avec `c` qui, cette fois, vaut 'r' et ainsi de suite jusqu'à la cinquième lettre, 't'. La première partie de l'affichage sera donc 11111. Au passage suivant, `c` vaut 'o'. Cependant, il existe déjà un 'o' dans le dictionnaire et, par conséquent, le code passe au `else` et l'affichage devient 11111ω. Il est maintenant clair qu'à la fin, nous devons obtenir un affichage tel que 11111ω111ω1.

Revenons à la fonction `histogram` initiale. Comme précédemment, au premier passage dans la boucle `for`, `c` vaut 'b'. Le code teste si 'b' se trouve parmi les clés du dictionnaire. Puisque ce dernier est vide, 'b' ne s'y trouve pas ; `d['b']` effectue l'entrée de 'b' comme clé du dictionnaire et l'entier 1 est affecté à la valeur `d['b']`.

En conséquence, l'état du dictionnaire est :

```
keys(d) = Any['b']
d = Dict{Any,Any}{'b'=> 1}
```

Au deuxième tour, l'état du dictionnaire passe à :

```
keys(d) = Any['r', 'b']
d = Dict{Any,Any}{'r'=> 1,'b'=> 1}
```

Au cinquième tour, le dictionnaire prend cette configuration (à l'ordre des clés près) :

```
keys(d) = Any['t', 'n', 'o', 'r', 'b']
d = Dict{Any,Any}{'t'=> 1,'n'=> 1,'o'=> 1,'r'=> 1,'b'=> 1}
```

Au passage suivant, le programme entre dans la partie `else` puisque 'o' se trouve déjà dans le dictionnaire. Sa valeur correspondante est incrémentée et devient 2. L'état du dictionnaire devient :

```
keys(d) = Any['t', 'n', 'o', 'r', 'b']
d = Dict{Any,Any}{'t'=> 1,'n'=> 1,'o'=> 2,'r'=> 1,'b'=> 1}
```

In fine l'état du dictionnaire est tel que :

```
keys(d) = Any['e','r','u','a','s','o','t', 'n', 'o', 'r', 'b']
d = Dict{Any,Any}{'e'=> 1,'u'=> 1,'a'=> 1,'s'=> 1,'t'=> 1,'n'=> 1,'o'=> 2,'r'=> 2,'b'=> 1}
```

Si vous appliquez l'appel `histogram("brontosauere")`, le programme retourne la valeur de `d` sous cette forme :


```
Dict{ Any,Any } with 9 entries:
```

```
'n' => 1
's' => 1
'a' => 1
'r' => 2
't' => 1
'o' => 2
'u' => 1
'e' => 1
'b' => 1
```

L'histogramme indique que les lettres *a*, *b*, *e*, *n*, *s*, *t* et *u* apparaissent une fois ; *o* et *r* apparaissent deux fois.

Les dictionnaires possèdent une fonction appelée `get` qui prend une clé et une valeur par défaut. Si la clé apparaît dans le dictionnaire, `get` retourne la valeur correspondante. Sinon, elle retourne la valeur par défaut. Par exemple :

```
julia> h = histogram("brontosauure");
julia> get(h, 'a', 0)
1
julia> get(h, 'o', 0)
2
julia> get(h, 'z', 0)
0
```

11.2.1 Exercice

Utilisez `get` pour réécrire `histogram` de manière plus concise. Vous devriez pouvoir éliminer la déclaration `if`.

11.3 Boucles et dictionnaires

Les clés d'un dictionnaire peuvent être parcourues grâce à une boucle `for`. Par exemple, `printhist` affiche chaque clé et la valeur correspondante :

```
function printhist(h)
    for c in keys(h)
        println(c, " ", h[c])
    end
end
```

Voici à quoi ressemble le résultat :

```
julia> histogram("perroquet");
julia> printhist(h)
 u  1
 e  2
 p  1
 r  2
 o  1
 q  1
 t  1
```

Là encore, les clés ne sont pas dans un ordre particulier. Pour parcourir les clés dans l'ordre alphabétique, une combinaison des fonctions `sort` et `collect` est pratique :

```
julia> for c in sort(collect(keys(h)))
    println(c, " ", h[c])
end
 e  2
 o  1
 p  1
 q  1
 r  2
 u  1
 t  1
```

11.4 Recherche inverse

Avec un dictionnaire `d` et une clé `k`, il est facile de trouver la valeur correspondante `v = d[k]`. Cette opération est appelée une *recherche directe* (ou *lookup*).

Ceci dit, comment procéder si nous disposons d'une valeur `v` et que nous souhaitons trouver sa clé `k` ? Ici, nous sommes confrontés à deux problèmes :

1. premièrement, il peut y avoir plus d'une clé qui correspond à la valeur `v` (par exemple, dans le cas du tableau `d` associé à « brontosauve », à la valeur « 2 » correspondaient les clés 'o' et 'r'),
2. deuxièmement, il n'y a pas d'astuce simple pour effectuer une *recherche inverse* (ou *reverse lookup*).

Voici une fonction qui prend une valeur et retourne la première clé qui correspond à cette valeur :

```
function reverselookup(d, v)
    for k in keys(d)
        if d[k] == v
            return k
        end
    end
    error("LookupError")
end
```

Cette fonction est un exemple de schéma de recherche inverse, mais elle utilise une fonction `error` non encore abordée jusqu'ici. La fonction `error` est exploitée pour produire une `ErrorException` qui interrompt le flux de contrôle normal. Dans ce cas, elle affiche le message `"LookupError"`, indiquant qu'une clé n'existe pas.

Si le programme arrive à la fin de la boucle, cela signifie que `v` n'est pas une valeur du dictionnaire. En conséquence, le programme émet une exception.

Voici un exemple de recherche inversée réussie :

```
julia> h = histogram("magnifique");
julia> key = reverselookup(h, 2)
'i': ASCII/Unicode U+0069 (category Ll: Letter, lowercase)
```

et voici un cas qui a échoué :

```
julia> key = reverselookup(h, 3)
ERROR: LookupError
```

Quand une exception se produit, l'effet est le même que lorsque Julia en émet une : une trace de pile (ou *stacktrace*) et un message d'erreur sont affichés.

Julia propose un moyen optimisé de faire une recherche inversée : `findall(isequal(3), h)`.

Avertissement. Une recherche inverse est beaucoup plus lente qu'une recherche classique. S'il est nécessaire d'y recourir souvent, ou si le dictionnaire est de grande taille, les performances du programme en souffriront.

11.5 Dictionnaires et tableaux

Les tableaux peuvent constituer des valeurs au sein d'un dictionnaire. Supposons un dictionnaire qui fait correspondre des lettres à des occurrences (voir section 11.2).

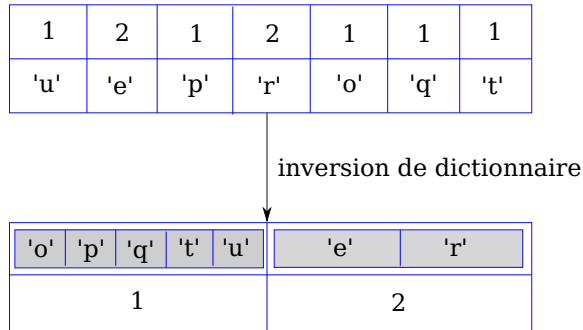


FIGURE 11.5.1 – Inversion de dictionnaire où les valeurs du dictionnaire inversé sont des tableaux.

On peut vouloir l'inverser, c'est-à-dire créer un dictionnaire qui fait correspondre des occurrences (en tant que clés) à des lettres (en tant que valeurs). Comme plusieurs lettres peuvent présenter la même occurrence, celles-ci peuvent être groupées, selon leur occurrence, sous forme de tableaux et ces tableaux peuvent devenir des valeurs dans le dictionnaire inversé.

La figure 11.5.1 permet de visualiser l'inversion d'un tableau (en l'espèce, l'histogramme du mot « perroquet »). Les zones grisées de la figure correspondent à deux tableaux constituant les deux valeurs du dictionnaire inversé.

Voici une fonction permettant d'inverser un dictionnaire :

```
function invertDict(d)
  inverse = Dict()
  for key in keys(d)
    val = d[key]
    if val ∉ keys(inverse)
      inverse[val] = [key]
    else
      push!(inverse[val], key)
    end
  end
  inverse
end
```

À chaque fois, dans la boucle, `key` récupère une clé à partir de `d` pendant que `val` récupère la valeur correspondante. Si `val` n'est pas dans `inverse`, cela signifie qu'elle n'a pas été détectée auparavant. En conséquence, un nouvel élément est créé et initialisé

avec un *singleton* (un tableau qui contient un seul élément). Si cette valeur a été repérée auparavant, la clé qui lui correspond est ajoutée au tableau à l'aide de la fonction [push!](#).

Voici un exemple :

```
julia> hist = histogram("perroquet");
julia> inverse = invertdict(hist)
Dict{Any,Any} with 2 entries:
 2 => ['e', 'r']
 1 => ['u', 'p', 'o', 'q', 't']
```

La figure 11.5.2 reprend les diagrammes d'état associés à [hist](#) et [inverse](#). Un dictionnaire est représenté sous la forme d'un cadre contenant des paires clé-valeur (que ces valeurs soient des entiers, des flottants ou des chaînes de caractères). Pour conserver la simplicité de lecture des diagrammes, les tableaux sont disposés dans des cadres à l'extérieur de ceux associés au dictionnaire.

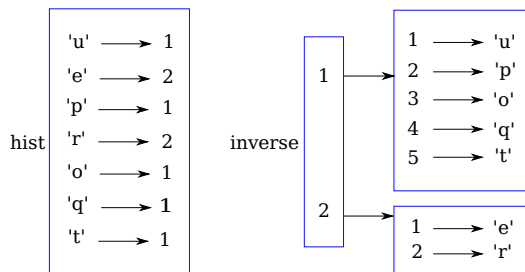


FIGURE 11.5.2 – Diagrammes d'état pour les fonctions [hist](#) et [inverse](#).

Note. Dans la section Dictionnaire et « mise en correspondance » (*mapping*), il a été mentionné qu'un dictionnaire est implémenté à l'aide d'une table de hachage et cela signifie que les clés doivent être hachables. Le hachage est une fonction qui prend une valeur (de n'importe quel type) et retourne un entier. Les dictionnaires utilisent ces entiers, appelés valeurs de hachage, pour enregistrer et rechercher des paires clé-valeur.

11.6 Mémos

Si vous avez quelque peu manié la fonction [fibonacci](#) de la section 6.7, vous avez remarqué que plus le nombre passé en argument est grand, plus la fonction prend de temps à effectuer le calcul. De surcroît, la durée d'exécution augmente de plus en plus rapidement.

Pour comprendre pourquoi, considérons la figure 11.6.1 qui montre le *graphe d'appel* pour `fibonacci` avec `n = 4` :

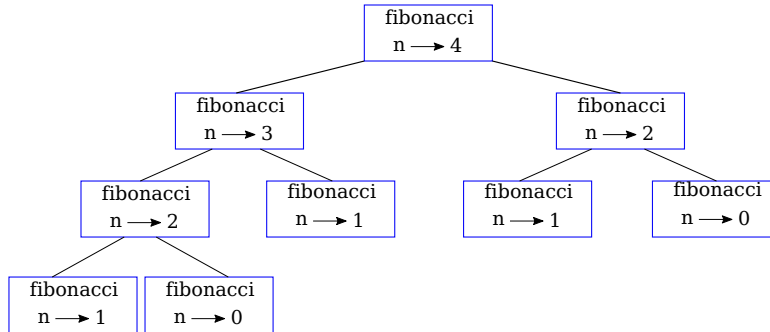


FIGURE 11.6.1 – Graphe d'appel associé à la fonction `fibonacci`.

Un graphique d'appel comporte un ensemble de cadres associés à une (ou des) fonction(s), avec des flèches reliant chaque cadre aux cadres des fonctions qu'il appelle. En haut du graphique, `fibonacci` avec `n = 4` appelle `fibonacci` avec `n = 3` et `n = 2`, et `fibonacci` avec `n = 3` appelle `fibonacci` avec `n = 2` et `n = 1`. Et ainsi de suite.

Il est facile de dénombrer les appels à `fibonacci(0)` et `fibonacci(1)` et de conclure à l'inefficacité de cette solution.

Un moyen de résoudre ce problème consiste à garder une trace des valeurs déjà calculées en les enregistrant dans un dictionnaire. Une valeur calculée précédemment et stockée pour une utilisation ultérieure s'appelle un *mémo*. Voici une version « mémo » de `fibonacci` :

```

known = Dict{0=>0, 1=>1}

function fibonacci(n)
    if n ∈ keys(known)
        return known[n]
    end
    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    res
end
  
```

`known` est un dictionnaire qui garde la trace des nombres de la suite de Fibonacci déjà calculés. Il commence par deux éléments : la clé 0 est associée à la valeur 0 et la clé 1 à la valeur 1.

Chaque fois que la fonction `fibonacci` est appelée, elle vérifie `known`. Si le résultat est déjà dans le dictionnaire, il est rappelé immédiatement. Sinon, `fibonacci` calcule la nouvelle valeur, l'ajoute au dictionnaire et la retourne.

À l'exécution, cette version de `fibonacci` s'avère beaucoup plus rapide que l'originale.

11.7 Variables globales

Dans l'exemple précédent, `known` est créé en dehors de la fonction. Ce dictionnaire appartient donc à la fonction `Main`. Les variables dans `Main` sont dites *globales* car elles sont accessibles depuis n'importe quelle fonction. Contrairement aux variables locales qui disparaissent lorsque leur fonction se termine, les variables globales persistent d'un appel de fonction à l'autre.

Il est courant d'utiliser des variables globales pour les drapeaux (*flags*), c'est-à-dire des variables booléennes qui indiquent si une condition est vraie (ou fausse). Par exemple, certains programmes utilisent un drapeau `verbose` pour contrôler le niveau de détail de la sortie :

```
verbose = true

function example1()
  if verbose
    println("Running example1")
  end
end
```

La réaffectation d'une variable globale n'entraîne pas la modification de cette dernière. L'exemple suivant est censé déterminer si la fonction (`example2`) a été appelée :

```
been_called = false

function example2()
  been_called = true  # ERRONÉ
end
```

À l'exécution, nous constatons que la valeur de `been_called` ne change pas. Le problème vient de ce qu'`example2` crée une nouvelle variable locale appelée `been_called`. La variable locale disparaît lorsque la fonction se termine. Par conséquent, cela n'affecte pas la variable globale.

Pour conférer le caractère global à une variable se trouvant à l'intérieur d'une fonction, il est nécessaire de placer le terme `global` immédiatement devant le nom de la variable :

```

been_called = false

function example2()
    global been_called
    been_called = true
end

```

Cette *déclaration globale* transmet à l'interpréteur une information signifiant : « Dans cette fonction, quand nous disons `been_called`, nous désignons la variable globale ; ne créez pas de variable locale ». Ainsi, quand à l'intérieur de la fonction, `true` est affecté à `been_called`, cela affecte la variable globale.

Voici un exemple qui tente erronément de mettre à jour une variable globale :

```

count = 0

function example3()
    count = count + 1  # ERRONÉ
end

```

À l'exécution, Julia retourne un message d'erreur :

```

julia> example3()
ERROR: UndefVarError: count not defined

```

Julia suppose que `count` est locale et que vous en prenez connaissance avant de modifier sa valeur. À nouveau, la solution est de déclarer `count` comme variable globale.

```

count = 0

function example3()
    global count
    count += 1
end

```

Si une variable globale fait référence à une séquence non persistante (comme un tableau ou un dictionnaire), il est possible de modifier les valeurs de cette séquence sans déclarer la variable comme étant globale :

```

known = Dict{0=>0, 1=>1}

function example4()
    known[2] = 1
end

```

Par conséquent, il est admissible d'ajouter, supprimer et remplacer des éléments d'un tableau ou d'un dictionnaire global. Cependant, la réaffectation de la variable au sein d'une fonction requiert une déclaration en tant que variable globale :


```
known = Dict(0=>0, 1=>1)

function example5()
  global known
  known = Dict()
end
```

Pour des raisons de performance, il est judicieux d’attribuer le mot-clé `const` à une variable globale. De cette manière, il devient impossible de la réaffecter. Si une variable globale `const` se réfère à une séquence non persistante, il reste néanmoins possible d’en modifier la valeur.

```
const known = Dict(0=>0, 1=>1)

function example4()
  known[2] = 1
end
```

Avertissement. Les variables globales peuvent s’avérer très utiles. Néanmoins, si un programme en contient un grand nombre et qu’elles sont modifiées fréquemment, elles peuvent rendre le programme difficile à déboguer et le conduire à mal fonctionner.

11.8 Débogage

Lorsqu’on travaille avec des jeux de données volumineux, il peut devenir difficile de déboguer en affichant et en vérifiant le résultat manuellement. Voici quelques suggestions pour le débogage de grands jeux de données :

- réduire la taille de l’entrée :

Si possible, réduire la taille du jeu de données. Par exemple, si un programme lit un fichier texte, on peut commencer par les 10 premières lignes, voire –si c’est possible avec le plus petit échantillon sur lequel des erreurs sont identifiables. Il est fortement conseillé de ne pas éditer les fichiers eux-mêmes mais plutôt de modifier le programme afin qu’il ne lise que les n premières lignes.

S’il y a une erreur, il convient de réduire n à la plus petite valeur qui révèle l’erreur. Par la suite, la valeur de n peut être augmentée progressivement à mesure que sont détectées et corrigées les erreurs,

- vérifiez les résumés et les types :

Au lieu d’afficher et de vérifier l’ensemble des données, il faut penser à afficher les résumés de données : par exemple, le nombre d’éléments dans un dictionnaire ou le total d’un tableau de nombres.

Une cause fréquente d’erreurs d’exécution tient en une valeur dont type est erroné. Pour déboguer ce type d’erreur, il suffit souvent d’afficher le type d’une valeur,

- rédigez des vérifications automatiques :

Parfois, il est habillé d’écrire un peu de code pour vérifier automatiquement la présence d’erreurs. Par exemple, si la moyenne d’un tableau de nombres est calculée, on peut aisément vérifier que le résultat n’est pas supérieur au plus grand élément du tableau ou inférieur au plus petit. Il s’agit là d’une « vérification de bon sens ».

Un autre type de contrôle consiste à comparer les résultats de deux calculs différents pour vérifier qu’ils sont cohérents. Ici, il est question d’une « vérification de cohérence »,

- formater les messages émis :

Le formatage des messages de débogage peut aider à repérer une erreur. Nous en avons vu un exemple dans la section 6.9. Là encore, le temps investi à construire un canevas peut considérablement réduire le temps de débogage.

11.9 Glossaire

mapping (association ou mise en correspondance) relation dans laquelle chaque élément d’un ensemble correspond à un élément d’un autre ensemble,

dictionnaire séquence permettant une mise en correspondance de clés avec leur valeur,

paire clé-valeur représentation de la mise en correspondance (*mapping*) d’une clé à une valeur,

item dans un dictionnaire, synonyme d’une paire clé-valeur,

clé objet qui apparaît dans un dictionnaire comme la première partie d’une paire clé-valeur,

valeur objet qui apparaît dans un dictionnaire comme la deuxième partie d’une paire clé-valeur. Ce terme est plus spécifique que notre utilisation précédente du terme « valeur »,

implémentation mise en œuvre d’un calcul,

table de hachage (*hash table*) algorithme utilisé pour mettre en œuvre les dictionnaires en Julia,

fonction de hachage (*hash function*) fonction utilisée par une table de hachage pour calculer l’emplacement d’une clé,

recherche directe (*lookup*) opération du dictionnaire qui prend une clé et trouve la valeur correspondante,

recherche inversée (*reverse lookup*) opération sur un dictionnaire qui prend une valeur et trouve une ou plusieurs clés qui y correspondent,

singleton tableau (ou tout autre séquence) avec un seul élément,

graphe d'appel diagramme qui montre chaque cadre créé pendant l'exécution d'un programme, avec une flèche de chaque appelant à chaque appelé,

mémo valeur calculée et enregistrée pour éviter tout calcul futur redondant,

variable globale variable définie dans [Main](#), en dehors de toute autre fonction. Les variables globales sont accessibles à partir de n'importe quelle fonction,

déclaration globale déclaration qui établit le nom d'un variable globale,

drapeau (*flag* ou *fanion*) variable booléenne utilisée pour indiquer si une condition est vraie,

déclaration énoncé tel que le mot [global](#) qui renseigne l'interpréteur sur une variable,

constante globale constante définie dans [Main](#) et ne pouvant être réaffectée.

11.10 Exercices

11.10.1 Exercice

Écrivez une fonction qui lit les mots dans la liste [mots_FR.txt](#) et les enregistre sous forme de clés dans un dictionnaire. Les valeurs n'ont pas d'importance. Vous pouvez alors utiliser l'opérateur `∈` comme moyen rapide de vérifier si une chaîne se trouve dans le dictionnaire.

Si vous avez résolu l'exercice 10.15.10, vous pouvez comparer la rapidité de cette implémentation avec l'opérateur `∈` par rapport à la recherche par bisection.

11.10.2 Exercice

Lisez la documentation de la fonction [get!](#) agissant sur les dictionnaires et utilisez-la pour rédiger une version plus concise d'[invertdict](#).

11.10.3 Exercice

Appliquez la technique « mémo » à la fonction d'Ackermann (exercice 6.11.2) et notez si cette technique permet d'évaluer la fonction avec des nombres en argument plus grands que dans sa forme initiale.

11.10.4 Exercice

Si vous avez résolu l'exercice 10.15.7, vous avez déjà une fonction appelée `has-duplicates` qui prend un tableau comme paramètre et retourne `true` si un objet apparaît plus d'une fois dans le tableau.

Utilisez un dictionnaire pour écrire une version plus rapide et plus simple de `has-duplicates`.

11.10.5 Exercice

Deux mots sont des « paires de rotation » si vous pouvez faire tourner l'un d'eux et obtenir l'autre (voir l'exercice 8.15.5).

Écrivez un programme qui lit un tableau de mots et trouve toutes les paires de rotation.

11.10.6 Exercice

Voici un autre casse-tête de Car Talk.

« Cette lettre a été envoyée par un certain Dan O'Leary. Il a récemment rencontré un mot commun d'une syllabe et de cinq lettres qui a la propriété unique suivante. Lorsque vous retirez la première lettre, les autres lettres forment un homophone². Remplacez la première lettre, c'est-à-dire remettez-la à sa place et supprimez la deuxième lettre et le résultat est encore un autre homophone du mot original. D'où la question : quel est le mot ? »

Je vais maintenant vous donner un exemple qui ne fonctionne pas. Examinons le mot de cinq lettres « wrack ». W-R-A-C-K, comme dans « to wrack with pain ». Si j'enlève la première lettre, il me reste un mot de quatre lettres, "R-A-C-K". Comme dans « Holy cow, did you see the rack on that buck! It must have been a nine-pointer! » C'est un homophone parfait. Si vous remettez le « w » et que vous enlevez le « r », vous obtenez le mot « wack », qui est un vrai mot mais pas l'homophone des deux autres termes.

2. Ce terme désigne des mots qui ont la même prononciation : « eau » et « haut » sont homophones.

Mais il y a au moins un mot que Dan et nous connaissons, qui donnera deux homophones si vous enlevez l'une des deux premières lettres pour faire deux nouveaux mots de quatre lettres. La question est de savoir quel est ce mot. »

Vous pouvez utiliser le dictionnaire de l'exercice 11.10.1 pour vérifier si une chaîne se trouve dans le tableau de mots.

Conseil. Pour vérifier si deux mots sont homophones, vous pouvez utiliser le Dictionnaire de prononciation anglaise de la CMU. Vous pouvez le télécharger sur le site [The CMU Pronouncing Dictionary](#).

Écrivez un programme qui pourrait afficher tous les termes de [mots_FR.txt](#) résolvant l'énigme.

Chapitre 12

Tuples

Ce chapitre présente un autre type intégré, les tuples¹. Par la suite, il montre comment les tableaux, les dictionnaires et les tuples peuvent être couplés. Nous présentons également une fonction utile pour les tableaux à arguments multiples ainsi que pour les opérateurs d'agrégation et de dispersion.

Note. Il n'y a pas de consensus sur prononciation de « tuple ». Vous pouvez écouter comment prononcer ce mot en anglais sur [ce site](#). Il semble qu'il n'y ait pas de prononciation française recommandée.

12.1 Les tuples sont persistants

Un tuple est une séquence de valeurs. Celles-ci peuvent être de type quelconque et elles sont indexées par des entiers, de sorte qu'à cet égard, les tuples ressemblent beaucoup aux tableaux. La différence notoire est que les tuples sont persistants (*immutable*). Chaque élément peut avoir son propre type (nombres, chaînes de caractères, tableaux, dictionnaires, tuples).

Syntaxiquement, un tuple est une liste de valeurs séparées par des virgules :

```
julia> t = 'a', 'b', 'c', 'd', 'e'
('a', 'b', 'c', 'd', 'e')
```

Sans que cela soit nécessaire, il est courant de mettre les tuples entre parenthèses :

1. Ce néologisme est basé sur le terme mathématique N-uplet : Table **UPLEt**

```
julia> t = ('a', 'b', 'c', 'd', 'e')
('a', 'b', 'c', 'd', 'e')
```

Pour créer un tuple avec un seul élément, il est nécessaire de ponctuer avec une virgule :

```
julia> t = ('a',)
('a',)
julia> typeof(t)
Tuple{Char}
```

Avertissement. Une valeur entre parenthèse sans virgule finale n'est pas un tuple :

```
julia> t2 = ('a')
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
julia> typeof(t2)
Char
```

Une autre manière de créer un tuple consiste à employer la fonction interne `tuple`. Sans aucun argument, elle crée un tuple vide :

```
julia> tuple()
()
```

Si plusieurs arguments sont fournis, le résultat est un tuple incorporant tous les arguments donnés :

```
julia> t3 = tuple(1, 'a', pi)
(1, 'a',  $\pi = 3.1415926535897...$ )
```

Comme `tuple` est le nom d'une fonction interne, il faut éviter de l'utiliser comme nom de variable.

La plupart des opérateurs agissant sur les tableaux fonctionnent également sur des tuples. L'opérateur crochet `[]` indice tout élément d'un tuple :

```
julia> t[2:4]
('b', 'c', 'd')
```

Cependant, comme les tuples sont persitants, vouloir en modifier un élément conduit à une erreur :

```
julia> t[1] = ('A')
ERROR: MethodError : no method matching
setindex!{::NTuple{5,Char}, ::Char, ::Int64}
```


Les opérateurs relationnels agissent sur les tuples comme avec d'autres séquences. Julia commence par comparer le premier élément de chaque séquence. S'ils sont égaux, Julia passe aux éléments suivants, et ainsi de suite jusqu'à trouver des éléments qui diffèrent. Les éléments ultérieurs ne sont pas pris en compte (même s'il s'agit, par exemple, de grands nombres).

```
julia> (0, 1, 2) < (0, 3, 4)
true
julia> (0, 1, 2000000) < (0, 3, 4)
true
```

12.2 Affectation des tuples

Il est souvent utile d'échanger les valeurs de deux variables. Dans le cas des affectations classiques, il est nécessaire d'utiliser une variable temporaire. Par exemple, pour permuter `a` et `b` :

```
temp = a
a = b
b = temp
```

Cette solution est remplacée par l'affectation de tuples qui s'avère plus élégante :

```
a, b = b, a
```

Le membre de gauche de l'affectation est un tuple de variables alors que le membre de droite est un tuple d'expressions. Chaque valeur est attribuée à sa variable respective. Toutes les expressions du membre de droite sont évaluées avant toute affectation.

Par ailleurs, le nombre de variables du membre de gauche doit être inférieur au nombre de valeurs du membre de droite :

```
julia> (a, b) = (1, 2, 3)
(1, 2, 3)
julia> a, b, c = 1, 2
ERROR: BoundsError: attempt to access (1, 2) at index [3]
```

Plus généralement, le membre de droite peut être tout type de séquence (chaîne, tableau ou tuple). Par exemple, pour séparer une adresse électronique en un nom d'utilisateur et un domaine, vous pouvez écrire ² :

2. Anecdotiquement, SPQR signifie *Senatus populusque romanus*

```
julia> addr = "jules.cesar@rome.spqr"
jules.cesar@rome
julia> uname, domain = split(addr, '@')
2-element Array{SubString{String},1} :
 "jules.cesar"
 "rome.spqr"
```

La valeur de retour de `split` est un tableau à deux éléments : le premier élément est attribué à `uname`, le second à `domain`. Si le séparateur avait été '.', Julia aurait affiché trois chaînes.

12.3 Les tuples en tant que valeurs retournées

À vrai dire, une fonction ne peut retourner qu'une seule valeur. Toutefois, si la valeur est un tuple, l'effet est le même que si elle retournait plusieurs valeurs. Par exemple, soit deux entiers dont on veut obtenir par division le quotient et le reste. Comme il est inefficace de calculer $x \div y$ puis $x \% y$, il est préférable de réaliser ces opérations simultanément.

La fonction interne `divrem` qui prend deux arguments retourne un tuple de deux valeurs, le quotient et le reste. Le résultat est enregistrable sous forme de tuple :

```
julia> t = divrem(7, 3)
(2, 1)
```

Il est aussi possible d'utiliser l'affectation des tuples pour enregistrer les éléments séparément :

```
julia> q, r = divrem(7, 3);
julia> @show q r;
q = 2
r = 1
```

Voici un exemple de fonction retournant un tuple :

```
function minmax(t)
    minimum(t), maximum(t)
end
```

`maximum` et `minimum` sont des fonctions internes qui identifient les éléments les plus petits et les plus grands d'une séquence. `minmax` calcule les deux et retourne un tuple de deux valeurs. La fonction interne `extrema` est plus efficace.

12.4 Tuples et arguments multiples

Les fonctions peuvent prendre un nombre variable d'arguments. Un nom de paramètre qui se termine par trois points consécutifs (...) *agrège* les arguments dans un tuple. Par exemple, `printall` prend un nombre quelconque d'arguments et les affiche :

```
function printall(args...)
    println(args)
end
```

Le nom du *paramètre d'agrégation* est libre quoique le terme `args` soit conventionnel. Voici comment procède la fonction :

```
julia> printall(1, 2.0, '3')
(1, 2.0, '3')
```

À l'agrégation s'oppose la *dispersion*³. Soit une séquence de valeurs qu'on souhaite passer à une fonction sous forme d'arguments multiples, il est alors possible d'utiliser l'opérateur `...`. Par exemple, `divrem` prend exactement deux arguments mais cette fonction ne peut pas manipuler directement de tuples :

```
julia> t = (7, 3);
julia> divrem(t)
ERROR: MethodError: no method matching divrem(::Tuple{Int64,Int64})
```

En revanche, il est envisageable de disperser le tuple :

```
julia> divrem(t...)
(2, 1)
```

De nombreuses fonctions internes utilisent des tuples à arguments en nombre variable. C'est le cas de `max` et `min` :

```
julia> max(1, 2, 3)
3
```

En revanche la fonction `sum` ne traite pas les tuples :

```
julia> sum(1, 2, 3)
ERROR: MethodError: no method matching sum(::Int64, ::Int64, ::Int64)
```

3. Dans le monde de Julia, l'agrégation est souvent appelée « *slurp* » et la dispersion « *splat* », en anglais du moins.

12.4.1 Exercice

Écrivez une fonction appelée `sumall` qui prend un nombre quelconque d'arguments et retourne leur somme.

12.5 Tableaux et tuples

`zip` est une fonction interne qui prend deux ou plusieurs séquences et retourne une collection de tuples où chacun contient un élément de chaque séquence. Le nom de la fonction fait référence à une fermeture éclair.

Cet exemple permet de « zipper » une chaîne et un tableau :

```
julia> s = "abc";
julia> t = [1, 2, 3];
julia> zip(s, t)
zip("abc", [1, 2, 3])
```

Le résultat est un *objet zip* capable d'itérer⁴ à travers les paires⁵. L'utilisation la plus courante de `zip` se rencontre dans une boucle :

```
julia> for pair in zip(s, t)
    println(pair)
end
('a', 1)
('b', 2)
('c', 3)
```

Un objet `zip` est une espèce d'itérateur agissant sur une séquence. D'une certaine manière, les itérateurs sont similaires aux tableaux, mais contrairement à ces derniers, il n'est pas possible d'utiliser les indices pour sélectionner un élément donné de l'itérateur.

Afin d'utiliser des opérateurs et des fonctions de tableau, il est astucieux d'utiliser un objet `zip` pour créer un tableau :

```
julia> collect(zip(s, t))
3-element Array{Tuple{Char,Int64},1}:
```

4. Itérer : avec la signification de parcourir les éléments d'un conteneur (au sens large). Itérateur : objet capable de parcourir tous les éléments contenus dans un autre objet.

5. La commande `typeof(zip(s,t))` retourne `Base.Iterators.Zip{Tuple{String,Array{Int64,1}}}`

```
('a', 1)
('b', 2)
('c', 3)
```

Le résultat obtenu est un tableau de tuples. Dans cet exemple, chaque tuple contient un caractère de la chaîne et l'indice correspondant du tableau.

Si les séquences ne sont pas de même longueur, la plus courte détermine la taille du résultat.

```
julia> collect(zip("Yann", "Le Cun"))
4-element Array{Tuple{Char,Char},1}:
('Y', 'L')
('a', 'e')
('n', ' ')
('n', 'C')
```

L'affectation des tuples est exploitable dans une boucle `for` pour parcourir un ensemble des tuples :

```
julia> t = [('a', 1), ('b', 2), ('c', 3)];
julia> for (letter, number) in t
    println(number, " ", letter)
end
1 a
2 b
3 c
```

À chaque fois, Julia sélectionne le tuple suivant dans le tableau et attribue les éléments à `letter` et `number`. Les parenthèses autour de `(letter, number)` sont obligatoires.

En combinant `zip`, `for` et l'affectation de tuples, on obtient une expression utile pour parcourir deux (voire plusieurs) séquences simultanément. Par exemple, `hasmatch` prend deux séquences, `t1` et `t2`, et retourne `true` s'il y a un indice `i` tel que `t1[i] == t2[i]` :

```
function hasmatch(t1, t2)
    for (x, y) in zip(t1, t2)
        if x == y
            return true
        end
    end
    false
end
```

S'il faut parcourir les éléments d'une séquence et leurs indices, la fonction interne `enumerate` s'avère utile :

```
julia> for (index, element) in enumerate("abc")
    println(index, " ", element)
end
1 a
2 b
3 c
```

Le résultat de la fonction `enumerate` est un objet recensé qui parcourt une séquence de paires. Chaque paire contient un indice (à partir de 1) et un élément de la séquence traitée.

