

Reverse Polish Lisp 2

Manuel de référence

BERTRAND Joël

26 septembre 2026

Copyright © 1989 à 2026, BERTRAND Joël.

Manuel de référence maintenu par l'auteur, tous droits réservés pour tous pays. Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites.

RPL/2 ® est une marque commerciale déposée par l'auteur à l'Institut National de Protection Industrielle.

Die Mathematik ist das Instrument, welches die Vermittlung bewirkt zwischen Theorie und Praxis, zwischen Denken und Beobachten: sie baut die verbindende Brücke und gestaltet sie immer tragfähiger. Daher kommt es, daß unsere ganze gegenwärtige Kultur, soweit sie auf der geistigen Durchdringung und Dienstbarmachung der Natur beruht, ihre Grundlage in der Mathematik findet.

David Hilbert

Les mathématiques sont l'instrument qui permet de mettre en relation théorie et pratique, pensée et observation : elles forment la passerelle, toujours plus solide, qui les relie. C'est pourquoi toute notre civilisation, dans la mesure où elle repose sur la pénétration intellectuelle et sur l'exploitation de la nature, trouve ses racines dans les mathématiques.

David Hilbert

Table des matières

Genèse du RPL/2	26
Réflexions	29
Historique	31
RPL 6502, version 1.0	33
RPL 6809, version 2.0	33
RPL 386, version 3.0	33
RPL 387, version 3.x	33
RPL/2 ou RPL 4.x, une renaissance	34
Pourquoi un nouveau langage ?	34
 I Concepts fondamentaux	 39
1 Notations	41
1.1 Notation polonaise inverse	41
1.2 Notation algébrique	42
1.3 Notation infixé	42
2 Types de données	43
2.1 Types exportables	43
2.1.1 Scalaires	43
2.1.1.1 Entiers	45
2.1.1.2 Booléens	45
2.1.1.3 Réels	45
2.1.1.4 Complexes	45
2.1.2 Vecteurs	45
2.1.3 Matrices	46
2.1.4 Listes	46
2.1.5 Tables	46
2.1.6 Expressions	46
2.1.6.1 Expressions algébriques	46

2.1.6.2	Expressions RPN	46
2.1.7	Noms	47
2.1.8	Chaînes de caractères	47
2.1.9	Entiers Binaires	47
2.2	Types non exportables	48
2.2.1	Fichiers	48
2.2.2	Sockets	48
2.2.3	Bibliothèques	49
2.2.4	Processus	49
2.2.5	Connecteurs SQL	50
2.2.6	Mutexes	50
2.2.7	Sémaphores nommés	50
2.3	Types internes	50
3	Niveaux et fils d'exécution	51
3.1	Niveau d'exécution	51
3.2	Contexte	52
3.3	Fil d'exécution	52
3.3.1	Processus	53
3.3.2	Processus léger	53
4	Variables	55
4.1	Niveau de variable	55
4.2	Classes de variables	55
4.2.1	Variable interne	56
4.2.2	Variable privée	56
4.2.2.1	Variable globale	56
4.2.2.2	Variable automatique ou locale	56
4.2.2.3	Variable implicite	56
4.2.3	Variable partagée	57
4.2.4	Variable virtuelle	57
4.3	Masque de variables	57
4.4	Portée de variables	58
4.5	Visibilité de variables	58
4.6	Protection des variables	59
4.7	Héritage	59
4.8	Noms de variable	59
5	Définitions	61
5.1	Définitions intrinsèques	61
5.2	Définitions extrinsèques	61
5.3	Définitions utilisateur	61
II	Appel	63
6	Différentes utilisations	65
6.1	Fonctionnement interactif	65
6.2	Exécution de programmes	66
6.2.1	Programmes interprétés	66

6.2.2	Programmes compilés	66
6.2.3	Utilisation du <i>sha bang</i>	66
6.3	Passage d'arguments	67
6.3.1	Arguments	67
6.3.2	Programmes	67
6.4	Option de débogage	68
6.4.1	Débogage des programmes	68
6.4.2	Débogage de l'interprète	68
6.4.3	Contrôles	68
6.5	Interaction avec le système d'exploitation	68
6.6	Autres options	68
7	Ligne de commande	71
7.1	Fonctionnement interactif	72
8	Exécution de programmes	75
8.1	Préprocesseur	75
8.1.1	Commentaires	75
8.1.2	<code>#define x y</code>	75
8.1.3	<code>#defeval x y</code>	76
8.1.4	<code>#undef x</code>	76
8.1.5	<code>#ifdef x</code>	76
8.1.6	<code>#ifndef x</code>	76
8.1.7	<code>#ifeq x y</code>	77
8.1.8	<code>#ifneq x y</code>	77
8.1.9	<code>#else</code>	77
8.1.10	<code>#endif</code>	77
8.1.11	<code>#include "file"</code>	77
8.1.12	<code>#exec command</code>	77
8.1.13	<code>#eval expr</code>	77
8.1.14	<code>#if expr</code>	78
8.1.15	<code>#elif expr</code>	78
8.1.16	<code>#mode keyword</code>	78
8.1.17	<code>#line</code>	78
8.1.18	<code>#file</code>	78
8.1.19	<code>#date fmt</code>	79
8.1.20	<code>#error msg</code>	79
8.1.21	<code>#warning msg</code>	79
8.2	Organisation des programmes	79
8.3	Extension	79
9	Convention d'appel librpl	81
III	Objets, pile et variables	83
10	Modifications automatiques	85
10.1	Fonctionnement des routines d'évaluation	85
10.1.1	Cas d'un programme interprété	88
10.1.2	Cas d'un programme compilé	89

