COURS ET EXERCICES

Premiers cours de programmation avec Scheme

Du fonctionnel pur aux objets avec DrRacket

Jean-Paul ROY

Table des matières

Introduction **13**
Les langages de programmation 13
La culture Lisp 15
Guide d'utilisation 18
Remerciements 19

I La Programmation Fonctionnelle **21**

1 Les Expressions Préfixées **23**
1.1 Les niveaux de langage dans Racket 23
1.2 La notation préfixée complètement parenthésée 25
1.3 Le toplevel 28
1.3.1 Un endroit où l'on cause 28
1.3.2 Un environnement de variables 29
1.4 Nommer le résultat d'un calcul avec `define` 30
1.5 Un *teachpack* pour l'apprentissage 31
1.6 Les nombres 33
1.6.1 Les entiers [$\mathbb{Z}$] 34
1.6.2 Les rationnels [$\mathbb{Q}$] 35
1.6.3 Les réels [$\mathbb{R}$] 37
1.6.4 Les complexes [$\mathbb{C}$] 38
1.7 Les booléens et les expressions conditionnelles 39
1.7.1 La conditionnelle `if` 39
1.7.2 La conditionnelle générale `cond` 40
1.7.3 Les séquenceurs `and` et `or` 41
1.8 Sur l'évaluation d'une expression arithmétique 41
1.9 Exercices 44

2 Les Fonctions **47**
2.1 Programmer et tester une fonction 47
2.2 Les fonctions anonymes 50
2.3 Le Garbage Collector 51
2.4 Les fonctions nommées 52
2.5 Les variables globales 52
2.6 Les variables locales 53
2.7 Exercices 54

3 Construire des Images **59**
3.1 Images géométriques de base 59
3.1.1 L'accent aigü ou *quote* 60
3.1.2 Les images textuelles 61
3.1.3 Dimensions d'une image 61
3.2 Les images composées 62
3.2.1 Superposition d'images 62
3.2.2 Alignement horizontal ou vertical d'images 63
3.2.3 Rotation d'une image 63
3.2.4 Changement d'échelle 64
3.2.5 Encadrement d'une image 64
3.2.6 Extraction d'une sous-image 64
3.2.7 Utiliser de véritables images 65
3.3 Exemple : construction d'un smiley 65
3.4 Exercices 66

4 Animer le Monde **69**
4.1 Trouver le modèle mathématique du monde 70
4.2 Décrire l'évolution du monde 70
4.3 Transformer le monde en une image 70
4.4 Prévoir la fin du monde 71
4.5 Le squelette d'une animation 71
4.6 Un exemple de monde à un seul paramètre 73
4.7 Interagir avec une animation 75
4.7.1 La gestion des évènements clavier 75
4.7.2 La gestion des évènements souris 76
4.8 Exercices 77

5 Les Structures **79**
5.1 Les structures comme n-uplets de données 79
5.2 Exemple : modélisation d'un nombre rationnel 80
5.3 Exemple : modélisation d'une balle physique 81

5.4 Animation d'un monde à plusieurs paramètres . . . 81
5.5 Déstructuration : la forme spéciale `match` . . . 85
5.6 Exercices . . . 86

6 Programmation par Récurrence **89**
6.1 La démonstration par récurrence . . . 89
6.2 La programmation par récurrence . . . 93
6.3 La récurrence enveloppée . . . 96
6.4 Exemples de fonctions récursives enveloppées . . . 98
6.4.1 Un problème combinatoire . . . 98
6.4.2 Construction récursive d'une image . . . 99
6.4.3 Calcul du PGCD . . . 99
6.5 La récurrence terminale ou itération . . . 101
6.6 Exemples de fonctions itératives . . . 103
6.7 Notion de complexité d'un calcul récursif . . . 105
6.8 Exercices . . . 108

7 Compléments sur la récursivité **113**
7.1 Liaison statique et liaison dynamique . . . 113
7.1.1 Quelle est la valeur d'une lambda-expression ? . . . 114
7.2 Les fonctions d'ordre supérieur . . . 116
7.2.1 Exemple : modéliser un couple par une fonction . . . 116
7.2.2 La curryfication . . . 117
7.2.3 Passer à l'ordre supérieur pour généraliser . . . 118
7.2.4 Programmation par passage à la continuation : CPS . . . 119
7.3 Exercices . . . 121

8 Les Listes Chaînées **125**
8.1 Construction d'une liste . . . 125
8.2 Les symboles et la citation . . . 126
8.3 Accès aux éléments d'une liste . . . 127
8.4 L'égalité en général : `equal?` . . . 128
8.5 Primitives de base sur les listes . . . 128
8.5.1 La longueur d'une liste : `length` . . . 128
8.5.2 L'appartenance d'un élément à une liste : `member` . . . 129
8.5.3 La construction d'une liste en extension : `list` . . . 129
8.5.4 La construction d'une liste en compréhension : `build-list` . . . 130
8.5.5 La concaténation de deux listes : `append` . . . 130
8.5.6 L'inversion d'une liste : `reverse` . . . 131
8.5.7 La sélection dans une liste : `filter` . . . 132
8.5.8 Décomposition d'une liste avec `match` . . . 132

8.6 Recherche dans une liste 133
8.6.1 Accès à l'élément numéro k d'une liste : `list-ref` 133
8.6.2 Chercher un élément vérifiant une condition donnée 134
8.6.3 Chercher un élément et poursuivre la recherche 135
8.6.4 Chercher le milieu d'une liste 136
8.6.5 Chercher une clé dans une A-liste : `assoc` 136
8.7 Les ensembles 137
8.7.1 Recherche en profondeur dans une liste 138
8.8 Deux manières de trier une liste 139
8.8.1 Le tri par insertion 139
8.8.2 Le tri par fusion 140
8.9 Animation et listes : un éditeur de polygone 141
8.10 L'ordre supérieur sur les listes 142
8.10.1 L'abstraction du traitement parallèle : `map` 142
8.10.2 L'abstraction du parcours séquentiel enveloppé : `foldr` 143
8.10.3 L'abstraction du parcours séquentiel itératif : `foldl` 144
8.10.4 L'applicateur fonctionnel : `apply` 145
8.10.5 Les fonctions d'arité variable 147
8.11 Exemple approfondi : analyse par combinateurs 148
8.11.1 Un reconnaisseur 148
8.11.2 Du reconnaisseur à l'analyseur syntaxique 151
8.12 La notion de type abstrait de donnée 151
8.12.1 Un type abstrait *vecteur 2D* 152
8.12.2 Le fichier contenant le type abstrait doit être un *module* 154
8.13 Exercices 155

9 Les Arbres 161
9.1 Les arbres binaires d'expressions 161
9.2 Parcours en profondeur et en largeur 162
9.3 Implémentation du type abstrait 164
9.4 Exemples de parcours d'arbres en profondeur 164
9.4.1 Hauteur d'un arbre 164
9.4.2 Recherche d'une feuille 165
9.4.3 Le feuillage 165
9.4.4 Parcours postfixe d'un arbre 165
9.4.5 Valeur d'un arbre 166
9.5 Le parcours préfixe plat et sa réciproque 168
9.5.1 Une solution par le théorème des poids 168
9.5.2 Mieux : une solution en un seul passage 169
9.6 Dessiner un arbre 170
9.7 Piles et parcours d'arbres itératifs 172