12.6 Dictionnaires et tuples

Les dictionnaires peuvent être utilisés comme des itérateurs qui permettent de parcourir les paires clé-valeur. Nous pouvons y recourir dans une boucle `for` de ce type :

```
julia> d = Dict{'a'=>1, 'b'=>2, 'c'=>3};
julia> for (key, value) in d
    println(key, " ", value)
end
a 1
c 3
b 2
```

Comme on peut s'y attendre, les paires clé-valeur ne sont pas classées de manière ordonnée.

Inversement, un tableau de tuples est utilisable pour initialiser un nouveau dictionnaire :

```
julia> t = [('a', 1), ('c', 3), ('b', 2)];
julia> d = Dict{t}
Dict{Char,Int64} with 3 entries :
'a' => 1
'c' => 3
'b' => 2
```

En combinant `Dict` et `zip`, on accède à un procédé concis de création d'un dictionnaire :

```
julia> d = Dict{zip("abc", 1:3)}
Dict{Char,Int64} with 3 entries :
'a' => 1
'c' => 3
'b' => 2
```

Il est courant d'utiliser les tuples comme clés dans les dictionnaires. Par exemple, un annuaire téléphonique peut établir une correspondance entre des paires « nom – prénom » d'une part et de l'autre des numéros de téléphone. En supposant que nous ayons défini `last` (pour le nom de famille), `first` (pour le prénom) et `number`, nous pourrions écrire :

```
directory[last, first] = number
```

L'expression entre crochets est un tuple. Parcourir ce dictionnaire peut se faire *via* l'affectation des tuples.

```
for ((last, first), number) in directory
    println(first, " ", last, " ", number)
end
```

Cette boucle parcourt les paires clé-valeur dans `directory`, ces paires étant des tuples. Elle attribue les éléments de la clé dans chaque tuple à `last` et `first` ainsi que la valeur à `number`. Ensuite, elle affiche le nom et le numéro de téléphone correspondant.

Il existe deux façons de représenter les tuples dans un diagramme d'état. La version détaillée montre les indices et les éléments tels qu'ils apparaissent dans un tableau. Par exemple, le tuple ("`Rabelais`", "`François`") apparaîtra comme dans la figure 12.6.1.

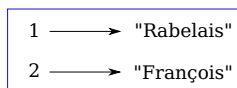


FIGURE 12.6.1 – Diagramme d'état associé au tuple ("`Rabelais`", "`François`")

Cependant, dans un diagramme plus étendu, il est pratique d'omettre les détails. Par exemple, un diagramme de l'annuaire téléphonique peut apparaître comme dans la figure 12.6.2.

Ici, les tuples sont représentés en utilisant la syntaxe de Julia comme raccourci graphique⁶.

6. Le numéro de téléphone indiqué dans le diagramme est le numéro du standard de Radio France (même si le numéro est public, ne l'appellez pas).

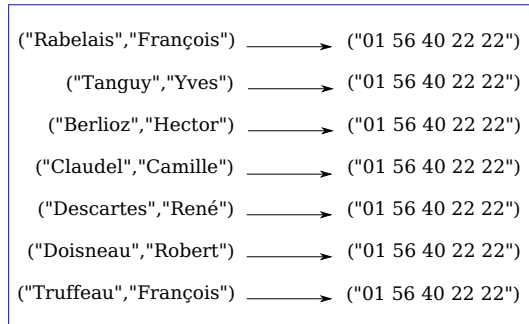


FIGURE 12.6.2 – Diagramme d'état associé au répertoire téléphonique.

12.7 Séquences de séquences

Notre attention s'est portée sur les tableaux de tuples. Néanmoins, presque tous les exemples de ce chapitre fonctionnent également avec des tableaux de tableaux, des tuples de tuples et des tuples de tableaux. Pour éviter d'énumérer les combinaisons possibles, il est parfois plus facile de parler de « séquences de séquences ».

Dans de nombreux contextes, les différents types de séquences (chaînes de caractères, tableaux et tuples) peuvent être utilisés de manière interchangeable. Alors, comment choisir l'une d'entre elles plutôt que les autres ?

Tout d'abord, les chaînes de caractères sont plus limitées que les autres séquences car les éléments doivent absolument être des caractères. Elles sont également persistantes. S'il faut pouvoir modifier les caractères d'une chaîne (plutôt que d'en créer une nouvelle), un tableau de caractères est plus pratique.

Les tableaux sont plus courants que les tuples, principalement parce qu'ils sont non persistants. Malgré cela, il existe quelques cas où les tuples leur sont préférés :

- dans certains contextes, comme une instruction de retour, il est syntaxiquement plus simple de créer un tuple qu'un tableau,
- quand une séquence est passée en argument d'une fonction, l'utilisation de tuples réduit le risque de comportements inattendus dû à l'*aliasing*,
- pour des raisons de performance, le compilateur pouvant se focaliser sur le type.

Comme les tuples sont persistants, ils ne fournissent pas de fonctions telles que `sort!` et `reverse!` qui modifient les tableaux existants. Cependant, Julia fournit la fonction interne `sort` qui prend un tableau et retourne un nouveau tableau avec les mêmes éléments triés et ordonnés. Julia met aussi à disposition la fonction `reverse` qui prend une séquence quelconque et retourne une séquence du même type dans l'ordre inverse.

12.8 Débogage

Les tableaux, les dictionnaires et les tuples sont des exemples de structures de données. Plus loin dans ce document, nous abordons les structures de données composites comme des tableaux de tuples ou des dictionnaires qui contiennent des tuples comme clés et des tableaux comme valeurs. Les structures de données composites sont utiles. Malheureusement, elles sont sujettes à des erreurs de forme, c'est-à-dire des erreurs causées lorsqu'une structure de données présente un mauvais type, une mauvaise taille ou une mauvaise structure. Par exemple, si vous attendez à un tableau avec un entier et que vous recevez un entier ordinaire (c'est-à-dire non incorporé à un tableau), la structure de données ne fonctionnera pas.

Julia permet d'attacher un type aux éléments d'une séquence. La manière de procéder est détaillée dans le [chapitre 17](#). Spécifier le type contribue à éliminer beaucoup d'erreurs de forme.

12.9 Glossaire

tuple séquence persistante d'éléments où chacun de ceux-ci peut avoir son propre type,

affectation de tuples affectation avec une séquence sur le membre de droite et un tuple de variables sur le membre de gauche. Le membre de droite est d'abord évalué et ses éléments sont ensuite affectés aux variables du membre de gauche,

agrégation opération consistant à assembler un tuple d'argument de longueur variable,

dispersion opération consistant à traiter une séquence comme une liste d'arguments,

objet zip résultat de l'appel de la fonction interne [zip](#). Objet qui itère à travers une séquence de tuples,

itérateur objet qui peut itérer à travers une séquence, mais qui ne fournit pas d'opérateurs ni de fonctions de tableau,

structure de données collection de valeurs connexes, souvent organisées en tableau, dictionnaires, tuples, etc.,

erreur de forme erreur causée par une valeur ayant une forme incorrecte, c'est-à-dire un type ou une taille incorrecte,

12.10 Exercices

12.10.1 Exercice

Écrivez une fonction appelée `mostfrequent` qui prend une chaîne de caractères et affiche les lettres par ordre décroissant d'occurrence. Trouvez des échantillons de texte de plusieurs langues différentes et voyez comment la fréquence des lettres varie d'une langue à l'autre. Comparez vos résultats avec les tableaux répertoriés sur les sites anglais et français.

12.10.2 Exercice

Encore quelques anagrammes...

1. Écrivez un programme qui lit une liste de mots à partir d'un fichier (voir la section 9.1) et imprime les ensembles de mots qui sont des anagrammes. Voici un échantillon de ce à quoi la sortie pourrait ressembler avec le fichier `mots_FR.txt` :

```
["mari", "mira", "rami", "rima"]  
["marche", "charme"]  
["mienne", "ennemi"]  
["alimenter", "terminale"]
```

Conseil. Si vous voulez construire un dictionnaire qui établit une correspondance entre une collection de lettres et une série de mots qui peuvent être orthographiés avec ces lettres, comment pouvez-vous représenter la collection de lettres de manière à ce qu'elle puisse servir de clé.

2. Modifiez le programme précédent de manière à ce qu'il affiche les anagrammes par ordre décroissant de taille.
3. Au jeu de Scrabble, un « scrabble⁷ » est un coup où les sept lettres de votre grille sont posées en les combinant à une lettre du tableau, pour former un mot de huit lettres. Quelle collection de 8 lettres forme le plus de « scrabble » possibles ?

7. En anglais, il s'agit d'un « bonus » (Grande-Bretagne) ou d'un « bingo » (États-Unis).

12.10.3 Exercice

Deux mots forment une paire de métathèse⁸ si vous pouvez transformer l'un en l'autre en échangeant deux lettres. Par exemple, « converser » et « conserver ». Écrivez un programme qui trouve toutes les paires de métathèse dans le dictionnaire.

Conseil. Ne testez pas toutes les paires de mots, et ne testez pas tous les échanges possibles.

Remarque. Cet exercice est inspiré d'un exemple se trouvant sur le site « Puzzlers.org ».

12.10.4 Exercice

Voici un autre exemple tiré du site [Car Talk](http://CarTalk.com).

« *Quel est le mot anglais le plus long, qui reste un mot anglais valide, car vous enlevez ses lettres une à une ?*

Maintenant, les lettres peuvent être enlevées à l'une ou l'autre extrémité, ou au milieu, mais vous ne pouvez pas réarranger les lettres. Chaque fois que vous laissez tomber une lettre, vous vous retrouvez avec un autre mot anglais. Si vous faites cela, vous finirez par avoir une lettre et ce sera aussi un mot anglais - un mot que l'on trouve dans le dictionnaire. Je veux savoir quel est le mot le plus long et combien de lettres il comporte ?

Je vais vous donner un petit exemple modeste : Sprite. D'accord ? On commence par sprite, on enlève une lettre, une de l'intérieur du mot, on enlève le r, et on se retrouve avec le mot spite, puis on enlève le e à la fin, on se retrouve avec spit, on enlève le s, on se retrouve avec pit, it, et I. »

Écrivez un programme pour trouver tous les mots qui peuvent être réduits de cette façon, puis trouvez le plus long.

Conseil. Le commentaire est associé à la langue anglaise. Utilisez [mots_FR.txt](#) (voir la section [Lecture de listes de mots](#)). Cet exercice est un peu plus difficile que la plupart des autres, aussi voici quelques suggestions :

1. vous pourriez écrire une fonction qui prend un mot et calcule un tableau de tous les mots qui peuvent être formés en enlevant une lettre. Ce sont les « mots-fils »,
2. récursivement, un mot est réductible si l'un de ses fils est réductible. Comme cas de base, vous pouvez considérer la chaîne vide comme réductible,
3. la chaîne vide peut être additionnée au dictionnaire,

8. Linguistiquement, il s'agit de la permutation de deux phonèmes dans la chaîne parlée (par exemple, *aéroplane* devient en français populaire *aréoplane*).

4. pour améliorer les performances de votre programme, vous pouvez utiliser un « mémo » pour les mots connus pour être réductibles.

Chapitre 13

Étude de cas: structures de données – choix

À ce stade, nous avons appris à connaître les structures de données de base en Julia et nous avons vu certains des algorithmes qui les exploitent.

Ce chapitre présente une étude de cas avec des exercices qui permettent de réfléchir à la sélection des structures de données en fonction du problème à résoudre.

13.1 Analyse de la fréquence des mots

Comme d’habitude, il convient au moins d’essayer de résoudre les exercices avant d’en lire les solutions.

13.1.1 Exercice

Écrivez un programme qui lit un fichier texte, découpe chaque ligne en mots et supprime les espaces ainsi que la ponctuation. Le programme doit ensuite convertir ces mots en minuscules.

Conseil. La fonction `isletter` teste si un caractère est alphabétique.

13.1.2 Exercice

Rendez-vous sur le site du [Projet Gutenberg](#) et téléchargez votre livre préféré en format texte brut (pour les livres en français, suivez [ce lien](#)).

Modifiez votre programme par rapport à l'exercice précédent pour parcourir le livre que vous avez téléchargé. Passez sur les informations d'en-tête au début du fichier et traitez le reste des mots comme dans l'exercice 13.1.1.

Modifiez ensuite le programme pour compter le nombre total de mots dans le livre et la fréquence d'utilisation de chaque mot.

Affichez le nombre de mots différents utilisés dans le livre. Comparez différents livres écrits à des époques différentes par des auteurs différents. Quel auteur emploie-t-il le vocabulaire le plus étendu ?

13.1.3 Exercice

Modifiez le programme de l'exercice 13.1.2 pour imprimer les 20 mots les plus fréquemment utilisés dans le livre que vous avez choisi.

13.1.4 Exercice

Modifiez le programme précédent pour lire une liste de mots, puis affichez tous les mots du livre qui ne figurent pas dans la liste de mots. Combien d'entre eux résultent de fautes de frappe ? Combien d'entre eux sont des mots courants qui devraient se trouver dans la liste de mots ? Combien d'entre eux sont rares ?

13.2 Nombres aléatoires

Avec les mêmes données en entrée, la plupart des programmes informatiques produisent les mêmes données en sortie à chaque exécution. Ces programmes sont dits déterministes. Le *déterminisme* est généralement souhaité puisque nous souhaitons que le même calcul donne le même résultat. Cependant, dans certaines applications (comme la sécurité informatique, l'analyse mathématique, les jeux, etc.), nous tenons à ce que l'ordinateur produise des données aléatoires.

Rendre un programme véritablement non-déterministe (c'est-à-dire stochastique) est une tâche difficile. Néanmoins, il existe des moyens de le rendre au moins quasi non-déterministe. Un de ces moyens consiste à utiliser des algorithmes qui fournissent des nombres pseudo-aléatoires. Ces nombres ne sont pas vraiment aléatoires parce qu'ils sont produits par un calcul déterministe, mais en première analyse, il est pratiquement impossible de les distinguer de nombres réellement sortis au hasard.

La fonction `rand` retourne un flottant aléatoire entre 0.0 et 1.0 (y compris 0.0 mais pas 1.0). Chaque fois que `rand` est appelée un nombre extrait d'une longue série est produit. Par exemple, nous pouvons faire tourner cette boucle :

```
for i in 1:10
    x = rand()
    println(x)
end
```

La fonction `rand` est susceptible de prendre un itérateur ou un tableau comme argument et elle retourne un élément aléatoire :

```
for i in 1:10
    x = rand(1:6)
    print(x, " ")
end
```

13.2.1 Exercice

Écrivez une fonction appelée `choosefromhist` qui prend un histogramme tel que défini dans la section 11.2 et qui retourne une valeur aléatoire de l’histogramme choisi avec une probabilité proportionnelle à la fréquence. Par exemple, pour cet histogramme :

```
julia> t = ['a', 'a', 'b'];

julia> histogram(t)
Dict{Any,Any} with 2 entries:
'a' => 2
'b' => 1
```

Votre fonction devrait retourner 'a' avec une probabilité de $2/3$ et 'b' avec une probabilité de $1/3$.

13.3 Histogramme des mots

Dans un premier temps, essayez les exercices précédents avant de poursuivre. Dans un second temps, rendez vous sur [cette page](#) où vous pouvez télécharger le texte brut UTF-8 de l’œuvre de Victor Hugo intitulée « *Notre Dame de Paris* ». Une fois le fichier enregistré sous `notre_dame_de_paris.txt`, supprimez l’en-tête jusqu’au titre « LIVRE PREMIER » non inclus.

À présent, voici un programme qui lit un fichier et construit un histogramme des mots inclus dans ce fichier :

```

function processfile(filename)
    hist = Dict()
    for line in eachline(filename)
        processline(line, hist)
    end
    hist
end;

function processline(line, hist)
    line = replace(line, '-' => ' ')
    for word in split(line)
        word = string(filter(isletter, [word...]))
        word = lowercase(word)
        hist[word] = get!(hist, word, 0) + 1
    end
end;

```

```
hist = processfile("/home/chemin_vers_fichier/notre_dame_de_paris.txt");
```

Ce programme lit le fichier `notre_dame_de_paris.txt`.

`processfile` parcourt en boucle les lignes du fichier, les passant une à une à `processline`. L'histogramme `hist` est utilisé comme un accumulateur.

`processline` utilise la fonction `replace` pour remplacer les traits d'union par des espaces avant d'utiliser `split` pour diviser la ligne traitée en un tableau de chaînes de caractères. La boucle `for` de `processline` traverse le tableau de mots et utilise `filter`, `isletter` et `lowercase` pour supprimer la ponctuation et convertir les caractères en minuscule.¹

Enfin, `processline` met à jour l'histogramme soit en créant un nouvel élément en cas de première détection, soit en incrémentant un élément existant.

Pour compter le nombre total de mots dans le fichier, il faut additionner les fréquences dans l'histogramme :

```

function totalwords(hist)
    sum(values(hist))
end

```

Le nombre de mots différents n'est que le nombre d'éléments dans le dictionnaire :

```

function differentwords(hist)
    length(hist)
end

```

1. C'est un raccourci que de déclarer : « les chaînes de caractères sont converties ». En effet, les chaînes sont persistantes. En réalité, une fonction comme `lowercase` retourne de nouvelles chaînes.

Voici le code permettant d'afficher les résultats :

```
julia> println("Nombre total de mots: ", totalwords(hist))
Nombre total de mots: 172053

julia> println("Nombre de mots différents: ", differentwords(hist))
Nombre de mots différents: 17510
```

13.4 Mots les plus communs

Pour trouver les mots les plus courants, nous pouvons (i) créer un tableau de tuples où chacun d'eux contient un mot et sa fréquence d'apparition et (ii) le trier. La fonction suivante prend un histogramme et retourne un tableau de tuples d'occurrence des mots :

```
function mostcommon(hist)
    t = []
    for (key, value) in hist
        push!(t, (value, key))
    end
    reverse(sort(t))
end
```

Dans chaque tuple, la fréquence apparaît en premier, de sorte que le tableau résultant est trié par fréquences. Voici une boucle qui affiche les 10 mots les plus courants :

```
t = mostcommon(hist)
println("Les mots les plus courants sont:")
for (freq, word) in t[1:10]
    println(word, "\t", freq)
end
```

Plutôt qu'un espace, un caractère de tabulation (`\t`) est employé comme séparateur. De cette manière, la deuxième colonne est alignée. Voici les résultats pour *Notre Dame de Paris* :

```
Les mots les plus courants sont:
de      8263
la      5427
et      4585
le      4124
à       3363
les     2456
```

il	2446
un	2042
que	2033
qui	1978

Conseil. Ce code peut être simplifié en utilisant le mot-clé `rev` associé à la fonction `sort`. À ce sujet, il est judicieux de consulter la page de [documentation de sort](#).

13.5 Paramètres optionnels

Nous avons vu des fonctions internes à Julia qui prennent des arguments optionnels (voir l'exercice 8.15.1). Il est également possible d'écrire des fonctions définies par le programmeur avec des arguments optionnels. Par exemple, voici une fonction qui affiche les mots les plus courants dans un histogramme :

```
function printmostcommon(hist, num=10)
    t = mostcommon(hist)
    println("Les mots les plus courants sont: ")
    for (freq, word) in t[1:num]
        println(word, "\t", freq)
    end
end
```

Le premier paramètre est obligatoire, le second facultatif. La valeur par défaut de `num` est 10.

Si un seul argument est fourni :

```
printmostcommon(hist)
```

`num` prend la valeur par défaut. Si vous fournissez deux arguments :

```
printmostcommon(hist, 20)
```

`num` prend la valeur de l'argument indiqué. En d'autres termes, l'argument optionnel remplace la valeur par défaut.

Si une fonction possède à la fois des paramètres obligatoires et facultatifs, tous les paramètres obligatoires apparaîtront les premiers.

13.6 Soustraction de dictionnaires

Trouver les mots d'un livre, qui ne sont pas dans la liste de mots de `mots_FR.txt` revient à un problème de soustraction d'ensembles, c'est-à-dire trouver « les mots du

livre moins les mots de la liste ».

`subtract` prend les dictionnaires `d1` et `d2` et, ensuite, retourne un nouveau dictionnaire qui contient toutes les clés de `d1` qui ne se trouvent pas dans `d2`. Comme nous ne nous soucions pas vraiment des valeurs, nous pouvons toutes les fixer à `nothing`.

```
function subtract(d1, d2)
  res = Dict()
  for key in keys(d1)
    if key ∉ keys(d2)
      res[key] = nothing
    end
  end
  res
end
```

En vue de trouver les mots de *Notre Dame de Paris* n'appartenant pas au fichier `mots_FR.txt`, nous pouvons utiliser `processfile` pour construire un histogramme associé à `mots_FR.txt`, et ensuite `subtract` :

```
words = processfile("mots_FR.txt")
diff = subtract(hist, words)
println("Mots du roman n'appartenant pas à la liste mots_FR.txt:")
for word in keys(diff)
  print(word, " ")
end
```

Voici un bref extrait des résultats obtenus dans le cas de *Notre Dame de Paris* :

```
Mots du roman n'appartenant pas à la liste mots_FR.txt:
lambrissé pâtisserie polyèdre magnétisme apostrophé damoiselles caballero gisante
bretonne ...
```

Certains de ces mots sont des noms, des adverbes, etc. D'autres, comme « damoiselles », ne sont plus d'usage courant en français. Toutefois, certains sont des mots courants qui devraient absolument figurer dans la liste.

13.6.1 Exercice

Julia fournit une structure de données appelée `Set` qui permet de nombreuses opérations usuelles. Vous pouvez les consulter à la section 20.2 du chapitre 20, et lire la documentation `Set-Like Collections`.

Écrivez un programme exploitant la soustraction associée à `Set` pour détecter des mots du roman *Notre Dame de Paris* qui ne sont pas dans la liste `mots_FR.txt`.

13.7 Mots aléatoires

Pour choisir un mot aléatoire dans l'histogramme, l'algorithme le plus simple consiste à construire un tableau contenant plusieurs copies de chaque mot selon la fréquence observée. Ensuite, il faut les extraire du tableau :

```
function randomword(h)
  t = []
  for (word, freq) in h
    for i in 1:freq
      push!(t, word)
    end
  end
  rand(t)
end
```

Une autre possibilité consiste à :

1. utiliser des clés pour obtenir un tableau des mots du livre,
2. construire un tableau qui contient la somme des occurrences d'un mot (voir l'exercice 10.15.2). Le dernier élément de ce tableau est le nombre total (n) de mots dans le livre,
3. choisir un nombre aléatoire compris entre 1 et n . Utiliser une recherche par bisection (voir exercice 10.15.10) pour trouver l'indice où le nombre aléatoire pourrait être inséré dans la somme,
4. utiliser l'indice pour trouver le mot correspondant dans le tableau de mots.

13.7.1 Exercice

Écrivez un programme qui applique cet algorithme pour choisir un mot aléatoire dans le livre.

13.8 Analyse de Markov

Si quelqu'un choisissait au hasard des mots du roman *Notre Dame de Paris*, il pourrait se faire une idée du vocabulaire, mais il n'obtiendrait très probablement pas une phrase cohérente. Voici le résultat d'une extraction aléatoire :

supplie des jours société de elle me vie en je

Une série de mots pris au hasard présente rarement du sens car il n'existe pas de relation entre les mots successifs. Par exemple, dans une phrase sémantiquement correcte, on attend qu'un article comme « le » soit très probablement suivi d'un adjectif ou d'un nom mais pas d'un verbe et encore moins d'un adverbe.

L'*analyse de Markov* est une manière d'établir ce genre de relations. Pour une séquence de mots donnée, cette analyse caractérise la probabilité qu'un mot en suive un autre. Par exemple, voici un extrait du poème *Liberté* de Paul Éluard :

Sur mes cahiers d'écolier
Sur mon pupitre et les arbres
Sur le sable sur la neige
J'écris ton nom
[...]
Sur la lampe qui s'allume
Sur la lampe qui s'éteint
Sur mes maisons réunies
J'écris ton nom
[...]
Sur mes refuges détruits
Sur mes phares écroulés
Sur les murs de mon ennui
J'écris ton nom
[...]
Et par le pouvoir d'un mot
Je recommence ma vie
Je suis né pour te connaître
Pour te nommer

Liberté

En l'occurrence, l'expression « la lampe » est toujours suivie du pronom relatif « qui », alors que l'expression « lampe qui » peut être suivie des verbes pronominaux conjugués au présent de l'indicatif « s'allume » ou « s'éteint ».

L'analyse de Markov consiste en une mise en correspondance de chaque préfixe (comme « la lampe » ou « lampe qui ») avec tous les suffixes possibles (comme « s'allume » ou « s'éteint »).

Grâce à cette correspondance, il devient possible de concevoir un texte aléatoire en commençant par n'importe quel préfixe et en choisissant aléatoirement parmi les suffixes possibles. Ensuite, la particule finale du préfixe peut être combinée avec le nouveau suffixe pour former le préfixe suivant. Le procédé est alors répété.

Par exemple, si nous commençons par le préfixe « le sable », le mot suivant doit être « sur » (c'est la seule combinaison entre ces deux termes observable dans l'extrait). Le préfixe suivant est « sable sur », donc le suffixe suivant doit être « la ». Ensuite, le préfixe suivant est « sur la » qui a pour suffixes soit « neige », soit « lampe ».

Dans cet exemple, le nombre de mots formant un préfixe vaut toujours deux, mais il est tout-à-fait envisageable d'entreprendre une analyse de Markov avec des préfixes dont les mots apparaissent en nombre quelconque, voire variable.

13.8.1 Exercice

Analyse de Markov :

1. Écrivez un programme destiné à lire un texte depuis un fichier et à effectuer une analyse de Markov. Le résultat doit être un dictionnaire qui établit une correspondance entre les préfixes et une collection de suffixes possibles. La collection peut être un tableau, un tuple ou un dictionnaire. Il est de votre ressort de faire un choix approprié. Vous pouvez tester votre programme avec un nombre de préfixes égal à deux, mais votre programme devrait permettre de tester facilement des nombres > 2 .
2. Ajoutez une fonction au programme précédent pour produire un texte aléatoire basé sur l'analyse de Markov. Dans un premier temps, utilisez un préfixe constitué de 2 mots sur le texte de *Notre Dame de Paris*.
Que se passe-t-il si vous augmentez la longueur du préfixe ? Le texte aléatoire a-t-il plus de sens ?
3. Une fois votre programme opérationnel, vous pouvez essayer une combinaison : si vous mélangez des textes provenant de deux ou plusieurs livres, le texte aléatoire résultant assemblera le vocabulaire et les phrases des diverses sources de manière intéressante.

Remarque. Ce problème est basé sur un exemple de Brian K. Kernighan et Rob Pike (voir la référence [9]).

Conseil. Vous devriez essayer cet exercice avant de poursuivre.

13.9 Structure de données

Utiliser l'analyse de Markov pour générer du texte aléatoire est amusant, cependant cet exercice a aussi un intérêt : la sélection de la structure des données. Dans les exercices précédents, vous deviez exercer des choix :

- comment représenter les préfixes,
- comment représenter la collection de suffixes possibles,
- comment représenter la correspondance entre chaque préfixe et la collection de suffixes possibles.

La dernière solution est simple. Un dictionnaire constitue le choix évident pour établir une correspondance entre les clés et les valeurs correspondantes.

Pour les préfixes, les options les plus évidentes sont les chaînes de caractères, les tableaux de chaînes de caractères ou les tuples de chaînes de caractères. Pour les suffixes, un tableau est une option ; un histogramme de dictionnaire en est une autre.

Comment choisir ? La première étape consiste à réfléchir aux opérations qu’il est nécessaire d’implémenter pour chaque structure de données. Pour les préfixes, nous devons pouvoir supprimer des mots du début et en ajouter à la fin. Par exemple, si le préfixe est « le sable », et que le mot suivant est « sur », il faut pouvoir former le préfixe suivant, « sable sur ». Un tableau pourrait constituer un premier choix car il est facile d’y ajouter et d’y supprimer des éléments. Pour la collection de suffixes, les opérations à effectuer comprennent l’ajout d’un nouveau suffixe (ou l’augmentation de la fréquence d’un suffixe existant) ainsi que le choix d’un suffixe aléatoire.

L’ajout d’un nouveau suffixe est tout aussi facile dans le cas d’un tableau que dans celui d’un histogramme. Choisir un élément aléatoire dans un tableau est commode ; en choisir un dans un histogramme s’avère plus difficile à réaliser efficacement (voir l’exercice 13.7.1).

Jusqu’à présent, la facilité de mise en œuvre a été privilégiée. Cependant, d’autres facteurs interviennent dans le choix de la structure de données. L’un d’eux est le temps d’exécution. Parfois, une raison théorique permet d’espérer qu’une structure de données se montre plus rapide qu’une autre. Par exemple, nous avons vu que l’opérateur `∈` est plus rapide pour les dictionnaires que pour les tableaux, du moins lorsque le nombre d’éléments est important.

Cependant, très souvent, nul ne sait *a priori* quelle implémentation sera la plus rapide. Une option consiste à développer deux implémentations et à observer la meilleure. Cette approche est appelée *analyse comparative* (ou *benchmarking*). Une autre option pratique consiste à choisir la structure de données la plus facile à mettre en œuvre, puis à analyser si elle est suffisamment rapide pour l’application prévue. Si c’est le cas, il n’est pas nécessaire de chercher plus loin. Sinon, des outils, comme le module [Profile](#) (voir la [documentation Julia](#) et [Profileview](#)), permettent d’identifier les blocs les plus lents d’un programme. Citons également [BenchmarkTools](#) (voir [BenchmarkTools.jl](#)) qui fournit diverses macros permettant d’automatiser les mesures de temps d’exécution sur plusieurs jeux de paramètres (aléatoires, par exemple) ².

2. Les paquets [Profile.jl](#), [ProfileView.jl](#) et [BenchmarkTools.jl](#) sont installables selon les méthodes indiquées

L'espace de stockage constitue un autre facteur à prendre en compte. Par exemple, l'utilisation d'un histogramme pour la collecte des suffixes peut prendre moins d'espace car il suffit d'enregistrer chaque mot une seule fois, quelle que soit sa fréquence dans le texte. Dans certains cas, l'économie d'espace peut également accélérer le fonctionnement d'un programme et, à l'extrême, ce dernier peut ne pas fonctionner du tout en cas de mémoire saturée. Néanmoins, pour de nombreuses applications, l'espace mémoire reste une considération secondaire par rapport au temps d'exécution.

Une dernière réflexion : dans cette discussion, nous avons laissé entendre qu'une seule structure de données était à exploiter à la fois pour l'analyse et le développement. Parce que ces deux phases sont distinctes, il est envisageable d'utiliser une structure pour l'analyse et de la convertir ensuite en une autre pour le développement. Le gain sera net si le temps épargné pendant le développement dépasse celui consacré à la conversion.

Conseil. Le paquet Julia [DataStructures](#) implémente 24 structures de données (voir la documentation sur les structures de données et [DataStructures.jl](#)).

13.10 Débogage

Lors du débogage d'un programme, cinq attitudes sont à adopter en particulier lorsque le bogue est très difficile à débusquer :

la lecture examinez votre code, relisez-le avec détachement, comme si c'était le programme de quelqu'un d'autre et vérifiez ce qu'il exécute réellement,

le fonctionnement expérimentez, en introduisant de petites modifications et en exécutant différentes versions. Souvent, si vous affichez le bon élément au bon endroit dans le programme, le problème saute aux yeux. Parfois, il faut construire un canevas.

la réflexion prenez le temps de réfléchir. De quel type d'erreur s'agit-il : syntaxique, d'exécution ou sémantique ? Quelles informations pouvez-vous tirer des messages d'erreur ou du rendu du programme ? Quel type d'erreur pourrait causer le problème que vous constatez ? Qu'avez-vous modifié en dernier lieu, avant que le problème n'apparaisse ?³

la méthode du canard en plastique (*rubber ducking*) si vous expliquez le problème à quelqu'un d'autre, vous trouvez parfois la réponse avant même d'avoir

dans l'appendice B, page 299.

3. À cet égard, le contrôle de version dans les environnements de développement intégré s'avère particulièrement utile.

fini de poser la question. Souvent, vous n'avez même pas besoin d'une autre personne. Vous pourriez simplement parler à un canard en plastique. Là, se trouve l'origine de la stratégie bien connue appelée « débogage par la méthode du canard en plastique »⁴.

Le « **lâcher prise** » à un moment donné, la meilleure chose à faire est de reculer et d'annuler les changements récents jusqu'à revenir à un programme qui fonctionne et que vous comprenez. Ensuite, une nouvelle période de reconstruction pourra commencer.

Les programmeurs débutants se trouvent parfois bloqués sur une de ces activités et oublient les autres. Chaque activité a ses propres limites. Par exemple, la lecture du code peut aider si le problème est d'ordre typographique, mais pas si le problème résulte d'un malentendu conceptuel. Si vous ne comprenez pas l'exécution de votre programme, vous pouvez le lire 100 fois et ne jamais repérer l'erreur car celle-ci est mentale.

Faire des expériences peut aider, surtout en procédant par de petits tests simples. En revanche, si vous faites des essais sans stratégie ou sans lire votre code, vous risquez de basculer dans de la « programmation en mode aléatoire » qui consiste à faire des changements aléatoires jusqu'à ce que le programme fasse ce à quoi il est destiné. Il va sans dire que la programmation en mode aléatoire est très chronophage et sans réelle garantie de résultat.

Vous devez prendre le temps de réfléchir. Le débogage est de même essence qu'une science expérimentale. Vous devez avoir au moins une hypothèse sur la nature du problème. S'il y a deux possibilités ou plus, essayez de penser à un test discriminant.

Pour tout dire, même les meilleures techniques de débogage échouent lorsqu'il y a trop d'erreurs ou si le code que vous essayez de corriger est trop volumineux et trop complexe. Parfois, la meilleure option est de battre en retraite, en simplifiant le programme jusqu'à arriver à un code qui fonctionne et qui est parfaitement compris.

Les programmeurs débutants sont souvent réticents à rebrousser chemin parce qu'ils ne supportent pas d'effacer une ligne de code (même lorsqu'elle est erronée). Si cela vous rassure, sauvegardez votre programme dans un autre fichier avant de commencer à le décortiquer. Vous pourrez ensuite recopier un à un les blocs analysés et corrects.

Pour trouver un bogue difficile, il est nécessaire de lire, d'expérimenter, de réfléchir et parfois... de reculer. Si vous êtes bloqué sur l'une de ces activités, n'hésitez pas à essayer les autres.