10.2	Opérations implicites	89
10.3	Opérations explicites	89
II	Manipulation des objets	91
II.1	Gestion de la pile	91
II.1.1	Clear	91
II.1.2	Depth	91
II.1.3	Last	91
II.2	Duplication d'éléments	91
II.2.1	Dup	91
II.2.2	Copy	92
II.2.3	Dup2	92
II.2.4	Dupn	92
II.2.5	Over	92
II.2.6	Pick	92
II.3	Suppression d'éléments	93
II.3.1	Drop	93
II.3.2	Drop2	93
II.3.3	Dropn	93
II.4	Modification de la hiérarchie	93
II.4.1	Swap	93
II.4.2	Rot	94
II.4.3	Roll	94
II.4.4	Rolld	94
II.4.5	Edit	94
II.5	Gestion des contextes	94
II.5.1	Pshcntxt	95
II.5.2	Pulcntxt	95
II.5.3	Dupcntxt	95
II.5.4	Dropcntxt	95
II.5.5	Swapcntxt	95
II.5.6	Clrcntxt	95
II.5.7	Exemple d'utilisation	95
12	Entrées et sorties	97
12.1	Sorties	97
12.1.1	Disp	97
12.1.2	Format	97
12.1.2.1	Séparateur décimal	98
12.1.2.2	Format standard	98
12.1.2.3	Format fixe	99
12.1.2.4	Format scientifique	99
12.1.2.5	Format ingénieur	99
12.1.2.6	Arrondi	100
12.1.2.7	Troncature	100
12.1.3	Clmf	100
12.2	Entrées	100
12.2.1	Input	100
12.2.2	Prompt	100
12.2.3	Key	101

12.3	Beep	101
13	Accessibilité des variables	103
13.1	Niveaux d'exécution	103
13.2	Évaluation implicite	104
13.3	Évaluation explicite	104
13.4	Liste des variables	106
13.4.1	Exemple	106
14	Variables globales	109
14.1	Création	109
14.2	Accès	109
14.3	Modification	110
14.4	Libération	110
14.5	Verrouillage	110
15	Variables locales	113
15.1	Création	113
15.2	Portée et visibilité	116
15.3	Modification	118
16	Arithmétique directe	119
16.1	Les quatre opérations	119
16.1.1	Addition	119
16.1.2	Soustraction	119
16.1.3	Multiplication	120
16.1.4	Division	120
16.2	Autres opérations	120
16.2.1	Inversion	120
16.2.2	Opposition	121
16.2.3	Conjugaison	121
IV	Contrôle	123
17	Conditions et tests	125
17.1	Tests simples	125
17.1.1	If...then...(else)...end	125
17.1.2	Ift	126
17.1.3	Ifte	126
17.2	Reprise sur erreur	127
17.2.1	Types d'erreurs	127
17.2.1.1	Erreurs système	127
17.2.1.2	Erreurs d'exécution	127
17.2.1.3	Exceptions mathématiques	127
17.2.2	Errn	127
17.2.3	Errm	127
17.2.3.1	Exemple	128
17.2.4	Clerr	128
17.2.5	Iferr...then...(else)...end	128

17.3	Tests multiples	129
17.3.1	If...then...elseif...then...(else)...end	129
17.3.2	Select...case...then...end...(default)...end	130
18	Boucles	133
18.1	Boucles définies	133
18.1.1	Boucle sans compteur	134
18.1.1.1	Start...(cycle)...next	134
18.1.1.2	Start...(cycle)...step	134
18.1.2	Boucle avec compteur	135
18.1.2.1	For...(cycle)...next	135
18.1.2.2	For...(cycle)...step	135
18.1.3	Exemples	136
18.2	Boucles indéfinies	136
18.2.1	While...repeat...end	136
18.2.2	Do...until...end	137
18.3	Instruction exit	137
19	Contrôle de l'exécution	139
19.1	Mode de fonctionnement	139
19.1.1	Indicateurs	139
19.1.2	Manipulation	139
19.1.2.1	Affectations	139
19.1.2.2	Tests	141
19.2	Exécution normale	141
19.2.1	Retour anticipé	141
19.2.2	Abandon	142
19.3	Débogage	142
19.3.1	Point d'arrêt	143
19.3.1.1	Exemple	143
19.3.2	Exécution pas à pas	143
19.3.3	Retour en exécution normale	144
V	Fonctions mathématiques	145
20	Les opérations de base	147
20.1	Notations	147
20.2	Addition : + ...	147
20.3	Soustraction : -	148
20.4	Multiplication : *	148
20.5	Division	149
20.5.1	Division standard : /	149
20.5.2	Inversion : inv	149
20.6	Puissance	149
20.6.1	Puissance standard : ** ou ^	149
20.6.2	Carré : sq	152
20.6.3	Racine carrée : sqrt	152
20.6.4	Racine $n^{\text{ième}}$: xroot	152

21	Constantes	155
21.1	Constantes booléennes : <code>true</code> et <code>false</code>	155
21.2	Constantes mathématiques	155
21.2.1	Base des logarithmes népériens : <code>e</code>	155
21.2.2	Entier de Gauß : <code>i</code>	155
21.2.3	Constante d'Archimède : π	156
21.3	Constantes physiques	156
22	Arithmétique générale	157
22.1	Arithmétique réelle	157
22.1.1	Incrémentation et décrémentation : <code>incr</code> et <code>decr</code>	157
22.1.2	Valeur absolue, module et norme de Frobenius : <code>abs</code>	157
22.1.3	Négation : <code>neg</code>	157
22.1.4	Non opération : <code>relax</code>	158
22.1.5	Signe : <code>sign</code>	158
22.1.6	Modulo : <code>mod</code>	158
22.1.7	Minimum et maximum : <code>min</code> et <code>max</code>	158
22.1.8	partie entière : <code>ip</code>	160
22.1.9	Partie fractionnaire : <code>fp</code>	160
22.1.10	Valeur plancher : <code>floor</code>	161
22.1.11	Valeur plafond : <code>ceil</code>	161
22.1.12	Mantisse : <code>mant</code>	161
22.1.13	Exposant : <code>xpon</code>	161
22.2	Arithmétique complexe	161
22.2.1	Partie réelle : <code>re</code>	161
22.2.2	Partie imaginaire : <code>im</code>	161
22.2.3	Conjugaison : <code>conj</code>	161
22.2.4	Arg	162
22.3	Conversions	162
22.3.1	$P \rightarrow r$	162
22.3.2	$R \rightarrow p$	162
22.3.3	$C \rightarrow r$	162
22.3.4	$R \rightarrow c$	162
22.3.5	$\rightarrow q$	162
22.4	Proportions	162
22.4.1	Instruction %	162
22.4.2	Instruction %CH	162
22.4.3	Instruction %T	162
23	Tests	165
23.1	Comparaisons	165
23.1.1	Égalité	165
23.1.2	Différence	165
23.1.3	Inférieur	165
23.1.4	Inférieur ou égal	165
23.1.5	Supérieur	165
23.1.6	Supérieur ou égal	165
23.1.7	Parmi	165
23.2	Opérateurs logiques	165
23.2.1	Et logique	165