9.7.1 Le type abstrait *pile fonctionnelle* . . . 172
9.7.2 Application au parcours itératif d'un arbre . . . 173
9.7.3 Utilisation du style CPS . . . 174
9.8 Files d'attente et parcours d'arbres en largeur . . . 175
9.8.1 Le type abstrait *file d'attente fonctionnelle* . . . 175
9.8.2 Application au parcours en largeur . . . 177
9.9 Arbres binaires de recherche . . . 177
9.9.1 Recherche d'un élément . . . 179
9.9.2 Insertion aux feuilles . . . 179
9.9.3 Les arbres équilibrés . . . 180
9.9.4 L'algorithme d'insertion équilibrante . . . 182
9.9.5 Suppression du plus grand élément . . . 184
9.9.6 Suppression d'un élément quelconque . . . 185
9.10 Les ensembles ordonnés . . . 185
9.11 Promenade multi-directionnelle dans un arbre . . . 186
9.12 Exercices . . . 188

10 Programmer avec des Arbres 193
10.1 La machine VRISC et la compilation des arbres . . . 193
10.1.1 Le traducteur vers VRISC1 . . . 194
10.1.2 L'interprète de code VRISC1 . . . 195
10.1.3 Les nœuds conditionnels . . . 197
10.1.4 La machine virtuelle VRISC2 . . . 197
10.1.5 Le traducteur vers VRISC2 . . . 198
10.1.6 L'interprète de code VRISC2 . . . 199
10.1.7 Exercices . . . 200
10.2 Introduction au calcul formel . . . 201
10.2.1 Le simplificateur . . . 203
10.2.2 Le dérivateur . . . 205
10.2.3 La série de Taylor d'un arbre . . . 205
10.2.4 Exercices . . . 206
10.3 Un démonstrateur automatique de théorèmes . . . 207
10.3.1 Les formules logiques sont des arbres . . . 208
10.3.2 Le type abstrait *formule bien formée* . . . 209
10.3.3 Exemple : élimination des implications . . . 210
10.3.4 L'algorithme de Wang . . . 210
10.3.5 Exercices . . . 213

11 Le Vrai Langage Racket **217**
11.1 Le niveau de langage est déterminé par le source 217
11.2 Les doublets : une nouvelle vision de la fonction `cons` 220
11.2.1 Les chaînages de doublets 221
11.2.2 La convention d'affichage du point-parenthèse 222
11.2.3 Les notations contractées `c--r` 222
11.2.4 L'égalité des doublets 222
11.2.5 La structure de liste 223
11.2.6 La quasiquote 224
11.3 Les booléens 224
11.4 Le déclenchement d'une erreur 225
11.4.1 La véritable syntaxe de la fonction `error` 225
11.4.2 Les erreurs sont des exceptions 225
11.5 Les variables locales en Scheme standard 227
11.5.1 `let` et les liaisons parallèles 227
11.5.2 `let*` et les liaisons séquentielles 228
11.5.3 `letrec` et les liaisons récursives 229
11.5.4 Les définitions internes 229
11.6 Les macros, ou extensions syntaxiques 230
11.6.1 Le mécanisme `define-syntax` 230
11.6.2 La mise au point des macros 232
11.6.3 Les points de suspension 232
11.6.4 Une macro `show` pour tester nos fonctions 233
11.6.5 La boucle `do` fonctionnelle 234
11.6.6 Les transformateurs syntaxiques 234
11.7 Exercices 236

II L'Impératif et les Objets **239**

12 La Mutation **241**
12.1 Le séquencement avec `begin` 241
12.2 `void` et les fonctions sans résultat 242
12.3 La mutation des variables ou affectation 242
12.4 Les boucles `do`, `while` et `for` 245
12.5 Retour sur l'appel par valeur 248
12.6 Les générateurs 249
12.7 Les fonctions à mémoire 251
12.8 Les acteurs et les envois de messages 253
12.9 Le graphisme impératif 256
12.9.1 Les primitives graphiques 257

12.9.2 Le graphisme de la tortue 258
12.9.3 Exemples de programmes tortue 261
12.10 La mutation des structures 263
12.10.1 Animation dans un monde mutable 264
12.11 La mutation des vecteurs 265
12.11.1 Échange de deux éléments 266
12.11.2 Exemple : le drapeau hollandais 267
12.11.3 Les matrices 267
12.11.4 Exemple : vectorisation de code VRISC 269
12.12 La mutation des chaînes de caractères 270
12.13 La mutation des doublets 270
12.14 Les tables de hash-code 275
12.15 Exercices 276

13 Le Texte et les Entrées-Sorties 283
13.1 Les caractères 283
13.1.1 Le codage Unicode 283
13.1.2 Le code ASCII et l'encodage UTF-8 285
13.2 Les chaînes de caractères 286
13.2.1 Construction d'une chaîne 286
13.2.2 Accès aux caractères et mutation 287
13.2.3 Pourquoi pas un `return` comme en JAVA ? 288
13.2.4 Exemple : le codage de César 289
13.2.5 Exemple : un analyseur lexical 290
13.2.6 Les expressions régulières 291
13.3 Écriture sur un port de sortie 295
13.3.1 Écriture à l'écran 295
13.3.2 Écriture dans un fichier sur disque 295
13.3.3 Écriture dans une chaîne 297
13.4 Lecture dans un port d'entrée 297
13.4.1 Lecture au clavier, rôle de `eval`, et gestion des erreurs 297
13.4.2 Lecture d'un fichier sur disque 299
13.4.3 Lecture dans une chaîne 302
13.5 Entrée-sortie sur Internet : un client Web 302
13.6 L'exploitation du système 305
13.6.1 Les fonctions RACKET liées au système de fichiers 305
13.6.2 Lancement de programmes UNIX à partir de RACKET 306
13.6.3 Exécution de scripts RACKET sous UNIX 308
13.7 Exercices 310

14 La Programmation par Objets et l'API graphique **313**
14.1 Rappel : les acteurs à états locaux 313
14.2 Classes et objets en RACKET 314
14.3 Les sous-classes et l'héritage 316
14.4 La construction d'interfaces graphiques 318
14.4.1 Présentation de résultats de calculs 319
14.4.2 Programmation d'un éditeur de texte 322
14.4.3 Dessins dans une fenêtre graphique 326
14.4.4 Déplacement d'objets à la souris 329
14.4.5 Boutons radio, cases à cocher, etc. 333
14.4.6 Animation gourmande et threads 334
14.4.7 Utilisation d'un double buffer 335
14.4.8 Utilisation d'une horloge 336
14.4.9 Simulation d'un mouvement planétaire 336
14.5 Exercices 338

III Syntaxe et Sémantique **341**

15 Des Analyseurs Syntaxiques **343**
15.1 LEX et la génération d'analyseurs lexicaux 343
15.2 YACC et la génération d'analyseurs syntaxiques 346
15.3 Incursion dans la théorie 348
15.4 Exemple : un traducteur du langage *Nano-C* 353
15.5 Exercices 358

16 Interprétation d'un sous-ensemble de SCHEME **361**
16.1 Le langage MISS 361
16.2 La gestion des environnements 363
16.3 Le cœur de l'interprète : `eval/apply` 366
16.4 Le traitement des fermetures 367
16.5 La conditionnelle `$if` 369
16.6 Les variables locales avec `$let` 369
16.7 Les fonctions locales co-récursives avec `$letrec` 370
16.8 Le séquencement avec `$begin` 371
16.9 L'affectation avec `$set!` 371
16.10 La définition au toplevel 371
16.11 La bibliothèque initiale 372
16.12 La boucle toplevel 372
16.13 Se protéger contre les erreurs 373
16.14 Exercices 374