4. La méthode du canard en plastique consiste à expliquer méticuleusement le code source qu'on a écrit à un collègue, à un simple passant, ou même à un objet inanimé comme un canard en plastique (voir Wikipedia EN).

13.11 Glossaire

déterministe terme qui se rapporte à un programme qui produit le même résultat à chaque exécution, avec les mêmes apports (voir la notion de fonction déterministe en informatique),

pseudo-aléatoire terme se rapportant à une séquence de nombres qui semble être aléatoire mais qui est conçue par un programme déterministe,

valeur par défaut valeur donnée à un paramètre optionnel si aucun argument n'est fourni,

outrépassement (*override*) remplacement d'une valeur par défaut par un argument fourni par l'utilisateur,

analyse comparative (*benchmarking*) processus consistant à choisir entre les structures de données en implémentant des alternatives et en les testant sur un échantillon d'entrées possibles,

méthode du canard en plastique (*rubber ducking*) méthode de débogage où le programmeur explique à un objet inanimé (tel qu'un canard en plastique) son problème en le décomposant pour se rendre compte de la ou des faille(s).

13.12 Exercices

13.12.1 Exercice

Le « rang » d'un mot est sa position dans un tableau de mots triés par fréquence : le mot le plus courant a le rang 1, le deuxième le plus courant, le rang 2, etc.

La loi de Zipf décrit une relation entre le rang et la fréquence des mots dans les langues naturelles. Plus précisément, elle prédit que la fréquence f du mot de rang r s'exprime comme :

$$f = cr^{-s}$$

où s et c sont des paramètres qui dépendent de la langue et du texte. Si vous prenez le logarithme des deux membres de cette équation, vous obtenez :

$$\log f = \log c - s \log r$$

Si vous tracez $\log f$ en fonction de $\log r$, vous devriez obtenir une droite de pente $-s$ interceptant l'ordonnée à l'origine à $\log c$.

Rédigez un programme qui lit un texte à partir d'un fichier, comptabilise la fréquence des mots et affiche une ligne pour chaque mot par ordre décroissant de fréquence, avec $\log f$ et $\log r$.

Installez la bibliothèque graphique [Plots](#) comme indiqué dans l'annexe B, page 299. Son utilisation est très facile :

```
julia> using Plots
julia> x = 0:10
julia> y = x.^2
julia> plot(x, y)
julia> savefig("/home/chemin_vers_repertoire/quadratic.pdf")
```

La figure 13.12.1 présente le graphique [quadratic.pdf](#).

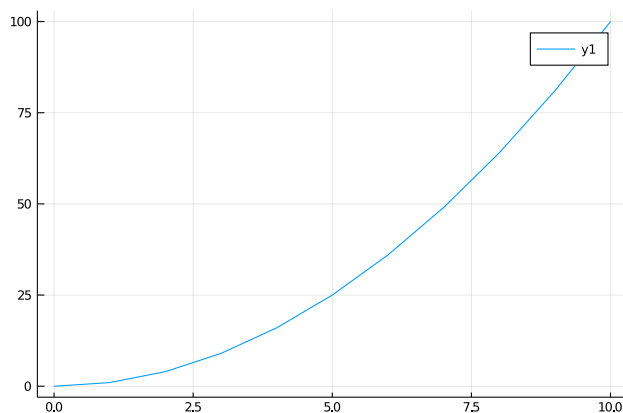


FIGURE 13.12.1 – Graphique « Plots » de la fonction $y = x^2$ pour $0 \leq x \leq 10$

Utilisez la bibliothèque [Plots](#) pour mettre en graphique les résultats et vérifier qu'ils forment bien une droite.

Chapitre 14

Fichiers

Ce chapitre introduit la notion de programmes persistants, capables de conserver les données en stockage permanent. Il montre comment utiliser différents types de stockage permanent tels que les fichiers et les bases de données.

14.1 Persistance

La plupart des programmes que nous avons étudiés jusqu'à présent sont transitoires : ils fonctionnent pendant une courte période et produisent un résultat. Cependant, lorsqu'ils se terminent, leurs données disparaissent. Lors d'une nouvelle exécution, le programme repart *ut nulla*.

D'autres programmes sont persistants : ils s'exécutent sur de longues périodes (voire continuellement) et conservent au moins une partie de leurs données sur un support de stockage permanent (un disque dur, par exemple). S'ils s'arrêtent puis redémarrent, ils reprennent au point d'arrêt.

Les systèmes d'exploitation qui fonctionnent pratiquement à chaque fois qu'un ordinateur est allumé sont des exemples de programmes persistants. C'est encore plus vrai pour les serveurs web, qui fonctionnent en permanence, attendant des requêtes en provenance du réseau.

Un des moyens les plus simples pour que les programmes conservent leurs données consiste à lire et à écrire des fichiers « texte ». Jusqu'ici, nous avons vu des programmes qui lisent des fichiers texte. Dans ce chapitre, nous en étudions dont le comportement consiste à écrire dans des fichiers.

Une autre possibilité consiste à stocker l'état d'un programme dans une base de

données. Nous étudions donc également la manière de tirer avantage d'une base de données simple.

14.2 Lire et écrire

Un fichier texte est une séquence de caractères enregistrée sur un support permanent comme un disque dur ou une mémoire flash. Nous avons vu comment ouvrir et lire un fichier dans la section 9.1.

Pour créer un fichier et y écrire, il faut l'ouvrir avec le mode `"w"` (*writing*) en deuxième paramètre :

```
julia> fout = open("output.txt", "w")
IOStream(<file output.txt>)
```

Si le fichier existe déjà, l'ouverture en mode écriture efface irréversiblement les anciennes données. Prudence, donc ! Si le fichier n'existe pas, il est nouvellement créé. La fonction `open` retourne un objet fichier et la fonction d'écriture `"w"` permet d'écrire diverses données dans le fichier.

```
julia> line1 = "Voici quatre lions, deux rouges, deux noirs,\n";
julia> write(fout, line1)
45
```

La valeur numérique retournée représente le nombre de caractères écrits dans le fichier. L'objet fichier conserve une trace de l'endroit où il se trouve dans l'arborescence de l'ordinateur. Donc, si elle est à nouveau appelée, la fonction `write` ajoute les nouvelles données à la suite des précédentes :

```
julia> line2 = "Tels qu'ils figurent sur les armoiries d'Élouges.\n";
julia> write(fout, line2)
51
```

Une fois la saisie terminée, il est pertinent de fermer le fichier.

```
julia> close(fout)
```

Si la fermeture du fichier n'a pas eu lieu, la terminaison du programme s'en chargera. L'état du fichier peut être contrôlé en l'ouvrant avec un éditeur de texte.

14.3 Formatage

L'argument en écriture doit impérativement être une chaîne de caractères. Ainsi, pour écrire d'autres valeurs, il est nécessaire de les convertir en chaînes de caractères.

Le moyen le plus simple est de procéder *via* l'interpolation de chaînes :

```
julia> fout = open("output.txt", "w")
IOStream(<file output.txt>)
julia> write(fout, string(7370))
4
```

Une autre possibilité consiste à utiliser la famille de fonctions `print(ln)`.

```
julia> chapelles = 2
julia> println(fout, "À Élouges, il y a $chapelles chapelles.")
```

Conseil. La macro `@printf` est très puissante du fait qu'elle utilise des commandes de formatage de style C. Voir la documentation de `printf`.

14.4 Noms de fichiers et chemins

Les fichiers sont organisés en répertoires (également appelés « dossiers »). Chaque programme en cours d'exécution possède un répertoire courant, c'est-à-dire celui par défaut pour la plupart des opérations. Par exemple, lorsqu'un fichier est ouvert aux fins de lecture, Julia le cherche dans le répertoire courant.

La fonction `pwd` retourne le nom du répertoire courant : ¹

```
julia> cwd = pwd()
"/home/aquarelle/Julia/Penser_en_Julia"
```

`cwd` signifie *current working directory* (répertoire de travail courant). Dans cet exemple, le résultat est `/home/aquarelle/Julia/Penser_en_Julia`, où `/home/aquarelle/` est le répertoire d'origine d'un utilisateur ² nommé `aquarelle`.

Une chaîne comme `/home/aquarelle/Julia/` identifie un répertoire et constitue un *chemin d'accès absolu*.

Un simple nom de fichier, comme `output.txt`, est considéré comme un *chemin relatif* car il se rapporte au répertoire courant. Si le répertoire courant est `/home/aquarelle/Julia-/Penser_en_Julia/`, le nom de fichier `output.txt` fera référence à `/home/aquarelle/Julia-/Penser_en_Julia/output.txt`.

Un chemin qui commence par `/` ne dépend pas du répertoire courant. Il s'agit d'un chemin absolu. Pour trouver le chemin absolu d'un fichier, il convient d'utiliser `abs-path` :

-
1. `pwd` –*print working directory*– est une commande des systèmes UNIX, GNU/LINUX et BSD.
 2. Ce répertoire est souvent noté `$HOME`.

```
julia> abspath("output.txt")
"/home/aquarelle/Julia/Penser_en_Julia/output.txt"
```

Julia propose d'autres fonctions pour travailler avec les noms de fichiers et les chemins d'accès. Par exemple, `ispath` vérifie si un fichier ou un répertoire existe :

```
julia> ispath("/home/aquarelle/Julia/Penser_en_Julia")
true
julia> ispath("/home/aquarelle/Julia/Penser_en_Julia/mots_FR.txt")
true
```

`isdir` vérifie l'existence d'un sous-répertoire appartenant au répertoire courant (les répertoires situés au-dessus du répertoire courant, y compris ce dernier, ne sont pas testés) :

```
julia> isdir("Chimie_Julia")
false
julia> isdir("Figures")
true
```

De même, `isfile` vérifie l'existence d'un fichier.

`readdir` retourne un tableau des fichiers (et autres répertoires) dans le répertoire courant :

```
julia> isfile("Figures/quadratic_1.pdf")
true
julia> readdir(cwd)
7-element Array{String,1} :
 "#Penser_en_Julia_v-0-24.lyx#"
 "Solutions_Exercices"
 "Figures"
 "Penser_en_Julia_v-0-24.lyx"
 "Penser_en_Julia_v-0-24.lyx~"
 "Penser_en_Julia_v-0-24.pdf"
 "Tests"
 "mots_FR.txt"
 "notre_dame_de_paris.txt"
```

Pour illustrer l'action de ces fonctions, la fonction `walk` explore un répertoire, affiche le nom de tous les fichiers et s'appelle récursivement au niveau des sous-répertoires.


```
function walk(dirname)
  for name in readdir(dirname)
    path = joinpath(dirname, name)
    if isfile(path)
      println(path)
    else
      walk(path)
    end
  end
end
```

La fonction interne `joinpath` prend un répertoire et un nom de fichier pour les réunir en un chemin complet.

Conseil. Julia propose une fonction appelée `walkdir` (voir la page de documentation de `walkdir`) qui est similaire à `walk` tout en s'avérant polyvalente. À titre d'exercice, lisez la documentation de `walkdir` et utilisez cette fonction pour afficher les noms des fichiers dans un répertoire donné et ses sous-répertoires.

14.5 Levée des exceptions

Lorsque des fichiers sont manipulés en lecture et en écriture, des erreurs peuvent survenir. Tenter d'ouvrir un fichier qui n'existe pas conduit à une `SystemError` :

```
julia> fin = open("fichier_inexistant.txt")
ERROR: SystemError: opening file "bad_file": Aucun fichier ou dossier de ce type
```

Si vous ne disposez pas des droits d'accès à un fichier (en l'occurrence en écriture), sous GNU/LINUX par exemple, une erreur se produit :

```
julia> fout = open("/etc/passwd", "w")
ERROR: SystemError: opening file "/etc/passwd": Permission non accordée
```

Afin d'éviter ces erreurs, il serait envisageable d'utiliser des fonctions comme `ispath` et `isfile`. Hélas, cela nécessiterait beaucoup de temps et de code pour vérifier toutes les possibilités.

La fonction `try` offre une solution élégante. La syntaxe est similaire à celle d'une instruction `if` :

```
try
  fin = open("fichier_inexistant.txt")
catch exc
  println("Une erreur s'est produite : $exc")
end
```

Julia commence par exécuter la clause `try`. Si tout se passe normalement, le système sort de la clause `try` et poursuit l'exécution du programme. Si une exception se produit, Julia passe à l'exécution de la clause `catch`.

Le traitement d'une exception avec une clause `try` s'appelle la *levée d'une exception*. Dans cet exemple, la clause d'exception affiche un message d'erreur (peu utile). Cependant, en général, capter une exception apporte une chance réelle de résoudre le problème, d'essayer à nouveau ou, au moins, de terminer le programme sagement.

Dans un code qui effectue des changements d'état ou qui utilise des ressources telles que des fichiers, un travail de nettoyage (fermeture de fichiers) doit généralement être réalisé lorsque le code est terminé. Les exceptions compliquent potentiellement cette tâche car elles peuvent provoquer la sortie d'un bloc de code avant d'atteindre sa fin normale. Le mot-clé `finally` permet d'exécuter un code lorsqu'un bloc se termine, quelle que soit la manière dont il prend fin :

```
f = open("output.txt")
try
    line = readline(f)
    println(line)
finally
    close(f)
end
```

De cette sorte, la fonction `close` est toujours exécutée.

14.6 Bases de données

Une base de données est un fichier spécialement organisé pour conserver des données. De nombreuses bases de données sont structurées comme un dictionnaire : elles établissent une correspondance entre des clés et leurs valeurs. La plus grande différence entre une base de données et un dictionnaire vient de ce que la première réside sur un support permanent. Par conséquent, elle persiste après la fin du programme.

ThinkJulia (et ThinkJuliaFR) fournit une interface au gestionnaire de base de données `GDBM` (GNU dbm³) pour la création et la mise à jour des fichiers de base de données. À titre d'exemple, créons une base de données qui contient des légendes associées à des fichiers contenant des images.

L'ouverture d'une base de données est similaire à l'ouverture de fichiers classiques :

3. `database manager`

```
julia> using ThinkJulia

➡ pour la version française :
julia> using ThinkJuliaFR

julia> db = DBM("légendes", "c")
DBM(<légendes>)
```

Le mode **"c"** signifie que la base de données doit être créée si elle n'existe pas déjà. Le résultat est un objet « base de données » qui peut être utilisé (pour la plupart des opérations) à l'instar d'un dictionnaire.

Lorsqu'un nouvel élément est créé, le gestionnaire GDBM met à jour le fichier de la base de données :

```
julia> db["rabelais.png"] = "Portrait de François Rabelais."
"Portrait de François Rabelais."
```

Lorsque vous accédez à un des éléments, le gestionnaire GDBM lit le fichier :

```
julia> db["rabelais.png"]
"Portrait de François Rabelais."
```

Si vous effectuez une nouvelle affectation à une clé existante, le gestionnaire GDBM remplace l'ancienne valeur :

```
julia> db["rabelais.png"] = "Portrait de François Rabelais promenant le chat
« Raminagrobis »."
"Portrait de François Rabelais promenant le chat « Raminagrobis »."
julia> db["rabelais.png"]
"Portrait de François Rabelais promenant le chat « Raminagrobis »."
```

Certaines fonctions ayant un dictionnaire comme argument, comme **key** et **value**, sont inopérantes avec les objets de la base de données. Toutefois, l'itération avec une boucle **for** fonctionne :

```
for (key, value) in db
    println(key, ": ", value)
end
```

Comme pour les autres fichiers, *in fine*, il est nécessaire de fermer la base de données :

```
julia> close(db)
```

14.7 Sérialisation

Une des limites du gestionnaire **GDBM** provient du fait que les clés et les valeurs doivent être des chaînes de caractères ou des tableaux d'octets. Si un autre type est utilisé, Julia retourne une erreur.

Cependant, les fonctions **serialize** et **deserialize** contribuent à contourner cette limitation. Elles traduisent presque tout type d'objet en un tableau d'octets (**iobuffer**⁴) adapté au stockage dans une base de données. Ensuite, elles retranscrivent les tableaux d'octets en objets :

```
julia> using Serialization
julia> io = IOBuffer();

julia> t = [1, 2, 3];

julia> serialize(io, t)
24
julia> print(take!(io))
UInt8[0x37, 0x4a, 0x4c, 0x07, 0x04, 0x00, 0x00, 0x00, 0x15, 0x00, 0x08, 0xe2, 0x01,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]
```

Le format n'est pas évident pour nous ; il est essentiellement censé être facile à interpréter pour Julia.

deserialize reconstitue l'objet :

```
julia> io = IOBuffer();

julia> t1 = [1, 2, 3];

julia> serialize(io, t1)
24
julia> s = take!(io);

julia> t2 = deserialize(IOBuffer(s));

julia> print(t2)
[1, 2, 3]
```

serialize et **deserialize** écrivent et lisent un objet **iobuffer** qui représente un flux d'entrées/sorties en mémoire. La fonction **take!** récupère le contenu du tampon d'entrées/sorties sous la forme d'un tableau d'octets et remet le tampon dans son état initial.

Bien que le nouvel objet ait la même valeur que l'ancien, il ne s'agit pas du même objet (du moins en général) :

4. mémoire tampon d'entrées/sorties

```
julia> t1 == t2
true
julia> t1 ≡ t2
false
```

En d’autres termes, la sérialisation puis la désérialisation ont le même effet que la copie de l’objet.

Vous pouvez l’utiliser pour enregistrer des structures de données autres que des chaînes de caractères dans une base de données.

Conseil. En fait, le stockage de structures de données autres que des chaînes de caractères dans une base de données est si courant qu’il a été encapsulé dans un paquet appelé JLD2 (voir la [documentation de JLD2](#)).

14.8 Objets de commande

La plupart des systèmes d’exploitation fournissent une interface en ligne de commande, également appelée « shell »⁵. Les shells fournissent entre autres des commandes pour naviguer dans le système de fichiers et aussi, pour lancer des applications. Par exemple, les systèmes UNIX permettent de changer de répertoire avec `cd`, d’afficher le contenu d’un répertoire avec `ls`, et de lancer un navigateur web en saisissant une commande comme `firefox` (par exemple) dans une console.

Tout programme pouvant être lancé depuis le shell l’est également depuis Julia à l’aide d’un objet de commande :

```
julia> cmd = `echo Julia est un langage très puissant.`
`echo Julia est un langage très puissant.`
```

Les guillemets inversés sont utilisés pour délimiter la commande. La fonction `run` exécute cette dernière :

```
julia> run(cmd)
Julia est un langage très puissant.
```

La partie « Julia est un langage très puissant. » représente la sortie de la commande `echo`, envoyée à STDOUT⁶. La fonction `run` elle-même retourne un objet processus et émet une `ErrorException` si la commande externe ne s’exécute pas correctement.

Si vous souhaitez lire la sortie de la commande externe, il convient d’utiliser `read` au lieu de `run` :

5. Un shell est l’interface entre un utilisateur et le noyau.

6. **ST**andard **OUT**put

```
julia> a = read(cmd, String)
"Julia est un langage très puissant.\n"
```

La plupart des systèmes UNIX fournissent une commande appelée `md5sum` ou `md5` qui lit le contenu d'un fichier et calcule une « somme de contrôle ». Vous pouvez en apprendre davantage sur la page [MD5 de Wikipédia](#). Cette commande fournit un moyen efficace de vérifier si deux fichiers présentent le même contenu. La probabilité que des contenus différents donnent la même somme de contrôle est extrêmement faible.

Vous pouvez utiliser une commande pour lancer `md5` à partir de Julia et obtenir le résultat :

```
julia> filename = "output.txt"
"output.txt"
julia> cmd = `md5 $filename`
`md5 output.txt`
julia> res = read(cmd, String)
"MD5 (output.txt) = d41d8cd98f00b204e9800998ecf8427e\n"
```

14.9 Modules

Supposons que vous ayez un fichier nommé `wc.jl` contenant les 9 lignes de code suivant :

```
function linecount(filename)
    count = 0
    for line in eachline(filename)
        count += 1
    end
    count
end

print(linecount("wc.jl"))
```

À l'exécution, ce programme se lit lui-même et affiche le nombre de lignes du fichier, soit 9. Il est permis de l'inclure dans le REPL comme ceci :

```
julia> include("wc.jl")
9
```

Julia introduit des modules permettant de créer des espaces de travail variables distincts, c'est-à-dire avec de nouvelles portées générales.

Un module commence par le mot-clé `module` et se termine par `end`. Les conflits de noms sont évités entre vos propres définitions de haut niveau et celles trouvées dans

le code de quelqu'un d'autre. `import` permet de contrôler quels noms d'autres modules sont visibles et `export` spécifie les noms de fichiers qui sont publics parmi les vôtres, c'est-à-dire ceux qui peuvent être utilisés en dehors du module sans être préfixés par le nom du module.

```
module LineCount
  export linecount
  function linecount(filename)
    count = 0
    for line in eachline(filename)
      count += 1
    end
    count
  end
end
```

L'objet `LineCount` fournit la commande `linecount` :

```
julia> using LineCount
julia> linecount("wc.jl")
11
```

14.9.1 Exercice

Saisissez cet exemple dans un fichier nommé `wc.jl`, incluez-le dans le REPL et entrez `using LineCount`.

Avertissement. Si vous réimportez un module, Julia ne réagira pas et ne relira pas le fichier même en cas de modification. Si vous voulez recharger un module, vous devez redémarrer le REPL. Il existe un paquet `Revise` qui permet de prolonger la durée des sessions (voir `Revise.jl`).

14.10 Débogage

Lorsque des fichiers sont utilisés en lecture et en écriture, des problèmes d'espace-ment sont susceptibles d'apparaître. Ces erreurs sont souvent difficiles à déboguer car les espaces, les tabulations et les nouvelles lignes sont normalement invisibles :

```
julia> s = "1 2\t 3\n 4";
julia> println(s)
1 2   3
4
```

Les fonctions internes `repr` ou `dump` sont des auxiliaires pratiques. Elles prennent tout objet en argument et retournent une représentation de l'objet sous forme de chaîne.

```
julia> repr(s)
"\1 2\t 3\n 4\"
julia> dump(s)
String "1 2\t 3\n 4"
```

Cela peut être utile pour le débogage.

Un autre problème susceptible d'être rencontré tient au fait que divers systèmes utilisent différents caractères pour indiquer la fin d'une ligne. Certains utilisent une nouvelle ligne, représentée par `\n`, d'autres un caractère de retour, représenté `\r`, d'autres encore les deux. Si des fichiers sont échangés entre différents systèmes, des incohérences de ce type peuvent survenir.

Pour la plupart des systèmes, il existe des applications permettant la conversion d'un format à un autre. Vous pouvez les trouver (et en apprendre plus sur cette question) en consultant la page Wikipédia traitant la *fin de ligne*. Bien entendu, vous pouvez en écrire une vous-même.

14.11 Glossaire

persistant terme qui se rapporte à un programme qui fonctionne « indéfiniment » et qui conserve au moins une partie de ses données sur un support permanent,

fichier texte séquence de caractères enregistrée en permanence sur un disque dur (par exemple),

répertoire collection de fichiers nommés (parfois appelée « dossier »),

chemin d'accès chaîne de caractères qui identifie un fichier,

chemin relatif chemin d'accès qui part du répertoire courant,

chemin absolu chemin d'accès qui part du répertoire racine (ou *root*, symbolisé par `/` sur les systèmes UNIX, GNU/LINUX et BSD),

catch commande pour éviter qu'une exception ne mette fin à un programme en utilisant les déclarations `try ... catch ... finally`,

base de données fichier dont le contenu est organisé comme un dictionnaire avec des clés correspondant à des valeurs,

shell programme qui permet aux utilisateurs de saisir des commandes et de les exécuter en lançant d'autres programmes (exemples : `bash`, `csh`, `sh`, etc.),

objet de commande objet qui représente une commande shell, permettant à un programme Julia d'exécuter des commandes et d'en lire le résultat.

14.12 Exercices

14.12.1 Exercice

Écrivez une fonction appelée `sed` qui prend comme arguments une chaîne de caractères modèle, une chaîne de remplacement et deux noms de fichiers. Cette fonction doit lire le premier fichier et écrire le contenu dans le second, en le créant si nécessaire. Si la chaîne modèle apparaît quelque part dans le fichier, elle doit être substituée par la chaîne de remplacement.

Si une erreur se produit lors de l'ouverture, de la lecture, de l'écriture ou de la fermeture d'un fichier, votre programme doit capter l'exception, afficher un message d'erreur puis quitter le flux de programmation.

14.12.2 Exercice

Si vous avez résolu l'exercice 12.10.2, vous savez qu'un dictionnaire est créé qui établit une correspondance entre une chaîne de lettres triée et la série de mots pouvant être orthographiés avec ces lettres. Par exemple, `"spot"` correspond au tableau `["post", "pots", "stop", "tops"]`.

Écrivez un module qui importe `anagramsets` et qui fournit deux nouvelles fonctions : (i) `storeanagrams` qui doit enregistrer le dictionnaire des anagrammes en utilisant JLD2, (ii) `readanagrams` qui doit rechercher un mot et retourner un tableau de ses anagrammes.

14.12.3 Exercice

Dans une grande collection de fichiers MP3, il peut y avoir plusieurs copies d'une même chanson stockées dans différents répertoires ou avec des noms de fichiers différents. L'objectif de cet exercice est de rechercher les doubles.

1. Écrivez un programme qui effectue une recherche dans un répertoire ainsi que tous ses sous-répertoires (donc, de manière récursive) et qui retourne un tableau de chemins complets pour tous les fichiers ayant un suffixe donné (comme `.mp3`).
2. Pour reconnaître les doubles, vous pouvez utiliser `md5sum` ou `md5` pour calculer une « somme de contrôle » associée à chaque fichier. Si deux fichiers ont la même somme de contrôle, ils ont plus que probablement le même contenu.
3. Pour effectuer une double vérification, vous pouvez utiliser la commande UNIX `diff`.

Chapitre 15

Structures et objets

À ce stade, nous savons comment utiliser d'une part les fonctions pour organiser le code et d'autre part les types intégrés (ou internes) afin d'organiser les données. L'étape suivante consiste à apprendre à construire nos propres types pour organiser à la fois code et données. C'est un sujet important : quelques chapitres seront nécessaires pour y parvenir.

15.1 Types composites

Jusqu'ici, nous avons utilisé de nombreux types internes à Julia. Nous allons maintenant définir un type personnel. À titre d'exemple, nous allons créer un type appelé `Point` qui représente un point dans un espace bidimensionnel. En notation mathématique, les points sont souvent écrits entre parenthèses avec une virgule séparant les coordonnées. Par exemple, dans un repère cartésien \mathbb{R}^2 , $(0,0)$ représente l'origine et (x,y) un point dont l'abscisse vaut x et l'ordonnée y .

Il y a plusieurs façons de traiter cette information en Julia :

1. les coordonnées pourraient être enregistrées séparément dans deux variables, `x` et `y`,
2. elles pourraient se retrouver sous forme d'éléments au sein d'un tableau ou d'un tuple,
3. enfin, nous pourrions créer un type personnalisé pour représenter les points sous forme d'objets.

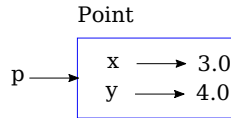


FIGURE 15.1.1 – Diagramme d’objet pour le type Point avec la valeur de ses attributs.

La création d’un nouveau type est plus compliquée que les autres options, mais elle présente des avantages qui apparaîtront bientôt.

Un *type composite* défini par le programmeur est également appelé une *structure*. La définition `struct` pour représenter un point ressemble à ceci :

```

struct Point
    x
    y
end
  
```

L’en-tête indique que la nouvelle structure s’appelle `Point`. Le corps définit les *attributs* ou les *champs* de la structure. La structure `Point` possède deux champs : `x` et `y`.

Une structure fonctionne comme une usine créant des objets. Pour créer un point, vous appelez `Point` comme s’il s’agissait d’une fonction ayant pour arguments les valeurs des champs. Lorsque `Point` est utilisé comme une fonction, on l’appelle un *constructeur*.

```

julia> p = Point(3.0, 4.0)
Point{Float64}(3.0, 4.0)
  
```

La valeur retournée est une référence à un objet `Point`, que nous affectons à `p`. La création d’un nouvel objet est une *instanciation* et l’objet est une *instance* du type. Lorsqu’une instance est affichée, Julia indique à quel type elle appartient et quelles sont les valeurs des attributs. Chaque objet est une instance d’un certain type, donc les termes « objet » et « instance » sont interchangeables. Cependant, dans ce chapitre, nous utilisons le terme « instance » pour désigner un type défini par le programmeur.

Un diagramme d’état qui rend compte d’un objet et de ses champs est appelé un diagramme d’objet (voir la figure 15.1.1).

15.2 Les structures sont persistantes

La valeur des champs (ou attributs) peut être extraite en utilisant la notation `.` de cette manière :

```
julia> x = p.x
3.0
julia> y = p.y
4.0
```

L'expression `p.x` signifie : « Allez à l'objet auquel `p` se réfère et extrayez la valeur de `x` ». Dans l'exemple, nous attribuons cette valeur à une variable nommée `x`. Il n'y a pas de conflit entre la variable `x` et le champ `x`.

La notation par points peut être exploitée dans le cadre de n'importe quelle expression. Par exemple, en considérant `p.x` et `p.y` comme des longueurs, le théorème de Pythagore est applicable :

```
julia> distance = sqrt(p.x^2 + p.y^2)
5.0
```

Ceci dit, par défaut, les structures sont persistantes. Après construction, les champs ne peuvent pas changer de valeur :

```
julia> p.y = 1.0
ERROR: setfield! immutable struct of type Point cannot be changed
```

Le caractère persistant des structures présente plusieurs avantages :

- l'efficacité,
- l'impossibilité de violer les invariants fournis par les constructeurs du type (voir la section 17.4),
- la facilité à raisonner sur du code utilisant des objets persistants.

15.3 Structures non-persistantes

S'il y échet, des types composites non persistants peuvent être déclarés avec le mot-clé `mutable struct`. Voici la définition d'un point modifiable (MPoint pour « Mutable ») :

```
mutable struct MPoint
    x
    y
end
```

Ainsi, il devient possible d'attribuer des valeurs à une instance d'une structure mutable en utilisant la notation par points :

```
julia> blank = MPoint(0.0, 0.0)
MPoint(0.0, 0.0)
julia> blank.x = 3.0
3.0
julia> blank.y = 4.0
4.0
```

15.4 Rectangles

Régulièrement, les champs d'un objet sont évidents. Néanmoins, dans certaines circonstances, il faut opérer des choix. Imaginons qu'il faille concevoir un type pour représenter des rectangles. Quels champs utiliser afin de spécifier l'emplacement et la taille d'un rectangle ? Pour simplifier les choses, supposons que le rectangle est disposé soit verticalement, soit horizontalement (pour éviter de traiter le problème de son inclinaison).

Il existe au moins deux possibilités :

- nous pouvons définir un coin du rectangle (ou son centre), la largeur et la hauteur,
- nous pouvons définir deux coins opposés.

À ce stade, il est difficile de dire quel est le meilleur choix. À titre d'exemple, mettons en œuvre le premier cas de figure.

```
" " "
Représentation d'un rectangle.
champs : largeur, hauteur, coin.
" " "

struct Rectangle
    largeur
    hauteur
    coin
end
```

La mini-documentation précise les champs : la **largeur** et la **hauteur** sont des nombres tandis que **coin** (qui requiert 2 coordonnées) est un objet **Point** qui définit le coin inférieur gauche.

Pour représenter un rectangle, il faut instancier un objet **Rectangle** :

```
julia> origine = MPoint(0.0, 0.0)
MPoint(0.0, 0.0)
julia> box = Rectangle(100.0, 200.0, origine)
Rectangle(100.0, 200.0, MPoint(0.0, 0.0))
```

Le diagramme d'objet (voir la figure 15.4.1) montre l'état de l'objet `box`. Un objet tel que `coin`, champ d'un autre objet, est dit *intégré* (*embedded*). Étant donné que l'attribut `coin` fait référence à un objet non persistant (`MPoint`), ce dernier est dessiné en dehors de l'objet `Rectangle`.

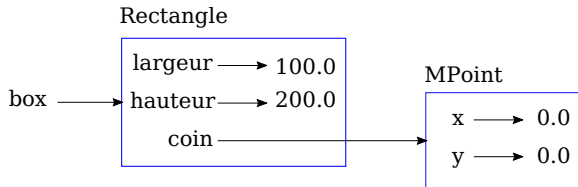


FIGURE 15.4.1 – Diagramme d'objet pour l'objet `Rectangle`.

15.5 Instances en arguments

Il est possible de faire passer une instance comme un argument selon la méthode habituelle. Par exemple :

```
function printpoint(p)
    println("( $(p.x), $(p.y) )")
end
```

`printpoint` prend une instance `Point` comme argument et l'affiche en notation mathématique. Pour l'invoquer, vous pouvez passer `p` comme argument :

```
julia> printpoint(blank)
(3.0, 4.0)
```

Si un objet struct non persistant est passé à une fonction en tant qu'argument, cette fonction peut modifier les champs de l'objet. Par exemple, `movepoint!` prend un objet `Point` non persistant et deux nombres, `dx` et `dy`, et ajoute –respectivement– ces nombres aux attributs `x` et `y` de `Point` :

```
function movepoint!(p, dx, dy)
    p.x += dx
    p.y += dy
    nothing
end
```

Voici un exemple qui en démontre l'effet :

```
julia> origine = MPoint(0.0, 0.0)
MPoint(0.0, 0.0)
julia> movepoint!(origine, 1.0, 2.0)

julia> origine
MPoint(1.0, 2.0)
```

À l'intérieur de la fonction, `p` est un alias d'`origine`. Donc, lorsque la fonction modifie `p`, `origine` est affecté.

Passer un objet `Point` persistant à `movepoint!` provoque une erreur :

```
julia> movepoint!(p, 1.0, 2.0)
ERROR: setfield! immutable struct of type Point cannot be changed
```

Cependant, il est permis de modifier la valeur d'un attribut non persistant au sein d'un objet persistant. Par exemple, `moverectangle!` a comme arguments un objet `Rectangle` et deux nombres (`dx` et `dy`). Cette fonction utilise `movepoint!` pour déplacer le coin du rectangle :

```
function moverectangle!(rect, dx, dy)
    movepoint!(rect.coin, dx, dy)
end
```

De ce fait, `p` dans `movepoint!` est un alias de `rect.coin`. Aussi, quand `p` est modifié, `rect.coin` l'est-il également :

```
julia> box
Rectangle(100.0, 200.0, MPoint(0.0, 0.0))
julia> moverectangle!(box, 1.0, 2.0)

julia> box
Rectangle(100.0, 200.0, MPoint(1.0, 2.0))
```

Avertissement. Vous ne pouvez pas réaffecter un attribut non persistant associé à un objet persistant :

```
julia> box.coin = MPoint(1.0, 2.0)
ERROR: setfield: immutable struct of type Rectangle cannot be changed
```

15.5.1 Exercice

Écrivez une fonction appelée `distancebetweenpoints` qui prend deux points comme arguments et retourne la distance entre eux.