23.2.2	Ou logique	165
23.2.3	Ou logique exclusif	165
23.2.4	Non	165
24	Arithmétique binaire	167
24.1	Opérations de conversion	168
24.1.1	$B \rightarrow r$	168
24.1.2	$R \rightarrow b$	168
24.2	Formats	168
24.2.1	Dec	168
24.2.2	Bin	168
24.2.3	Oct	168
24.2.4	Hex	168
24.2.5	Stws	168
24.2.6	Rcws	168
24.3	Opérations sur des octets	168
24.3.1	Rlb	168
24.3.2	Rrb	168
24.3.3	Slb	168
24.3.4	Srb	168
24.4	Opérations sur des bits	168
24.4.1	Asl	168
24.4.2	Asr	168
24.4.3	RI	168
24.4.4	Rr	168
24.4.5	Sl	168
24.4.6	Sr	168
24.4.7	Fonction et bit à bit	168
24.4.8	Fonction ou bit à bit	168
24.4.9	Fonction ou exclusif bit à bit	168
24.4.10	Fonction complément à 1	168
25	Fonctions trigonométriques	169
25.1	Cosinus	170
25.1.1	Cos	170
25.1.2	Acos	170
25.2	Sinus	170
25.2.1	Sin	170
25.2.2	Asin	170
25.3	Tangente	170
25.3.1	Tan	170
25.3.2	Atan	170
25.4	Fonctions de conversion	170
25.4.1	Deg	170
25.4.2	Rad	170
25.4.3	$\rightarrow hms$	170
25.4.4	$hms \rightarrow$	170
25.4.5	$D \rightarrow r$	170
25.4.6	$R \rightarrow d$	170
25.5	Arithmétique sexadécimale	170

25.5.1	Hms+	170
25.5.2	Hms-	170
26	Fonctions hyperboliques	171
26.1	Cosinus hyperbolique	171
26.1.1	Cosh	171
26.1.2	Acosh	171
26.2	Sinus hyperbolique	171
26.2.1	Sinh	171
26.2.2	Asinh	171
26.3	Tangente hyperboliques	171
26.3.1	Tanh	171
26.3.2	Atanh	171
27	Fonctions logarithmiques	173
27.1	Logarithme naturel	173
27.1.1	Ln	173
27.1.2	Lnpi	173
27.1.3	Exp	173
27.1.4	Expm	173
27.2	Logarithme vulgaire	173
27.2.1	Log	173
27.2.2	Alog	173
VI	Algèbre linéaire	175
28	Vecteurs et matrices	177
28.1	→array	178
28.2	array→	178
28.3	Diag→	178
28.4	→diag	178
28.5	Size	178
28.6	Idn	178
28.7	Trn	178
28.8	Redimensionnement	178
28.9	Con	178
28.10	Col+	178
28.11	Col-	178
28.12	Col→	178
28.13	→col	178
28.14	Row+	178
28.15	Row-	178
28.16	Row→	178
28.17	→row	178
28.18	Échange de colonnes	178
28.19	Échange de lignes	178
28.20	Rci	178
28.21	Rcij	178
28.22	Get	178

28.23	Geti	178
28.24	Getc	178
28.25	Getr	178
28.26	Put	178
28.27	Puti	178
28.28	Putc	178
28.29	Putr	178
28.30	Min	178
28.31	Max	178
28.32	Sq	178
28.33	Produit scalaire	178
28.34	Produit vectoriel	178
28.35	Norme de colonne	178
28.36	Norme de ligne	178
29	Résolution	179
29.1	Inversion	179
29.2	Système linéaire	179
29.3	Cond	179
29.4	Rank	179
29.5	Déterminant	179
29.6	Moindres carrés	179
29.7	Moindres carrés généralisés	179
29.8	Résidus	179
30	Décompositions	181
30.1	Vecteurs propres	181
30.2	Vecteurs propres généralisés	181
30.3	Décomposition de Cholesky	181
30.4	Décomposition LU de Crout	181
30.5	Décomposition LQ	181
30.6	Décomposition QR	181
30.7	Décomposition de Schur	181
30.8	Décomposition en valeurs singulières	181
VII	Analyse	183
31	Analyse de Fourier	185
31.1	Transformée discrète	185
31.1.1	DFT	185
31.1.2	IDFT	185
31.2	Transformée rapide	185
31.2.1	FFT	185
31.2.2	IFFT	185

32	Calcul différentiel et intégral	187
32.1	Dérivation et intégration	187
32.1.1	Der	187
32.1.2	Int	187
32.2	Développements limités	187
32.2.1	Taylr	187
32.2.2	Mclrin	187
VIII	Fonctions spéciales	189
33	Fonctions mathématiques	191
33.1	Fonctions Γ	191
33.2	Fonction de Bessel	191
34	Conversion d'unités	193
35	Fonctions temporelles	195
35.1	Horodatage	195
35.1.1	Date	195
35.1.2	Jdate	195
35.1.3	Rdate	195
35.2	Attente	195
35.2.1	Alarm	195
35.2.2	Wait	195
36	Accès au système d'exploitation	197
36.1	Syseval	197
36.2	Temps processeur consommé	197
36.3	Journalisation	197
36.4	Répertoire de travail	197
37	Informations diverses	199
37.1	Mémoire utilisée	199
37.2	Version	199
37.3	Copyright	199
37.4	Garantie	199
37.5	Splash	199
37.6	Aide	199
37.7	Trace	199
38	Profilage	201
38.1	Pshprfl	201
38.2	Pulprfl	201
39	Autres fonctions	203
39.1	Vérification	203