17 La prise en main du Contrôle **377**
17.1 Rappels sur CPS . . . 377
17.1.1 Du style direct à CPS . . . 377
17.1.2 Abandon et capture de continuation . . . 378
17.1.3 Les continuations à plusieurs variables . . . 381
17.1.4 Mise en attente d'un calcul sans pile . . . 381
17.1.5 Générateurs et calculs pas à pas . . . 382
17.1.6 Application au retour arrière : les N dames . . . 383
17.2 Les continuations de première classe avec `call/cc` . . . 384
17.2.1 S'échapper d'un calcul récursif . . . 386
17.2.2 Capturer la continuation du toplevel . . . 387
17.2.3 Capturer une continuation pour simuler un `GOTO` . . . 387
17.2.4 Vers un interprète à continuation . . . 391
17.2.5 Traduction automatique vers CPS . . . 391
17.3 La programmation non déterministe . . . 392
17.3.1 Un chercheur de nombres premiers . . . 393
17.3.2 Un puzzle logique . . . 394
17.3.3 Une implémentation de `amb` . . . 395
17.4 La Programmation Paresseuse . . . 396
17.4.1 Le langage HASKELL . . . 397
17.4.2 LAZY RACKET . . . 400
17.4.3 Rendre MISS paresseux . . . 404
17.4.4 Une implémentation des flots en SCHEME strict . . . 406
17.5 Exercices . . . 411

18 Annexe : le teachpack `valrose` **415**

Bibliographie **419**

Introduction

Les langages de programmation

Les langages de programmation ont pour vocation de faciliter la communication avec un ordinateur, pour lui faire accomplir des tâches variées correspondant à ce que le programmeur aura spécifié dans le texte de son programme. Mais il est aussi un moyen de communiquer avec d'autres programmeurs et surtout avec soi-même, dans cet élan réflexif qui permet d'exprimer, à travers une langue ésotérique, un comportement opératoire.

Ce caractère opérationnel du langage de programmation s'est historiquement exprimé à travers une volonté d'*exécution* transportée par les phrases du langage. Phrases ayant reçu bien à propos le qualificatif d'*instructions*. Le travail du programmeur, dans cette optique, consiste à donner des ordres *impératifs* à une machine : fais ceci, fais cela, modifie telle valeur, etc.

Remontons dans le temps et suivons en effet la genèse de ces outils que sont les langages. Il suffira pour notre propos de ramener les horloges aux années cinquante, qui virent l'apparition de langages robustes destinés à programmer des ordinateurs adultes. Parmi ces langages, et si l'on excepte le binaire pur, on trouve en premier lieu le langage d'assemblage, permettant au programmeur d'écrire de longues suites d'instructions très élémentaires manipulant la mémoire de la machine d'une manière explicite. Un logiciel spécialisé, l'assembleur, avait alors pour charge de traduire (d'assembler) ce texte en langage binaire directement exécutable par le processeur. Les programmeurs de la première génération ont reçu comme image de l'ordinateur une peinture qui ressemblait vaguement à une vaste suite de boîtes de chaussures numérotées dont il fallait consulter ou modifier le contenu.

FORTRAN apparut vers 1955. Ce FOR*mula* TRAN*slator* bien nommé, dû à John Backus, avait une spécialité : le *calcul numérique*. Sa capacité à traduire en langage d'assemblage les formules mathématiques permettait au programmeur de s'éloigner du niveau spartiate des cases-mémoire et de se rapprocher d'une culture plus familière, celle des notations mathématiques. L'introduction de variables symboliques X ou Y et de formules comme $2 * X + Y$ l'autorisait à faire abstraction des emplacements mémoire destinés à recevoir les contenus de X et Y. De plus, il pouvait penser la formule $2*X+Y$

comme un tout, sans avoir à calculer d'abord la multiplication intermédiaire $2 * X$, pour lui additionner ensuite Y. Le compilateur FORTRAN (programme chargé de la traduction en langage machine) était particulièrement optimisé pour les calculs arithmétiques. Le programmeur *spécifiait* la formule $2*X+Y$ en laissant au compilateur le soin d'organiser le calcul des étapes intermédiaires dans le sens d'une efficacité maximale. Mises à part cette facilité et quelques autres, un texte FORTRAN restait avant tout dans la lignée du langage d'assemblage, une suite d'instructions impératives destinées à être exécutées pour modifier la mémoire. Les algorithmes devaient contenir une description pas-à-pas de la stratégie de résolution du problème en cours [1].

La branche des langages *impératifs*, successeurs du langage d'assemblage et de FORTRAN, s'étendit rapidement, notamment dans la foulée du langage algorithmique ALGOL (1960) qui réglait le problème de la rigueur grammaticale et introduisait nombre de constructions puissantes qui font encore défaut dans bien des langages d'aujourd'hui. Le langage PASCAL, malgré de nombreuses faiblesses, eut son heure de gloire dans l'enseignement, notamment grâce à l'apparition de compilateurs rapides. Assez contraignant et d'une rigidité exemplaire (ce que certains prirent pour une vertu), il fut vite remplacé par le langage C, beaucoup plus laxiste et diffusé avec le système d'exploitation UNIX.

Le paradigme impératif est basé sur l'obligation de faire progresser un calcul par des modifications explicites, soit de la mémoire de la machine (par des *affectations*), soit de la manière dont l'exécution du programme est contrôlée (par des *branchements*). Voyez ci-contre un calcul de 10! dans ce style, où la flèche ← se lit *devient égal à*, et où `GOTO` se traduit par *continuer en.*

```
    N ← 10
    F ← 1
a:  if N=0 then GOTO b
    F ← F×N
    N ← N-1
    GOTO a
b:  print F
    stop
```

L'un des mérites de l'école dite de *programmation structurée* des années 70 fut d'éliminer la gestion explicite par le programmeur de ces branchements inconditionnels comme `GOTO` [Dij68], au profit de nouvelles structures de contrôle comme la boucle `while`, le compilateur — PASCAL par exemple — générant automatiquement les branchements correspondants en langage machine. Le programmeur passait à un niveau d'abstraction plus élevé pour penser (panser) ses algorithmes. Ce que cette école n'a pas vu, ou n'a pas voulu voir, c'est que l'on pouvait aussi et plus radicalement se débarasser à la fois du concept de boucle et de celui d'affectation en exploitant à fond le concept de fonction, joint au principe de récurrence. Bien exploité, ce dernier fournit l'itération comme un cas particulier, phénomène auquel les mathématiciens n'ont pas attaché une attention particulière.

1. Lors du discours de réception du Turing Award (une sorte de prix Nobel pour les informaticiens) qu'il reçut en 1977 pour ses travaux sur FORTRAN, Backus reniera ce dernier au profit d'une programmation purement fonctionnelle [Bac78]. *Cette indication entre crochets renvoie à la bibliographie.*

Dans toutes les sciences qui modélisent du virtuel et jouent avec des concepts abstraits (espaces mathématiques, particules physiques étranges ou arbres informatiques), les métaphores se sont toujours avérées d'une puissance créatrice incomparable. L'une des plus belles métaphores de l'histoire de l'informatique est sans doute celle des *objets*. À l'origine, le Programmeur était un chef d'orchestre gérant (impérativement ou fonctionnellement) le calcul sur ses données. Jusqu'à la fin des années 60, où les données devinrent des objets dotés de la capacité de recevoir et d'envoyer eux-mêmes des messages vers d'autres objets. Le paradigme de la Programmation Orientée Objet (POO) était né, avec des langages comme Smalltalk, CLOS, Java, ou les extensions de C vers les objets (C++, Objective-C), et allait vite envahir le monde logiciel.

La culture Lisp

Face à ce monde d'instructions et d'affectations, existe un monde bien plus ancien, bien mieux connu, celui des expressions mathématiques. Hélas, les mathématiques forment bien un langage, mais qui n'est pas directement exécutable sur ordinateur. Un bon compromis émerge avec les langages fonctionnels dont l'ancêtre est Lisp [Mac60]. Les fonctions, augmentées du principe de récurrence, permettent une description des calculs dans laquelle l'idée de modification n'a pas grand intérêt et se trouve remplacée par celle de substitution temporaire d'une valeur à une autre. L'avantage réside dans la bonne maîtrise de la rigueur et des techniques de démonstration ou construction (par récurrence), alors que la preuve d'un programme impératif, dont les variables changent de valeur au cours du temps, nécessite de nouvelles techniques logiques plus difficiles à mettre en œvre [Man74].