15.6 Instances en tant que valeurs retournées

Les fonctions peuvent renvoyer des instances. Par exemple, `findcenter` prend `Rectangle` comme argument et retourne un `Point` qui contient les coordonnées du centre du rectangle :

```
function findcenter(rect)
    Point(rect.coin.x + rect.largeur / 2, rect.coin.y + rect.hauteur / 2)
end
```

L'expression `rect.coin.x` signifie : « Allez à l'objet `rect` et sélectionnez le champ nommé `coin`, puis rendez vous à cet objet et sélectionnez le champ nommé `x` ».

Voici un exemple qui passe `box` en argument et affecte le `Point` résultant à `centre` :

```
julia> centre = findcenter(box)
Point(51.0, 102.0)
```

15.7 Copies

L'*aliasing* peut rendre un programme difficile à lire parce que les modifications apportées à un endroit peuvent avoir des effets inattendus ailleurs dans le code. Il est difficile de garder une trace de toutes les variables qui pourraient se référer à un objet donné.

La copie d'un objet est souvent une alternative judicieuse à l'*aliasing*. Julia propose une fonction appelée `deepcopy` qui peut dupliquer tout objet :

```
julia> p1 = MPoint(3.0, 4.0)
MPoint(3.0, 4.0)
julia> p2 = deepcopy(p1)
MPoint(3.0, 4.0)
julia> p1 == p2
false
julia> p1 === p2
false
```

L'opérateur `==` indique que `p1` et `p2` ne constituent pas le même objet, ce qui est attendu. En revanche, on aurait pu attendre que `==` donne `true`, car ces points contiennent les mêmes données. Dans ce cas, il faut être attentif au fait que pour les objets non persistants, le comportement par défaut de l'opérateur `==` est le même que celui de l'opérateur `===`. L'opérateur `==` vérifie l'identité de l'objet et non son équivalence. C'est parce que pour les types composites non persistants, Julia ne sait pas ce qui doit être considéré comme équivalent (du moins, pas encore).

15.7.1 Exercice

Créez une instance de `Point`, faites-en une copie et vérifiez l'équivalence ainsi que l'égalité de la copie et de son original. Le résultat peut vous surprendre, mais il montre pourquoi l'*aliasing* ne constitue pas un problème pour un objet persistant.

15.8 Débogage

Lorsqu'on commence à travailler avec des objets, de nouvelles exceptions sont susceptibles d'apparaître. Par exemple, une tentative d'accès à un champ qui n'existe pas se solde par une erreur :

```
julia> p = Point(3.0, 4.0)
Point(3.0, 4.0)
julia> p.z = 1.0
ERROR: type Point has no field z Stacktrace: [1] setproperty!{::Point, ::Symbol, ::Float64} at ./sysimg.jl:19 [2] top-level scope at none :0
```

Au cas où vous ne seriez pas sûr du type d'objet, il convient de le demander à Julia :

```
julia> typeof(p)
Point
```

La fonction `isa` est également utilisable pour vérifier qu'un objet est bel et bien une instance d'un type :

```
julia> p isa Point
true
```

S'il n'est pas sûr qu'un objet possède un attribut particulier, la fonction interne `fieldnames` s'avère utile :

```
julia> fieldnames(Point)
(:x, :y)
```

Alternativement, la fonction `isdefined` est utilisable :

```
julia> isdefined(p, :x)
true
julia> isdefined(p, :z)
false
```

Le premier argument désigne tout objet. Le deuxième argument est composé du symbole `:` suivi du nom du champ.

15.9 Glossaire

struct type composite,

constructeur fonction portant le même nom qu'un type qui crée des instances de ce type,

instance objet qui appartient à un type,

instancier créer un nouvel objet,

attribut ou champ une des valeurs nommées associées à un objet,

objet intégré objet « enchassé » dans le champ d'un autre objet,

duplication (*deep copy*) action de copier le contenu d'un objet ainsi que tous les objets qui y sont intégrés, et ainsi de suite. La duplication est mise en œuvre par la fonction `deepcopy`,

diagramme d'objet diagramme qui montre les objets, leurs champs et les valeurs des champs.

15.10 Exercices

15.10.1 Exercice

1. Rédigez une définition pour un type nommé `Circle` avec des champs `center` et `radius`, où `center` est un objet `Point` et `radius` est un nombre.
2. Instancier un objet `circle` qui représente un cercle dont le centre est à (150, 100) et le rayon = 75.
3. Écrivez une fonction appelée `pointincircle` qui prend un objet `Circle` et un objet `Point` et qui retourne `true` si le point se trouve dans ou sur la circonférence du cercle.
4. Écrivez une fonction appelée `rectincircle` qui prend un objet `Cercle` et un objet `Rectangle` et qui retourne `true` si le rectangle se trouve entièrement dans le cercle ou touche la circonférence.
5. Écrivez une fonction appelée `rectcircleoverlap` qui prend un objet `Cercle` et un objet `Rectangle` et retourne `true` si l'un des coins du rectangle se trouve à l'intérieur du cercle. Ou, dans une version plus complexe, elle retourne `true` si une partie quelconque du rectangle intersecte le cercle.

15.10.2 Exercice

1. Écrivez une fonction appelée `drawrect` qui prend un objet « turtle » et un objet `Rectangle` et utilise la tortue pour dessiner le rectangle. Voir le chapitre 4 pour des exemples d'utilisation d'objets `Turtle`.
2. Écrivez une fonction appelée `drawcircle` qui prend un objet `Turtle` et un objet `Cercle` et, qui dessine le cercle.

Chapitre 16

Structures et fonctions

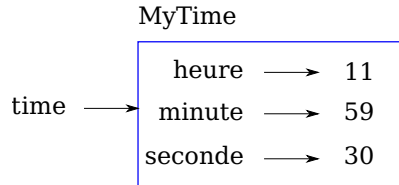
Dès lors que nous savons créer de nouveaux types composites, l'étape suivante consiste à écrire des fonctions qui prennent des objets définis par le programmeur comme paramètres et les retournent en tant que résultats. Dans ce chapitre, nous présentons également le « style de programmation fonctionnelle » et deux nouveaux plans de développement de programmes.

16.1 Heures, minutes et secondes

Comme autre exemple de type composite, nous allons définir une `struct` appelée `MyTime` qui enregistre l'heure associée à un moment de la journée. Voici la définition de la structure :

```
" " "  
Représentation d'un moment de la journée.  
Champs : heure, minute, seconde  
" " "  
  
mutable struct MyTime  
    heure  
    minute  
    seconde  
  
end
```

Le mot `Time` étant déjà utilisé dans Julia, choisissons `MyTime` pour éviter tout conflit et créons un nouvel objet `MyTime` :

FIGURE 16.1.1 – Diagramme d'objet associé à `MyTime`.

```
julia> time = MyTime(11, 59, 30)
MyTime(11, 59, 30)
```

Le diagramme d'objet pour `MyTime` est représenté à la figure 16.1.1.

16.1.1 Exercice

Écrivez une fonction appelée `printtime` qui prend un objet `MyTime` et l'affiche sous la forme `heure:minute:seconde`. La macro `@printf` du module `StdLib Printf` affiche un entier avec le format `"%02d"` en utilisant au moins deux chiffres, y compris un zéro de tête si nécessaire.

16.1.2 Exercice

Écrivez une fonction booléenne appelée `isafter` qui prend deux objets `MyTime`, `t1` et `t2`, et qui retourne `true` si `t1` suit chronologiquement `t2` et `false` dans le cas contraire. Défi : n'utilisez pas de test `if`.

16.2 Fonctions pures

Dans les prochaines sections, nous allons écrire deux fonctions qui ajoutent des valeurs de temps. Le but est d'appréhender deux types de fonctions : les fonctions pures et les modificateurs. Nous verrons également un plan de développement appelé *prototype et correctifs*. Un plan de développement est un procédé permettant de s'attaquer à un problème complexe en commençant par un prototype simple et en y incorporant graduellement des éléments qui le complexifient.

Voici un prototype simple `addtime` :

```
function addtime(t1, t2)
    MyTime(t1.heure + t2.heure, t1.minute + t2.minute, t1.seconde + t2.seconde)
end
```

La fonction crée un nouvel objet `MyTime`, initialise ses champs et retourne une référence au nouvel objet. On parle de *fonction pure* car elle ne modifie aucun des objets qui lui sont transmis en tant qu'arguments. En outre, elle n'a aucun autre effet (tel l'affichage d'une valeur ou l'obtention d'une entrée utilisateur) que le renvoi d'une valeur.

Pour tester cette fonction, créons deux objets `MyTime` :

- `start` contient l'heure de début d'un film, comme *Le nom de la rose* de Jean-Jacques Annaud.
- `duration` contient la durée du film (2 heures 11 minutes).

`addtime` indique quand le film sera terminé.

```
julia> start = MyTime(9, 55, 0);
julia> duration = MyTime(2, 11, 0);
julia> done = addtime(start, duration);
julia> printtime(done)    # voir exercice 16.1.1
11:66:00
```

Le résultat 11:66:00 est à peine inattendu. Le problème vient de ce que cette fonction ne traite pas les cas où le nombre de secondes ou de minutes dépasse 60. Lorsque cela se produit, il est nécessaire de « reporter » les secondes supplémentaires dans la colonne des minutes et/ou les minutes supplémentaires dans la colonne des heures. Voici une version améliorée :

```
function addtime(t1, t2)
    seconde = t1.seconde + t2.seconde
    minute = t1.minute + t2.minute
    heure = t1.heure + t2.heure
    if seconde >= 60
        seconde -= 60
        minute += 1
    end
    if minute >= 60
        minute -= 60
        heure += 1
    end
    MyTime(heure, minute, seconde)
end
```

Bien que cette fonction soit correcte, elle commence à s'allonger significativement. Ultérieurement (section 16.4), nous considérerons une version plus courte.

16.3 Modificateurs

Il est parfois utile pour une fonction de modifier les objets qu'elle reçoit en paramètre. Dans ce cas, les modifications sont visibles pour l'appelant. Les fonctions qui procèdent de cette manière sont appelées des *modificateurs*.

La fonction `increment!` qui ajoute un nombre donné de secondes à un objet `MyTime` peut être écrite naturellement comme un modificateur. Voici une ébauche :

```
function increment!(time, secondes)
    time.seconde += secondes
    if time.seconde >= 60
        time.seconde -= 60
        time.minute += 1
    end
    if time.minute >= 60
        time.minute -= 60
        time.heure += 1
    end
end
```

La première ligne effectue l'opération de base ; le reste traite des cas spéciaux que nous avons vus auparavant.

Cette fonction est-elle correcte ? Que se passe-t-il si `secondes` est bien supérieur à 60 ?

Dans ce cas, il ne suffit pas d'effectuer l'opération une seule fois. Nous devons poursuivre jusqu'à ce que `time.seconde` soit inférieur à soixante. Une solution consiste à remplacer les déclarations `if` par des déclarations `while`. Cela rendrait la fonction correcte, mais peu efficace.

Tout ce qui peut être fait avec des modificateurs peut également l'être avec des fonctions pures. En fait, certains langages de programmation n'autorisent que des fonctions pures. Il est prouvé que les programmes qui utilisent des fonctions pures sont plus rapides à développer et moins sujets aux erreurs que ceux recourant aux modificateurs. Cependant, les modificateurs sont parfois pratiques et les programmes fonctionnels ont tendance à être moins efficaces à l'exécution.

En général, il est recommandé d'écrire des fonctions pures chaque fois que cela est raisonnable et de ne recourir aux modificateurs que s'il existe un avantage incontestable. Cette approche pourrait être appelée : *style de programmation fonctionnelle*.

16.3.1 Exercice

Écrivez une version correcte de la fonction `increment!` qui ne contienne aucune boucle.

16.3.2 Exercice

Écrivez une version pure de la fonction `increment`, qui crée et retourne un nouvel objet `MyTime` plutôt que de modifier le paramètre.

16.4 Prototypage ou planification ?

Le plan de développement exposé ci-dessus est dénommé « prototype et correctifs » (*prototype and patches*). Pour chaque fonction, nous avons écrit un prototype qui a effectué un calcul de base. Ensuite après l’avoir testé, nous en avons corrigé les erreurs graduellement.

Cette approche peut être efficace notamment lorsque le programmeur n’a pas encore une compréhension approfondie du problème. Ceci étant, les corrections incrémentales peuvent conduire à un code inutilement compliqué –puisque’il traite de nombreux cas particuliers– et peu fiable –étant donné qu’il est difficile de connaître toutes les erreurs traitées.

Une autre option consiste en un développement planifié dans lequel une compréhension de haut niveau du problème peut rendre la programmation nettement plus aisée. Dans ce cas, l’idée est qu’un objet `Time` est en réalité un nombre composé de trois nombres en base 60 (voir le système sexagésimal).

Lorsque nous avons écrit `addtime` and `increment!`, nous faisons effectivement de l’addition en base 60, c’est pourquoi nous devons passer d’une colonne à l’autre.

Cette observation suggère une autre approche de l’ensemble du problème : nous pouvons convertir les objets `MyTime` en nombres entiers et profiter du fait que l’ordinateur est capable de faire de l’arithmétique des nombres entiers.

Voici une fonction qui convertit les objets `MyTime` en nombres entiers :

```
function timetoint(time)
  minutes = time.heure * 60 + time.minute
  secondes = minutes * 60 + time.seconde
end
```

Et voici une fonction qui convertit un entier en un objet de type `MyTime` (rappelons que `divrem` divise le premier argument par le second et retourne le quotient ainsi que le reste sous forme de tuple) :

```
function inttotime(secondes)
    (minutes, seconde) = divrem(secondes, 60)
    heure, minute = divrem(minutes, 60)
    MyTime(heure, minute, seconde)
end
```

Il faudra peut-être réfléchir un peu et faire quelques tests pour se convaincre que ces fonctions sont correctes. Une façon de les tester est de vérifier que `timetoint(inttotime(x)) == x` pour de nombreuses valeurs de `x`. Ceci constitue un exemple de contrôle de cohérence.

Une fois convaincu que ces fonctions sont correctes, il devient possible de les utiliser afin de réécrire `addtime` :

```
function addtime(t1, t2)
    secondes = timetoint(t1) + timetoint(t2)
    inttotime(secondes)
end
```

Cette version est plus courte que l'originale et plus facile à vérifier.

Il est plus difficile de passer de la base 60 à la base 10 (et inversement) que de jongler avec les heures, minutes et secondes. La conversion de base est plus abstraite. La manipulation de valeurs heures/minutes/secondes est plus spontanée.

Cependant, si nous pouvions traiter ces valeurs comme des nombres en base 60¹ et faisons l'investissement d'écrire les fonctions de conversion (`timetoint` et `inttotime`), nous obtiendrions un programme plus court, plus facile à lire et à déboguer et... plus fiable.

Il est également plus aisé d'ajouter des fonctions par la suite. Par exemple, imaginons qu'il faille soustraire deux `MyTime` pour trouver la durée qui les sépare. Une approche naïve serait de mettre en œuvre la soustraction par emprunt. L'utilisation des fonctions de conversion serait plus facile en base 60 et aurait plus de chances d'être correcte.²

16.4.1 Exercice

Réécrivez `increment` ! en utilisant `timetoint` et `inttotime`.

1. comme le faisaient les astronomes et géographes grecs.

2. Ironiquement, le fait de poser un problème sous une forme plus générale (ce qui demande un investissement) le rend parfois plus facile à résoudre du fait qu'il y a moins de cas particuliers et moins de possibilités d'erreur.

16.5 Débogage

Un objet `MyTime` est bien formé si les valeurs `minute` et `seconde` sont comprises entre 0 et 60 (y compris 0 mais pas 60) et si l'attribut `heure` est positif. `heure` et `minute` devraient être des valeurs entières, mais nous pourrions permettre à `seconde` d'être exprimée avec une partie fractionnaire.

De telles exigences sont appelées des *invariants* car elles doivent toujours être vraies. En d'autres termes, si elles ne sont pas vraies, c'est que quelque chose dys-fonctionne.

L'écriture d'un code pour vérifier les invariants peut aider à détecter les erreurs et à en trouver les causes. Par exemple, nous pourrions disposer d'une fonction comme `isvalidtime` qui prend un objet `MyTime` et retourne `false` si ce dernier viole un invariant :

```
function isvalidtime(time)
  if time.heure < 0 || time.minute < 0 || time.seconde < 0
    return false
  end
  if time.minute >= 60 || time.seconde >= 60
    return false
  end
  true
end
```

Au début de chaque fonction, il est pertinent de vérifier les arguments pour s'assurer qu'ils sont valides :

```
function addtime(t1, t2)
  if !isvalidtime(t1) || !isvalidtime(t2)
    error("objet MyTime non valide dans add_time")
  end
  seconds = timetoint(t1) + timetoint(t2)
  inttotime(seconds)
end
```

Une macro `@assert` peut être utilisée, telle qu'elle vérifie un invariant donné et émet une exception en cas d'échec :

```
function addtime(t1, t2)
  @assert(isvalidtime(t1) && isvalidtime(t2), "objet MyTime non valide dans
add_time")
  seconds = timetoint(t1) + timetoint(t2)
  inttotime(seconds)
end
```

Les macros `@assert` sont utiles car elles permettent de distinguer le code qui traite des conditions normales de celui qui vérifie les erreurs.

16.6 Glossaire

prototype et correctifs plan de développement qui implique la rédaction d'une ébauche de programme, le test et la correction des erreurs au fur et à mesure que ces dernières sont détectées,

planification plan de développement qui implique une compréhension de haut niveau d'un problème et une planification plus poussée que le développement progressif ou le développement de prototypes/correctifs,

fonction pure fonction qui ne modifie aucun des objets qu'elle reçoit comme arguments. La plupart des fonctions pures possèdent une valeur de retour,

modificateur fonction qui modifie un ou plusieurs des objets qu'elle reçoit comme arguments. La plupart des modificateurs sont vides (ou nuls), c'est-à-dire qu'ils ne retournent rien,

style de programmation fonctionnelle style de conception de programme dans lequel la majorité des fonctions sont pures,

invariant paramètre ou attribut qui ne devrait jamais changer pendant l'exécution d'un programme.

16.7 Exercices

16.7.1 Exercice

Écrivez une fonction appelée `multitime` qui prend un objet `MyTime` ainsi qu'un nombre et qui retourne un nouvel objet `MyTime` contenant le produit du `MyTime` original et du nombre.

Utilisez ensuite `multitime` pour écrire une fonction qui prend un objet `MyTime` représentant le temps d'arrivée dans une course ainsi qu'un nombre figurant la distance. Cette fonction doit retourner un objet `MyTime` qui donne l'allure moyenne (durée par kilomètre parcouru).

16.7.2 Exercice

Julia fournit des objets « temps » similaires à ceux de `MyTime` vu dans ce chapitre. Toutefois, ils offrent un riche ensemble de fonctions et d'opérateurs. Lisez la documen-

tation à l'adresse [Dates](#).

1. Écrivez un programme qui recueille la date du jour et affiche le jour de la semaine.
2. Écrivez un programme qui accepte une date d'anniversaire en entrée et affiche l'âge de l'utilisateur ainsi que le nombre de jours, d'heures, de minutes et de secondes jusqu'à son prochain anniversaire.
3. Pour deux personnes nées à des jours différents, il existe un jour où l'une d'elles est deux fois plus âgée que l'autre. Écrivez un programme qui prend deux dates anniversaires et calcule [DoubleJour](#).
4. Défi : écrivez une version plus générale qui calcule le jour où une personne est n fois plus âgée qu'une autre.

Chapitre 17

Dispatch multiple

Julia offre la possibilité d'écrire du code capable de fonctionner sur différents types, ce qui est connu comme de la *programmation générique*. Dans ce chapitre, nous abordons l'utilisation des déclarations de type en Julia et nous présentons des méthodes permettant d'implémenter différents comportements pour une fonction selon les types associés à ses arguments. Il s'agit là du dispatch multiple (*multiple dispatch*).

17.1 Déclarations de types

L'opérateur `::` associe des *annotations de type* aux expressions et aux variables :

```
julia> (1 + 2) :: Float64
ERROR: TypeError: in typeassert, expected Float64, got Int64
julia> (1 + 2) :: Int64
3
```

Cela permet de confirmer qu'un programme fonctionne de manière adéquate.

Par ailleurs, l'opérateur `::` peut être ajouté dans le membre de gauche d'une affectation ou dans le cadre d'une déclaration.

```
julia> function returnfloat()
    x::Float64 = 100
    x
end
```

```
returnfloat (generic function with 1 method)
julia> x = returnfloat()
100.0
julia> typeof(x)
Float64
```

En l'occurrence, la variable `x` est toujours de type `Float64` et la valeur est convertie en virgule flottante si nécessaire.

Une annotation de type peut également être jointe à l'en-tête de la définition d'une fonction :

```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    sin(x)/(x)
end
```

La valeur de retour de `sinc` est toujours convertie en type `Float64`.

Par défaut, lorsque les types ne sont pas précisés, Julia considère les valeurs comme étant de type quelconque (`Any`).

17.2 Méthodes

Dans la figure 16.1.1, nous avons défini une structure appelée `MyTime` et dans la section 16.1, nous avons écrit une fonction appelée `printtime` :

```
using Printf

struct MyTime
    heure :: Int64
    minute :: Int64
    seconde :: Int64
end

function printtime(time)
    @printf("%02d:%02d:%02d", time.heure, time.minute, time.seconde)
end
```

Comme on peut le constater, les déclarations de type peuvent (et, pour des raisons de performance, *devraient*) être associées aux champs d'une définition de structure.

Pour appeler la fonction `printtime`, il est nécessaire de passer un objet `MyTime` en argument :


```
julia> start = MyTime(9, 55, 0)
MyTime(9, 55, 0)
julia> printtime(start)
09:55:00
```

Pour ajouter à la fonction `printtime` une *méthode* qui n'accepte comme seul argument qu'un objet `MyTime`, il suffit d'ajouter `::` suivi de `MyTime` à l'argument `time` dans la définition de la fonction :

```
function printtime(time::MyTime)
    @printf("%02d:%02d:%02d", time.heure, time.minute, time.seconde)
end
```

Une méthode est une définition de fonction avec une *signature* spécifique : `printtime` a un argument de type `MyTime`.

Appeler la fonction `printtime` avec un objet `MyTime` produit le même résultat :

```
julia> printtime(start)
09:55:00
```

À présent, nous pouvons redéfinir la première méthode sans l'annotation de type `::`, ce qui permet l'usage d'un argument de type quelconque :

```
function printtime(time)
    println("Je ne sais pas comment afficher l'argument time.")
end
```

Si nous appelons la fonction `printtime` avec un objet différent de `MyTime`, nous obtenons :

```
julia> printtime(150)
Je ne sais pas comment afficher l'argument time.
```

17.2.1 Exercice

Réécrivez `time` et `inttotime` pour spécifier leur argument (voir la section 16.4).

17.3 Exemples supplémentaires

Voici une version de la fonction `increment` (voir la section 16.3) réécrite pour spécifier ses arguments :

```
function increment(time::MyTime, secondes::Int64)
    secondes += timetoint(time)
    inttotime(secondes)
end
```

À présent, il s'agit d'une fonction pure, et non plus d'un modificateur. Voici comment cette fonction `increment` peut être invoquée :

```
julia> start = MyTime(9, 45, 0)
MyTime(9, 45, 0)
julia> increment(start, 1337)
MyTime(10, 7, 17)
```

Si les arguments apparaissent dans le mauvais ordre, Julia retourne une erreur :

```
julia> increment(1337, start)
ERROR: MethodError: no method matching increment(::Int64, ::MyTime)
```

La signature de la méthode est `increment(time::MyTime, seconds::Int64)` et non `increment(seconds::Int64, time::MyTime)`.

Réécrire `isafter` pour agir uniquement sur les objets `MyTime` est aisé :

```
function isafter(t1::MyTime, t2::MyTime)
    (t1.heure, t1.minute, t1.seconde) > (t2.heure, t2.minute, t2.seconde)
end
```

Au fait, les arguments optionnels sont implémentés comme syntaxe pour les définitions de méthodes multiples. Par exemple, cette définition :

```
function f(a=1, b=2)
    a + 2b
end
```

se traduit par les trois méthodes suivantes :

```
f(a, b) = a + 2b
f(a) = f(a, 2)
f() = f(1, 2)
```

En Julia, ces expressions sont des définitions valides de méthode. Il s'agit d'une notation abrégée pour la définition des fonctions/méthodes.

17.4 Constructeurs

Un *constructeur* est une fonction spéciale qui est appelée pour créer un objet. Les méthodes de constructeur par défaut de `MyTime` ont les signatures suivantes :

```
MyTime(heure, minute, seconde)
MyTime(heure::Int64, minute::Int64, seconde::Int64)
```

Nous pouvons également ajouter nos propres méthodes de construction externes :

```
function MyTime(time::MyTime)
    MyTime(time.heure, time.minute, time.seconde)
end
```

Cette dernière méthode est appelée *constructeur de copie* car le nouvel objet `MyTime` est une copie de son argument.

Pour forcer l'usage des invariants, il est nécessaire de recourir à la méthode des *constructeurs internes* :

```
struct MyTime
    heure :: Int64
    minute :: Int64
    seconde :: Int64
    function MyTime(heure::Int64=0, minute::Int64=0, seconde::Int64=0)
        @assert(0 ≤ minute < 60, "Minute n'est pas entre 0 et 60.")
        @assert(0 ≤ seconde < 60, "Seconde n'est pas entre 0 et 60.")
        new(heure, minute, seconde)
    end
end
```

La structure `MyTime` dispose maintenant de 4 méthodes à constructeurs internes :

```
MyTime()
MyTime(heure::Int64)
MyTime(heure::Int64, minute::Int64)
MyTime(heure::Int64, minute::Int64, seconde::Int64)
```

Une méthode à constructeur interne est toujours définie à l'intérieur du bloc d'une déclaration de type. Elle a accès à une fonction spéciale appelée `new` qui crée des objets du type nouvellement déclaré.

Avertissement. Le constructeur par défaut n'est pas disponible si un constructeur interne est défini. Il faut écrire explicitement tous les constructeurs internes dont on a besoin.

Une deuxième méthode avec la fonction locale `new` sans arguments existe :

```
mutable struct MyTime
    heure :: Int
    minute :: Int
    seconde :: Int
    function MyTime(heure::Int64=0, minute::Int64=0, seconde::Int64=0)
        @assert(0 ≤ minute < 60, "Les minutes sont comprises entre 0 et 60.")
        @assert(0 ≤ seconde < 60, "Les secondes sont comprises entre 0 et 60.")
        time = new()
        time.heure = heure
        time.minute = minute
        time.seconde = seconde
        time
    end
end
```

Cela permet de construire des structures de données récursives, c'est-à-dire une structure dont un des champs est la structure elle-même. Dans ce cas, la structure doit être non persistante puisque ses champs sont modifiés après l'instanciation.

17.5 show

`show` est une fonction spéciale qui retourne une représentation en chaîne d'un objet. Par exemple, voici une méthode `show` pour les objets `MyTime` :

```
using Printf
function Base.show(io::IO, time::MyTime)
    @printf(io, "%02d:%02d:%02d", time.heure, time.minute, time.seconde)
end
```

Le préfixe `Base` est nécessaire parce que nous voulons ajouter une nouvelle méthode à la fonction `Base.show`.

Lorsqu'un objet est affiché, Julia invoque la fonction `show` :

```
julia> time = MyTime(9, 45)
09:45:00
```

Lorsque nous écrivons un nouveau type composite, il est pertinent :

- de commencer presque toujours par écrire un constructeur extérieur (ceci facilite l'instanciation des objets) et,
- d'utiliser `show` (qui est utile pour le débogage).

17.5.1 Exercice

Écrivez une méthode de construction extérieure pour la classe `Point` qui prend `x` ainsi que `y` comme paramètres optionnels et les affecte aux champs correspondants.

17.6 Surcharge d'opérateurs

En définissant les méthodes des opérateurs, nous pouvons spécifier leur comportement sur des types définis par le programmeur. Par exemple, si nous définissons une méthode nommée `+` avec deux arguments `MyTime`, nous pouvons utiliser l'opérateur `+` capable d'additionner des objets `MyTime`.

Voici à quoi pourrait ressembler la définition :

```
import Base.+  
  
function +(t1::MyTime, t2::MyTime)  
    secondes = timetoint(t1) + timetoint(t2)  
    inttotime(secondes)  
end
```

La déclaration d'importation ajoute l'opérateur `+` au champ d'application local afin que des méthodes puissent être ajoutées.

Voici comment nous pouvons l'utiliser :

```
julia> start = MyTime(9, 55)  
09:55:00  
julia> duration = MyTime(2, 11, 0)  
02:11:00  
julia> start + duration  
12:06:00
```

Lorsque l'opérateur `+` est appliqué aux objets `MyTime`, Julia invoque la méthode nouvellement ajoutée. Lorsque le REPL affiche le résultat, Julia invoque `show`. Il se passe donc beaucoup de choses en coulisses.

L'ajout au comportement d'un opérateur pour qu'il fonctionne avec des types définis par le programmeur s'appelle la *surcharge de l'opérateur*.

17.7 Dispatch multiple

Dans la section précédente, nous avons additionné deux objets `MyTime`. Cependant, il est également possible d'ajouter un entier à un objet `MyTime` :

```
function +(time::MyTime, secondes::Int64)
    increment(time, secondes)
end
```

Voici un exemple qui utilise l'opérateur `+` avec un objet `MyTime` et un entier :

```
julia> start = MyTime(9, 55)
09:55 ::00
julia> start + 1337
10:07:17
```

L'addition étant un opérateur commutatif, il faut donc ajouter une méthode complémentaire.

```
function +(secondes::Int64, time::MyTime)
    time + secondes
end
```

Nous obtenons alors le même résultat :

```
julia> 1337 + start
10:07:17
```

Le choix de la méthode à exécuter lorsqu'une fonction est appliquée s'appelle un *dispatch*. Julia permet au processus de *dispatching* de choisir la méthode d'une fonction à appeler selon le nombre d'arguments passés et les types de chacun des arguments de la fonction. L'utilisation de tous les arguments d'une fonction pour laisser le choix de la méthode à invoquer est connue sous le nom de *dispatch multiple*¹.

17.7.1 Exercice

Écrivez des méthodes `+` pour les objets point (voir le chapitre 15) :

1. Si les deux opérandes sont des objets point, la méthode doit retourner un nouvel objet point dont la coordonnée `x` est la somme des coordonnées `x` des opérandes. De même pour les coordonnées `y`.
2. Si le premier ou le second opérande est un tuple, la méthode doit ajouter le premier élément du tuple à la coordonnée `x` et le second élément à la coordonnée `y`, et retourner un nouvel objet point avec le résultat.

1. Parfois appelé également *multiméthode*.

17.8 Programmation générique (généricité)

Lorsqu'il est nécessaire, le dispatch multiple est d'une grande utilité. Ce n'est malheureusement pas toujours le cas. Souvent, il est possible de l'éviter en écrivant des fonctions qui se comportent correctement pour des arguments de types différents.

De nombreuses fonctions que nous avons écrites pour des chaînes de caractères fonctionnent également pour d'autres types de séquences. Par exemple, dans la section 11.2, nous avons utilisé la fonction `histogram` pour compter le nombre d'occurrences de chaque lettre apparaissant dans un mot.

```
function histogram(s)
    d = Dict{Char, Int}()
    for c in s
        if c ∉ keys(d)
            d[c] = 1
        else
            d[c] += 1
        end
    end
    d
end
```

Cette fonction agit également sur les tableaux, les tuples et même les dictionnaires à condition que les éléments de `s` soient hachables, afin qu'ils puissent être utilisés comme clés dans le dictionnaire `d`.

```
julia> t = ("poêle", "œuf", "poêle", "poêle", "jambon", "poêle")
("poêle", "œuf", "poêle", "poêle", "poêle", "poêle")
julia> histogram(t)
Dict{Any,Any} with 3 entries:
  "jambon" => 1
  "poêle"  => 4
  "œuf"    => 1
```

Les fonctions capables de manipuler plusieurs types sont dites polymorphiques. Le *polymorphisme* contribue à la réutilisation du code.

Par exemple, la fonction interne `sum`, qui ajoute les éléments d'une séquence, remplit son rôle tant que les éléments de la séquence supportent l'addition.

Comme une méthode `+` est fournie pour les objets `MyTime`, ceux-ci fonctionnent avec `sum` :

```
julia> t1 = MyTime(1, 7, 2)
01:07:02
julia> t2 = MyTime(1, 5, 8)
01:05:08
julia> t3 = MyTime(1, 5, 0)
01:05:00
julia> sum((t1, t2, t3))
03:17:10
```

En général, si toutes les opérations à l'intérieur d'une fonction remplissent leur rôle avec un type donné, la fonction fera de même avec ce type.

Le meilleur type de polymorphisme est le type involontaire, où vous découvrez qu'une fonction que vous avez déjà écrite peut être appliquée à un type jusque-là imprévu.

17.9 Interface et implémentation

Un des objectifs du dispatch multiple est de rendre les logiciels plus faciles à gérer et à entretenir. Cela signifie qu'il est possible d'une part de continuer à faire fonctionner le programme lorsque d'autres parties du système changent et, de l'autre, de modifier le programme pour répondre à de nouvelles exigences.

Un principe de conception qui contribue à atteindre cet objectif consiste à garder les interfaces séparées des implémentations. Autrement dit, les méthodes ayant un argument annoté avec un type ne doivent pas dépendre de la façon dont les champs de ce type sont représentés.

Par exemple, dans ce chapitre, nous avons développé une structure qui représente un moment de la journée. Les méthodes ayant un argument annoté avec ce type comprennent `timetoint`, `isafter` et `+`.

Nous pourrions implémenter ces méthodes de plusieurs manières. Les détails de l'implémentation dépendent de la façon dont nous représentons `MyTime`. Dans ce chapitre, les champs d'un objet `MyTime` étaient l'heure, la minute et la seconde.