IX	Probabilités et statistiques	205
40	Probabilités	207
40.1	Itinialisation d'un générateur	207
40.1.1	Rdgn	207
40.2	Rdz	207
40.3	Tirages aléatoires	207
40.4	Loi uniforme	207
40.5	Loi normale	207
41	Dénombrement	209
41.1	Fact	209
41.2	Arrangements	209
41.3	Permutations	209
42	Statistiques	211
42.1	Matrice de statistique	212
42.1.1	Destruction	212
42.1.2	S+	212
42.1.3	S-	212
42.1.4	Stos	212
42.1.5	Rcls	212
42.1.6	Spar	212
42.1.7	Xcol	212
42.1.8	Ycol	212
42.1.9	Cols	212
42.1.10	Nombre d'éléments	212
42.2	Valeurs courantes	212
42.2.1	Maxs	212
42.2.2	Mins	212
42.2.3	Sx	212
42.2.4	Sx2	212
42.2.5	Sy	212
42.2.6	Sy2	212
42.2.7	Sxy	212
42.2.8	Tot	212
42.3	Corrélation	212
42.4	Moyenne	212
42.5	Variance et écart-type	212
42.5.1	Var	212
42.5.2	Pvar	212
42.5.3	Sdev	212
42.5.4	Psdev	212
42.5.5	Cov	212
42.5.6	Pcov	212
42.6	Graphiques	212
42.6.1	Scls	212
42.6.2	Drws	212

43	Lois de probabilité cumulées	213
43.1	Distribution de Laplace-Gauß dite normale	213
43.2	Distribution du χ^2	213
43.3	Distribution de Fisher	213
43.4	Distribution de Student	213

X Accès aux éléments constituant les objets 215

44	Chaînes de caractères	219
44.1	→str	219
44.2	str→	219
44.3	Size	219
44.4	Chr	219
44.5	Num	219
44.6	Sub	219
44.7	Repl	219
44.8	Pos	219
44.9	Tokenize	219
44.10	Trim	219
44.11	Ucase	219
44.12	Lcase	219
44.13	Recode	219
44.14	Localization	219
44.15	Currenc	219

45	Vecteurs et matrices	221
45.1	Get	221
45.2	Put	221
45.3	Geti	221
45.4	Puti	221
45.5	Size	221

46	Listes	223
46.1	→list	223
46.2	list→	223
46.3	Get	223
46.4	Sub	223
46.5	Put	223
46.6	Geti	223
46.7	Puti	223
46.8	Size	223
46.9	Repl	223
46.10	Pos	223
46.11	Head	223
46.12	Tail	223
46.13	Revlist	223
46.14	Sort	223

47	Expressions	225
47.1	Evaluation	225
47.2	Size	225
47.3	Obget	225
47.4	Obsub	225
47.5	Exget	225
47.6	Exsub	225

48	Tables	227
48.1	Crtab	227
48.2	→table	227
48.3	table→	227
48.4	Get	227
48.5	Put	227
48.6	Size	227
48.7	Pos	227
48.8	Sort	227

XI Fichiers et sockets 229

49	Variable virtuelle	231
49.1	Store	231
49.2	Recall	231

50	Fichiers	233
50.1	Open	234
50.2	Close	234
50.3	Create	234
50.4	Delete	234
50.5	Append	234
50.6	Rewind	234
50.7	Seek	234
50.8	Backspace	234
50.9	Backspace	234
50.10	Inquire	234
50.11	Sync	234
50.12	Lock	234
50.13	Unlock	234
50.14	Wflock	234
50.15	Fichiers à accès séquentiel	234
50.15.1	Read	234
50.15.2	Format	234
50.15.3	Write	234
50.16	Fichiers à accès direct	234
50.16.1	Read	234
50.16.2	Format	234
50.16.3	Write	234
50.17	Fichiers à accès indexé	234
50.17.1	Read	234

50.17.2	Format	234
50.17.3	Write	234
51	Bases de données	235
51.1	Sqlconnect	235
51.2	Sqldisconnect	235
51.3	Sqlquery	235
52	Sockets	237
52.1	Domaine Unix	237
52.2	Domaine IP	237
52.3	Instructions génériques	237
52.3.1	Open	237
52.3.2	Close	237
52.3.3	Format	237
52.3.4	Read	237
52.3.5	Write	237
52.3.6	Wfsock	237
52.3.7	Target	237
XII	Processus	239
53	Processus courant	241
53.1	Suspend	241
53.2	Stop	241
53.3	Continue	241
53.4	Nrproc	241
53.5	Daemonize	241
53.6	Sched	241
53.7	Yield	241
53.8	Wfproc	241
53.9	Traitement différé des requêtes d'arrêt	241
53.9.1	cstop	241
53.9.2	rstop	241
53.10	Fusibles	241
53.10.1	Fuse	241
53.10.2	Rfuse	241
53.10.3	Clrfuse	241
54	Processus détachés	243
54.1	Vie d'un processus détaché	243
54.2	Detach	243
55	Processus légers	245
55.1	Vie d'un processus léger	245
55.2	Spawn	245

56	Mutexes et sémaphores	247
56.1	Mutexes	247
56.1.1	Crmtx	247
56.1.2	Clrmtx	247
56.1.3	Mtxlock	247
56.1.4	Mtxtrylock	247
56.1.5	Mtxunlock	247
56.1.6	Mtxstatus	247
56.2	Sémaphores	247
56.2.1	Crsmphr	247
56.2.2	Clrsmphr	247
56.2.3	Smphrdecr	247
56.2.4	Smphrtrydecr	247
56.2.5	Smphrinchr	247
56.2.6	Smphrgetv	247
57	Interruptions	249
57.1	Stoswi	249
57.2	Rclswi	249
57.3	Clrswi	249
57.4	Swi	249
57.5	Swilock	249
57.6	Swiunlock	249
57.7	Swiqueue	249
57.8	Swistatus	249
57.9	Isci	249
57.10	Wfswi	249
58	Communication interprocessus	251
58.1	Vers le processus père	251
58.1.1	Par interruption	251
58.1.2	Par scrutation	251
58.1.3	Mécanisme de communication	251
58.1.3.1	Send	251
58.1.3.2	Recv	251
58.1.3.3	Wfdata	251
58.1.3.4	Wfack	251
58.2	Vers le processus fils	251
58.2.1	Peek	251
58.2.2	Poke	251
58.2.3	Wfpoke	251
58.2.4	Atpoke	251
58.2.5	clratpoke	251
XIII	Graphisme	253
59	Instructions générales	255
59.1	Variables	256
59.1.1	Steq	256