L'année 1956 vit s'ouvrir un séminaire au Dartmouth College (USA) intitulé *Summer Research Project on Artificial Intelligence*, dont les organisateurs étaient John McCarthy (MIT), Marvin Minsky (Harvard University), Nathaniel Rochester (IBM) et Claude Shannon (Bell Telephone). Ces chercheurs et leurs équipes allaient inaugurer une nouvelle branche de la recherche informatique, l'Intelligence Artificielle (IA).

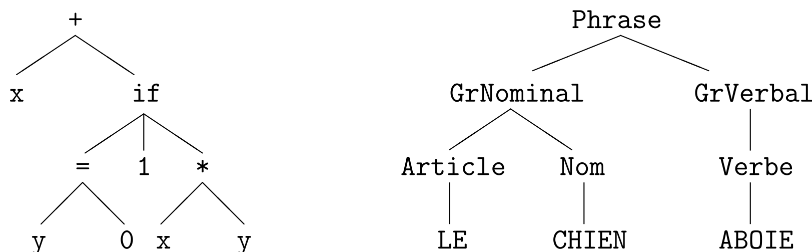
Bien sûr, l'idée d'une mécanisation de l'intelligence humaine remonte loin : Wilhelm Leibniz, Georges Boole, plus tard Alan Turing, ou John Von Neumann lui-même, et bien d'autres, ont tous rêvé d'un *calculus ratiocinator* à la Leibniz, d'un calcul universel permettant de raisonner de manière mécanique. Mais ce qui nous intéresse ici, ce n'est pas l'intelligence artificielle proprement dite, avec son histoire chaotique où la science se mêle d'idéologie, mais plutôt l'incidence qu'elle a eue sur l'évolution des langages de programmation.

Car Fortran n'était pas vraiment l'outil idéal pour programmer un jeu d'échecs avec ses plans d'actions, et ses arborescences de décision. Bien entendu, tout langage *minimal* suffit pour programmer à peu près n'importe quoi, mais ce résultat théorique ne sert qu'au théoricien, le programmeur souhaitant, quant à lui, disposer d'outils et de langages suffisamment sophistiqués, lui permettant d'exprimer ses problèmes et leurs

solutions à un bon niveau d'abstraction. En l'occurrence, les *arbres* furent vite perçus comme une structure de donnée fondamentale permettant de représenter nombre de connaissances de nature symbolique : une stratégie au jeu d'échecs, une expression algébrique, une phrase en langage naturel. Il était tentant de greffer sur FORTRAN des bibliothèques de programmes fournissant des données arborescentes ; on vit alors naître des mutants comme IPL (Information Processing Language) ou FLPL (Fortran List Processing Language). Mais ces rustines ne furent pas satisfaisantes.

C'est à ce moment – la fin des années 50 – que John McCarthy, qui avait organisé le séminaire de Dartmouth, travaillait sur un nouveau langage de programmation nommé LISP. Ce langage devait notamment permettre d'écrire des programmes de démonstration automatique de théorèmes, ces derniers ainsi que leurs preuves étant exprimés dans la notation LISP. Autrement dit, un programme écrit en LISP devait être capable de traiter des données étant elles-mêmes des textes écrits en LISP, recevant la dénomination d'*expressions symboliques* [Mac60] ou *S-expressions* (*sexpr*).

Ces *sexpr* pouvant dénoter aussi bien une formule mathématique qu'une assertion logique, une structure moléculaire, une loi du code pénal ou un programme, il était urgent de trouver un concept suffisamment unificateur pour y fondre toutes ces formes. La structure d'**arbre** se révéla un excellent candidat.



Un programme LISP sera donc un arbre, ou un ensemble d'arbres. Bien entendu, faute de technologie adéquate et habitué à une écriture linéaire, le programmeur n'utilisera pas une notation arborescente à deux dimensions.

Ce texte LISP, que ce soit un programme ou une donnée, sera donc pensé comme un arbre, mais représenté de manière unidimensionnelle sous la forme d'une liste complètement parenthésée [2] :

```
(+ x (if (= y 0) 1 (* x y)))
(Phrase (GrNominal (Article LE) (Nom CHIEN)) (GrVerbal (Verbe ABOIE)))
```

LISP devenait ainsi un *langage de traitement de listes* : **LISt Processor**. Ce fut l'idée géniale de McCarthy de faire le pas décisif – toujours évident *a posteriori* – consistant à rationaliser cette exigence d'*auto-référence* du langage. La vision mathématique qu'avait

2. Il est assez amusant et triste à la fois de voir tout le tapage fait autour de XML qui ne fait que revenir aux sources de LISP, avec une notation inutilement compliquée.

McCarthy de la programmation fit de LISP un langage fonctionnel de traitement de listes, dans lequel le texte d'une fonction était lui-même représenté... par une liste ! La boucle était bouclée. Il devenait dès lors facile d'écrire en LISP un interprète ou un compilateur LISP, et en particulier de décrire la sémantique – le sens – d'une expression LISP, en expliquant comment elle serait calculée. La souplesse du langage permit aussi d'expérimenter diverses stratégies de calcul, de la plus banale à la plus sophistiquée, sans avoir à payer chaque fois le prix de l'écriture d'un gros compilateur.

L'idée de fonder le langage sur la notion d'arbre provient donc des exigences du calcul symbolique. Celle de mettre en avant les fonctions est issue du *lambda-calcul* d'Alonzo Church qui tenta dans les années 1940 de fonder les mathématiques sur le concept primitif de fonction (les entiers étant eux-mêmes définis par des fonctions !). Un peu comme Nicolas Bourbaki [3], au même moment de l'histoire, reconstruira l'édifice mathématique sur la pierre ensembliste [Bou69]. Ces entreprises auront eu un écho profond et durable sur l'avancée des recherches et leurs retombées, même si cette ambition totalitaire paraît quelque peu dérisoire aujourd'hui. Il reste que LISP en a gardé une double image, celle d'un langage apprécié du théoricien (y compris des mathématiciens [Cha03]) pour sa concision et son auto-référence, mais aussi du *hacker* (dans le sens noble du terme) qui voit en lui un langage-machine de très haut niveau.

Si comme on le prétend l'informatique est née de la rencontre fortuite de la logique formelle et d'un fer à souder, l'épistémologie de cette jeune discipline n'est pas très avancée, et le choix d'un langage de programmation, notamment pour l'enseignement, relève souvent de considérations douteuses. C'est vrai que la multiplication des langages, et des dialectes à l'intérieur des langages, n'y est pas étrangère. Issu de la branche LISP, le langage SCHEME a l'avantage d'être simple, aux fondations claires, compatible avec la pensée mathématique usuelle et doté d'excellentes implémentations. Extensible dès l'origine, minimal à ses débuts, puis doté de riches bibliothèques (dont les objets), il excelle aux deux bouts de la chaîne : très bon langage d'initiation agréable à utiliser, il s'avère aussi un redoutable outil de recherche fondamentale sur la sémantique des langages et le prototypage ([Que07], [EOPL]), jusqu'à la programmation du Web [Ser07].