Comme autre option, nous aurions pu remplacer ces champs par un seul entier représentant le nombre de secondes depuis minuit. Cette implémentation rendrait certaines fonctions, comme `isafter`, plus faciles à écrire. En revanche, d'autres seraient plus difficiles à développer.

Après avoir déployé un nouveau type, vous pourriez découvrir une meilleure implémentation. Si d'autres parties du programme utilisent votre type, cela peut être chronophage et provoquer des erreurs par changement de l'interface.

Cependant, si vous avez conçu l'interface avec soin, vous pouvez modifier l'implémentation sans changer l'interface. La conséquence immédiate est que les autres parties du programme ne doivent pas être modifiées.

17.10 Débogage

Appeler une fonction avec les bons arguments peut être ardu lorsque plus d'une méthode est spécifiée pour la fonction. Julia permet d'effectuer une introspection des signatures associées aux méthodes d'une fonction.

Pour savoir quelles méthodes sont disponibles pour une fonction donnée, la fonction `methods` vient à point :

```
julia> methods(printtime)
# 2 methods for generic function "printtime":
[1] printtime(time::MyTime) in Main at REPL[3]:2
[2] printtime(time) in Main at REPL[4] ::2
```

17.11 Glossaire

annotation de type l'opérateur `::` suivi d'un type indiquant qu'une expression ou une variable est de ce type,

méthode définition d'un comportement possible d'une fonction,

dispatch choix de la méthode à mettre en œuvre lorsqu'une fonction est exécutée,

signature c'est le nombre et le type des arguments d'une méthode permettant au dispatch de sélectionner la méthode la plus spécifique d'une fonction lors de l'appel de fonction,

constructeur externe constructeur défini en dehors de la définition de type pour spécifier les méthodes utiles à la création d'un objet,

constructeur interne constructeur défini à l'intérieur de la définition de type pour imposer des invariants ou pour construire des objets récursifs,

constructeur par défaut constructeur interne disponible lorsqu'aucun constructeur interne défini par le programmeur n'est fourni,

copie du constructeur méthode de construction extérieure d'un type avec comme seul argument un objet du type. Cette méthode crée un nouvel objet, copie de l'argument,

surcharge d'opérateur extension du comportement d'un opérateur (comme +) pour que celui fonctionne avec un type défini par le programmeur,

dispatch multiple (multiméthode) dispatch basé sur l'ensemble des arguments d'une fonction,

programmation générique (généricité) rédaction d'un code susceptible d'opérer avec plusieurs types.

17.12 Exercices

17.12.1 Exercice

Modifiez les champs de `MyTime` pour qu'il s'agisse d'un seul nombre entier représentant les secondes depuis minuit. Ensuite, modifiez les méthodes définies dans ce chapitre pour qu'elles fonctionnent avec la nouvelle implémentation.

17.12.2 Exercice

Rédigez une définition pour un type nommé `Kangaroo` avec un champ nommé `putinpocket` du type `Array` et les méthodes suivantes :

1. Un constructeur qui initialise `pouchcontents` dans un tableau vide.
2. Une méthode nommée `putinpouch` qui prend un objet `Kangaroo` et un objet de n'importe quel type et l'ajoute à `pouchcontents`.
3. Une méthode `show` qui retourne une représentation en chaîne de caractères de l'objet `Kangaroo` et du contenu de la poche.

Testez votre code en créant deux objets `Kangaroo`, en les affectant à des variables nommées `kanga` et `roo`, puis en ajoutant `roo` au contenu de la poche de `kanga`.

Chapitre 18

Sous-typage

Dans le chapitre précédent, nous avons présenté le dispatch multiple et les méthodes polymorphes. En ne précisant pas le type d'arguments, on obtient une méthode qui peut être appelée avec des arguments de tout type. La spécification d'un sous-ensemble de types autorisés dans la signature d'une méthode est l'étape suivante logique.

Dans ce chapitre, nous explicitons la notion de sous-typage en utilisant des types qui représentent des cartes à jouer, des jeux de cartes et des « mains » au poker.

Si vous ne jouez pas au poker, vous pouvez vous informer sur le lien [Wikipédia: Poker](#). Cependant, ce n'est pas obligatoire ; tout au long des exercices, nous irons à l'essentiel.

18.1 Cartes

Dans un paquet, il y a cinquante-deux cartes, chacune appartenant à une des quatre couleurs et à un des treize rangs (ou hauteurs, voir l'exercice 18.13.3). Au poker, les couleurs sont pique (♠), cœur (♥), carreau (♦) et trèfle (♣). Les rangs sont : as (A), 2, 3, 4, 5, 6, 7, 8, 9, 10, valet (J^1), dame (Q) et roi (K). Selon le jeu auquel on participe, un as peut être supérieur au roi ou inférieur à 2.

Si nous souhaitons définir un nouvel objet pour représenter une carte à jouer, il est évident que les attributs doivent être le rang et la couleur. En revanche, le type de ces attributs n'est pas aussi intuitif à déterminer. Une option consiste à utiliser des chaînes de caractères contenant des mots comme « pique » pour les couleurs et « dame »

1. J : Jack, Q : Queen, K : King

pour les rangs. Un problème relatif à cette mise en œuvre provient de la difficulté de comparer les cartes en termes de préséance de rang ou de couleur.

Une autre option consiste à utiliser des nombres entiers pour coder les rangs et les couleurs. Dans ce contexte, « coder » signifie que nous allons définir une correspondance entre des nombres et les couleurs ainsi qu'entre des nombres et les rangs. Ce type de codage n'est pas secret, autrement, il s'agirait d'un chiffrement.

Par exemple, ce tableau illustre la correspondance « couleurs \mapsto nombres entiers » :

```
— ♠  $\mapsto$  4
— ♥  $\mapsto$  3
— ♦  $\mapsto$  2
— ♣  $\mapsto$  1
```

Cette correspondance permet de comparer facilement les cartes. La hiérarchie des couleurs correspond à la hiérarchie des nombres. Nous procédons de la même manière pour les rangs avec 13 nombres (13 \mapsto roi, 12 \mapsto dame, etc.).

Le symbole \mapsto est employé pour indiquer clairement que ces correspondances ne font pas partie du programme Julia. Elles relèvent de la conception du programme, sans toutefois apparaître explicitement dans le code.

La définition de la struct pour `Carte` se présente de cette manière :

```
struct Carte
    couleur :: Int64
    rang :: Int64
    function Carte(couleur::Int64, rang::Int64)
        @assert(1 ≤ couleur ≤ 4, "la couleur n'est pas entre 1 et 4")
        @assert(1 ≤ rang ≤ 13, "le rang n'est pas entre 1 et 13")
        new(couleur, rang)
    end
end
```

Pour créer une carte, nous appelons `Carte` avec la couleur et le rang souhaités :

```
julia> dame_de_carreau = Carte(2, 12)
Carte(2, 12)
```

18.2 Variables globales

Afin d'afficher les objets `Carte` de sorte que tout le monde puisse les lire facilement, il faut établir une correspondance entre les couleurs et leurs entiers ainsi qu'une correspondance entre les rangs et leurs entiers. Une manière naturelle de procéder consiste à

utiliser deux tableaux de chaînes de caractères ² :

```
const noms_couleurs = ["♣", "♦", "♥", "♠"]
const noms_rangs = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "V", "D", "R"]
```

Les variables `noms_couleurs` et `noms_rangs` sont globales. La déclaration `const` signifie que la variable ne peut être attribuée qu'une seule fois. Cela résout le problème de la performance des variables globales.

Dès lors, nous pouvons mettre en œuvre une méthode `show` idoine :

```
function Base.show(io::IO, carte::Carte)
    print(io, noms_rangs[carte.rang], noms_couleurs[carte.couleur])
end
```

L'expression `noms_rangs[carte.rang]` signifie « utiliser le champ `rang` de l'objet `carte` comme indice dans le tableau `noms_rangs`, et sélectionner la chaîne appropriée ».

Avec les méthodes dont nous disposons à ce stade, nous pouvons créer et afficher des cartes :

```
julia> Carte(3, 11)
♥♥
```

18.3 Comparaison de cartes

Pour les types internes, il existe des opérateurs relationnels (`<`, `>`, `==`, etc.) qui comparent les valeurs et déterminent quand l'une est supérieure, inférieure ou égale à l'autre. Pour les types définis par le programmeur, nous pouvons remplacer le comportement des opérateurs intégrés en fournissant une méthode nommée `<`.

La préséance correcte des cartes n'est pas évidente. Par exemple, du 3 de trèfle ou du 2 de carreau qui l'emporte ? L'un a un rang plus élevé, mais l'autre a une couleur plus élevée. Pour comparer les cartes, il faut décider si c'est le rang ou la couleur qui l'emporte.

La réponse peut dépendre du jeu. Pour simplifier, nous supposons que la couleur a priorité sur le rang. De la sorte, toutes les piques l'emportent sur toutes les carreaux, etc. Ainsi, dans l'exemple cité au début de cette section, $3♣ < 2♦$.

Ceci fixé, nous pouvons écrire `<` :

```
import Base.<

function <(c1::Carte, c2::Carte)
    (c1.couleur, c1.rang) < (c2.couleur, c2.rang)
end
```

2. Pour `♣` : `\:clubs : TAB` ; pour `♦` : `\:diamonds : TAB` ; pour `♥` : `\:hearts : TAB` ; pour `♠` : `\:spades : TAB`

18.3.1 Exercice

Écrivez une méthode `<` pour les objets `MyTime`. Vous pouvez utiliser la comparaison de tuples mais vous pourriez aussi considérer la comparaison d'entiers.

18.4 Tests unitaires

Les *tests unitaires* (T.U.) permettent de vérifier l'exactitude d'un code en comparant les résultats effectifs de ce dernier à ce qu'on en attend. Cela peut être utile pour s'assurer que le code est toujours correct après modifications. Cette technique est aussi une manière de prédéfinir le comportement correct du code pendant la phase de développement.

Des tests unitaires simples peuvent être effectués avec les macros `@test` :

```
julia> using Test
julia> @test Carte(1, 4) < Carte(2, 4)
Test Passed
julia> @test Carte(1, 3) < Carte(1, 4)
Test Passed
```

`@test` retourne « `Test Passed` » si l'expression est `true`, « `Test Failed` » si elle est `false` et « `Error Result` » si elle ne peut pas être évaluée.

18.5 Paquets de cartes

Maintenant que nous avons les cartes, l'étape suivante consiste à créer des paquets de cartes. Il est naturel que chaque paquet contienne un tableau de cartes comme attribut.

Voici une structure composite de `Paquet`. Le constructeur crée les cartes des champs et produit un jeu de 52 cartes :

```
struct Paquet
    cartes :: Array{Carte, 1}
end

function Paquet()
    paquet = Paquet(Carte[])
    for couleur in 1:4
        for rang in 1:13
            push!(paquet.cartes, Carte(couleur, rang))
        end
    end
end
```

```

end
paquet
end

```

La manière la plus simple de constituer un paquet consiste à utiliser deux boucles. La boucle extérieure permet de créer les couleurs (de 1 à 4) et la boucle intérieure permet de produire les rangs (de 1 à 13). Chaque itération crée une nouvelle carte qui entre dans `paquet.cartes`. Ainsi, la première carte produite est l'as de trèfle, puis le 2 de trèfle, etc., jusqu'à la dernière carte, le roi de pique.

Voici la méthode `show` pour `Paquet` :

```

function Base.show(io::IO, paquet::Paquet)
    for carte in paquet.cartes
        print(io, carte, " ")
    end
    println()
end

```

Voici le résultat :

```

julia> Paquet()
A♠ 2♠ 3♠ 4♠ 5♠ 6♠ 7♠ 8♠ 9♠ 10♠ V♠ D♠ R♠ A♦ 2♦ 3♦ 4♦ 5♦ 6♦ 7♦ 8♦ 9♦
10♦ V♦ D♦ R♦ A♥ 2♥ 3♥ 4♥ 5♥ 6♥ 7♥ 8♥ 9♥ 10♥ V♥ D♥ R♥ A♣ 2♣ 3♣ 4♣ 5♣ 6♣
7♣ 8♣ 9♣ 10♣ V♣ D♣ R♣

```

18.6 Ajouter, supprimer, mélanger et trier

Pour distribuer des cartes, il faudrait pouvoir utiliser une fonction qui retire une carte du jeu et l'affiche. La fonction `pop!` est utile à cet égard :

```

function Base.pop!(paquet::Paquet)
    pop!(paquet.cartes)
end

```

Vu son mode d'action (retrait du dernier élément d'une structure de données), `pop!` agit sur le bas de la pile.

Ajouter une carte peut se faire en utilisant la fonction `push!` :

```

function Base.push!(paquet::Paquet, carte::Carte)
    push!(paquet.cartes, carte)
    paquet
end

```

Une telle méthode qui exploite une autre méthode en n’effectuant que peu de traitement supplémentaire s’appelle du *placage*. La métaphore vient de la marqueterie, où un placage est une fine couche de bois noble collée à la surface d’une pièce de bois de moindre qualité afin d’en améliorer l’apparence.

Dans ce cas, `push!` est une méthode « légère » qui adapte une opération sur les tableaux au cas des paquets de cartes. Elle améliore l’interface de l’implémentation.

Comme autre exemple, pour battre les cartes, nous pouvons écrire une méthode appelée `shuffle!` en utilisant la fonction `Random.shuffle!` :

```
using Random

function Random.shuffle!(paquet::Paquet)
    shuffle!(paquet.cartes)
    paquet
end
```

18.6.1 Exercice

Écrivez une fonction appelée `sort!` qui utilise la fonction `sort!` pour trier les cartes dans un `Paquet`. `sortstring!` utilise la méthode `<` que nous avons définie pour déterminer l’ordre de préséance des cartes (voir la section 18.3).

18.7 Types abstraits et sous-typage

Pour jouer aux cartes, il faut créer une « main », c’est-à-dire un ensemble de cartes détenues par un joueur. Pour cela, il est nécessaire d’y associer un type. Or, une main est similaire à un jeu de cartes : les deux sont composés d’une collection de cartes et les deux recourent à des opérations comme l’ajout et le retrait de cartes.

Toutefois, il existe des différences. Certaines opérations spécifiques aux mains n’ont pas de sens pour un jeu de cartes complet. Par exemple, au poker, on peut comparer deux mains pour déterminer laquelle est gagnante. Au bridge, on peut calculer le score d’une main pour faire une offre.

Nous devons donc trouver un moyen de regrouper les *types spécifiques* lorsqu’ils sont apparentés. En Julia, et pour notre problème, la technique consiste à définir un *type abstrait* qui sert de parent à la fois pour le paquet de cartes complet (`Paquet`) et pour une main (`UneMain`³). Cette manière de procéder s’appelle le *sous-typage*.

Nommons le type abstrait `EnsembleDeCartes` :

3. En français, nous devons éviter l’usage d’une fonction personnelle `Main` du fait de l’existence de la fonction générale `Main`.


```
abstract type EnsembleDeCartes end
```

Un nouveau type abstrait est créé avec le mot-clé `abstract type`. Dans la déclaration que nous venons de faire, un type « parent » peut être facultativement spécifié en précisant (après le nom) le symbole `<` suivi lui-même du nom d'un type abstrait existant.

Lorsqu'aucun *supertype* n'est indiqué, Julia utilise le supertype par défaut `Any`, un type abstrait prédéfini. Tous les objets en sont des instances. Tous les types en sont des sous-types.

À présent, exprimons que `Paquet` est un sous-type descendant du type abstrait `EnsembleDeCartes` :

```
struct Paquet <: EnsembleDeCartes
    cartes :: Array{Carte, 1}
end

function Paquet()
    paquet = Paquet(Carte[])
    for couleur in 1:4
        for rang in 1:13
            push!(paquet.cartes, Carte(couleur, rang))
        end
    end
    paquet
end
```

L'opérateur `isa` vérifie si un objet est d'un type donné :

```
julia> paquet = Paquet();
julia> paquet isa EnsembleDeCartes
true
```

Une main est aussi un sous-type du type parent `EnsembleDeCartes` :

```
struct UneMain <: EnsembleDeCartes
    cartes :: Array{Carte, 1}
    label :: String
end

function UneMain(label::String="")
    UneMain(Carte[], label)
end
```

Au lieu de peupler une main avec 52 nouvelles cartes, le constructeur associé à `UneMain` initialise `cartes` sous la forme d'un tableau vide. Un argument optionnel peut être passé au constructeur, ce qui permet de donner une étiquette à `UneMain`.

```
julia> main = UneMain("nouvelle main")
UneMain(Carte[], "nouvelle main")
```

18.8 Types abstraits et fonctions

Ceci fait (section 18.7), nous pouvons écrire les opérations communes à [Paquet](#) et [UneMain](#) comme des fonctions ayant pour argument [EnsembleDeCartes](#) :

```
function Base.show(io::IO, edc::EnsembleDeCartes)
    for carte in edc.cartes
        print(io, carte, " ")
    end
end

function Base.pop!(edc::EnsembleDeCartes)
    pop!(edc.cartes)
end

function Base.push!(edc::EnsembleDeCartes, carte::Carte)
    push!(edc.cartes, carte)
    nothing
end
```

À présent, nous pouvons utiliser [pop!](#) et [push!](#) pour distribuer une carte :

```
julia> paquet = Paquet()
julia> shuffle!(paquet)
julia> carte = pop!(paquet)
julia> push!(main, carte)
```

L'étape suivante consiste naturellement à encapsuler ce code dans une fonction appelée [move!](#) :

```
function move!(edc1::EnsembleDeCartes, edc2::EnsembleDeCartes, n::Int)
    @assert 1 ≤ n ≤ length(edc1.cartes)
    for i in 1:n
        carte = pop!(edc1)
        push!(edc2, carte)
    end
    nothing
end
```

La fonction `move!` prend trois arguments : deux objets `EnsembleDeCartes` et le nombre de cartes à distribuer. Elle modifie les deux objets `EnsembleDeCartes` et renvoie `nothing`.

Dans certains jeux, les cartes sont déplacées d'une main à l'autre alors que dans d'autres jeux, elles sont échangées depuis une main vers le paquet. La fonction `move!` est donc utilisable pour ces opérations : `edc1` et `edc2` peuvent être soit de type `Paquet`, soit de type `UneMain`.

18.9 Diagrammes de types

Jusqu'à présent, nous avons vu des diagrammes de pile (qui résume l'état d'un programme) et des diagrammes d'objet (qui mettent en évidence les attributs d'un objet et leurs valeurs). Ces diagrammes représentent un instantané de l'exécution d'un programme. En conséquence, ils évoluent au fur et à mesure de l'exécution. Ils sont également très détaillés, voire trop pour certains usages.

Un diagramme de types est une esquisse plus abstraite de la structure d'un programme. Au lieu de dépeindre des objets individuels, il représente de manière synthétique les types et les relations qu'ils entretiennent (voir la figure 18.9.1).

Il existe plusieurs catégories de relation entre les types :

- les objets d'un type spécifique peuvent contenir des références à des objets d'un autre type. Par exemple, chaque `Rectangle` (voir la section 15.4) contient une référence à un `Point`. Dans le présent chapitre, chaque `Paquet` contient des références à un tableau de `Cartes`. Ce type de relation est appelé *HAS-A*, comme dans l'expression en anglais : « *a Rectangle has a Point* »,
- un type spécifique peut avoir un type abstrait comme supertype. Cette relation est appelée *IS-A*, comme dans la formulation en anglais : « *UneMain is a kind of EnsembleDeCartes* ».
- un type peut dépendre d'un autre dans la mesure où les objets d'un type prennent les objets du second type comme paramètres ou alors, utilisent les objets du second type dans le cadre d'un calcul. Ce type de relation est appelé une *dépendance*.

Dans la figure 18.9.1, les flèches à extrémité creuse représentent une relation *IS-A*. Dans le cas présent, elle indique qu'`Unemain` et `Paquet` ont comme supertype `EnsembleDeCartes`.

Les flèches à extrémité pleine représentent une relation *HAS-A*. Dans le cas présent, `Paquet` a des références aux objets `Carte`.

L'étoile (★) près de la pointe d'une flèche désigne une *multiplicité*. Elle indique que `Paquet` ou `UneMain` possèdent un certain nombre de `Cartes`. La multiplicité peut être

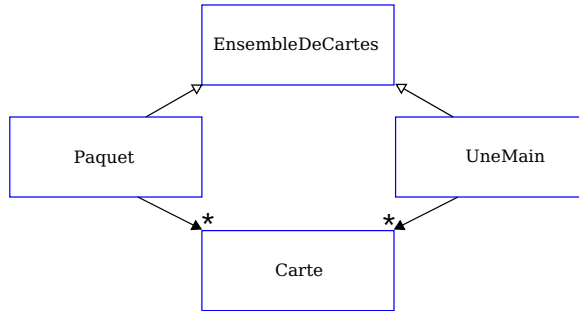


FIGURE 18.9.1 – Diagramme de type et relations entre types.

un simple nombre (comme 52), une fourchette (5 :7) ou une étoile (★). Ce dernier cas indique que Paquet peut avoir un nombre quelconque de Cartes.

Il n’y a pas de dépendances dans ce diagramme. Elles sont normalement indiquées par une flèche pointillée. Cependant, si les dépendances sont nombreuses, il est judicieux d’omettre leur représentation.

Un diagramme plus détaillé peut montrer qu’un Paquet contient un tableau de Cartes, mais les types internes à Julia comme les tableaux et les dictionnaires ne sont généralement pas repris dans les diagrammes de types.

18.10 Débogage

Le sous-typage peut rendre le débogage difficile. En effet, lorsqu’une fonction est appelée avec un objet comme argument, il peut être compliqué de déterminer quelle méthode est invoquée.

Supposons que nous écrivions une fonction qui manipule des objets `UneMain`. Idéalement, il faudrait qu’elle fonctionne avec toute sorte de type `UneMain+s`, comme `+PokerUneMain+s`, `+BridgeUneMain+s`, etc. Si nous invoquons une méthode comme `+sort!`, il se pourrait que nous travaillions effectivement avec celle définie pour le type abstrait `UneMain`. Toutefois, s’il existe une méthode `sort!` ayant comme argument un des sous-types, nous manipulerons cette version plutôt que celle définie pour le type abstrait `UneMain`. Ce comportement est généralement sain mais, *a priori*, il peut s’avérer déroutant.

```

function Base.sort!(main::UneMain)
    sort!(main.cartes)
end
  
```

Lorsqu'on n'est pas sûr de la manière dont l'exécution d'un programme procède, la solution la plus simple consiste à ajouter des instructions d'affichage au début des méthodes concernées. Si `shuffle!` imprime un message tel que « `Running shuffle! Paquet` », cela signifie que le programme est exécuté correctement.

Une meilleure manière de pratiquer consiste à utiliser la macro `@which` :

```
julia> @which sort!(main)
sort!(main::UneMain) in Main at REPL[5]:1
```

Donc, la méthode `sort!` associée à `main` a pour argument un objet de type `UneMain`.

À ce stade, voici une suggestion relative à la conception d'un programme. Lorsque vous passez outre une méthode, l'interface de la nouvelle méthode doit être la même que l'interface de l'ancienne. Elle devrait prendre les mêmes paramètres, retourner le même type et obéir aux mêmes conditions *a priori* et *a posteriori*. Si vous suivez cette règle, vous constaterez que toute fonction conçue pour utiliser une instance d'un supertype (comme `EnsembleDeCartes`), fonctionnera également avec des instances de ses sous-types (comme `Paquet` et `UneMain`).

Si vous enfrezignez cette règle, connue comme le « principe de substitution de Liskov », votre code s'effondrera comme un château de cartes.

La fonction `supertype` peut être utilisée pour trouver le supertype direct d'un type.

```
julia> supertype(Paquet)
EnsembleDeCartes
```

18.11 Encapsulation de données

Les chapitres 15 et 16 présentent un plan de développement qui pourrait s'appeler « conception orientée type ». Nous avons identifié les objets à traiter – comme `Point`, `Rectangle` et `MyTime` – et nous avons défini des structures pour les représenter. Dans chaque cas, il existait une correspondance évidente entre l'objet et une entité du monde réel (du moins, sa représentation mathématique).

Ceci étant, il est parfois moins facile de déterminer les objets dont nous avons besoin et comment ils doivent interagir. S'il en va ainsi, il faut un plan de développement mieux adapté que la conception orientée type. De la même manière que nous avons découvert les interfaces de fonctions par encapsulation et généralisation, nous allons découvrir les interfaces de types par *encapsulation de données*.

L'analyse de Markov (section 13.8) fournit un bon exemple. Supposons que vous téléchargez le code [sous ce lien](#). Il est facile de constater que deux variables, `suffixes` et `prefix`, sont utilisables en lecture et écriture par plusieurs fonctions.

```
suffices=Dict()
prefix=[]
```

Du fait que ces variables sont globales, nous ne pouvons effectuer qu’une seule analyse à la fois. S’il arrivait que deux textes soient lus, leurs préfixes et suffixes seraient ajoutés aux mêmes structures de données (ce qui produirait un texte intéressant).

Pour effectuer plusieurs analyses séparément, il convient d’encapsuler l’état de chaque analyse dans un objet. Voici comment procéder :

```
struct Markov
  order :: Int64
  suffixes :: Dict{Tuple{String,Vararg{String}}, Array{String, 1}}
  prefix :: Array{String, 1}
end

function Markov(order::Int64=2)
  new(order, Dict{Tuple{String,Vararg{String}}, Array{String, 1}}(), Array{String, 1}())
end
```

Ensuite, les fonctions sont transformées en méthodes. Par exemple, voici `processword` :

```
function processword(markov::Markov, word::String)
  if length(markov.prefix) < markov.order
    push!(markov.prefix, word)
    return
  end
  get!(markov.suffixes, (markov.prefix...,), Array{String, 1}())
  push!(markov.suffixes[(markov.prefix...,)], word)
  popfirst!(markov.prefix)
  push!(markov.prefix, word)
end
```

La transformation conduisant à un programme comme celui-ci – à savoir, changer la conception sans modifier le comportement – constitue un autre exemple de refonte (voir la section 4.7).

Cet exemple propose un plan de développement pour la conception des types :

- commencer par écrire des fonctions qui lisent et écrivent des variables globales (si nécessaire),
- une fois le programme opérationnel, rechercher les associations entre les variables globales et les fonctions qui les utilisent,
- encapsuler les variables afférentes sous forme de champs d’une structure composite,

- transformer les fonctions associées en méthodes avec comme argument les objets du nouveau type.

18.11.1 Exercice

Téléchargez le code Markov [sous ce lien](#). Suivez les étapes décrites ci-dessus pour encapsuler les variables globales comme attributs d'une nouvelle structure appelée Markov.

18.12 Glossaire

encoder représenter un ensemble de valeurs à l'aide d'un autre ensemble de valeurs en établissant une correspondance entre elles,

test unitaire moyen normalisé de tester l'exactitude d'un code,

placage méthode (ou fonction) qui fournit une interface différente à une autre fonction sans ajouter de calculs supplémentaires,

sous-typage possibilité de définir une hiérarchie de types apparentés,

type abstrait type qui peut agir comme parent pour un autre type,

type spécifique type qui peut être construit à partir d'un type abstrait,

sous-type type qui a comme parent un type abstrait,

supertype type abstrait, parent d'un autre type,

relation IS-A relation entre un sous-type et son supertype,

relation HAS-A relation entre deux types où les instances d'un type contiennent des références aux instances de l'autre,

dépendance relation entre deux types où les instances d'un type utilisent des instances d'un autre type, mais ne les enregistrent pas en tant que champs,

diagramme de type diagramme qui montre les types d'un programme et les relations entre eux,

multiplicité notation dans un diagramme de type qui montre, pour une relation HAS-A, combien de références existent à des instances d'une autre classe,

encapsulation de données plan de développement d'un programme qui comprend un prototype utilisant des variables globales et une version finale transformant les variables globales en champs d'instance.

18.13 Exercices

18.13.1 Exercice

Pour le programme suivant, dessinez un diagramme de types qui décr ces types et les relations entre eux.

```
abstract type PingPongParent end

struct Ping <: PingPongParent
  pong :: PingPongParent
end

struct Pong <: PingPongParent
  pings :: Array{Ping, 1}
  function Pong(pings=Array{Ping, 1}())
    new(pings)
  end
end

function addping(pong::Pong, ping::Ping)
  push!(pong.pings, ping)
  nothing
end

pong = Pong()
ping = Ping(pong)
addping(pong, ping)
```

18.13.2 Exercice

Rédigez une méthode appelée `deal!` qui prend trois paramètres : `Paquet`, le nombre de mains et le nombre de cartes par main. Elle doit créer le nombre *ad hoc* d'objets `UneMain`, distribuer le nombre approprié de cartes par main et retourner un tableau de `+Main+s`.

18.13.3 Exercice

Voici les mains possibles au poker (avec 5 cartes), par ordre décroissant de valeur et par ordre croissant de probabilité (voir le classement des mains au poker) :

Quinte Flush Royale il s'agit d'une suite de la même couleur (♠, ♥, ♦, ou ♣) allant du 10 à l'as. C'est la combinaison la plus forte au poker et d'une grande rareté. Par exemple : A♥, R♥, D♥, V♥, 10♥.

Quinte Flush toute quinte de la même couleur (♠, ♥, ♦, ou ♣) inférieure à la quinte flush royale. Par exemple : 9♠, 8♠, 7♠, 6♠, 5♠.

Carré toute combinaison de 4 cartes identiques de même rang. Si deux joueurs ont le même carré en même temps (ce qui veut dire que le carré est déjà sur la table), c'est le joueur avec la plus forte carte en mains (le Kicker) qui remporte le pot. Si la cinquième plus forte carte possible est aussi la cinquième du tapis, il y a égalité, et le pot est partagé. Par exemple : 4♠, 4♥, 4♦, 4♣, R♥.

Full House un Full House (« main pleine » en français) est l'association d'un Brelan et d'une Paire, c'est à dire 3 cartes identiques de n'importe quelle couleur et 2 autres cartes identiques à côté. On évalue toujours la force du Brelan en premier pour comparer deux Full. Par exemple : A♥, A♠, A♣, R♠, R♥.

Couleur (Flush en anglais) toute combinaison de 5 cartes de la même couleur (qui ne se suivent pas, sinon c'est une quinte flush). La carte la plus haute de la couleur détermine sa force. L'exemple montre une « couleur hauteur as », la plus forte couleur possible. Si deux joueurs ont une couleur de la même hauteur, on compare alors leur deuxième carte de la couleur la plus forte, et ainsi de suite. Exemple : A♠, 10♠, 7♠, 6♠, 2♠.

Quinte suite de 5 cartes consécutives qui ne sont pas toutes de la même couleur. Les as peuvent tout autant compter pour une quinte basse (A-2-3-4-5) qu'on appelle aussi la roue, ou pour une quinte haute (10-V-D-R-A). Plus la quinte est haute, plus elle est forte. Exemple : 5♣, 4♦, 3♠, 2♥, A♥.

Brelan 3 cartes identiques. L'exemple montre un Brelan d'as, avec un roi et une dame en kickers (cartes accompagnantes), soit le meilleur brelan possible : A♥, A♠, A♣, R♠, D♥.

Double paire 2 cartes identiques, accompagnées de 2 autres cartes identiques. L'exemple montre la meilleure combinaison de deux paires possible, une double paire as – rois. Lorsqu'on compare une double paire, on commence toujours par la paire la plus forte. Ainsi, une double paire A-A-5-5 est plus forte qu'une double paire R-R-V-V. Exemple : A♠, A♥, R♥, R♠, D♦.

Paire 2 cartes de rang identique. L'exemple montre la meilleure main possible à une paire : A♥, A♠, R♥, D♠, V♦.

Hauteur toute main qui ne possède aucune des combinaisons citées ci-dessus. Dans ce cas on évalue la main par rapport à sa carte la plus forte. Par exemple,

une main R-V-9-4-2 (qui ne sont pas de la même couleur) est une « Hauteur roi ». Si deux joueurs ont la même « hauteur », on regarde ensuite la deuxième carte la plus forte. Exemple : A♥, R♥, D♦, V♣, 9♠.

L'objectif de cet exercice est d'estimer la probabilité de ces différentes mains.

1. Ajoutez les méthodes appelées `haspair`, `hastwopair`, etc. qui retournent `true` ou `false` selon que la main répond ou non aux critères pertinents. Votre code devrait fonctionner correctement pour des « mains » qui contiennent un nombre quelconque de cartes (bien que 5 et 7 soient les configurations les plus courantes).
2. Rédigez une méthode appelée `classify` qui détermine le classement de plus haute valeur pour une main et définissez le champ `label` en conséquence. Par exemple, si une main de 7 cartes contenait une « flush » et une « paire », elle devait être étiquetée « flush ».
3. Lorsque vous êtes convaincu que vos méthodes de classement fonctionnent, l'étape suivante consiste à estimer les probabilités des différentes mains. Rédigez une fonction qui mélange un jeu de cartes, le divise en mains, classe les mains et compte le nombre de fois que les différents classements apparaissent.
4. Imprimez un tableau des classements et de leurs probabilités. Exécutez votre programme avec un nombre de mains de plus en plus important jusqu'à ce que les valeurs de sortie convergent vers un degré de précision raisonnable. Comparez vos résultats aux valeurs sur `classement des mains au poker` ou `Hand Ranking` (EN).

Chapitre 19

Bonus : À propos de la syntaxe

Un des objectifs de ce livre a été d'apprendre l'essentiel de Julia. Lorsque deux façons de procéder coexistaient, l'une a été choisie tout en évitant de mentionner l'autre. Parfois, la seconde méthode faisait partie d'un exercice.

À présent, revenons sur quelques éléments que nous avons négligés. Julia fournit un certain nombre de fonctionnalités pas toujours nécessaires. Il est possible d'écrire un bon code sans elles mais leur usage permet parfois d'écrire un code plus concis, plus lisible, plus efficace. Ces trois qualités peuvent même se conjuguer.

Ce chapitre et le suivant abordent les éléments laissés en suspens dans les chapitres précédents :

- des suppléments relatifs à la syntaxe,
- les fonctions, types et macros directement disponibles dans [Base](#),
- les fonctions, types et macros de la bibliothèque standard (*Standard Library*).

19.1 Tuples nommés

On peut désigner les composants d'un tuple en lui attribuant un nom :

```
julia> x = (a=1, b=1+1)
(a = 1, b = 2)
julia> x.a
1
```

Avec les tuples nommés, il devient possible d'accéder aux champs par leur nom en utilisant la syntaxe par point ([x.a](#)).