59.1.2	Rceq	256
59.2	Paramètres	256
59.2.1	Indep	256
59.2.2	Depnd	256
59.2.3	Pmin	256
59.2.4	Pmax	256
59.2.5	Res	256
59.2.6	Ppar	256
59.3		256
59.4	Cllcd	256
59.5	Redraw	256
59.6	Drax	256
59.7	Dgtiz	256
59.8	Persist	256
59.9	Label	256
59.10	Title	256
59.11	Keylabel	256
59.12	Keytitle	256
59.13	Dimensions	256
59.13.1	Autoscale	256
59.13.2	Scale	256
59.13.3	Slicescale	256
59.13.4	Logscale	256
59.13.5	Centr	256
59.13.6	Axes	256
59.13.7	*d	256
59.13.8	*h	256
59.13.9	*s	256
59.13.10	*w	256
59.13.11	eyept	256
60	Sauvegarde et récupérations	257
60.1	Lcd→	257
60.2	→lcd	257
61	Dessin	259
61.1	Newplane	259
61.2	Line	259
61.3	Mark	259
61.4	Plot	259
62	Fonctions	261
62.1	Types de fonction	261
62.1.1	Function	261
62.1.2	Parametric	261
62.1.3	Polar	261
62.1.4	Wireframe	261
62.1.5	Slice	261
62.2	Draw	261

63	Statistiques	263
63.1	Type de tracé	263
63.1.1	Plotter	263
63.1.2	Scatter	263
63.1.3	Histogram	263
63.2	Drws	263
63.3	Scls	263
63.4	Nuages de points	263
63.5	Histogrammes	263
XIV	Impression	265
64	Gestion de l'impression	267
65	Commandes	269
65.1	Format du papier	269
65.2	Effacement des fichiers graphiques	269
65.3	Print	269
65.4	Impression de données	269
65.4.1	pri	269
65.4.2	prst	269
65.4.3	prstc	269
65.4.4	prusr	269
65.4.5	prvar	269
65.4.6	prmd	269
65.5	Impression de graphiques	269
XV	Interfaces externes	271
66	RPL/C	273
66.1	Définition du langage	273
66.2	Utilisation d'une bibliothèque	273
66.2.1	Use	273
66.2.2	Remove	273
66.2.3	Externals	273
67	Convention d'appel	275
67.1	Depuis une convention C	275
67.2	Vers une convention C	275
XVI	Optimisations	277
68	Du bon usage des variables	279
68.1	Création de variables	279
68.2	Utilisation de la pile	279
69	Bibliothèques partagées	281

XVII	Fonctionnement interne	283
70	Objets	285
71	Utilisation de la mémoire	287
71.1	Allocateur	287
71.2	Mémoire cache	287
72	Gestion des processus	289
72.1	Communication interprocessus	289
72.2	Utilisation des signaux	289
XVIII	Exemples	291
73	Programmes RPL/2	293
73.1	Premier exemple simple	293
73.2	Programme complexe	295
74	Bibliothèques	311
	Index	314

Avant-propos

Genèse du RPL/2

Réflexions

LORSQU'AU début des années 1990, mon activité m'a conduit à écrire des algorithmes de traitement du signal, je me suis aperçu rapidement qu'il n'existait aucun langage informatique atteignant le formalisme mathématique minimal nécessaire à leurs implantations efficaces par des chercheurs qui n'étaient pas des spécialistes en analyse numérique.

La plupart des langages informatiques disponibles et portables avait été conçus par des informaticiens pour des informaticiens et la mode était aux langages objets comme le C++ voire à Java qui venait d'apparaître. Contrairement aux langages procéduraux comme le C, les langages objets permettent de définir assez simplement de nouveaux types de données et d'y associer un certain nombre d'opérations. Ils parlent de classes et de méthodes, de bibliothèques, d'héritages multiples et de polymorphisme, concepts bien souvent étrangers au mathématicien qui se trouve à les utiliser sans en comprendre tous les tenants et les aboutissants.

Ces langages étaient souvent perçus comme la panacée car ils offraient de possibilités d'extensions que ne permettaient pas les langages qui n'étaient pas orientés objet. Toute une série de bibliothèques ont vu le jour pour adjoindre aux langages objet des capacités mathématiques évoluées, qu'il s'agisse de calculs sur l'ensemble des complexes, d'algèbre linéaire voire d'analyse numérique. Si du point de vue du programmeur ces nouvelles fonctionnalités étaient séduisantes, elles sont du point de vue du processeur extrêmement coûteuses. En effet, au langage de base s'ajoute une multitude de classes, d'héritages et de fonctions virtuelles qui rendent le code effectivement exécuté par la machine peu efficace en terme de temps de calcul ou d'occupation de la mémoire. En ajoutant à la notion de langage objet celle des ramasse-miettes permettant de libérer la mémoire de façon asynchrone, nous obtenons une famille de langages certes d'une utilisation aisée mais qui, du point de vue de la machine, gaspillent les ressources disponibles. Une boutade prétend qu'en C++, lorsqu'une fonction demande à avoir accès à une banane, il faut qu'elle appelle un singe pour qu'il apporte cette banane sur un plateau et que ce singe amène malgré lui tous ses copains, c'est-à-dire la moitié de la jungle. C'est malheureusement assez vrai.