Si Niklaus Wirth, inventeur de PASCAL, a pu insister en son temps sur l'équation *Algorithmes + Structures de Données = Programmes*, les théoriciens du monde LISP ont fourni un travail profond et difficile sur les *structures de contrôle*, limitées à la séquence, la conditionnelle et la boucle par la programmation structurée. La notion abandonnée de *saut* (`GOTO`), reprend sa place sous un habillage plus formel et mieux compris avec le concept moderne de *continuation*, introduit par SCHEME et repris récemment avec plus ou moins de bonheur par de nouveaux langages. Deux visions de la programmation [impérative et fonctionnelle] évoluent ainsi en parallèle, s'enrichissant mutuellement. Les étudiants sont de nos jours formés à divers styles de pensée, leur permettant d'aborder avec profit des langages dont le niveau d'abstraction ne cesse de croître [Nar05].

3. Célèbre mathématicien polycéphale de l'Université de Nancago, né en 1935 à Besse-en-Chandesse.

Guide d'utilisation

La rédaction de ce livre fait suite à plus de vingt années d'enseignement de LISP et de SCHEME à divers publics, enseignants de lycée et étudiants, notamment deux cours semestriels (introductif et avancé) de programmation à la Faculté des Sciences de l'Université de Nice Sophia-Antipolis.

La première partie (chapitres 1 à 11) constitue la matière d'un cours d'introduction à la programmation fonctionnelle – voire à la programmation tout court – pour de grands débutants. Je l'ai augmentée de manière à ce qu'elle puisse servir à un public auto-didacte non contraint par un volume horaire. Elle présente la programmation fonctionnelle et sa pureté, au sein d'un monde mathématiquement propre, dans lequel on peut raisonner sur des données qui ne changent pas dans le temps.

Les chapitres 12, 13 et 14 constituent un cours complémentaire où est abordé le paradigme impératif qui traite de la mutation des données et de sa dangeureuse efficacité, suivi de l'incontournable modélisation par des objets afin de construire des interfaces graphiques, bien qu'il ne s'agisse pas ici d'enseigner la conception par objets. Ce qui permet au final de disposer d'un langage de programmation moderne et efficace, dont les principes sont assez vites exportables.

Enfin, les chapitres 15, 16 et 17 sont plus avancés et se placeraient bien dans un cadre de L3-Informatique. Ils introduisent des thèmes plus avancés, où il est question de la syntaxe d'un langage de programmation (quelles sont les phrases bien formées ?), et de sa sémantique (que signifient-elles ?). Presqu'au sens psychanalytique du terme, la question posée est : *comment ça calcule ?...*

Nous concrétiserons ces idées à travers le système RACKET (anciennement connu sous les noms PLT-SCHEME ou DRSCHEME) :

`http://www.racket-lang.org`

Il s'agit d'une excellente implémentation universitaire (gratuite sous Linux, MacOS-X et Windows), très complète, avec un éditeur syntaxique intégré, une interface graphique soignée, et toutes les bibliothèques largement documentées en ligne sur le graphisme, l'accès à Internet, la programmation par objets à la Java, etc. Le noyau portable du langage SCHEME est décrit dans un rapport du Comité International de Normalisation [4], mais certaines extensions proposées par RACKET seront utilisées, que ce soit pour des raisons pédagogiques ou simplement parce que la norme ne les prévoit pas.
Il existe d'autres bons systèmes SCHEME, dont vous trouverez la trace sur Internet, comme *Bigloo* (cocorico), *Gambit* (de nos amis canadiens), etc. Chacun a ses qualités propres et fonctionne sur tous les systèmes d'exploitation.

Il est recommandé au lecteur de s'abonner au forum *comp.lang.scheme* sur Internet. Il existe parallèlement une *mailing list* réservée aux utilisateurs de RACKET, pour des

4. Le rapport de normalisation est inclus dans l'aide en ligne de DRRACKET.

problèmes ou des développements impliquant ce logiciel. Prenez un clavier, quand même quelques neurones, et bienvenue dans le monde des parenthèses !

Vous trouverez sur ma page Web :

`http://deptinfo.unice.fr/~roy`

un secteur réservé à ce livre (`PCPS`).

Remerciements

Après avoir débuté avec Basic et Pascal, j'ai découvert Lisp à travers un drôle de petit livre, *The Little Lisper* [Fri96], qui en expliquait la tournure d'esprit sous la forme d'un dialogue socratique et de bandes dessinées. Je l'ai lu sans clavier sous la main, pendant les vacances d'été, et je dois avouer qu'il m'a fortement remué. À la rentrée, j'ai décidé de suivre les cours de Jean-François Perrot puis de Christian Queinnec à l'Université Pierre et Marie Curie, et ce sont eux qui sont directement responsables de l'inoculation du virus Lisp (aucun antidote connu à ce jour). Qu'ils en soient chaleureusement remerciés, je n'ai jamais rencontré d'enseignants aussi capables de faire passer un tel mélange de rigueur et d'enthousiasme.

Par la suite, nombreuses furent les influences amicales ou livresques. Jacques Arsac, malgré son extraordinaire maîtrise du monde impératif [Ars83], tenta vainement de me guérir, et m'apprit quand même énormément de choses alors que nous travaillions à la formation des professeurs de lycées parisiens à la programmation dans les années 80. Jacques Chazarain fut un collègue proche à la Faculté des Sciences de Nice, avec lequel j'aurais aimé signer ce livre, ou le sien [Cha96]. Je traîne depuis des lustres tous les scrupules du monde à ne pas avoir choisi dès le début l'excellent dialecte *Bigloo* de Manuel Serrano, de l'INRIA de Sophia Antipolis, mais je n'ai su résister à la puissance de frappe pédagogique de l'équipe Racket.

Une influence forte fut le livre [SICP] qui eut une portée considérable dans le monde des programmeurs et des formateurs, puisqu'il formait le cœur du cursus de première année du célèbre MIT, le *Massachusetts Institute of Technology*. Il popularisait l'idée qui fait son chemin que l'apprentissage de la programmation passe par l'étude *réflexive* du langage de programmation [comment peut-il se décrire lui-même ?], et pas seulement par son utilisation. L'équipe Racket [HtDP] semble avoir pris le relais, avec des pédagogues-chercheurs aussi remarquables que Matthias Felleisen, Matthew Flatt, Robby Findler, Shriram Krishnamurti et bien d'autres.

Merci à la NASA et au projet THEMIS pour leur image ci-dessous du lambda dans un cratère, qui montre clairement à mes yeux une λ-présence sur la planète Mars.

Merci à mes professeurs de mathématique pour le plus important, le goût de l'abstraction. Merci à mes relecteurs, enseignants et étudiants. Bravo à Apple pour son MacBook Pro et à Richard Koch pour son logiciel TeXShop.

Et surtout merci à toi, Catherine, pour ta patience et ton amour.