19.2 Fonctions

En Julia, les fonctions peuvent être définies à l'aide d'une syntaxe compacte :

```
julia> f(x,y) = x + y  
f (generic function with 1 method)
```

19.2.1 Fonctions anonymes

Une fonction peut être définie de manière non-nominative :

```
julia> x -> x^2 + 2x - 1  
#1 (generic function with 1 method)  
julia> function (x)  
    x^2 + 2x - 1  
end  
#3 (generic function with 1 method)
```

Ces exemples correspondent à des fonctions « anonymes ». Les fonctions anonymes sont souvent utilisées comme argument pour une autre fonction :

```
julia> using Plots  
julia> plot(x -> x^2 + 2x - 1, 0, 10, xlabel="x", ylabel="y")
```

La figure 19.2.1 montre le résultat de ces instructions.

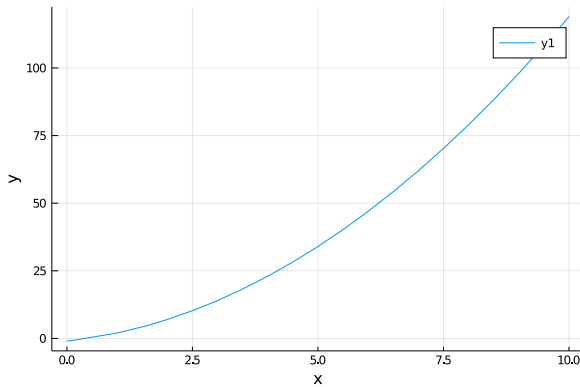


FIGURE 19.2.1 – Graphique de la fonction anonyme $x \rightarrow x^2 + 2x - 1$ avec Plots.

19.2.2 Arguments nommés

Les arguments de fonctions peuvent également être nommés :

```
julia> function myplot(x, y ; style="solid", width=1, color="black")
    ###
end
myplot (generic function with 1 method)
julia> myplot(0:10, 0:10, style="dotted", color="blue")
```

Les arguments nommés dans une fonction sont spécifiés après un point-virgule dans la signature mais peuvent être appelés avec une virgule.

19.2.3 Fermetures

Une fermeture est une technique qui permet à une fonction de capturer une variable définie en dehors du champ d'application de la fonction.





```
julia> foo(x) = ()->x
foo (generic function with 1 method)
julia> bar = foo(1)
#1 (generic function with 1 method)
julia> bar()
1
```

Dans cet exemple, la fonction `foo` retourne une fonction anonyme qui a accès à l'argument `x` de la fonction `foo`. `bar` pointe sur la fonction anonyme et retourne la valeur de l'argument de `foo`.

19.3 Blocs

Un *bloc* est un moyen de regrouper un certain nombre de déclarations. Un bloc commence par le mot-clé `begin` et se termine par `end`.

La macro `@svg` a été présentée dans le chapitre 4 :

```
 = Turtle()
@svg begin
    forward(, 100)
    turn(, -90)
    forward(, 100)
end
```

Dans cet exemple, la macro `@svg` possède un seul argument, c'est-à-dire un bloc regroupant 3 appels de fonction.

19.3.1 Bloc `let`

Un bloc `let` s'avère utile pour créer de nouvelles associations (ou ligatures), c'est-à-dire des variables locales pointant vers des valeurs.

```
julia> x, y, z = -1, -1, -1 ;
julia> let x = 1, z
    @show x y z;
end
x = 1
y = -1
ERROR: UndefVarError: z not defined
julia> @show x y z;
x = 1
y = -1
z = -1
```

Dans cet exemple, la première macro `@show` affiche `x` en local, `y` en global et `z` (indéfini) en local. Les variables globales ne sont pas touchées.

19.3.2 Blocs `do`

Dans la section 14.2, nous avons montré qu'il fallait fermer les fichiers après avoir effectué les opérations d'écriture. En réalité, cela peut être exécuté automatiquement en utilisant un *bloc do* :

```
julia> data = "Appréciez votre liberté,\nou vous la perdrez !\n"
"Appréciez votre liberté,\nou vous la perdrez !\n"
julia> open("output.txt", "w") do fout
    write(fout, data)
end
47
```

Dans cet exemple¹, `fout` est le flux de fichiers utilisé pour la sortie. En termes de fonctionnement, ceci est équivalent à :

1. Citation de Richard M. Stallman.

```
julia> f = fout -> begin
    write(fout, data)
end
#3 (generic function with 1 method)
julia> open(f, "output.txt", "w")
47
```

La fonction anonyme est utilisée comme premier argument de la fonction `open` :

```
function open(f::Function, args...)
    io = open(args...)
    try
        f(io)
    finally
        close(io)
    end
end
```

Un bloc `do` peut « capturer » des variables à l'extérieur de son périmètre d'application. Par exemple, dans l'exemple ci-dessus, la variable `data` de « `open ... do` » est capturée alors qu'elle figure comme variable globale.

19.4 Structure de contrôle

19.4.1 Opérateur ternaire

L'opérateur ternaire `?` : est une alternative à une déclaration `if-elseif` lorsqu'un choix entre plusieurs valeurs d'une seule expression s'avère nécessaire.

```
julia> a = 150
150
julia> a % 2 == 0 ? println("even") : println("odd")
even
```

L'expression précédant le signe `?` désigne une condition. Si la condition est `true`, l'expression précédant `:` est évaluée, sinon l'expression suivant `:` est exécutée.

19.4.2 Évaluation en court-circuit

Les opérateurs `&&` et `||` effectuent une évaluation en mode court-circuit : l'argument qui les suit n'est évalué que si nécessaire pour déterminer la valeur finale.

Par exemple, une routine factorielle récursive pourrait être définie ainsi :

```
function fact(n::Integer)
    n >= 0 || error("n ne peut pas être négatif")
    n == 0 && return 1
    n * fact(n-1)
end
```

19.4.3 Tâches ou co-routines

Une *tâche* est une structure de contrôle qui peut passer le contrôle de manière co-opérative sans retour. En Julia, une tâche peut être implémentée comme une fonction ayant en premier argument un objet [Channel](#). Un canal est utilisé pour transmettre des valeurs de la fonction à l'énoncé qui l'appelle.

La suite de Fibonacci peut être calculée à l'aide d'une tâche.

```
function fib(c::Channel)
    a = 0
    b = 1
    put!(c, a)
    while true
        put!(c, b)
        (a, b) = (b, a+b)
    end
end
```

[put!](#) enregistre les valeurs dans un objet canal et [take!](#) lit les valeurs à partir de celui-ci :

```
julia> fib_gen = Channel{Int}(10);
julia> take!(fib_gen)
0
julia> take!(fib_gen)
1
julia> take!(fib_gen)
1
julia> take!(fib_gen)
2
julia> take!(fib_gen)
3
```

Le constructeur [Channel](#) crée la tâche. La fonction [fib](#) est suspendue après chaque appel [put!](#) et reprise après [take!](#) Pour des raisons de performance, plusieurs valeurs

de la séquence sont mises en mémoire tampon dans l'objet canal pendant un cycle de reprise/suspension.

Un objet canal peut également être utilisé comme itérateur :

```
julia> for val in Channel(fib)
    print(val, " ")
    val > 20 && break
end
0 1 1 2 3 5 8 13 21
```

19.5 Types

19.5.1 Types primitifs

Un type spécifique (ou particulier) composé de bits est appelé un type primitif. Contrairement à la plupart des langages, Julia permet de déclarer nos propres types primitifs. Les types primitifs standard sont définis de la même manière :

```
primitive type Float64 <: AbstractFloat 64 end
primitive type Bool <: Integer 8 end
primitive type Char <: AbstractChar 32 end
primitive type Int64 <: Signed 64 end
```

Ces déclarations précisent le nombre de bits requis.

L'exemple suivant crée un type primitif `Byte` et un constructeur :

```
julia> primitive type Byte 8 end

julia> Byte(val::UInt8) = reinterpret(Byte, val)
Byte
julia> b = Byte(0x01)
Byte(0x01)
```

La fonction `reinterpret` sert à enregistrer les bits d'un entier non signé de 8 bits (`UInt8`) dans l'octet.

19.5.2 Types paramétriques

Le système de types de Julia est paramétrique, ce qui signifie que les types peuvent avoir des paramètres.

Les paramètres des types sont introduits après le nom du type, entouré d'accolades :

```
struct Point{T<:Real}
  x::T
  y::T
end
```

Ceci définit un nouveau type paramétrique, `Point{T<:Real}`, contenant deux « coordonnées » de type `T`, qui peut être n'importe quel type ayant `Real` comme supertype.

```
julia> Point(0.0, 0.0)
Point{Float64}(0.0, 0.0)
```

En plus des types composites, les types abstraits et les types primitifs peuvent également avoir un paramètre de type.

Conseil. Pour des raisons de performance, il est vivement recommandé d'avoir des types spécifiques pour les champs de structure. C'est là une bonne méthode pour rendre `Point` à la fois rapide et flexible.

19.5.3 Unions de types

À l'image d'une structure, une union de types est un regroupement d'objet de types différents. À la différence d'une structure, une union de types ne peut contenir qu'un seul de ses membres à la fois :

```
julia> IntOrString = Union{Int64, String}
Union{Int64, String}
julia> 150 :: IntOrString
150
julia> "Julia" :: IntOrString
"Julia"
```

Julia permet à ses utilisateurs de tirer parti des unions car un code efficace est produit en terme d'utilisation de la mémoire. L'accès aux champs d'une union est simplifié lorsque l'union est rendue « anonyme » au sein d'une autre structure ou d'une autre union.

19.6 Méthodes

19.6.1 Méthodes paramétriques

Les définitions de méthodes peuvent également comporter des paramètres de type caractérisant leur signature :

```
julia> isintpoint(p::Point{T}) where {T} = (T === Int64)
isintpoint (generic function with 1 method)
julia> p = Point(1, 2)
Point{Int64}(1, 2)
julia> isintpoint(p)
true
```

19.6.2 Objets foncteurs

En Julia, tous les objets sont « appelables ». Ces objets « appelables » sont des *foncteurs* (*functors*), c'est-à-dire des objets qui peuvent être traités à la manière d'une fonction.

```
struct Polynomial{R}
    coeff::Vector{R}
end

function (p::Polynomial)(x)
    val = p.coeff[end]
    for coeff in p.coeff[end-1:-1:1]
        val = val * x + coeff
    end
    val
end
```

Pour évaluer le polynôme :

```
julia> p = Polynomial([1,10,100])
Polynomial{Int64}([1, 10, 100])
julia> p(3)
931
```

19.7 Constructeurs

Les types paramétriques peuvent être construits explicitement ou implicitement :

```
julia> Point(1,2)           # implicit T
Point{Int64}(1, 2)
julia> Point{Int64}(1, 2)  # explicit T
Point{Int64}(1, 2)
julia> Point(1,2.5)        # implicit T
ERROR: MethodError: no method matching Point{::Int64, ::Float64}
```

Des constructeurs internes et externes par défaut sont générés pour chaque **T** :

```
struct Point{T<:Real}
  x::T
  y::T
  Point{T}(x,y) where {T<:Real} = new(x,y)
end

Point(x::T, y::T) where {T<:Real} = Point{T}(x,y);
```

x et **y** doivent tous deux être du même type.

Lorsque **x** et **y** ont un type différent, le constructeur extérieur peut être défini comme suit :

```
Point(x::Real, y::Real) = Point(promote(x,y)...);
```

La fonction **promote** est traitée dans la sous-section 19.8.2.

19.8 Conversions et promotions

Julia dispose d'un système permettant de promouvoir des arguments de différentes sortes en un type commun. Bien qu'elle ne soit pas automatique, la *promotion* peut facilement être effectuée.

19.8.1 Conversion

Une valeur peut être convertie d'un type vers un autre :

```
julia> x = 12
12
julia> typeof(x)
Int64
julia> convert(UInt8, x)
0x0c
julia> typeof(ans)
UInt8
```

Nous pouvons ajouter nos propres méthodes de conversion :

```
julia> Base.convert{::Type{Point{T}}}(x::Array{T, 1}) where {T<:Real} = Point(x...)
julia> convert(Point{Int64}, [1, 2])
Point{Int64}(1, 2)
```

19.8.2 Promotion

La *promotion* est la conversion des valeurs de types mixtes en un seul type commun :

```
julia> promote(1, 2.5, 3)
(1.0, 2.5, 3.0)
```

Les méthodes pour la fonction de promotion ne sont normalement pas directement définies. Cependant, la fonction auxiliaire `promote_rule` est utilisée pour spécifier les règles de promotion :

```
promote_rule(::Type{Float64}, ::Type{Int32}) = Float64
```

19.9 Méta-programmation

Le code Julia peut être représenté comme une structure de données de la langue elle-même. Cela permet à un programme de transformer et de produire son propre code.

19.9.1 Expressions

En Julia, chaque programme commence par une chaîne :

```
julia> prog = "1 + 2"
"1 + 2"
```

L'étape suivante consiste à analyser chaque chaîne de caractères en un objet appelé *expression*, représenté par le type `Expr` :

```
julia> ex = Meta.parse(prog)
:(1 + 2)
julia> typeof(ex)
Expr
julia> dump(ex)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 2
```

La fonction `dump` affiche les objets `expr` avec des annotations.

Les expressions sont construites directement avec le préfixe `:` suivi de parenthèses ou bien en utilisant un bloc *quote* :

```
julia> ex = quote
      1 + 2
end;
```

eval

Julia est à même d'évaluer un objet d'expression en utilisant `eval` :

```
julia> eval(ex)
3
```

Chaque module a sa propre fonction d'évaluation qui calcule les expressions dans son champ d'application.

Avertissement. Lorsque le nombre de recours à la fonction `eval` est élevé, cela signifie souvent qu'un programme est mal conçu. L'usage d'`eval` est considéré comme une « mauvaise pratique ».

19.9.2 Macros

Les macros peuvent inclure le code produit dans un programme. Une *macro* associe un ensemble d'objets `Expr` directement à une expression compilée :

Voici une macro simple :

```
macro containervariable(container, element)
    return esc(:($(Symbol(container,element)) = $container[$element])) end
```

Les macros sont appelées en préfixant leur nom par le symbole `@`. L'appel de macro `@containervariable lettres 1` est remplacé par :

```
:(lettres1 = lettres[1])
```

`@macroexpand @containervariable lettres 1` retourne cette expression qui est extrêmement utile pour le débogage.

Cet exemple illustre comment une macro accède au nom de ses arguments, ce qu'une fonction ne peut pas faire. L'expression retournée nécessite l'usage d'un échappement avec `esc`, du fait qu'elle doit être résolue dans l'environnement d'appel de la macro.

Note. Pourquoi utiliser des macros ? Les macros produisent et incluent des fragments de code personnalisé pendant que l'analyse a lieu, c'est-à-dire avant que le programme complet ne soit exécuté.

19.9.3 Fonctions générées

La macro `@generated` crée un code spécialisé pour les méthodes en fonction des types d'arguments :

```
@generated function square(x)
    println(x)
    :(x * x)
end
```

Le corps retourne une expression citée comme une macro.

Pour l'appelant, la fonction générée se comporte comme une fonction régulière :

```
julia> x = square(2); # note: sortie après l'instruction println() dans le corps
Int64
julia> x              # maintenant nous écrivons x
4
julia> y = square("bon");
String
julia> y
"bonbon"
```

19.10 Valeurs manquantes

Les *valeurs manquantes* peuvent être représentées via l'objet `missing`, qui est l'instance unique du type `Missing`.

Les tableaux sont susceptibles de contenir des valeurs manquantes :

```
julia> a = [1, missing]
2-element Array{Union{Missing, Int64},1}:
 1
missing
```

Le type d'élément d'un tel tableau est `Union{Missing, T}` avec `T`, le type des valeurs non manquantes.

Les fonctions de réduction retournent les valeurs manquantes lorsqu'elles sont invoquées sur des tableaux qui contiennent des valeurs manquantes :

```
julia> sum(a)
missing
```

Dans ce cas, il convient d'employer la fonction `skipmissing` pour passer outre les valeurs manquantes :

```
julia> sum(skipmissing([1, missing]))
```

```
1
```

19.11 Appel du code C et Fortran

Une quantité considérable de code est écrit en C ainsi qu'en Fortran. Réutiliser du code testé est une meilleure pratique que d'écrire sa propre version d'un algorithme. Julia peut appeler directement des bibliothèques C ou Fortran existantes en utilisant la syntaxe `ccall`.

Dans la section 14.6 (relative aux bases de données), nous avons introduit une interface Julia à la bibliothèque GDBM des fonctions de base de données. La bibliothèque est écrite en C. Pour fermer la base de données, un appel de fonction à `close(db)` doit être effectué :

```
Base.close(dbm::DBM) = gdbm_close(dbm.handle)

function gdbm_close(handle::Ptr{Cvoid})
    ccall((::gdbm_close, "libgdbm"), Cvoid, (Ptr{Cvoid},), handle)
end
```

Un objet `dbm` a un champ `handle` de type `Ptr{Cvoid}`. Ce champ contient un pointeur C qui fait référence à la base de données. Pour fermer la base de données, la fonction C `gdbm_close` doit être invoquée en ayant comme seul argument le pointeur C pointant vers la base de données et aucune valeur de retour. Julia effectue cela directement avec la fonction `ccall` ayant comme arguments :

- un tuple consistant en un symbole contenant le nom de la fonction que nous voulons appeler : `:gdbm_close` et la bibliothèque partagée spécifiée sous forme de chaîne : `"libgdbm"`,
- le type de retour : `Cvoid`,
- un tuple des types d'arguments : `(Ptr{Cvoid},)` et
- les valeurs de l'argument : `handle`.

La cartographie complète de la bibliothèque GDBM peut être trouvée à titre d'exemple dans les sources de ThinkJulia (ou ThinkJuliaFR).

19.12 Glossaire

fermeture fonction qui saisit les variables dans sa zone de portée,

bloc `let` bloc créant de nouvelles associations (ou ligatures) de variables locales vers des valeurs,

- fonction anonyme** fonction définie de manière non nominative,
- tuple nommé** tuple dont les composants sont nommés,
- arguments nommés** arguments identifiables par leur nom, c'est-à-dire pas seulement par leur position,
- bloc `do`** construction syntaxique utilisée pour définir et appeler une fonction anonyme similaire à un bloc de code usuel,
- opérateur ternaire** opérateur de flux de contrôle prenant trois opérandes : une condition, une expression à exécuter lorsque la condition retourne `true` et une autre expression à exécuter lorsque la condition retourne `false`,
- évaluation en court-circuit** évaluation d'un opérateur booléen pour lequel le deuxième argument est exécuté ou évalué, uniquement si le premier argument ne suffit pas à déterminer la valeur de l'expression,
- tâche (ou co-routine)** fonction de contrôle du flux qui permet la suspension et la reprise des calculs de manière flexible,
- type primitif** type spécifique dont les données sont constituées de simples bits,
- type union** type qui inclut comme objets toutes les instances de l'un ou l'autre de ses paramètres de type,
- type paramétrique** type qui possède des paramètres (autrement-dit : type paramétré),
- foncteur** objet avec une méthode associée lui permettant d'être invoqué,
- conversion** procédé permettant de convertir une valeur d'un type en un autre,
- promotion** conversion des valeurs de types différents en un seul type commun,
- expression** type Julia qui contient une construction linguistique,
- macro** procédé permettant d'inclure du code généré dans le corps final d'un programme,
- fonctions générées** fonctions capables de produire un code spécialisé selon les types d'arguments,
- valeurs manquantes** instances qui représentent des données sans qu'une valeur leur soit attribuée.

Chapitre 20

Bonus : Bibliothèque de base et standard

Outre un environnement en développement permanent, Julia est livrée –de base– avec de nombreux outils. Le module de base contient les fonctions, les types et les macros les plus utiles. Julia fournit également un grand nombre de modules spécialisés dans sa bibliothèque standard (dates, calcul distribué, algèbre linéaire, profilage, nombres aléatoires, etc.). Les fonctions, types et macros définis dans la bibliothèque standard doivent être importés avant d’être utilisés :

- `import Module` importe le module souhaité et `Module.fn(x)` appelle la fonction `fn`,
- `using Module` importe toutes les fonctions, types et macros du `Module`.

Des fonctionnalités supplémentaires sont ajoutées à partir d’une collection croissante de paquets (voir [Julia Observer](#)).

Ce chapitre ne remplace pas la [documentation officielle](#) de Julia. Ne sont cités que quelques exemples pour illustrer ce qui est possible sans toutefois être exhaustif. Les fonctions déjà introduites ailleurs ne sont pas incluses. Une vue d’ensemble complète est disponible sur [Julia Documentation](#).

20.1 Mesures de performance

Nous avons vu que certains algorithmes sont plus performants que d’autres. La fonction `fibonacci` en section 11.6 (Mémorisation) est beaucoup plus rapide que `fib` écrite en section 6.7. La macro `@time` permet de quantifier la différence :

```
julia> fib(1)
1
julia> fibonacci(1)
1
julia> @time fib(40)
0.567546 seconds (5 allocations : 176 bytes)
102334155
julia> @time fibonacci(40)
0.000012 seconds (8 allocations : 1.547 KiB)
102334155
```

`@time` affiche le temps d'exécution de la fonction, le nombre d'allocations et la mémoire allouée avant de retourner le résultat. La version « mémo » est effectivement beaucoup plus rapide mais elle requiert davantage de mémoire.

« Rien n'est gratuit ».

Conseil. En Julia, lors de sa première exécution, une fonction est compilée. La comparaison de deux algorithmes requiert que ceux-ci soient implémentés en tant que fonctions pour être compilés et, la première fois qu'ils sont appelés doit être exclue de la mesure de performance, sinon la durée de compilation est prise en compte.

Le paquet `BenchmarkTools` fournit la macro `@btime` qui permet de faire de l'analyse de performance de la bonne manière. Utilisez-le.¹

20.2 Collections et structures de données

Dans la section 13.6 (Soustraction de dictionnaires), des dictionnaires ont été utilisés pour trouver les mots qui apparaissent dans un document mais pas dans un tableau de mots. La fonction que nous avons écrite prend `d1` contenant les mots du document comme clés et `d2` qui renferme le tableau de mots. Elle retourne un dictionnaire qui contient les clés de `d1` absentes dans `d2`.

```
function subtract(d1, d2)
    res = Dict{<type>(), <type>}()
    for key in keys(d1)
        if key ∉ keys(d2)
            res[key] = nothing
        end
    end
    res
end
```

1. Le lecteur consulera la référence [10]

Dans tous ces dictionnaires, les valeurs sont `nothing` parce qu'elles ne sont jamais utilisées. Par conséquent, nous gaspillons un peu d'espace de stockage.

Julia propose un autre type interne appelé un « *set* ». Ce type se comporte comme un ensemble de clés de dictionnaire sans valeurs. L'ajout d'éléments à un *set* est rapide, tout comme la vérification d'appartenance à un *set*. Les *sets* fournissent des fonctions et des opérateurs pour effectuer des opérations courantes.

Par exemple, la soustraction d'un *set* est disponible sous la forme d'une fonction appelée `setdiff`. Nous pouvons donc réécrire la soustraction comme suit :

```
function subtract(d1, d2)
    setdiff(d1, d2)
end
```

Le résultat est un *set* au lieu d'un dictionnaire.

Certains des exercices de ce livre peuvent être réécrits de manière concise et efficace avec des *sets*. Par exemple, voici une solution pour `hasduplicates`, de l'exercice 10.15.7 qui utilise un dictionnaire :

```
function hasduplicates(t)
    d = Dict()
    for x in t
        if x ∈ d
            return true
        end
        d[x] = nothing
    end
    false
end
```

Lorsqu'un élément apparaît pour la première fois, il est ajouté au dictionnaire. Si le même élément apparaît à nouveau, la fonction retourne `true`.

En utilisant des *sets*, la même fonction peut être réécrite comme ceci :

```
function hasduplicates(t)
    length(Set(t)) < length(t)
end
```

Un élément ne peut apparaître qu'une seule fois dans un *set*. Donc si un élément apparaît plus d'une fois dans `t`, le *set* sera plus petit que `t`. S'il n'y a pas de répétitions d'élément(s), l'ensemble aura la même taille que `t`.

Nous pouvons également utiliser des *sets* pour faire certains des exercices du chapitre 9. Par exemple, voici une version d'`useonly` (section 9.3) avec une boucle :

```
function useonly(word, available)
    for letter in word
        if letter ∉ available
            return false
        end
    end
    true
end
```

`useonly` vérifie si toutes les lettres contenues dans `word` se trouvent dans `available`. Cette fonction peut être réécrite ainsi :

```
function usesonly(word, available)
    Set(word) ⊆ Set(available)
end
```

L'opérateur \subseteq (`\subseteq` TAB) vérifie si un *set* est inclus dans un autre *set*, en ce compris la possibilité qu'ils soient égaux. Dans ce dernier cas, cela signifie que toutes les lettres de `word` apparaissent dans `available`.

20.2.1 Exercice

Réécrivez la fonction `avoids` (section 9.3) avec les *sets*.

20.3 Mathématiques

Les nombres complexes sont pris en charge par Julia. La constante globale `im` est liée au nombre complexe `i` (avec $i^2 = -1$).

L'identité d'Euler est vérifiable,

```
julia> e^(im*pi)+1
```

Le symbole e (`\euler` TAB) est la base des logarithmes naturels.

Illustrons le caractère complexe des fonctions trigonométriques :

$$\cos x = \frac{e^{ix} + e^{-ix}}{2}$$

Nous pouvons tester cette formule pour différentes valeurs de x .

```
julia> x = 0:0.1:2pi
0.0:0.1:6.2
julia> cos.(x) == 0.5*(e.^(im*x)+e.^(-im*x))
true
```

Ceci est un autre exemple d'application de l'opérateur « point ». Julia permet également de juxtaposer des littéraux numériques avec des identificateurs sous forme de coefficients comme dans 2π .

20.4 Chaînes

Dans les chapitres 8 et 9, nous avons mené quelques recherches élémentaires dans les objets de type String. Cependant, Julia gère des expressions rationnelles compatibles avec le langage Perl. Ceci facilite la recherche de motifs complexes dans les chaînes de caractères.

La fonction `usesonly` peut être mise en œuvre comme une expression rationnelle :

```
function usesonly(word, available)
    r = Regex("[^$(available)]")
    !occursin(r, word)
end
```

L'expression régulière recherche un caractère qui n'est pas dans la chaîne `available` et `occursin` retourne `true` si le motif est trouvé dans le mot.

```
julia> usesonly("bonbon", "bno")
true
julia> usesonly("bonbons", "bno")
false
```

Les expressions rationnelles peuvent également être construites comme des chaînes de caractères non normalisées préfixées par un `r` :

```
julia> match(r"[^bno]", "bonbon")
julia> m = match(r"[^bno]", "bonbons")
RegexMatch("s")
```

Dans ce cas, l'interpolation de chaînes n'est pas autorisée. La fonction `match` ne retourne rien si le motif (une commande) n'est pas trouvé et retourne un objet `regexmatch` dans le cas contraire.

Nous pouvons extraire les informations suivantes d'un objet `regexmatch` :

- la sous-chaîne entière correspondant : `m.match`
- les sous-chaînes capturées comme un tableau de chaînes de caractères : `m.captures`
- le décalage auquel commence l'ensemble de la correspondance : `m.offset`
- les décalages des sous-chaînes capturées sous la forme d'un tableau : `m.offsets`

```
julia> m.match
"s"
julia> m.offset
7
```

Les expressions rationnelles constituent un outil très puissant. La page de manuel de Perl fournit tous les détails pour mener à bien des recherches très poussées, voire sophistiquées.

20.5 Tableaux

Dans le chapitre 10, nous avons utilisé l'objet `Array` (tableau) comme conteneur unidimensionnel avec des indices permettant de retrouver ses éléments. Julia manipule aussi les tableaux multidimensionnels.

Créons une matrice de 2 par 3 contenant des zéros :

```
julia> z = zeros{Float64, 2, 3}
2×3 Array{Float64,2} :
 0.0 0.0 0.0
 0.0 0.0 0.0
julia> typeof(z)
Array{Float64,2}
```

Le type de cette matrice est un tableau à 2 dimensions contenant des nombres à virgule flottante.

La fonction `size` renvoie un tuple décrivant le nombre d'éléments dans chaque dimension :

```
julia> size(z)
(2, 3)
```

La fonction `ones` construit une matrice avec des éléments de valeur 1 :

```
julia> s = ones{String, 1, 3}
1×3 Array{String,2} :
 "" "" ""
```

L'élément unitaire « chaîne » représente une chaîne vide.

Avertissement. `s` n'est pas un tableau unidimensionnel :

```
julia> s == ["", "", ""]
false
```

`s` est une matrice à une ligne, tandis que `["", "", ""]` est une matrice à 1 colonne.

Une matrice peut être saisie directement en utilisant un espace pour séparer les éléments d'une ligne et un point-virgule ; pour séparer les lignes :

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2} :
 1 2 3
 4 5 6
```

Pour traiter des éléments individuels, des crochets sont utilisables :

```
julia> z[1,2] = 1
1
julia> z[2,3] = 1
1
julia> z
2×3 Array{Float64,2} :
 0.0 1.0 0.0
 0.0 0.0 1.0
```

La sélection d'un sous-groupe d'éléments peut être réalisée par segmentation :

```
julia> u = z[:,2:end]
2×2 Array{Float64,2} :
 1.0 0.0
 0.0 1.0
```

L'opérateur `.` effectue une distribution sur chaque éléments dans toutes les dimensions :

```
julia> e.^(im*u)
2×2 Array{Complex{Float64},2} :
 0.540302+0.841471im 1.0+0.0im
 1.0+0.0im 0.540302+0.841471im
```

20.6 Interfaces

Julia tire parti de certaines interfaces informelles pour définir des comportements, c'est-à-dire des méthodes ayant un objectif spécifique. Lorsque ces méthodes sont étendues à un type, des objets de ce type peuvent être utilisés pour construire ces comportements.

« Si ça ressemble à un canard, si ça nage comme un canard et si ça cancanne comme un canard, c'est un canard. »

Dans la section 6.7, nous avons mis en œuvre la fonction `fib` qui retourne le $n^{\text{ème}}$ élément de la suite de Fibonacci. La recherche des valeurs d'une suite constitue une des ces interfaces. Créons un itérateur qui retourne la suite de Fibonacci :

```
struct Fibonacci{T<:Real} end
Fibonacci(d::DataType) = d<:Real ? Fibonacci{d}() : error("No Real type!")