Le langage Fortran était à la même époque considéré à tort comme obsolète. La norme Fortran 90 venait pourtant d'être publiée et permettait de corriger un Fortran 77 assez rébarbatif puisqu'il utilisait encore des lignes au format fixe hérité des cartes perforées. Compiler les caractères sur un terminal n'était pas très pratique d'autant que très peu d'éditeurs de texte étaient capables d'être configurés correctement pour s'adapter aux spécificités des

RÉFLEXIONS

fichiers source du Fortran. Si les normes Fortran 90 puis 95 ont été réellement un progrès, les premiers compilateurs portables Fortran 95 n'ont été disponibles qu'au début des années 2000.

Ainsi, entre 1990 et 2000, alors que j'avais sur mon bureau une station Alpha, la seule solution pour effectuer du calcul scientifique était soit d'utiliser un langage objet particulièrement inefficace, soit d'utiliser un langage obsolète. Je ne parlais pas des outils comme Matlab qui commençaient à faire leur apparition, ces outils n'étant ni portables ni efficaces.

Toutes ces raisons m'ont conduit à une réflexion sur les langages informatique et à concevoir un langage de haut niveau qui devait au moins satisfaire aux conditions suivantes :

- être portable au moins sur tout système POSIX, BSD ou System V ;
- fonctionner de la même façon sur toutes les architectures matérielles cible ;
- être capable d'utiliser simultanément tous les processeurs d'un calculateur ;
- être capable de fonctionner sur des machines à ressources limitées, typiquement sur des calculateurs embarqués ;
- être capable de manipuler simplement les objets mathématiques de base (matrices, listes, tables, expressions aléatoires...);
- savoir intercepter les erreurs et fournir un mécanisme de reprise sur erreur efficace ;
- utiliser efficacement la mémoire ;
- procurer un mécanisme simple d'interface avec d'autres langages permettant d'ajouter de nouvelles instructions au cœur du langage ;
- être utilisable par un utilisateur qui n'est pas un spécialiste de l'informatique ;
- être aussi rapide que possible,

et pour des raisons de commodités, je désirais aussi que ce langage sépare le fond de la forme — l'algorithme des contingences matérielles comme la gestion des ressources du système que sont par exemple les fils d'exécutions ou les mécanismes d'allocation et de libération de la mémoire.

L'ensemble de ces conditions conduit à une complexité que n'ont pas les langages traditionnels. Le coût de ces exigences en terme de temps de calcul est non négligeable et a abouti au choix d'un langage utilisant la notation polonaise inversée puisque celle-ci ne requiert pas d'analyse de ligne. Le gain de rapidité d'exécution apportée par cette logique de programmation a permis d'optimiser les routines de calcul de manière à atteindre une précision optimale sans dégradation notable des performances.

Les concepts de base de ce langage sont directement issus du langage RPL¹ développé par Hewlett-Packard pour son calculateur HP-28S. Il contient de nouvelles primitives le transformant en un langage réellement moderne. Le RPL/2 n'est pas une simple évolution du RPL, il s'agit d'un nouveau langage semi-compilé qui ne cherche pas à être compatible avec le RPL originel.

Tous les concepts de ce langage ont été pensés pour être portables. Il s'ensuit qu'à partir du moment où un programme RPL/2 fonctionne sur une machine donnée, il fonctionnera de la même manière sur une autre machine même si celle-ci est d'architecture matérielle radicalement différente. Toute la gestion de la mémoire est faite de manière synchrone et est

¹Reverse Polish Lisp d'après ses concepteurs, ROM-based Procedural Language d'après Hewlett-Packard

laissée à la discrétion de l'interprète qui gère un allocateur spécifique permettant de cacher la mémoire. L'utilisateur peut alors se concentrer sur l'algorithme qu'il plante et non sur sa transcription informatique. Le RPL/2 est aussi conçu comme un langage ouvert et extensible. L'utilisateur peut adjoindre aux instructions intrinsèques du langage ses propres instructions, écrites en RPL/C, un langage proche du C.

RÉFLEXIONS

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Historique

JUSQU'à sa version 4, ce langage était un langage expérimental qui n'avait d'autre intérêt que celui de permettre une réflexion sur ce qu'il devait être et sur les voies à suivre. En particulier, sa syntaxe n'était pas figée et son fonctionnement interne loin d'être optimal. Plusieurs pistes ont été explorées pour atteindre le but recherché, nombre d'entre elles ont été des voies sans issue.

RPL 6502, version 1.0

La première version opérationnelle de ce langage fut codée en 1989 sur un Oric Atmos 48 Ko en Tangerine Basic V1.1. Elle ne permettait de travailler qu'en notation polonaise inverse stricte et sur des réels codés en simple précision (32 bits). L'exiguïté de la mémoire disponible ainsi que la modeste puissance de la machine rendait impossible les traitements des objets complexes comme les matrices. Pour les mêmes raisons, seul le mode interactif était disponible.

RPL 6809, version 2.0

Une deuxième version, commencée en 1990, n'a jamais été achevée faute d'avoir pu coder un éditeur efficace et compact. Elle était censée fonctionner sur un SMT-Goupil G3/6809 muni de 320 Ko de mémoire, mais toute la mémoire au delà de 64 Ko n'était accessible qu'au travers de tableaux virtuels sur un système de fichiers en mémoire. Cette station fonctionnait sous TSC-Flex9 et l'interprète était codé en Sbasic. La grande nouveauté résidait en l'exploitation des disques, en l'introduction de commandes de gestion des fichiers, ainsi qu'en l'apparition des calculs symboliques.

RPL 386, version 3.0

En 1991 a débuté l'écriture d'une troisième version d'interprète fonctionnant sur un IBM PS/2 modèle P70, un i386DX/16 MHz d'une puissance très raisonnable pour l'époque. Les instructions graphiques font leurs apparitions. Le séquenceur reste toujours écrit en Basic, mais cette fois-ci en Turbo-Basic V1.1 sous IBM-DOS 5.00.