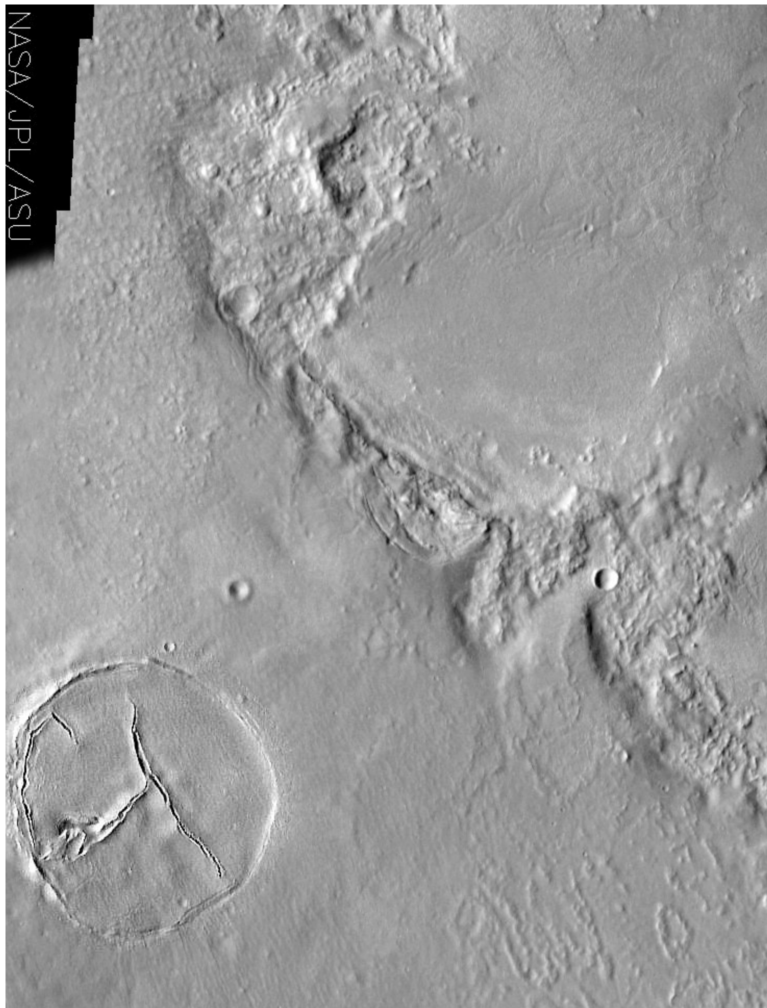


Image : NASA/JPL/Arizona State University

Bibliographie

[R⁶RS] Richard Kesley, William Clinger and Jonathan Rees ed. *Revised^6 Report on the Algorithmic Language Scheme.* Voir `http://www.r6rs.org`.

[Abe81] Harold Abelson & Andrea diSessa. *Turtle Geometry.* MIT Press, 1981.

[Ars83] Jacques Arsac. *Les Bases de la Programmation.* Dunod, 1983.

[Bac78] John Backus. *Can Programming Be Liberated from the von Neumann Style ? A Functional Style and Its Algebra of Programs.* Communications of the ACM, August 1978. Reproduit dans *ACM Turing Award Lectures, 1966-1985*, Addison-Wesley, 1987.

[Bis95] Charles Donnelly and Richard Stallman. *Bison. The YACC-compatible Parser Generator.* Free Software Foundation, November 1995. Voir `http://www.gnu.org/software/bison`.

[Bou69] Nicolas Bourbaki. *Éléments d'Histoire des Mathématiques.* Springer, 2007.

[Bry04] Anne Brygoo, Titou Durand, Maryse Pelletier, Christian Queinnec et Michèle Soria. *Programmation Récursive (en Scheme).* Dunod, 2004. Un cours de 1ère année professé à l'Université Pierre et Marie Curie, dont les Annales corrigées sont publiées aux éditions Paracamplus.

[Bur82] F.W. Burton. *An efficient implementation of FIFO queues.* Information Processing Letters, vol. 14, pp. 205-206.

[Cha03] Gregory J. Chaitin. *The Limits of Mathematics.* Springer, 2003.

[Cha96] Jacques Chazarain. *Programmer avec* SCHEME. Thomson Publishing, 1996.

[Cor94] Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. *Introduction à l'Algorithmique*, Dunod, 2002. Texte et vidéos du cours disponibles en anglais sur le site *MIT OpenCourseWare*, `http://ocw.mit.edu`.

[Dan04] Olivier Danvy. *Sur un exemple de Patrick Greussay.* BRICS Report RS-04-41, University of Aarhus, voir `http://www.brics.dk`.

[Dij68] Edsger Dijkstra. *GOTO statements considered harmful.* Communications of the ACM, Vol. 11, No. 3, March 1968.

[Dyb09] Kent Dybvig. *The* SCHEME *Programming Language.* MIT Press, 4th edition, 2009. Consultable en ligne sur `http://www.scheme.com`.

[EOPL] Daniel Friedman, Mitchell Wand and Christopher Haynes. *Essentials of Programming Languages.* MIT Press, 3rd edition, 2008.

[Fri76] Daniel Friedman & David Wise. *CONS should not Evaluate its Arguments.* Automata, Languages and Programming Symposium, Edinburgh, 1976.

[Fri88] Daniel Friedman. *Applications of Continuations.* Fifteenth Annual ACM Symposium on Principles of Programming Languages, January 13-15, 1988. Voir `http://www.cs.indiana.edu/~dfried`.

[Fri96] Daniel Friedman, Matthias Felleisen. *The Little Schemer.* MIT Press, 4th edition, 1996. Voir `http://www.ccs.neu.edu/home/matthias/books.html` ou bien `http://www.cs.indiana.edu/~dfried`.

[Gru02] D. Grüne, H. Bal, C. Jacobs et K. Langendoen. *Compilateurs.* Éditions Dunod, 2002.

[Hew76] Carl Hewitt. *Viewing Control Structures as Patterns of Passing Messages.* MIT Artificial Intelligence Lab Memo 410, December 1976.

[HtDP] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt and Shriram Krishnamurthi. *How to Design Programs.* MIT Press, 2001. Lecture en ligne sur `http://www.htdp.org`.

[Hue97] Gérard Huet. *Functional Pearl : the Zipper.* Journal of Functional Programming, September 1997.

[Ive62] Kenneth Iverson. *A Programming Language.* Addison-Wesley, 1962.

[Kar00] Jerzy Karczmarczuk. *Traitement paresseux et optimisation des suites numériques.* Journées francophones des langages applicatifs, Janvier 2000.

[LAM] La série des *Lambda Papers* de Guy Steele et Gerald Sussman. Voir `http://library.readscheme.org/page1.html`.

[Mac60] John McCarthy. *Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I.* Communications of the ACM, Vol. 3, No. 4, pages 185–195, April 1960. Voir `http://www-formal.stanford.edu/jmc`.

[Mac63] John McCarthy. *A Basis For A Mathematical Theory Of Computation.* Computer Programming and Formal Systems, North-Holland, 1963.

[Man74] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[Mar04] Bruno Martin. *Codage, Cryptologie et Applications.* Presses Polytechniques Universitaires Romandes, 2004.

[Meu09] Pierre Meunier. *Algèbre, avec applications à l'algorithmique et à la cryptographie.* Ellipses, 2009.

[Nar05] Philippe Narbel. *Programmation fonctionnelle, générique et objet.* Vuibert, 2005.

[Pei92] H. Peitgen, H. Jürgens, D. Saupe. *Chaos and Fractals.* Springer-Verlag, 1992.

[PLAI] Shriram Krishnamurti. *Programming Languages : Applications and Implementation.* Voir `http://www.cs.brown.edu/~sk`.

[Que00] Christian Queinnec. *Inverting back the inversion of control, or Continuations versus page-centric programming.* SIGPLAN Notices, 2001, vol. 38.

[Que07] Christian Queinnec. *Principes d'Implantation de Scheme et Lisp.* Paracamplus, 2007. Le chapitre 1 est téléchargeable sur la *homepage* de l'auteur, ou bien sur `http://www.paracamplus.com/Books/Cours/LiSP/4/extrait.pdf`, sa lecture est vivement recommandée.

[Ser07] Manuel Serrano. *HOP, un langage de programmation pour le Web 2.0.* Deux articles du magazine *Programmez!* numéros 104 et 105, en 2007. Voir aussi `http://hop.inria.fr`.

[SICP] Harold Abelson et Gerald Jay Sussman avec Julie Sussman. *Structure et Interprétation des Programmes Informatiques.* InterÉditions, 1989, épuisé. Préférer la seconde édition en anglais *Structure and Interpretation of Computer Programs*, lisible en ligne sur `http://www.mitpress.mit.edu/sicp`.

[Sit98] Dorai Sitaram. *Teach Yourself Scheme in Fixnum Days.* Voir `http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html`

[Ski98] Steven Skiena. *The Algorithm Design Manual.* Springer-Verlag, 1998. Voir `http://cs.sunysb.edu/~algorith`.