Base.iterate(::Fibonacci{T}) where {T<:Real} = (zero(T), (one(T), one(T)))
Base.iterate(::Fibonacci{T}, state::Tuple{T, T}) where {T<:Real} = (state[1],
(state[2], state[1] + state[2]))
```

Nous avons mis en œuvre un type paramétrique sans champs `Fibonacci`, un constructeur extérieur et deux méthodes `iterate`. La première est appelée pour initialiser l'itérateur et retourne un tuple composé de la première valeur (0) et d'un état. L'état est un tuple contenant la deuxième et la troisième valeur : 1 et 1.

La seconde itération est appelée pour obtenir la valeur suivante de la séquence de Fibonacci et retourne un tuple avec comme premier élément la valeur suivante et comme second élément un état qui consiste en un tuple avec les deux valeurs suivantes.

À ce stade, nous pouvons appeler `Fibonacci` dans une boucle `for` :

```
julia> for e in Fibonacci{Int64}()
    e > 100 && break
    print(e, " ")
end
0 1 1 2 3 5 8 13 21 34 55 89
```

Cela semble magique, mais l'explication est simple. Une boucle `for` en Julia :

```
for i in iter
    # corps de la boucle
end
```

est convertie en :

```
next = iterate(iter)
while next != nothing
    (i, state) = next
    # corps de la boucle
    next = iterate(iter, state)
end
```

C'est là un très bon exemple de la façon dont une interface bien conçue permet à une implémentation d'utiliser toutes les fonctions disponibles *via* cette interface.

20.7 Utilitaires interactifs

Nous avons déjà rencontré le module `InteractiveUtils` dans la section 18.10. La macro `@which` n'est que la partie émergée de l'iceberg.

Le code Julia est transformé par la bibliothèque LLVM (*Low Level Virtual Machine*) en code machine en plusieurs étapes. Nous pouvons directement visualiser la sortie de chaque étape.

Donnons un exemple simple :

```
function squaresum(a::Float64, b::Float64)
    a^2 + b^2
end
```

La première étape consiste à examiner le code de bas niveau :

```
julia> using InteractiveUtils

julia> @code_lowered squaresum(3.0, 4.0)
CodeInfo(
  1 - %1 = (Core.apply_type)(Base.Val, 2)
    | %2 = (%1)()
    | %3 = (Base.literal_pow)(:~, a, %2)
    | %4 = (Core.apply_type)(Base.Val, 2)
    | %5 = (%4)()
    | %6 = (Base.literal_pow)(:~, b, %5)
    | %7 = %3 + %6
    └─ return %7
)
```

La macro `@code_lowered` retourne sous forme d'un tableau une représentation intermédiaire du code utilisé par le compilateur pour produire du code optimisé.

L'étape suivante ajoute des informations sur le type :

```
julia> @code_typed squaresum(3.0, 4.0)
CodeInfo(
  1 - %1 = (Base.mul_float)(a, a)::Float64
    | %2 = (Base.mul_float)(b, b)::Float64
    | %3 = (Base.add_float)(%1, %2)::Float64
    └─ return %3 ) => Float64
```

Nous observons que le type de résultats intermédiaires et la valeur de retour sont correctement déduits.

Cette représentation du code est transformée en code LLVM :

```
julia> @code_llvm squaresum(3.0, 4.0)
; @ none:2 within 'squaresum'
define double @julia_squaresum_14821(double, double) {
top :
; @ intfuncs.jl:243 within 'literal_pow'
; | @ float.jl:399 within '*'
; |   %2 = fmul double %0, %0
; |   %3 = fmul double %1, %1
; |   LL
; | @ float.jl:395 within '+'
; |   %4 = fadd double %2, %3
; |   L
; |   ret double %4
; }
}
```

Finalement, le *code machine* est produit :

```
julia> @code_native squaresum(3.0, 4.0)
.section __TEXT,__text,regular,pure_instructions
; @ none :2 within 'squaresum'
; | @ intfuncs.jl:243 within 'literal_pow'
; | | @ none:2 within '*'
; | |   vmulsd %xmm0, %xmm0, %xmm0
; | |   vmulsd %xmm1, %xmm1, %xmm1
; | |   LL
; | @ float.jl:395 within '+'
; |   vaddsd %xmm1, %xmm0, %xmm0
; |   Lretl
; |   nopl    (%eax)
; |   L
; }
```

20.8 Débogage

Les macros [Logging](#) fournissent une alternative aux canevas avec des déclarations d’affichage :

```
julia> @warn "Abandon printf debugging, all ye who enter here!"
[ Warning : Abandon printf debugging, all ye who enter here !
[ @ Main REPL[1]:1
```

Les déclarations de débogage ne doivent pas être retirées du code. Par exemple, contrairement à [@warn](#) ci-dessus, le code :

```
julia> @debug "The sum of some values $(sum(rand(100)))"
```

ne produira, par défaut, aucun résultat. Dans ce cas, `sum(rand(100))` ne sera jamais évaluée à moins que la *journalisation du débogage* ne soit activée.

Le niveau de journalisation peut être sélectionné par une variable d'environnement `JULIA_DEBUG` :

```
$ JULIA_DEBUG=all julia -e '@debug "The sum of some values $(sum(rand(100)))"'
└ Debug: The sum of some values 47.116520814555024
  @ Main none:1
```

En l'occurrence, nous avons utilisé `all` pour extraire toutes les informations de débogage. Cependant, il est possible de ne produire que les informations associées à un fichier ou à un module spécifique.

20.9 Glossaire

regex expression rationnelle, une séquence de caractères qui définit un modèle de recherche,

matrice tableau à deux dimensions,

représentation intermédiaire structure de données utilisée en interne par un compilateur pour représenter le code source.

code machine instructions qui peuvent être exécutées directement par l'unité centrale d'un ordinateur,

enregistrement de débogage stockage des messages de débogage dans un journal.

Chapitre 21

Débogage

Lors du débogage, il est nécessaire de distinguer les différents types d'erreurs afin de les repérer rapidement :

- les erreurs de syntaxe sont découvertes par le compilateur lors de la traduction à la volée du code source en code compilé. Elles indiquent une formulation erronée au sein de la structure du programme. Il en va ainsi, par exemple, de l'omission du mot-clé `end` à la fin d'un bloc de fonction qui engendre le message d'erreur quelque peu redondant : `ERROR: LoadError: syntax: incomplete: function requires end`.
- les erreurs d'exécution sont produites par le compilateur si un dysfonctionnement apparaît au cours de l'exécution d'un programme. La plupart des messages d'erreur d'exécution contiennent des informations sur la localisation de l'erreur et les fonctions en cours d'exécution. Par exemple, une récursion infinie provoque une erreur d'exécution telle que `ERROR: StackOverflowError`.
- les erreurs sémantiques sont liées à un programme qui s'exécute sans produire de messages d'erreur mais qui ne fait pas ce à quoi il est destiné. À titre d'exemple, une expression peut ne pas être évaluée dans l'ordre prévu, ce qui donne un résultat incorrect.

La première étape du débogage consiste à déterminer le type d'erreur auquel on est confronté. Bien que les sections suivantes soient organisées par type d'erreur, certaines techniques sont applicables dans plus d'une situation.

21.1 Erreurs de syntaxe

Les erreurs de syntaxe sont généralement faciles à corriger dès qu'on a compris de quoi il s'agit. Malheureusement, d'une manière générale, les messages d'erreur ne sont pas très explicites. Les messages les plus courants sont du type `ERROR: LoadError: syntax: incomplete: premature end of input` et `ERROR: LoadError: syntax: unexpected "="`. Aucune des deux formulations n'est véritablement instructive.

En revanche, le message indique l'endroit du programme où le problème s'est produit. En fait, il indique où Julia a remarqué un problème, ce qui n'est pas nécessairement l'endroit exact où se situe l'erreur. Il n'est pas rare que celle-ci se trouve en amont de l'emplacement indiqué par le message d'erreur (souvent sur la ligne précédente).

Si vous construisez un programme de manière incrémentale, vous devriez avoir une bonne idée de l'endroit où se trouve l'erreur. Elle se cache soit dans la dernière ligne soit dans le dernier bloc ajouté.

Si vous copiez le code depuis un livre ou une page de l'internet, commencez par comparer très soigneusement votre code à celui contenu dans la source. Vérifiez chaque caractère. En même temps, n'oubliez pas que la source peut elle-même contenir un code erroné. Si bien que si vous voyez quelque chose qui ressemble à une erreur de syntaxe, il peut s'agir de ce dernier cas de figure.

Voici quelques moyens d'éviter les erreurs de syntaxe les plus courantes :

1. assurez-vous que vous n'employez pas un mot-clé de Julia pour un nom de variable,
2. vérifiez que le mot-clé `end` clôture chaque déclaration composée, y compris les blocs `for`, `while`, `if` et les blocs de fonction,
3. assurez-vous que toutes les chaînes de caractères du code sont entre guillemets et que tous les guillemets sont des "droits" et non "bouclés",
4. si vous avez écrit des chaînes multilignes avec des guillemets triples, assurez-vous que vous avez clôturé la chaîne correctement. Une chaîne inachevée peut provoquer une erreur « *invalid token* » à la fin d'un programme. En outre, elle peut traiter la partie suivante du programme comme une chaîne jusqu'à ce qu'elle parvienne à la chaîne suivante. Dans ce dernier cas, il se peut que la compilation ne produise aucun message d'erreur,
5. un opérateur d'ouverture sans son vis-à-vis – `(`, `{`, ou `[` – amène Julia à considérer la ligne suivante comme appartenant à la déclaration courante. En général, une erreur se produit presque immédiatement à la ligne suivante,
6. vérifiez le classique `=` au lieu de `==` à l'intérieur d'un test conditionnel,

7. si des caractères non-ASCII se trouvent dans le code (y compris des chaînes de caractères et des commentaires), cela peut poser un problème bien que Julia gère généralement les caractères non-ASCII. Soyez attentif si vous collez du texte provenant d'une page web ou d'une autre source.

Si rien de ceci ne fonctionne, passez à la section suivante.

21.1.1 Je continue à faire des changements mais sans effet

Si le REPL indique qu'il y a une erreur et que vous ne la détectez pas, c'est peut-être parce que vous ne considérez pas le même code que le REPL. Vérifiez votre environnement de programmation afin de déterminer si le programme que vous éditez est bien celui que Julia tente d'exécuter.

Si vous n'en êtes pas sûr, essayez d'introduire une erreur de syntaxe évidente et délibérée au début du programme. Exécutez-le à nouveau. Dans le cas où le REPL ne trouve pas la nouvelle erreur, vous pouvez conclure que vous n'exécutez pas le nouveau code.

Il y a quelques coupables potentiels :

- vous avez édité le fichier et oublié de sauvegarder les modifications avant de l'exécuter à nouveau. Certains environnements de programmation le font pour vous, d'autres non,
- vous avez changé le nom du fichier, cependant vous utilisez toujours l'ancien nom,
- quelque chose dans votre environnement de développement est mal configuré,
- si vous écrivez un module et que vous l'utilisez, assurez-vous de ne pas donner à votre module le même nom qu'un des modules standard de Julia,
- si vous employez le mot-clé `using` pour importer un module, n'oubliez pas qu'il est impératif de redémarrer le REPL lorsque vous modifiez le code du module.

Si vous importez à nouveau le module, il ne se passe rien.

Si vous êtes bloqué et que vous ne pouvez pas comprendre ce qui se produit, une approche consiste à recommencer avec un nouveau programme comme « Hello, World ! » et à vous assurer que vous pouvez faire fonctionner un programme opérationnel connu. Ensuite, ajoutez progressivement les éléments du programme d'origine au nouveau programme.

21.2 Erreurs d'exécution

Dès que votre programme est syntaxiquement correct, Julia peut le lire et *a minima* commencer à l'exécuter. Qu'est-ce qui alors pourrait dysfonctionner ?

21.2.1 Mon programme ne fait absolument rien

Ce problème est assez fréquent lorsque votre fichier se compose de fonctions et de classes sans invoquer réellement une fonction pour en lancer l'exécution. Cela peut être intentionnel si vous prévoyez d'importer ce module uniquement pour fournir des classes et des fonctions.

Si ce n'est pas le cas, assurez-vous qu'un appel de fonction est bien présent dans le programme. Ensuite, vérifiez que le flux d'exécution l'atteint bel et bien (voir le paragraphe 21.2.2).

21.2.2 Mon programme est « suspendu »

Si un programme s'arrête et semble ne rien faire, il est « suspendu ». Souvent, cela signifie qu'il est engagé dans une boucle ou une récursion infinies :

- si vous soupçonnez qu'une boucle particulière est à l'origine du problème, ajoutez un message d'affichage immédiatement avant la boucle qui stipule « entrer dans la boucle » et un autre immédiatement après tel que « sortir de la boucle ». Exécutez à nouveau le programme. Si vous obtenez le premier message et non le second, vous êtes confronté à une boucle infinie. Passez au paragraphe 21.2.2 ci-dessous,
- la plupart du temps, une récursion infinie entraînera l'exécution du programme pendant un certain temps puis, produira une [ERROR: LoadError: Error Stack-OverflowError](#). Si c'est le cas, passez au paragraphe 21.2.2 ci-dessous.
Si vous n'obtenez pas cette erreur quoique vous soupçonniez un problème avec une méthode ou une fonction récursive, vous pouvez toujours utiliser les techniques du paragraphe 21.2.2,
- si aucune de ces étapes n'apporte de solution, commencez à tester d'autres boucles ainsi que d'autres fonctions et méthodes récursives,
- en cas d'échec, il est possible que vous ne compreniez pas le flux d'exécution de votre programme. Rendez-vous au paragraphe 21.2.2.

Boucle infinie

Si vous pressentez une boucle infinie et que vous pensez avoir identifié la boucle responsable du problème, ajoutez une déclaration d'affichage à la fin de la boucle, qui communique les valeurs des variables de la condition et la valeur de la condition.

Par exemple :

```
while x > 0 && y < 0
  # modifier x
  # modifier y
  @debug "variables" x y
  @debug "condition"
  x > 0 && y < 0
end
```

Maintenant, lorsque vous exécutez le programme en mode de débogage, vous verrez la valeur des variables et la condition pour chaque passage dans la boucle. Au dernier passage, la condition devrait être fausse. Si la boucle continue, vous pourrez visualiser les valeurs de `x` ainsi que de `y`, et vous comprendrez peut-être pourquoi elles ne sont pas mises à jour correctement.

Récursion infinie

La plupart du temps, une récursion infinie amène un programme à calculer pendant un certain temps pour produire une [ERROR: LoadError: Error StackOverflowError](#).

Si vous pensez qu'une fonction provoque une récursion infinie, contrôlez la présence d'un cas de base. Il devrait y avoir une condition amenant la fonction à effectuer un retour sans exécuter une invocation récursive. Si ce n'est pas le cas, vous devez repenser l'algorithme et identifier un cas de base.

Si un cas de base existe mais que le programme ne semble pas l'atteindre, ajoutez une instruction d'affichage au début de la fonction, qui indique les paramètres. À présent, lorsque vous exécutez le programme, vous verrez quelques lignes de sortie chaque fois que la fonction est invoquée et vous pourrez analyser les valeurs des paramètres. Si les paramètres ne se déplacent pas vers le cas de base, vous aurez une idée des raisons pour lesquelles il en va ainsi.

Flux d'exécution

Si vous n'êtes pas sûr de la manière dont le flux d'exécution percole dans votre programme, ajoutez un message au début de chaque fonction comme « entrer dans la fonction *unetelle* », où *unetelle* est le nom de la fonction.

À ce stade, lorsque vous exécutez le programme, celui-ci affichera une trace de chaque fonction au fur et à mesure qu'elle sera invoquée.

21.2.3 Quand j'exécute mon programme, j'obtiens une exception

Si quelque chose capote pendant l'exécution, Julia affiche un message qui comprend le nom de l'exception, la ligne du programme où le problème a surgi et une trace de pile (*stacktrace*).

La trace de pile identifie la fonction en cours d'exécution, puis la fonction qui l'a appelée, celle qui à son tour a appelé cette dernière, et ainsi de suite. En d'autres termes, elle retrace la séquence des appels de fonction qui ont conduit à la situation à laquelle vous êtes confronté, y compris le numéro de ligne de votre fichier où chaque appel s'est produit.

La première étape consiste à examiner l'endroit du programme où l'erreur s'est produite et à déterminer si vous pouvez comprendre ce qui s'est passé. Il s'agit là de certaines des erreurs d'exécution les plus courantes :

ArgumentError un des arguments à un appel de fonction n'est pas dans l'état attendu,

BoundsError une opération sur les indices dans un tableau a tenté d'accéder à un élément hors limites,

DomainError l'argument d'une fonction ou d'un constructeur est en dehors du domaine de validité,

DivideError une division entière a été tentée avec une valeur de dénominateur égale à 0,

EOFError il n'y avait plus de données disponibles à lire à partir d'un fichier ou d'un flux,

InexactError ne peut pas exactement se convertir à un type,

KeyError une opération d'indexation dans un objet [AbstractDict](#) ([Dict](#)) ou [Set](#) a tenté d'accéder ou de supprimer un élément inexistant,

MethodError une méthode avec la signature requise n'existe pas dans la fonction générique donnée. Alternativement, il n'existe pas de méthode unique plus spécifique,

OutOfMemoryError une opération a alloué trop de mémoire pour que le système ou le récupérateur de mémoire puisse la gérer correctement,

OverflowError le résultat d'une expression est trop grand pour le type spécifié et provoque un débordement,

StackOverflowError un appel de fonction tente d'utiliser plus d'espace que celui disponible dans la pile d'appels. Cela se produit généralement lorsqu'un appel est répété à l'infini.

StringIndexError Une erreur s'est produite lors de la tentative d'accès à un indice de chaîne invalide,

SystemError un appel système a échoué avec un code d'erreur,

TypeError erreur d'assertion de type, ou erreur produite lors de l'appel d'une fonction intégrée avec un type d'argument incorrect,

UndefVarError un symbole dans l'environnement en cours n'est pas défini.

21.2.4 J'ai ajouté beaucoup de déclarations d'affichage ; je suis inondé de sorties

L'utilisation intensive des instructions d'affichage pour le débogage se traduit régulièrement en une avalanche de messages. Deux façons d'y remédier : simplifier la sortie ou simplifier le programme.

Pour simplifier la sortie, il est pertinent de supprimer ou commenter les déclarations d'affichage qui n'apportent pas d'aide concrètes, les combiner (le cas échéant) ou encore, formater la sortie de manière à être plus facile à interpréter.

Pour simplifier un programme, il existe plusieurs approches. Tout d'abord, réduisez le problème sur lequel le programme travaille. Par exemple, si vous traiter une liste, testez une liste courte. Si le programme capte diverses données de l'utilisateur, donnez-lui l'information la plus élémentaire qui cause le problème.

Ensuite, nettoyez le programme. Supprimez le code mort et réorganisez le programme pour le rendre aussi lisible que possible. Imaginons que le problème se situe dans une partie profondément imbriquée du programme. Essayez de réécrire cette partie avec une structure allégée. Si vous présumez l'implication d'une fonction de grande taille, essayez de la diviser en fonctions de moindre taille afin de les tester séparément.

Souvent, le processus de recherche d'un test minimal vous mène au bogue. Si vous trouvez qu'un programme fonctionne dans une situation mais pas dans une autre, cela aussi vous donne un indice.

De même, réécrire un morceau de code peut aider à débusquer des bogues malicieux. Si vous apportez une modification qui, selon vous, ne devrait pas affecter le programme alors qu'il en est ainsi, vous tenez peut-être bien un indice à défaut de tenir la solution.

21.3 Erreurs sémantiques

Les erreurs sémantiques sont les plus difficiles à déboguer par le simple fait que le compilateur à la volée ne fournit aucune information. Vous seul savez ce que le

programme est censé faire.

La première étape consiste à établir un lien entre le code du programme et le comportement observé. Il faut émettre une hypothèse concernant la manière dont le programme procède. Une des causes qui rendent la dépiage des erreurs sémantiques si difficile provient de la vitesse d'exécution des ordinateurs.

On voudrait souvent pouvoir ralentir le programme jusqu'à « vitesse humaine ». L'insertion judicieuse de quelques instructions d'affichage est souvent plus fructueuse et plus rapide que la mise en place d'un débogueur, l'insertion et la suppression de points d'arrêt et la gestion du « pas » du programme pour converger vers l'endroit où l'erreur se produit.

21.3.1 Mon programme ne fonctionne pas

Le moment est venu de se poser quelques questions :

- le programme était-il censé faire quelque chose qui semble ne pas se produire ? Trouvez la section du code qui remplit cette fonction et vérifiez qu'il s'exécute au moment où vous pensez qu'il devrait procéder ainsi,
- quelque chose survient-il qui ne devrait pas se produire ? Trouvez dans votre programme le code qui remplit cette fonction et assurez-vous qu'il s'exécute au moment où il ne devrait pas.
- une partie du code produit-elle un effet qui n'est pas celui auquel vous vous attendiez ? Assurez-vous de bien comprendre le code en question, surtout s'il s'agit de fonctions ou de méthodes associées à d'autres modules de Julia. Lisez la documentation relative aux fonctions que vous appelez. Essayez-les en écrivant des cas de test simples et en vérifiant les résultats.

Pour pouvoir programmer, vous devez disposer d'un modèle mental du fonctionnement des programmes. Si vous écrivez un programme qui n'exécute pas ce que vous attendez, souvent le problème ne se situe pas dans le programme mais dans votre modèle mental.

La meilleure façon de corriger votre modèle mental est de fractionner le programme en ses composantes (généralement les fonctions et les méthodes) et de tester chacune d'elles indépendamment. Une fois que vous avez trouvé la distorsion entre votre modèle et la réalité, vous êtes en mesure de résoudre le problème.

Bien entendu, vous devez construire et tester les composants au fur et à mesure que vous développez le programme. Si vous rencontrez un problème, il ne devrait y avoir qu'une petite quantité de nouveau code susceptible d'être incorrect.

21.3.2 J'ai une expression embroussaillée qui ne fait pas ce que j'attends

L'écriture d'expressions complexes reste une bonne pratique pour autant qu'elles restent lisibles. Néanmoins, elles sont souvent difficiles à déboguer. En général, il est judicieux de décomposer une expression complexe en une série d'instructions comportant des variables temporaires.

Par exemple, l'instruction :

```
addcard(game.hands[i], popcard(game.hands[findneighbor(game, i)]))
```

peut être reformulée en :

```
neighbor = findneighbor(game, i)
pickedcard = popcard(game.hands[neighbor])
addcard(game.hands[i], pickedcard)
```

La version explicite est nettement plus facile à lire parce que les noms des variables fournissent une documentation *per se*. Elle est aussi plus facile à déboguer car il est possible de vérifier les types des variables intermédiaires et d'afficher leurs valeurs.

Un autre problème susceptible de survenir avec les expressions de grande taille vient de l'ordre d'évaluation qui n'est peut-être pas celui auquel on s'attend *a priori*. Par exemple, l'expression $\frac{x}{2\pi}$ pourrait être écrite en Julia comme ceci :

```
y = x / 2 * π
```

Toutefois, cette formulation est erronée car la multiplication et la division ayant la même priorité sont évaluées de gauche à droite. Cette expression calcule donc $\frac{x}{2}\pi$.

Une bonne manière de déboguer les expressions consiste à ajouter des parenthèses pour rendre l'ordre d'évaluation explicite :

```
y = x / (2 * π)
```

Lorsque vous n'êtes pas sûr de l'ordre d'évaluation, utilisez des parenthèses. Non seulement le programme sera correct (dans le sens où il exécutera votre intention) mais, de surcroît, il sera lisible pour ceux qui n'ont pas mémorisé l'ordre des opérations.

21.3.3 J'ai une fonction qui ne retourne pas ce que j'attends

Si vous écrivez une déclaration de retour avec une expression complexe, vous vous privez de la possibilité d'afficher la valeur de retour avant que l'instruction `return` ne soit exécutée. Là encore, il est astucieux de tirer parti d'une variable temporaire. Par exemple, au lieu de :

```
return removematches(game.hands[i])
```

il est approprié d'écrire :

```
count = removematches(game.hands[i])  
return count
```

Ainsi, devient-il possible d'afficher la valeur de `count` avant que l'instruction `return` ne soit exécutée.

21.3.4 Je suis vraiment, vraiment coincé et j'ai besoin d'aide

D'abord, essayez de vous éloigner de l'ordinateur pendant quelques minutes. Travailler avec un ordinateur peut provoquer ces symptômes :

- frustration et rage,
- croyances superstitieuses (« l'ordinateur me déteste ») et pensées magiques (« le programme ne fonctionne que lorsque je porte ma casquette à l'envers »),
- tenter la programmation aléatoire (c'est-à-dire programmer en écrivant tous les programmes possibles et en choisissant celui qui procède de manière adéquate).

Si vous souffrez de l'un de ces symptômes, levez-vous et allez faire une promenade. Lorsque vous reviendrez au calme, pensez au programme. Que fait-il ? Quelles sont les causes possibles de son comportement ? Quand avez-vous reçu du programme de travail lors de la dernière exécution, et qu'avez-vous fait ensuite ?

Parfois, il faut juste du temps pour trouver un bug. Les très bons programmeurs trouvent régulièrement des bugs quand ils se trouvent loin de leur(s) ordinateur(s) et qu'ils laissent leur esprit vagabonder. Les meilleurs endroits pour trouver des bugs sont les trains, les douches et le lit, juste avant de s'endormir.

21.3.5 Non, j'ai vraiment besoin d'aide

Cela arrive. Même les meilleurs programmeurs se retrouvent occasionnellement bloqués. Parfois, vous travaillez sur un programme depuis si longtemps que vous ne pouvez plus voir l'erreur. Vous avez besoin d'un regard neuf.

Avant de faire appel à quelqu'un d'autre, assurez-vous que vous êtes prêt. Votre programme doit être aussi simple que possible et vous devez travailler sur la plus petite zone possible qui déclenche l'erreur. Vous devez avoir sous la main les instructions d'affichage aux endroits appropriés (et le résultat qu'elles produisent doit être compréhensible). Vous devez comprendre le problème suffisamment bien pour le décrire de manière concise et précise.

Lorsque vous appelez à l'aide, veuillez donner les informations nécessaires :

- s’il y a un message d’erreur, de quoi s’agit-il et quelle partie du programme indique-t-il ?
- quelle est la dernière modification insérée avant que cette erreur se produise ?
- quelles sont les dernières lignes de code introduites, et/ou quel est la nouvelle batterie de tests qui échoue ?
- qu’avez-vous essayé jusque là et qu’en avez-vous tiré ?

Lorsque vous trouvez le bogue, prenez un peu de temps pour réfléchir à ce que vous auriez pu faire pour le trouver plus rapidement. La prochaine fois que vous verrez quelque chose de similaire, vous pourrez trouver le bogue plus rapidement.

N’oubliez pas que le but n’est pas seulement de *faire* fonctionner le programme. L’objectif fondamental est d’*apprendre* à faire fonctionner le programme.

Annexe A : Entrées Unicode

Le tableau suivant énumère quelques caractères Unicode parmi les nombreux qui peuvent être saisis par tabulation d’abréviations de type L^AT_EX dans le REPL de Julia (ainsi que dans d’autres environnements associés à divers éditeurs).

Caractères	Séquence de complétion	Représentation ASCII
²	\^2	
₁	_1	
₂	_2	
🍏	\:apple :	
🍌	\:banana :	
🐪	\:camel :	
🍐	\:pear :	
🐢	\:turtle :	
∩	\cap	
≡	\equiv	===
<i>e</i>	\euler	
∈	\in	in
≥	\ge	>=
≤	\le	<=
≠	\ne	!=
∉	\notin	
π	\pi	pi
⊆	\subseteq	
ε	\varepsilon	

Annexe B : Installation de Julia

Situation au 1er janvier 2021

Depuis la publication du livre Think Julia [1], deux modifications importantes ont été décidées par les développeurs :

- l’abandon de la gratuité *sans restriction* de JuliaBox (voir la section 21.3.5) qui a des conséquences notamment sur l’utilisation de JuliaBox pour des cours Julia en direct,
- le gel du développement de l’environnement de développement intégré Atom/Juno –toujours installable et utilisable– et la migration vers Visual Studio Code ou vers Visual Studio Codium.

Ce double choix a amené le traducteur à modifier la section « Utilisation des exemples de codes » ainsi que l’introduction du chapitre 4 et, à proposer la présente annexe.

Julia en mode local

Julia est installable sur MS-Windows, MacOS, GNU/LINUX et BSD.

Le téléchargement se fait à partir du site [Download Julia](#). L’installation pour les différentes plateformes est décrite sur [Platform Specific Instructions for Official Binaries](#) et [Platform Specific Instructions for Unofficial Binaries](#).

Installation de binaires sous Ubuntu

Les binaires de la version stable (par exemple : julia-1.5.3) se trouvent [sous ce lien](#). En supposant que le fichier [julia-1.5.3-linux-x86_64.tar.gz](#) ait été téléchargé dans le répertoire `$HOME/Téléchargements` :

```
sudo cp Téléchargements/julia-1.5.3-linux-x86_64.tar.gz /usr/local
cd /usr/local
tar -zxvf julia-1.5.3-linux-x86_64.tar.gz && rm julia-1.5.3-linux-x86_64.tar.gz
```

Ensuite, il convient de se rendre dans le fichier `$HOME/.bashrc` et d'y ajouter ¹ :

```
PATH=/usr/local/julia-1.5.3/bin:$PATH
```

Pour pouvoir travailler avec les caractères accentués et grecs, par exemple, repérez le fichier `startup.jl` et ajoutez-y : ²

```
ENV["GKS_ENCODING"]="utf-8"
```

Installation de Visual Studio Codium

À côté d'éditeurs puissants comme Vim et Emacs, Visual Studio Codium est un auxiliaire de développement très répandu. Sous Ubuntu, l'installation procède ainsi :

```
sudo snap install codium
```

Le lancement se fait à partir de la ligne de commande (`$ codium`) ou par l'icône déposée sur le bureau, ce qui permet d'obtenir une fenêtre comme celle représentée à la figure 21.3.1. Pour obtenir cette fenêtre, il convient après l'ouverture de codium de se rendre sur l'onglet Terminal → New Terminal. L'écran principal est divisé en deux parties dont un terminal `bash`. Derrière l'invite, saisissez `julia`.

À ce stade, cliquez sur le bouton Extensions (5^{ème} bouton de la barre verticale gauche) et installez Julia 1.0.10 – Julia Language Support (ce numéro n'est pas lié au numéro de version de Julia proprement-dit).

Installation de modules

Tout au long de l'utilisation de Julia, il sera nécessaire d'installer des paquets (modules, installation) complétant la bibliothèque standard.

L'installation de nouveaux paquets se fait dans le REPL de VSCodium (*Read-Eval-Print Loop*) qui se trouve dans la partie inférieure de la figure 21.3.1 (la hauteur de cet espace de travail est ajustable).

1. Pour que la modification de fichier soit prise en compte, il est nécessaire de sortir de la session et d'y revenir. Ceci permet de relire `.bashrc`.

2. Ceci peut se faire avec l'utilisation des commandes : `sudo updatedb` suivi (en mode utilisateur) de `locate`.

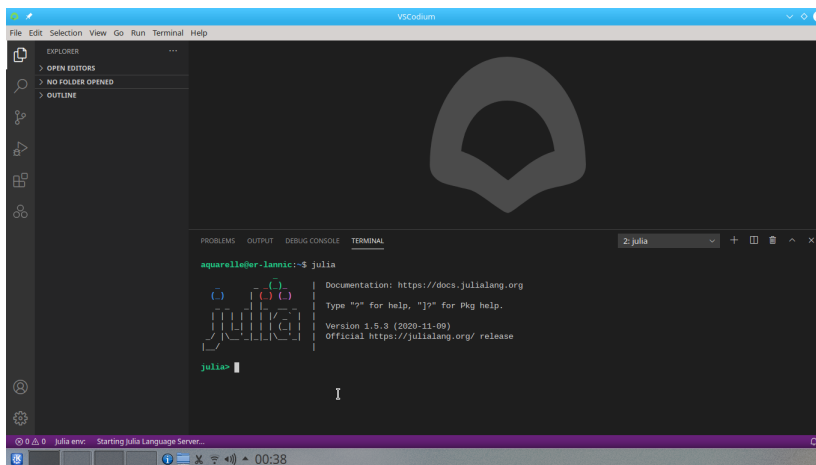


FIGURE 21.3.1 – Écran de Visual Studio Codium.

Ancienne Méthode

Si, par exemple, on souhaite utiliser le paquet [Plots](#) (voir la sous-section 13.12.1 pour réaliser des graphiques), il est possible d'appliquer la méthode suivante. Dans le REPL et une session active :

```
julia> using Pkg
Pkg.add("Plots")
```

L'installation est automatique. Julia se charge d'établir les dépendances, les télécharger et les installe dans le répertoire [\\$HOME/julia/packages](#).

Nouvelle Méthode

Il existe toutefois une méthode plus moderne de procéder. Dans le REPL, derrière l'invite `julia>`, saisissons le caractère `]` (Alt-J) :

```
julia> ]
```

Immédiatement, Julia nous fait basculer en mode gestionnaire de paquets (en l'occurrence pour la version 1.5) :

```
(@v1.5) pkg>
```

Après quoi, il est possible de demander l'incorporation d'un paquet. Par exemple, pour le paquet [Plots](#) :

```
(@v1.5) pkg > add Plots
```

Pour quitter le gestionnaire de paquets, on peut utiliser la touche *backspace* ou la combinaison [Ctrl-C](#).

Carnets de travail : Jupyter et Pluto

Jupyter

Le carnet Jupyter (Jupyter Notebook) est un éditeur de feuilles de calcul qui fonctionne en local dans un navigateur. Il permet d'exécuter sur une même feuille des commandes et d'afficher leurs résultats, y compris des graphiques.

Sous Ubuntu, l'installation procède selon :

```
sudo apt-get install jupyter
```

De retour dans VSCode, Julia activé et en mode gestionnaire de paquet :

```
(@v1.5) pkg > add IJulia
```

Pour lancer Jupyter, dans un terminal bash de VSCode :

```
jupyter notebook
```

Le navigateur présente une feuille du carnet Jupyter. En cliquant sur le bouton « Nouveau » (coin supérieur droit de la feuille) → Julia-1.5.3, il est possible d'entrer du code. La figure 21.3.2 donne un exemple.

Pluto

Pluto est un carnet —écrit en Julia— et dont les caractéristiques principales sont :

- la réactivité : lors de la modification d'une fonction ou d'une variable, Pluto met automatiquement à jour toutes les cellules affectées,
- le caractère intuitif et moderne : Pluto permet de créer des documents exportés, de belles factures avec des thèmes personnalisés.

Pour installer Pluto :

```
(@v1.5) pkg > add Pluto
```

Après quoi, de retour dans Julia :

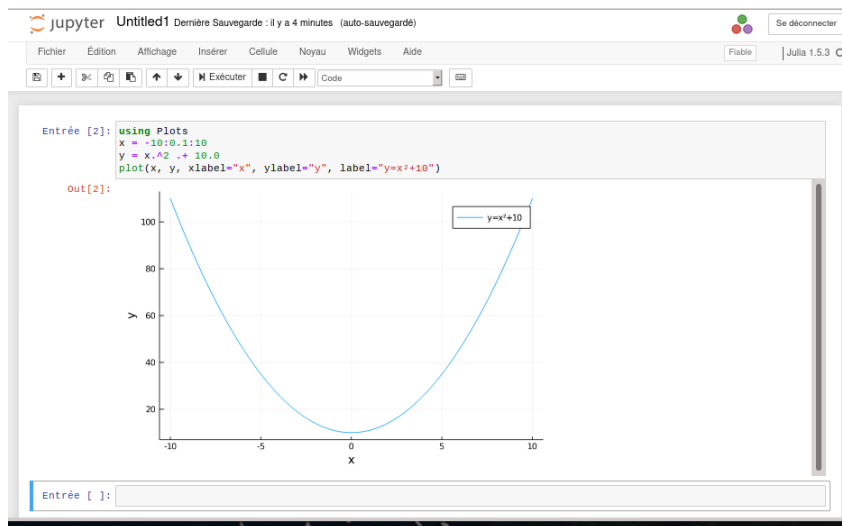


FIGURE 21.3.2 – Exemple d’une feuille de carnet Jupyter.