HISTORIQUE

RPL 387, version 3.1

L'ajout d'un i387DX dans la machine de développement a rendu possible l'adjonction de fonctions utilisant le coprocesseur arithmétique. Au regard du gain apporté par ce composant — un facteur cinq en terme de vitesse d'exécution des calculs en virgule flottante —, il a été possible d'effectuer les calculs en double précision sur l'ensemble des complexes. Pour simplifier la gestion des différents objets et la complexité du code qui devait tenir dans les 640 Ko du DOS, l'interprète considérait que toutes les valeurs numériques étaient des nombres complexes contenus dans des chaînes de caractères. Cette hypothèse était faite au détriment des performances qui étaient particulièrement mauvaises. Toute la mémoire était gérée de manière statique et tous les objets étant enregistrés sous forme de chaînes de caractères, il était hors de question de chercher à traiter des objets aussi complexes que des listes, des matrices ou simplement des vecteurs. En revanche, cette version a permis de poser les bases de l'évaluation des expressions algébriques.

RPL/2 ou RPL 4.x, une renaissance

Au début de 1998, les choix pris une dizaine d'années plus tôt n'étaient plus pertinents et entravaient le développement de l'interprète. Si en 1989 ces choix se justifiaient pleinement, la puissance des nouvelles stations de travail ainsi que les systèmes d'exploitation de type Unix qui commençaient à sortir des ordinateurs des centres de calculs rendaient caducs le fait de travailler en mémoire statique, d'effectuer tous les calculs en complexe ou de traiter les matrices comme des chaînes de caractères.

Par ailleurs, la disposition grâce au projet GNU de compilateurs C et Fortran portables et efficaces m'a incité à réécrire totalement l'interprète de manière à obtenir un outil portable et sans aucune autre limite que les limites imposées par les machines hôtes, qu'il s'agisse de la mémoire, de la gestion des fils d'exécution ou plus généralement des accès aux ressources matérielles. Les niveaux de variables, qui peuvent être volatiles, statiques ou partagées, verrouillées ou non, font leur apparition. Un préprocesseur est ajouté, facilitant l'écriture des programmes, ainsi qu'un semi-compilateur et qu'un ensemble de fonctionnalités nouvelles.

Les tâtonnements des trois versions précédentes ont permis de stabiliser le langage qui n'a plus subi d'évolution majeure rendant une version nouvelle incompatible avec la précédente.

Le RPL/2 s'ouvre vers les autres langages puisqu'un programme RPL/2 peut maintenant être appelé à partir d'un autre langage pour peu qu'il soit possible d'avoir depuis ce langage une convention d'appel de type C. Il est aussi possible d'ajouter de nouvelles fonctions au langage sous la forme de bibliothèques partagées écrites en RPL/C.

Un debugger interne est ajouté. Il devient possible d'analyser des erreurs *post mortem* et d'optimiser ses programmes grâce à un outil de profilage. Les programmes sont soit interprétés soit semi-compilés. Cette semi-compilation n'est pas une compilation qui génère un fichier binaire autonome. Elle revient à chaîner en mémoire les différents objets pour n'exécuter que du code utile et augmenter sa vitesse de traitement.

Pourquoi un nouveau langage ?

Critère de Turing

Le RPL/2 est un langage procédural Turing-équivalent à inférence de type. Chaque procédure — appelée définition — se comporte comme une nouvelle instruction du langage prenant ses arguments sur la pile et renvoyant ses résultats sur la même pile.

Types et variables

Le RPL/2 est un langage à inférence de type. Un type est associé à un objet lors de sa création et le type de cet objet peut varier au cours de la vie de cet objet en fonction des opérations menées sur cet objet. Il faut noter que les calculs sont toujours effectués au moindre coût et à précision maximale, ce qui revient à toujours associer à un objet le type minimal permettant d'en garder une représentation interne exacte.

Le concept de variable est très différent de celui qui prévaut dans la plupart des autres langages. Une variable est en RPL/2 un objet qui contient un nom et qui est associé à un autre objet. Ainsi, seul le nom d'une variable et non le type de l'objet associé, doit être déclaré. Contrairement à un langage classique, une variable n'a pas à être associée à un type car un objet peut être manipulé en dehors de toute référence à une variable. L'objet associé à une variable contient un type et la variable ne fait que donner un accès à cet objet.

Les variables sont de différentes portées :

- les variables non volatiles : les variables non volatiles sont des variables qui survivent à la fin de l'exécution d'un programme. Typiquement, il s'agit de variables écrites sur une mémoire de masse et qui peuvent être appelées d'une instance RPL/2 à une autre ;
- les variables globales : les variables globales sont des variables qui sont accessibles depuis tous les points d'un programme. Elles restent accessibles à travers une commande spécifique si elles sont masquées par une variable automatique mais ne sont pas partagées entre deux fils d'exécution ;
- les variables locales : les variables locales sont des variables automatiques. À ce titre, elles n'existent que dans un bloc de programme. Elles sont créées automatiquement lors de l'ouverture du bloc et sont détruite à sa fermeture. Les objets qu'elles contenaient sont alors perdus ;

POURQUOI UN NOUVEAU LANGAGE ?

- les variables statiques : les variables statiques sont des variables automatiques. À la différence des variables locales, les objets qu'elles contiennent sont sauvegardés à la fin du bloc les définissant. Si elles doivent être recréées, elles ne seront plus initialisées par leurs valeurs par défaut mais par l'objet sauvegardé précédemment ;
- les variables partagées : les variables partagées sont des variables statiques partagées entre plusieurs fils d'exécution communs au même processus.

Domaines d'application

Si les premières versions du RPL/2 étaient orientées vers le calcul scientifique, le RPL/2 se veut actuellement d'usage le plus général possible. Il possède ce qu'il faut pour interagir efficacement avec un système d'exploitation — jusqu'à l'écriture de daemons au sens Unix du terme —, des fonctions de traitement de données statistiques, des possibilités de représentation graphique, des fonctions de mise en forme de résultat sous la forme de fichiers PostScript générés par un moteur $\text{\TeX}$, des instructions de gestion de fichiers et de sockets, des instructions permettant d'accéder à des bases de données SQL, cette liste n'étant de loin pas exhaustive. De plus, un langage de macroinstructions appelé RPL/C permet de l'étendre en ajoutant aux fonctions intrinsèques du langage des fonctions compilées définies par l'utilisateur pour un besoin spécifique.