[Tur99] Alan Turing et Jean-Yves Girard. *La machine de Turing.* Éditions du Seuil, 1999.

Index

`!`, 401
`!!`, 401
$\hat{\lambda}$, 386
∼a, 48
`+ - * /`, 38
`+i`, 38
`...`, 232
`< <= > >=`, 37
`=`, 36, 37
`#:mutable`, 263
`#:transparent`, 263
`#:exists`, 296
`#:mode`, 296
`#f`, 39
`#lang`, 217, 301, 415
`#t`, 39
`2htdp/image`, 59, 415
`2htdp/universe`, 69, 415

A-liste, 136
abandon de continuation, 378, 382
`above`, 63
`above/align`, 63
`abs`, 37
accesseur, 152
`acos`, 38
acteur, 253
`adt-arbre23.rkt`, 197, 416
`adt-fbf.rkt`, 418
`adt-pile.rkt`, 417
affectation, 14, 31, 242, 371
    multiple, 103, 244, 276
afficheur, 152
aiguillage, 197
aléatoire entier, 35
aléatoire réel, 37
alignement
    d'images, 62, 63
    d'un panneau, 319
`all-defined-out`, 218
`all-from-out`, 237
`amb`, 392
analyse
    lexicale, 290, 343
    syntaxique, 27, 151, 346
`and`, 41, 225, 237
`angle`, 38
animation, 69, 264, 334
APL, 25, 145
appel par valeur, 43, 51, 248
appel récursif, 96
`append`, 130
`apply`, 145, 367, 405
arbre binaire
    d'expression, 161, 193, 416
    de recherche, 177
arbre-23 conditionnel, 197, 416
argument d'une fonction, 51
arité, 50
    variable, 147
arithmétique de Peano, 106
ASCII, 285
`asin`, 38
`assert`, 396
`assoc`, 136
`atan`, 38

atome (fbf), 209
axes de coordonnées, 60

backtrack, 383, 393, 395
balle rebondissante, 81
bash, 308
`begin`, 241, 371
`beside`, 63
`beside/align`, 63
Bezout, 100, 105
`big-bang`, 71
Bigloo, 18
`bitmap`, 65
`bitmap-dc%`, 335
`bitmap%`, 335
booléen, 39
`border`, 322
boucle, 103
boucle `do`
  fonctionnelle, 234
  impérative, 245
`brf`, 198
`build-list`, 130
`build-string`, 286
`build-vector`, 265
`button%`, 319, 321

`c--r`, 222
calcul formel, 201, 403
`call`, 193
`call-with-input-file`, 299
`call-with-input-string`, 302, 346
`call-with-output-file`, 295
`call/cc`, 384
`callback`, 320
capture de continuation, 378, 385, 387, 395
`car`, 221
caractère, 283
`case`, 166
`cdr`, 221
champ
  d'un objet, 315
  d'une structure, 79
`change-style`, 325
`char->integer`, 284
`char-ref`, 287
`char-set!`, 287
`char<=?`, 284
`check-expect`, 49
`check-within`, 49
chirurgie des doublets, 272
`circle`, 59
citation, voir `quote`
`class`, 314
client Web, 302
closure, voir fermeture
`collapse`, 325
combinateurs, 148
commentaire, 47
compilation d'un arbre, 194
complexe, 38
complexité, 92, 106–108, 111, 129–131, 134, 136, 139–141, 381
`compose`, 56, 116
compréhension de liste, 399
`cond`, 40, 237
conflit shift-reduce, 352
`conjugate`, 38
`cons`, 125, 220
constructeur, 152, 314
conteneur, 319
continuation, 120, 384
  à plusieurs variables avec CPS, 381
  avec CPS, 119, 174, 378, 384
  du toplevel, 387
`control-event%`, 320
convention du `.(`, 222
corps d'une fonction, 50
`cos`, 38
couper/copier/coller, 325
couple comme fonction, 57
CPS, voir continuation

crochets, 54, 228
cryptologie, 289
curryfication, 117, 397

`dc%`, 330
décimal (nombre), 36
`define`, 30, 371
`define-empty-tokens`, 344
`define-lex-abbrevs`, 345
`define-struct`, 79
`define-syntax`, 230
`define-tokens`, 344
`define-values`, 303
`define/override`, 317, 331
définition
  interne, 229
  locale, 53
`delay`, 404, 407
délégation de message, 254
démonstration automatique, 207
`denominator`, 35
dérécursiver avec CPS, 121
dérivée
  approchée, 56
  exacte, 205
dichotomie, 97, 180
dictionnaire, 30, 227, 243, 363
dimensions de l'écran, 322
discrétisation, 74
distance de deux points, 330
division euclidienne, 34
double buffer, 69, 335
drag and drop, 331
drapeau hollandais, 267
`draw-bitmap%`, 335
`draw-line`, 330
`draw-point`, 330

échange
  dans un vecteur, 266
  de variables, 249
échappement, 120, 386
écriture sur un port, 295
éditeur de polygone, 141
effet de bord, 243
`ellipse`, 59, 65
Emacs, 113, 114, 397
encodage, 285
ensemble, 137
  non ordonné, 137
  ordonné, 185
enveloppe, 96, 101
environnement, 227, 248, 363
  étendu, 228, 364
  de création, 114, 313, 367, 368
  global, 30, 363, 372
`eof-object?`, 300
`eq?`, 223, 372
`equal?`, 128, 222
équilibrage par AVL, 180
Eratosthènes (crible), 402
erreur de lecture, 299
étiquette, 199
Étudiant Avancé, 54
Étudiant avancé, 23, 32
euro, 284
`eval`, 298, 323
eval/apply, 366
eval4TEX, 376
évaluation, 41–42, 366
  en CPS, 378–379
  retardée, 404–405
`even?`, 34
exact, 34
`exact->inexact`, 37
`exact?`, 36
`except-out`, 237
exception, 225, 299, 373
`exn-fail?`, 226, 373
`exn-message`, 373
`exn:fail:read?`, 299
`exp`, 38

expansion d'une macro, 231
`expr->string`, 220
expression régulière, 150, 291, 305
`expt`, 38, 97

`false`, 39
fbf, 418
fenêtre, 319
Fermat, 112
fermeture, 115, 313, 367
feuille d'un arbre, 161
Fibonacci, 98, 121, 381, 399–400, 406, 410
fichier sur disque
 en écriture, 295
 en lecture, 299
file d'attente fonctionnelle, 175
`file-exists?`, 305
`file-stream-buffer-mode`, 304
`filter`, 132
`first`, 127
`floor`, 37
flot, 402, 408
`foldl`, 144
`foldr`, 143
fonction, 42, 47
 à mémoire, 251
 anonyme, 50
 en style équationnel, 132, 397
 itérative, 101, 378
`for`, 246
`for-each`, 248
`force`, 401, 404, 407
`format`, 61, 153, 295
forme spéciale, 40, 366
formule bien formée, 208, 418
`fprintf`, 295
`frame`, 64
`frame%`, 319
fréquence d'horloge, 336

GAMBIT, 18
garbage collector (GC), 51, 273
gcc, 242
`gcd`, 34–35
généraliser, 101, 118
générateur, 249, 382
`gensym`, 199, 287
gestion automatique de la mémoire, 51
`get-display-size`, 322
`get-event-type`, 320
`get-key-code`, 333
`get-output-string`, 307
`get-text`, 323
`get-value`, 320
`GOBACK`, 389
GOOGLE, 145
`GOTO`, 14, 390
grammaire, 346
graphe, 307
`graphics`, 256
Graphviz, 307
grep, 300
GUI, 318

hash-code, 275, 363
`hash-has-key?`, 364
`hash-set!`, 365
HASKELL, 43, 397
hauteur d'un arbre, 164
héritage, 316
 multiple, 317
`horizontal-panel%`, 321
horloge, voir fréquence d'horloge
How to Design Programs, 59
HTTP, 304
hygiène, 235, 249