```
julia> using Pluto
julia> Pluto.run()
```

La dernière commande ouvre une page du carnet dans votre navigateur. **Ctrl-C** permet de quitter Pluto et le retour dans une nouvelle session Julia.

Julia en nuage

Il est possible d’utiliser Julia sur le *cloud* à l’intervention de NextJournal, CoCalc, mybinder ou Repl.it, par exemple.

Aide et documentation

Outre quelques livres cités dans la bibliographie, notamment les références [1–5] et [10] (sans que cela ne soit exhaustif), le lecteur peut se référer à la documentation officielle pour laquelle existe un document pdf téléchargeable. Par ailleurs, Julia Discourse offre un espace de discussions techniques et d’aide extrêmement précieux.

Annexe C : Notes de traduction

Plusieurs modifications, décrites ci-après, ont été introduites par rapport au document original.

Notes de bas de page

Toutes les notes de bas de page sont des compléments associés à la traduction sauf en ce qui concerne les notes associées aux pages 15, 74 (note 3), 86, 110, 182 et 224 (note 2), qui proviennent du document original.

Modifications du texte

Les sections suivantes ont été ajoutées au texte original en anglais :

- « Droits associés à la version française »,
- « Préface de l'édition française ».

Des modifications du texte original ont eu lieu dans :

- les sections « Utilisation des exemples de code » en xxix et 2.4 page 14 (pour faire face à la situation décrite à la section 21.3.5) ainsi que dans la section 11.2 page 148 (pour expliciter pas à pas le fonctionnement de la fonction `histogram("brontosauure")`),
- le choix du répertoire de mots français (`mots_FR.txt`, source : référence [8], chapitre 9) au lieu du répertoire de mots anglais `words.txt`,
- le choix du roman *Notre Dame de Paris* (section 13.3) au lieu du roman *Emma* utilisé dans l'édition anglaise,
- le choix d'un extrait du poème *Liberté* (section 13.8) au lieu d'*Eric, the Half a Bee* dans l'édition anglaise.

Modifications des exemples de programmes

Aucune modification de noms de fonctions ou structures composites n'a eu lieu, sauf dans le chapitre 18 pour des raisons de clarté. En revanche, les arguments passés aux fonctions ou les paramètres des structures composites ont été francisés.

Ajouts de figures

Les figures 4.1.1, 4.1.2, 8.2.1, 8.6.1, 10.2.1, 11.1.1, 11.5.1, 21.3.1 sont des ajouts associés à la traduction.

Un croquis a été inséré dans la section 7.1.

Compte tenu de la situation décrite à la section 21.3.5, un écran de Jupyter (figure 21.3.2) fonctionnant en mode local sans JuliaBox substitue l'écran de Jupyter de l'édition originale en anglais.

Modifications de tableaux

Les tableaux 2.1 et 5.1 sont une mise en forme des informations originales.

Bibliographie

Dans la version française, la bibliographie a été concentrée dans une partie *ad hoc*. Les citations [1–5], [8] et [10] constituent des ajouts associés à la traduction.

Bibliographie

- [1] Ben Lauwens et Allen B. Downey, *Think Julia: How to Think Like a Computer Scientist*, O'Reilly, Sebastopol CA95472, 2019
- [2] Georg Root, *The Julia Language Handbook*, publication indépendante, 2019
- [3] Andrian Salceanu, *Julia Programming Projects: Learn Julia 1.x by building apps for data analysis, visualization, machine learning, and the web*, Packt, Birmingham, 2018
- [4] Ivo Balbaert, Adrian Salceanu, *Julia 1.0 Programming Complete Reference Guide: Discover Julia, a high-performance language for technical computing*, Packt, Birmingham, 2019
- [5] Olivier Garet, *Introduction à Julia: Programmer des mathématiques*, publication indépendante, 2020
- [6] Steve Oualline, *Practical C Programming*, O'Reilly Media, 1997
- [7] Michael Sipser, *Introduction to the Theory of Computation*, 3th. Ed., Cengage Learning, 2012
- [8] Christophe Pythoud, *Français-GUTenberg: un nouveau dictionnaire français pour ISPELL. Problèmes résolus et intégration de contributions extérieures*, Cahiers GUTenberg, n° 28-29, pp. 252-275, 1998
- [9] Brian K. Kernighan et Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999
- [10] Avik Sengupta, *Julia High Performance*, Packt, Birmingham, 2nd. Ed., 2019

Index des fonctions extrinsèques

+, 238
+, dispatch multiple, 235
+, surcharge, 235
<, méthode, 246
<, surcharge, 243

absvalue, 68, 69
ack, 81
addall, 128
adding, 254
addtime, 220, 223, 224, 226
anagramsets, 207
anylowercase1, 109
anylowercase2, 109
anylowercase3, 109
arc, 42, 45, 50
area, 67
avoids, 114, 116, 276

baddeletehead, 137

capitalizeall, 129, 131
Carte, 242, 244
cattwice, 28, 29
checkfermat, 64
choosefromhist, 181
circle, 42, 49
circle, 45
circlearea, 72
classify, 256

compare, 69
containervariable, 268
convert, 266
count, 103
countdown, 58, 60, 85
cumulsum, 140

deal, 254
deletehead, 135
differentwords, 182
distance, 69, 70, 72
distancebetweenpoints, 214
dofour, 34
dotwice, 33
DoubleJour, 227
draw, 66
drawcircle, 218
drawrect, 218

eleve_au_carre, 27
estimatepi, 93
evalloop, 93
example1, 157
example2, 157
example3, 158
example4, 158
example5, 158

f(), 232
fact, 74, 75, 77–79

- fibonacci, 155
- find, 102, 103
- findcenter, 215
- first, 81
- firstindex, 98
- grid, 34
- hasduplicates, 142, 162, 275
- hasmatch, 171
- hasno_e, 114, 115
- haspair, 256
- hastwopair, 256
- histogram, 148, 150, 151, 237
- hypotenuse, 71
- inbisect, 142
- inboth, 104
- increment, 222–224, 231
- interior, 141
- inttotime, 223, 224, 231
- invertdict, 154, 161
- isabecedarian, 115, 117, 118
- isafter, 220, 232
- isanagramme, 141
- isbetween, 73
- isdivisible, 72, 73, 76
- isintpoint, 264
- ispalindrome, 82, 109, 119
- ispuissance, 82
- isreverse, 105, 106, 119
- issort, 141
- istriangle, 65
- isvalidtime, 225
- iterate, 280
- last, 81
- linecount, 204
- LineCount, module, 205
- Markov, 252
- middle, 81
- mostcommon, 183
- mostfrequent, 176
- move, 248
- movepoint, 213, 214
- moverectangle, 214
- multime, 226
- mysqrt, 92
- MyTime, 233
- nestedsum, 140
- onlyupper, 130
- Paquet, 244, 245, 247
- pgcd, 82
- pointincircle, 217
- polygon, 41, 43, 44
- polyline, 46, 47, 50
- Polynomial, 265
- Pong, 254
- pop, 248
- pouchcontents, 240
- print, 34
- printall, 169
- printhist, 151
- printlyrics, 24, 25
- printmostcommon, 184
- printn, 59, 60, 87
- printpoint, 213
- printspam, 34
- printtime, 220, 230, 231, 239
- printtwice, 28–30
- processfile, 181, 182, 185
- processline, 181, 182
- processword, 252
- push, 248
- putinpouch, 240

randomword, 186
readanagrams, 207
rectcircleoverlap, 217
rectincircle, 217
recurse, 60, 65
reinterpret, 263
repeatlyrics, 24, 25
returnfloat, 229
reverselookup, 153
reversepairs, 143
rightjustify, 33
rotateword, 110

sed, 207
seq, 86
shuffle, 248
sinc, 230
sort, 246
square, 41–43, 269
squaresum(), 281
storeanagrams, 207
subtract, 185
subtract(), 274
sumall, 170

testsquareroot, 92
time, 231
timetoint, 223, 224, 238
totalwords, 182
type, annotation, 229

UneMain, 247
useall, 116, 117
useonly, 115, 116, 275, 277

walk, 198

Index

!, 55
!=, 55
#, commentaire, 17
\$, 101
\$, interpolation, 101, 197, 204
<, 55
<=, 55
>, 55
>=, 55
||, 55
 α , 88
 \div , 53, 63
 \equiv , 133, 215
 \geq , 55
 \in , 104, 126, 142, 147
 \leq , 55
 \neq , 55
 \notin , 116, 155
★, 249
 ε , 90
%, 54
&&, 55, 280
(@v1.x) pkg, 37, 302
*, 3
*, 16
+, 3
+=, 129
::, opérateur, 229
-, 3
->, 258, 259
.
.bashrc, 300
.jl, 14
/, 4
/, 53
:, 61
=, 54, 286
==, 54, 73, 215, 286
===, 133
=>, 146
=>, dictionnaire, 146
?, 47
?, caractère, 47
?, opérateur ternaire, 261
@, 268
@assert(), 225
@code_llvm, 281
@code_lowered, 281
@code_native, 282
@code_typed, 281
@debug, 282, 288
@generated, 269
@printf(), 197, 220
@show, 70, 106, 149, 168, 260
@svg, 38, 39, 259, 260
@test, 244
@time, 273
@warn, 282
@which, 251, 281
[n:m], 100
évaluation en court-circuit, 261

-], 37, 301
- \wedge , 16
- \wedge , 4
- \neg , 12
- $\setminus 1$, 70
- { }, type paramétrique, 263
- [], 96, 123, 146
- [], tuple, 166
- abécédaire, 115
- abs, 50
- abs(), 69, 90
- abspath(), 197
- abstract type, 13
- AbstractDict(), 290
- accumulateur, 129, 139
- Ackermann, fonction d', 81
- add, 37
- addition, 16
- affectation, 18
- affectation de tuple, 167, 175
- affectation, déclaration, 11
- agrégation, 175
- agrégation, paramètre, 169
- aka, 134
- aléatoire, 180
- algorithme, 90, 92
- alias, 134, 214
- aliasing, 134, 140, 174, 215
- aliasing, persistance, 135
- \backslash alpha, 88
- ambiguïté, 6
- anagramme, 141, 176, 207
- analyse, 8
- analyse comparative, 189, 192
- analyse syntaxique, 6
- Annaud, Jean-Jacques, 221
- anniversaire, paradoxe des, 142
- annotation de type, 229, 230, 239
- appel, graphe, 161
- appel, graphe d', 156
- appel, trace, 33
- appel, trace d', 33
- append!(), 128
- Archimède, spirale, 51
- args..., 169
- argument, 21, 26, 32
- argument annoté, 238
- argument facultatif, 140
- argument nommé, 259, 271
- argument optionnel, 232
- argument, validité, 225
- ArgumentError, 290
- arguments multiples, tuple, 169
- arrondi, 22
- ASCII, 95, 107, 297
- association, 260
- Atom, xxix, 299
- attribut, 217
- baremodule, 13
- Base, 257
- base de données, 195, 200, 206
- base sexagésimale, 224
- Base.<, 243
- Base.convert(), 266
- Base.iterate, 280
- Base.pop\subitem(), 245
- Base.push\subitem(), 245
- Base.show, 234, 243
- Base.show(io::IO, ...), 245
- bash, 14, 101
- begin, 13, 259
- begin, 38
- Benchmark Tools, 189
- benchmarking, 189, 192
- BenchmarkTools, 274
- BenchmarkTools.jl, 189

- Bezanson, Jeff, xxi
- bibliothèque standard, 257
- bissection, 91, 142
- bloc, 259
- bloc do, 260, 271
- bloc let, 260, 270
- bogue, 8, 120
- Bool, type, 54
- boucle, 49
- boucle infinie, 86, 92, 288
- boucles, indices, 117
- BoundsError, 126, 290
- box, 215
- branche, 63
- break, 13, 87, 92, 263, 280
- BSD, 299
- bug, 8, 120
- C, 270
- canal, 262
- canal et itérateur, 263
- canard en plastique, méthode du, 191, 192
- canevas, 71, 80
- Car Talk, 120, 121, 162, 177
- caractère, 95
- caractère de retour, 206
- caractères, 297
- carnet, xxix
- Carroll, Lewis, 74
- cas atypique, 120
- cas de base, 64, 289
- cas particulier, 120
- catch, 13, 199, 206
- ccall, 270
- cd, 203
- chaîne conditionnelle, 64
- chaîne de caractères, 4
- chaîne multiligne, 286
- chaîne vide, 108
- chaîne, interpolation, 101, 108
- chaîne, persistance, 100
- chaîne, segment, 99
- chaîne, traversée de, 98, 107
- chaînes, 95, 277
- chaînes, bibliothèque, 103
- chaînes, comparaison, 104
- chaînes, documentation, 108
- champ, 217
- Channel, 262
- Channel, constructeur, 262
- Char, 8, 96
- char, 108
- chemin absolu, 197, 206
- chemin d'accès, 206
- chemin relatif, 197, 206
- clé, 160
- clé-valeur, 145, 160
- clode(db), 270
- close(), 113, 200
- CMU Pronouncing Dictionary, 163
- co-routine, 262
- CoCalc, 303
- codage, 242
- code bas niveau, 281
- code César, 110
- code machine, 282, 283
- code mort, 80
- Codium, xxix, 2, 14, 299, 300
- codium, 300
- Collatz, conjecture de, 86
- collect(), 132, 152
- collect(), zip(), 170
- commentaire, 17, 19, 47
- complétion, séquence de, 297
- composition, 23, 33, 71
- compteur, 103, 108
- concaténation, 16, 101
- concaténer, 19

- condition, 55, 63
- condition *a posteriori*, 49
- condition *a posteriori*, 48, 79
- condition *a priori*, 48, 49, 79
- condition préalable, 48, 49
- conditions imbriquées, 57
- const, 13, 159, 243
- constante globale, 161
- constructeur, 210, 211, 217, 233, 265
- constructeur de copie, 233
- constructeur externe, 239
- constructeur interne, 233, 239
- constructeur par défaut, 239
- constructeur, copie, 239
- constructeur, méthode externe, 233
- constructeur, méthode par défaut, 233
- constructeur, signature, 233
- continue, 13, 88, 92
- conversion, 266, 271
- copie, 215
- Core.eval(), 92
- coroutine, 271
- corps, 24, 32
- cos(), 23, 276
- count(), 108
- count(), documentation, 108
- court-circuit, 271
- current working directory, 197
- Cvoid, 270
- cwd, 197
- débogage et sous-typage, 250
- déclaration d'importation, 235
- définition de fonction, 230
- dépendance, 249, 253
- Database Manager, 200
- DataStructures.jl, 190
- DBM, 200
- dbm, 270
- débogage, 7, 8, 18, 31, 48, 62, 79, 91, 105, 119, 138, 159, 175, 190, 205, 216, 225, 239, 250, 282, 285
- déclaration, 8, 13, 19, 161
- déclaration conditionnelle, 63
- déclaration d'affectation, 11
- déclaration d'affichage, 291
- déclaration d'utilisation, 49
- déclaration globale, 158, 161
- décrémentation, 92
- deepcopy(), 215, 217
- deleteat!(), 131
- deletehead!(), 135
- délimiteur, 132, 140
- deserialize(), 202
- déterminisme, 180
- déterministe, 192
- développement incrémental, 69, 80
- développement progressif, 69, 80
- développement, plan de, 47
- diagramme d'état, 12, 19, 155
- diagramme d'objet, 210, 213, 217, 220
- diagramme de pile, 28, 33, 59, 75, 136
- diagramme de type, 249, 253
- Dict(), zip(), 172
- dictionnaire, 145, 160
- dictionnaire et tableau, 153
- dictionnaire et tuple, 172
- dictionnaire inversé, 154
- dictionnaire, compteur, 148
- dictionnaire, inversion, 154
- diff, 207
- Dijkstra, Edsger W., 120
- Disparition, La*, 114
- dispatch, 239
- dispatch multiple, 229, 235, 238, 240
- dispersion, 169, 175
- distributivité, 130
- DivideError, 290

- division, 16
- division décimale, 4, 53, 63
- division entière, 53
- division euclidienne, 53, 63
- divrem(), 168, 223
- do, 13, 271
- docstring, 47, 49
- documentation interne, 47
- documentation officielle, 273
- DomainError, 290
- données, encapsulation de, 251, 253
- données, structure de, 95, 175
- dot, 130, 139
- dot, opérateur, 130
- dot, syntaxe, 139
- Downey, Allen B., xxi, xxiii
- drapeau, 157, 161
- dump(), 205, 267
- duplication, 217

- eachindex(), 126
- Edelman, Alan, xxi
- EDI, 190
- élément, 139, 145
- élément, tableau, 123
- else, 13, 56, 57
- elseif, 13, 56
- Éluard, Paul, 187
- Emacs, 14
- emboîtement de deux mots, 143
- emoji, 96
- en-tête, 24, 32
- encapsulation, 42, 47, 49, 251
- encapsulation de données, 253
- encapsuler, 248
- enchaînement de conditions, 56
- encoder, 253
- end, 13, 24, 96, 199, 204, 210, 286
- enumerate(), 171

- environnement de développement, 190
- EOFError, 290
- équivalence, 134, 140
- équivalence, non-identité, 134
- erreur d'exécution, 287
- erreur de forme, 175
- erreur de syntaxe, 12, 286
- erreur sémantique, 292
- error, 153
- Error Result, @test, 244
- ErrorException, 153, 203
- esc(), 268
- espace de stockage, 190
- euclidienne, division, 53
- Euler, identité, 276
- eval, 268
- exécution alternative, 56
- exécution conditionnelle, 55
- exécution, erreur d', 18, 19
- exception, 18, 19, 290
- exception, levée, 78, 199
- exp(), 23, 67
- exponentiation, 16
- export, 13, 205
- Expr, type, 267
- expression, 13, 19, 271
- expression booléenne, 54, 63
- expression rationnelle, 277
- Extensions, 300
- extrema(), 168

- factoriel, 74
- false, 54, 72
- fanion, 157, 161
- Fermat, théorème de, 64
- fermeture, 259, 270
- fibonacci(), 156
- Fibonacci, suite de, 77, 156, 280
- fichier, 195

- fichier texte, 206
- fieldnames(), 216
- filtre, 128, 130, 140
- fin de ligne, 206
- finally, 13, 200, 206
- findall(), 153
- findfirst(), 103
- findnext, 104
- firefox, 203
- flèche à extrémité creuse, 249
- flèche à extrémité pleine, 249
- flag, 157, 161
- Float, 8
- float(), 22
- Float64, 4
- flux d'exécution, 26, 33, 288, 289
- flux de fichier, 120
- foncteur, 265, 271
- fonction, 21
- fonction anonyme, 258, 261, 271
- fonction avec retour, 30, 32, 67
- fonction booléenne, 72
- fonction de hachage, 160
- fonction et méthode, 252
- fonction factorielle, 74
- fonction générée, 269
- fonction mathématique, 22
- fonction nulle, 30, 67
- fonction polymorphique, 237
- fonction pure, 220, 226, 232
- fonction récursive, 59
- fonction vide, 30, 67
- fonction void, 32
- fonction, appel de, 21, 26, 32
- fonction, définition, 24, 32
- fonction, distributivité, 130
- fonction, objet de, 32
- fonction, syntaxe compacte, 258
- fonctionnement, 190
- for, 13, 40, 85, 286
- Fortran, 270
- forward(), 38
- fréquence, 149, 153
- Français-GUTenberg, 113
- function, 13, 24
- généricité, 237, 240
- gauche ↔ droite, 74
- GDBM, 200
- gdbm_close, 270
- \ge, 55
- généralisation, 43, 47, 49
- get!(), documentation, 161
- get(), 151
- Git, xxix
- GitHub, xxix
- global, 13, 98, 157, 161
- globale, variable, 136
- GNU/LINUX, 299
- golden, 30
- graphe d'appel, 156, 161
- Greenfield, Larry, 32
- guillemet triple, 48
- guillemets bouclés, 286
- guillemets droits, 3, 286
- guillemets inversés, 203
- guillemets, simples, 24
- GUTenberg, projet, 113
- hachage, 155
- hachage, fonction, 160
- hachage, table, 160
- hachage, valeur de, 155
- handle, 270
- hash function, 160
- hash table, 147, 160
- help?>, 47
- histogramme, 149, 151, 154, 181

- homophonie, 163
- Hugo, Victor, 181
- IDE, 190
- identité, 134, 140
- if, 13, 55, 286
- if, expression booléenne, 55
- IJulia, 302
- IJulia, installation, 302
- imbrication conditionnelle, 64
- imbrication de conditions, 57
- immuabilité, 100
- implémentation, 148, 160
- import, 13, 205, 273
- import Base.<, 243
- importall, 13
- importation, déclaration d', 235
- in, 13, 126
- \in, 104
- include(), 204
- incrémentation, 92, 103, 129, 139
- indice, 96, 107
- InexactError, 290
- Inf, 63
- initialisation, 92
- insert !(), 132, 138
- insertion, 131
- instance, 210, 215, 217
- instance en argument, 213
- instanciation, 133, 210, 212
- instruction, 13
- instruction composée, 56, 63
- Int, 8
- Int64, 4
- InteractiveUtils, 281
- interface, 44, 49, 239, 279
- interface ↔ implémentation, 238
- interpolation, 108
- interpolation de chaîne, 101
- introspection des signatures, 239
- invalid token, 286
- invariant, 211, 225, 226, 233
- invite, 7
- io, 243
- iobuffer, 202
- isa(), 216
- isa, opérateur, 247
- isafter, 238
- isdefined(), 216
- isdir(), 198
- isfile(), 199
- ispath(), 199
- itérateur, 170, 175, 181
- itération, 83, 92
- itérer, 170
- item, 139, 160
- item, tableau, 123
- Jabberwocky, 74
- JLD2, 203
- JLD2, documentation, 203
- join(), 132
- joinpath(), 199
- journalisation, débogage, 283
- julia, 2
- Julia Discourse, 303
- Julia Discourses, xxvii
- Julia Language Support, 300
- Julia Observer, 37
- Julia, cloud, 303
- Julia, documentation officielle, 303
- JULIA_DEBUG, 283
- JuliaBox, xxix, 299
- JuliaCollections, 190
- Juno, xxix, 299
- Jupyter, 14, 49, 302
- Jupyter, installation, 302

- Karpinski, Stefan, xxi
- Kernighan, Brian K., 188
- KeyError, 290
- keys(), 147
- lâcher prise, débogage, 191
- langage formel, 5
- langage naturel, 5
- lastindex(), 106
- Lauwens, Ben, xxi, xxiii
- le, 55
- Le nom de la rose*, 221
- length(), 33, 97
- length, tableau, 126
- let, 13, 260, 270
- levée d'exception, 78, 199
- Liberté*, 187
- libgdm, 270
- ligature, 260
- LINUX, 32
- lipogramme, 114
- littéralité, 6
- LLVM, 281
- ln, 22
- LoadError: syntax, 285
- local, 13
- locale, variable, 136
- locate, 38, 300
- log(), 22
- log10(), 22, 62
- logarithme népérien, 22
- logarithme naturel, 22
- logarithme, base 10, 22
- Logging, 282
- lookup, 152, 161
- Low Level Virtual Machine, 281
- lowercase(), 182
- lpad(), 121
- ls, 203
- Luxor, 38
- méta-programmation, 267
- méthode, 230, 231, 238, 239
- méthode et fonction, 252
- méthode multiple, 232
- méthode paramétrique, 264
- MacOS, 299
- macro, 13, 39, 268, 271
- Main, 28, 60, 98, 157
- mapping, 128, 130, 140, 145, 160
- Markov, analyse de, 186
- match(), 277
- matrice, 278, 283
- max(), tuple, 169
- maximum(), 168
- md5, 204, 207
- md5sum, 204, 207
- mémo, 155, 161
- Meta.parse(), 92, 267
- métathèse, 177
- MethodError, 290
- methods(), 239
- min(), tuple, 169
- minimum(), 168
- minmax(), 168
- mise à jour, 92
- mise en correspondance, 130, 140, 145, 160
- missing, objet, 269
- Missing, type, 269
- MIT, licence, xxi
- mode écriture, 196
- mode interactif, 19
- modificateur, 222, 226, 232
- module, 13, 37, 49, 204, 273
- module(s), 273
- modules, 300
- modulo, 53, 63

- mot-clé, 19
- mots, fréquence, 179
- mots, occurrence, 179
- mots-clés, liste, 12
- mp3, 207
- MS-Windows, 299
- multiméthode, 236, 240
- multiplication, 16
- multiplicité, 249, 253
- mutabilité, 211
- mutable struct, 13
- MyTime, 219

- \n, 206
- \ne, 55
- new, 233
- new, méthode sans argument, 234
- new, première méthode, 233
- NEWLINE, 113
- Newton, méthode de, 88
- nextind(), 98
- NextJournal, 303
- nombre aléatoire, 180
- nombre complexe, 276
- nombre irrationnel, 90
- nombre rationnel, 90
- non-ASCII, caractère, 287
- non-persistance, structure, 211
- notation par ., 211
- notebook, xxix
- nothing, 31, 33, 69
- \notin, 116
- Notre Dame de Paris*, 181
- nouvelle ligne, 206

- obèle, 53
- objet, 133, 140
- objet aliasé, 135
- objet callable, 265
- objet de commande, 206
- objet enchassé, 217
- objet et instance, 210
- objet intégré, 213, 217
- objet persistant, 211
- objet, diagramme, 210, 213, 217, 220
- objet, diagramme d'état, 133
- objet, structure de données, 133
- occurrence, 149, 153
- official binaries, 299
- opérande, 19
- opérateur, 8
- opérateur ::, 229
- opérateur ., 279
- opérateur logique, 55, 63
- opérateur point, 130, 139
- opérateur relationnel, 54, 63
- opérateur ternaire, 271
- opérateur ternaire ?, 261
- opérateur, distributivité, 130
- opérateur, surcharge, 235
- opérateurs arithmétiques, 3
- opérateurs, préséance, 15
- opérateurs, priorité, 15
- opération de réduction, 129
- open(), 113, 196, 199, 261
- Oualline, Steve, 34
- OutOfMemoryError, 290
- outrepassement, 192
- OverflowError, 290
- override, 192

- périmètre d'application, 261
- paire clé-valeur, 160
- palindrome, 81
- Pallier, Christophe, 113
- paquet, 49
- paquet(s), 273
- paquets, ancienne méthode, 301

- paquets, gestionnaire, 301
- paquets, installation, 300
- paquets, nouvelle méthode, 301
- paramétrique, type, 271
- paramètre, 26, 32
- paramètre d'agrégation, 169
- parcourir un tableau, 126
- parenthèses, 16
- parse, 8, 62
- parse(), 21, 26
- PATH, 300
- PEMDAS, 16
- pendown, 39
- penup, 39
- Perec, Georges, 114
- performances, mesure de, 273
- persistance, 100, 108, 206
- persistance, aliasing, 216
- persistance, programme, 195
- persistance, tuple, 165
- PGCD, 82
- \pi, 11
- pictogramme, 96
- Pike, Rob, 188
- pile, diagramme de, 28, 33
- pile, trace de, 29
- Pkg, 37
- Pkg.add(), 301
- placage, 246, 253
- plan de développement, 49
- planification, 226
- plot(), 258
- Plots, 301
- Plots(), 193
- Plots, fonction anonyme, 258
- Plots, installation, 301
- Pluto, 14, 49, 302
- Pluto, installation, 302
- Pluto.run(), 302
- point, syntaxe, 139
- pointeur C, 270
- polymorphisme, 237, 238
- pop, 245
- pop!(), 131, 138
- popfirst!(), 131, 136, 138
- préséance d'opérateurs, 293
- prevind(), 106
- primitive type, 13
- printf(), 197
- printf(), documentation, 197
- println(), 3, 21
- profile, 189
- profile, documentation, 189
- Profile.jl, 189
- ProfileView.jl, 189
- profondeur de récursion, 61
- programmation aléatoire, 191
- programmation fonctionnelle, 222
- programmation générique, 229, 237, 240
- programme, 2
- programme suspendu, 288
- projet GUTenberg, 179
- promote(), 267
- promote_rule(), 267
- promotion, 267, 271
- prototypage↔planification, 223
- prototype et correctif, 220, 226
- pseudo, pseudonyme, 134
- pseudo-aléatoire, 180, 192
- pseudonymie, 134, 140
- Ptr{Cvoid}, 270
- push, 245
- push!(), 127, 131, 136, 138, 155, 183
- pushfirst!(), 131, 138
- pwd(), 197
- Pythagore, théorème de, 69, 211
- Pythoud, Christophe, 113

- quote, 13
- quotient, 53
-
- \r, 206
- racine carrée, 88
- Ramanujan, Srinivasa, 93
- rand(), 180
- Random.shuffle, 246
- read(), 203
- Read-Eval-Print Loop, 2
- readdir(), 198
- readline(), 61, 87, 113
- réaffectation, 83, 92, 157
- recherche, 102, 108, 115
- recherche binaire, 142
- recherche directe, 152, 161
- recherche inverse, 152, 161
- rect.coin.x, 215
- réursion, 53, 58, 64, 73, 85
- réursion infinie, 60, 64, 289
- redondance, 6
- réduction, 117, 120, 128, 139
- réduction, opération de, 129
- refactoring, 45, 49
- référence, 135, 140
- référencement, 135
- réflexion, 190
- refonte, 45, 49, 252
- regex, 283
- Regex(), 277
- règles de préséance, 19
- relation de dépendance, 249
- relation HAS-A, 249, 253
- relation IS-A, 249, 253
- relecture, 190
- répertoire, 206
- répertoire courant, 197
- REPL, 2, 3, 7, 19, 47, 287
- replace(), 108
- repr(), 205
- représentation intermédiaire, 283
- retour, valeur, 21
- return, 13, 59, 67, 102
- rev(), sort(), 184
- reverse lookup, 152, 161
- reverse !(), 174
- Revise.jl, 205
- ROT13, 110
- rubber ducking, 191, 192
- run(), 203
-
- savefig(), 193
- scaffolding, 71, 80
- script, 14, 19
- search, 47
- segment, 108
- segment de chaîne, 99
- segmentation, 137, 279
- sémantique, 19
- sémantique, erreur, 18, 19
- sentinelle, 78, 80
- séquence, 95, 107, 167
- séquence de séquences, 174
- séquence indicée, 96
- séquence, tableau, 123
- serialize(), 202
- Set, 185, 290
- set, 275
- Set(), 275
- Set, structure de données, 185
- setdiff(), 275
- sexagésimale, base, 224
- Shah, Viral B., xxi
- shell, 203, 206
- show, 234, 243
- show(), 30
- shuffle, 246
- signature, 239

- signature spécifique, 231
- sin(), 23, 67
- singleton, 155, 161
- Sipser, Michael, 74
- size(), 278
- sizeof(), 97
- skipmissing(), 269
- slice, 99, 108
- slurp, 169, 175
- sort!(), 128, 138, 174
- sort(), 128, 139, 152
- sort, documentation, 184
- sous-typage, 241, 246, 253
- sous-type, 247, 253
- soustraction, 16
- soustraction d'ensembles, 184
- spirale, 51
- splat, 169, 175
- splice!(), 131
- split(), 132, 168
- sqrt(), 23, 30, 70
- StackOverflowError, 285, 288–290
- stacktrace, 29, 33, 61, 290
- Standard Library, 257
- start, 108
- startup.jl, 300
- StdLib, 220
- STDOUT, 203
- stochasticité, 180
- stockage permanent, 195
- stockage, espace de, 190
- String, 4, 8
- string, 108
- string!(), 138
- string(), 22
- StringIndexError, 291
- strip(), 108
- Struct, 210
- struct, 13, 212, 217, 219
- structure, 210
- structure de contrôle, 261
- structure de données, 95, 175
- structure, attribut, 210
- structure, champ, 210
- structure, persistance, 210
- structures et fonctions, 219
- structures et objets, 209
- \subsetq, 276
- \subseteq , 276
- sum(), 129, 182
- sum(), tuple, 169
- supertype, 247, 253
- supertype(), 251
- suppression, 131
- surcharge d'opérateur, 235, 240
- SVG, 39
- syntaxe, 8
- syntaxe, erreur de, 12, 18, 19, 286
- SystemError, 199, 291
- \t, 183
- tâche, 262, 271
- TAB, 11
- table de hachage, 147, 160
- Table UPLEt, 165
- tableau, 123, 139, 278
- tableau bidimensionnel, 278
- tableau et dictionnaire, 153
- tableau et tuple, 170
- tableau imbriqué, 123, 126, 139
- tableau vide, 124, 127
- tableau, copie de, 127
- tableau, diagramme d'état, 125
- tableau, dimension, 124
- tableau, indice, 125
- tableau, non-persistance, 124
- tableau, segment, 127
- tableau, segmentation, 127

- tableau, traversée, 126
- tableaux, bibliothèque, 127
- tables d'indice, 96
- tabulation, \t, 183
- take\subitem(), 263
- tan(), 23
- téléchargement, 299
- Test Failed, 244
- Test Passed, 244
- test unitaire, 244, 253
- ThinkJulia, 200, 270
- ThinkJulia, module, 37
- ThinkJuliaFR, 200, 270
- ThinkJuliaFR, module, 37
- time, 219
- time(), 64
- Torvalds, Linus, 32
- trace d'appel, 33, 61
- trace de pile, 29, 61, 290
- traversée de chaîne, 98
- true, 54, 72
- trunc(), 22, 45
- try, 13, 199, 206
- tuple, 165, 175
- tuple et dictionnaire, 172
- tuple et tableau, 170
- tuple nommé, 257, 271
- tuple(), 166
- tuple, affectation, 167
- tuple, affectation de, 175
- tuple, diagramme d'état, 173
- tuple, persistance, 165
- tuple, [], 166
- Turing, Alan M., 74
- Turing, thèse de, 74
- turn(), 39
- Turtle, 37, 218
- type, 4, 8
- type abstrait, 246, 253
- type abstrait et fonction, 248
- type abstrait prédéfini, 247
- type Bool, 54
- type composite, 209
- type paramétrique, 263, 271
- type parent, 247
- type primitif, 263, 271
- type primitif et constructeur, 263
- type spécifique, 246, 253, 263
- type union, 271
- type, diagramme de, 249, 253
- TypeError, 291
- typeof(), 4, 124
- UInt8, 263
- UndefinedVarError, 29, 291
- Unicode, 12, 24, 107
- Union, 271
- union, 264
- unofficial binaries, 299
- updatedb, 38, 300
- uppercase(), 103
- using, 13, 38, 193, 273, 287
- using InteractiveUtils, 281
- using Object, 205
- using Pkg, 301
- using Random, 246
- using Test, 244
- UTF-8, 97, 107
- valeur, 4, 8, 133, 160
- valeur de retour, 21, 32
- valeur par défaut, 192
- valeur retournée, tuple, 168
- valeurs manquantes, 269, 271
- values(), 147
- \varepsilon, 90
- variable, 18
- variable globale, 136, 157, 161, 243

- variable locale, 28, 32, 136
- variable temporaire, 80
- vcat(), 136–138
- verbose, mode, 157
- Verlaine, Paul, 132
- Vim, 14
- Visual Studio Code, 299
- Visual Studio Codium, 299, 300
- void function, 30, 67
- vorpal, 74
- VSCodium, 14

- "w", mode écriture, 196
- walkdir(), 199
- where, 13
- while, 13, 83, 85, 92, 98, 102, 286

- zip(), 170
- zip(), collect(), 170
- zip(), Dict(), 172
- zip, objet, 170, 175
- Zipf, loi de, 192
- zipper, 170

Quatrième de couverture

Si vous souhaitez commencer à programmer, Julia est un excellent langage compilé à la volée (*Just In Time*, JIT), dynamiquement typé et doté d'une syntaxe élégante et soignée. Ce guide pratique utilise Julia (version ≥ 1.0) pour apprendre à programmer de manière progressive, en commençant par les concepts de base de la programmation avant de passer à des fonctionnalités plus avancées, telles que la création de nouveaux types, le polymorphisme, la méta-programmation et le *dispatch multiple*.

Conçu dès le départ pour être très performant, Julia est un langage polyvalent, idéal non seulement pour l'analyse numérique et les sciences informatiques, mais aussi pour la programmation web ou la création de scripts. Grâce à des exercices accompagnant chaque chapitre, vous pourrez tester les notions de programmation au fur et à mesure de leur apprentissage.

« Think Julia » est idéal pour les ingénieurs, les étudiants de niveau secondaire et universitaire ainsi que pour les autodidactes, les étudiants scolarisés à domicile et les professionnels devant apprendre les bases de la programmation moderne.

- Commencez par les bases, y compris la syntaxe et la sémantique du langage
- Accédez une définition claire de chaque notion de programmation
- Apprenez à manipuler les valeurs, les variables, les énoncés, les fonctions et les structures de données dans une progression logique
- Découvrez comment exploiter des fichiers et des bases de données
- Créez et gérez les types, les méthodes et le *dispatch multiple*
- Utilisez des techniques de débogage performantes pour corriger les erreurs de syntaxe, d'exécution et de sémantique
- Explorez la conception d'interfaces et les structures de données à travers des études de cas