`if`, 39, 237, 369
`imag-part`, 38
`image-height`, 61
`image-width`, 61
impératif, 14, 89, 242

implication, 208, 210
indentation, 40
`inexact->exact`, 37
inférence de type, 398
`inherit-field`, 317
insérer une image, 65
instruction, 241
`integer->char`, 284
`integer?`, 35
intégrale, 110
intelligence artificielle, 15
interface, 318
interprétation, 195, 361
irrationnel, 36
itération, 102, 173, 174

jeu du chaos, 257
`jmp`, 198, 387
joker, 48, 283

`key=?`, 75

lambda-calcul, 89
lambda-expression, 50, 52
`lang/posn`, 329
lazy, 396
Lazy Racket, 400
`lcm`, 34
`length`, 128
`let`, 227, 369
`let*`, 228, 237
`let-values`, 322
`let/ec`, 288
`letrec`, 229, 370
LeX, 343
lexème, 344
`lexer`, 345
liaison, 364
  dynamique, 114, 368, 375
  statique, 114
ligne de commande, 306
`line`, 59–60
Lisp, 15–17, 26, 52, 113, 221
LISP2TEX, 376
`list`, 129
`list-ref`, 133
`list?`, 223
liste, 125, 223
  circulaire, 274
  infinie, 399
  mutable, 270
`local`, 54
`log`, 38
logique des propositions, 116, 208

machine à pile, 193
macro, 230
  récursive, 231
Macro Stepper, 232
`magnitude`, 38
`make-base-namespace`, 298
`make-color`, 60
`make-hash`, 364
`make-object`, 315, 327
`make-polar`, 38
`make-rectangular`, 38
`map`, 143
MapReduce, 145
`match`
  avec un prédicat, 146
  sur les listes, 132
  sur les structures, 85, 188
  sur les vecteurs, 266
matrice, 267
`max`, 37
Maxima, 201
maxima, 84
`mcons` (`mcar`, *etc*), 270
`member`, 129, 225
mémo-fonction, 251
`menu%`, 325
message, 253, 315
`message%`, 319

méthode, 314
min, 37
modificateur, 152
module, 32, 154, 217, 244, 300
modulo, 34
molécule (fbf), 209
monde, 69
Monte-Carlo, 326
mot clé, 40, 230
mot réservé, 230, 232
mouse=?, 76
mutation
  d'un doublet, 270
  d'un vecteur, 265
  d'une chaîne, 287
  d'une structure, 263
  d'une variable, 242
  entre modules, 244
mzlib/string, 302, 323

nand, 208
niveau de langage, 23
nombre de chiffres, 44, 94
nombres, 33
  premiers, 112
  premiers entre eux, 35
not, 39
notation
  préfixée, 25
  scientifique, 37
notify-callback, 336
null?, 125, 223
number->string, 61, 94, 328
numerator, 35

odd?, 34
on-char, 333
on-draw, 71
on-event, 331
on-key, 75
on-mouse, 76
on-paint, 326, 329
on-tick, 71
one-liner, 145
only-in, 296
open-input-text-editor, 324
open-output-string, 307
open-output-text-editor, 324
or, 41, 225, 237
ordre
  applicatif, 43
  d'évaluation, 42
  normal, 43
  supérieur, 116, 142, 145
'outline, 59

paint-callback, 328
pair?, 221
paramètre d'une fonction, 50–51
parcours d'un arbre, 162
  en largeur, 163, 177
parenthèses, 27
parser, 347
parser-tools/lex, 343
parser-tools/yacc, 346
partie
  imaginaire, 38
  réelle, 38
PATH, 308
path->string, 305
Peano, 106
périodique (développement), 36, 158
pgcd, 99
pile, 97, 172, 417
  comme acteur, 255
  comme classe, 314
PLaneT, 220, 392
poids d'un opérateur, 26, 168
polymorphe, 140
port
  d'entrée, 297
  de sortie, 295

posn, 79, 329
précédence d'opérateur, 352
précision infinie, 34
prédicat, 39
pretty-print, 302
printf, 48, 61, 283, 295
priorité, voir précédence
procédure, 42
process, 306
programmation
 fonctionnelle, 47
 impérative, 241
 non déterministe, 392
 par objets, 15, 253, 314
 paresseuse, 396
promesse, 401, 407
provide, 154, 217–218
push, 193

quasiquote, 224
quicksort en HASKELL, 399
quote, 60, 127
quotient, 34

racine
 d'un arbre, 161
 d'une équation, 111
racket, 24, 308
racket, 217
racket/base, 298
racket/gui, 318
racket/mpair, 271
racket/system, 306
raise, 227
random, 34, 37
rationnel, 35, 80
read-accept-reader, 301
read-from-string, 302
read-from-string-all, 323
read-line, 299, 300, 346
real-part, 38
recherche en profondeur, 138
reconnaisseur, 152–153
record?, 72
rectangle, 59–60
récurrence, 89
 double, 98
 enveloppée, 96
 sans cas de base, 399
 terminale, 101
récursivité gauche, 346
redéfinition d'une méthode, 317
redex, 384
réel, 37
regexp-match, 291
regexp-replace, 294
réglages de DRRACKET, 24
règles de Wang, 212
rename-out, 237
repeat, 262
require, 154, 218, 219
rest, 127
return, 288, 379
reverse, 131
rotate, 63
rotation
 dans le plan complexe, 56
 dans un AVL, 183
round, 37

saut à étiquette, 198, 387–390
scale, 64
script, 308
send, 255, 315
séquence, 241, 371
séquent, 211
série
 de Taylor, 205
 formelle, 403
set!, 242, 371
show, 233, 321, 415
simplification formelle, 167, 202–203

`sin`, 38
situation générale, 267
SLaTEX, 376
smiley, 65
`'solid`, 59
souris, 76, 141, 329
sous-classe, 316
`sqrt`, 38
`stop-when`, 71
stream, 408
string, 60, 286
`string->number`, 44, 328
`string->symbol`, 287
`string-append`, 287
`string-length`, 94
`string=?`, 287
string?, 42
structure, 79
style MIT, 52
`style-delta%`, 325
`substring`, 286
`super`, 318
`symbol->string`, 287
symbole, 30
`syntax-rules`, 235
système d'exploitation, 305
`system`, 306

tabulation, 283
`take`, 399, 401
tampon, 304
`tan`, 38
`tcp-connect`, 303
teachpack, 31, 59, 69
tester une fonction, 48
`text`, 60–61
`text-field-enter`, 320
`text-field%`, 319
`text/font`, 60
théorème, 210
`this`, 254, 315
`thread`, 334
`timer%`, 336
token, 290, 344
toplevel, 28–29, 41, 47, 298, 372
tortue, 258
transformateur syntaxique, 234
tri
    par fusion, 140
    par insertion, 139
    par pivot, 399
`true`, 39
Turing, 89, 421
typage dynamique, 34
type abstrait, 152, 162, 217

`underlay`, 62
`underlay/align`, 62
Unicode, 283
UNIX, 306
UTF-8, 285

valeur d'un arbre, 166
`valrose.rkt`, 32
variable
    globale, 53
    liée, 114
    libre, 114, 313
    locale, 53, 369
vecteur, 265
vectorisation, 269
Vérifier, 104
`vertical-panel%`, 320
`void`, 242
Von Koch, 263
VRISC, 193

Wang, 210
`while`, 246
`with-handlers`, 226, 299, 373

`zero?`, 34
zipper, 186