Particularités

Tous les problèmes d'allocation de mémoire sont traités implicitement. À aucun moment un programme RPL/2 n'est autorisé à demander explicitement une allocation. Ces allocations implicites sont totalement transparentes et utilisent un mécanisme de cache pour qu'elles soient le plus efficaces possible. Ainsi, le programmeur n'a jamais à se préoccuper des problèmes inhérents à la gestion de la mémoire, qu'il utilise des listes, des matrices, des tables ou tout autre objet du langage. Par ailleurs, lorsqu'un programme lance un fil d'exécution parallèle, même si celui-ci s'exécute dans le même plan d'adressage, ce nouveau fil d'exécution utilise un contexte qui lui est propre. L'utilisateur n'a ainsi pas à se préoccuper des accès concurrents aux données, ceux-ci étant par construction impossibles sauf s'il utilise des variables partagées.

Le mécanisme de reprise sur erreur ne fonctionne ni par exception comme en C++ ni par valeur de retour comme en C. Il fonctionne par interruption du programme, cette interruption pouvant dans certains cas être récupérée par une structure spécifique du langage. Le RPL/2 distingue trois types d'erreurs :

- erreur système : l'erreur n'est pas due au programme RPL/2 en cours d'exécution mais à des ressources indisponibles (dépassement des capacités mémoire par exemple) et n'est pas récupérable. Le RPL/2 essaie d'arrêter le mieux possible le programme en cours qui ne peut continuer ;
- erreur d'exécution : l'erreur est due au programme en cours d'exécution. Elle peut être récupérée et traitée par le mécanisme de reprise sur erreur classique ;
- exception : une exception est une erreur dont le traitement dépend de paramètres de configuration. Un exemple typique est une division par zéro qui, en fonction de la configuration, renvoie une exception de type division par zéro qui doit être récupérée comme une erreur d'exécution ou une valeur numérique de type *+infinity*, *-infinity* ou *undefined*.

Si une erreur ne peut pas être récupérée, soit parce qu'il s'agit d'une erreur système, soit parce qu'aucun mécanisme de reprise sur erreur n'a été utilisé, le RPL/2 peut écrire son état interne dans un fichier de débogage. Ce fichier de débogage contient :

- la pile des processus;
- la pile système du processus courant;
- la pile *last*, optionnelle, contenant les arguments de l'instruction ayant levé l'erreur;
- un pointeur sur l'instruction fautive;
- l'ensemble des variables courantes.

Optimisation des programmes

Un outil de profilage permet de connaître le nombre d'appels aux définitions d'un programme ainsi que le temps passé dans chacune de ces fonctions. Il existe deux niveaux de profilages, le premier indiquant le temps passé dans les différentes définitions et instructions extrinsèques, le second comptabilisant jusqu'au temps passé dans les instructions intrinsèques.

Accès aux données externes

Le RPL/2 gère nativement des bases de données de type SQL ainsi que des fichiers et des sockets des domaines locaux, IPv4 et IPv6. Les données sont enregistrées ou transmises sous forme formatée, non formatée (les données contiennent des informations rendant impossible une lecture erronée) ou flux (données brutes sans information annexe). Les fichiers sont quant eux de type séquentiel, à accès direct ou à accès indexé.

Utilisation interactive

Il est possible d'utiliser l'interprète en mode interactif, grâce à une ligne de commande. Cela permet d'effectuer des calculs simples comme ils pourraient être conduits sur un calculateur de poche fonctionnant en notation polonaise inverse.

POURQUOI UN NOUVEAU LANGAGE ?

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Première partie

Concepts fondamentaux

1

Notations

POUR atteindre une certaine efficacité de traitement, le fonctionnement interne du RPL/2 se fait exclusivement en notation polonaise inversée. Cette notation est la logique principale utilisée par les programmes écrits en RPL/2. Néanmoins, pour des raisons de lisibilité, deux autres notations peuvent être rencontrées : la notation algébrique pour les expressions symbolique et la notation infixe pour les structures de contrôle.

1.1 Notation polonaise inverse

La logique principalement utilisée par le séquenceur RPL/2 s'appuie sur une formalisation mathématique connue sous le nom de « notation polonaise inverse » et développée par le logicien polonais Jean Łukasiewicz (1878-1956). Si la notation algébrique conventionnelle place les opérateurs *entre* les nombres ou variables d'une expression lors de son évaluation, la notation introduite par Łukasiewicz spécifie les opérateurs *avant* les variables. Une variante de cette logique spécifie les opérateurs *après* les variables et s'appelle alors « notation polonaise inverse ». L'idée principale de la notation polonaise inverse est que les nombres et les autres objets sont tout d'abord saisis et, après eux, une commande qui agit sur ces nombres ou objets appelés « arguments » de cette commande. La « pile opérationnelle » est une suite de ces objets qui attendent d'être utilisés. La plupart des commandes prennent des arguments dans cette pile et y renvoient leurs résultats d'où ils peuvent à leur tour être utilisés comme arguments d'autres commandes.

Le RPL/2 utilise cette logique pour plusieurs raisons, la première étant qu'elle évite la coûteuse interprétation nécessaire à une l'évaluation de la notation algébrique, mais aussi parce qu'elle présente la souplesse nécessaire permettant l'exécution uniforme des calculs permis par le langage. En effet, tout devient un objet soumis à évaluation, qu'il s'agisse d'un mot clef du langage, d'un nom de variable ou d'une simple constante. Toutes les opérations, même celles qui ne peuvent pas être exprimées sous forme algébrique, sont exécutées de la même manière, les arguments étant pris dans la pile et les résultats y étant renvoyés. Cela étant dit, l'utilisation de la notation polonaise inverse reste sans doute l'obstacle majeur que rencontreront les utilisateurs du RPL/2, car si elle est très efficace, elle nécessite un réarrangement mental des séquences d'instructions d'une expression avant de pouvoir en calculer le résultat. Cependant, le RPL/2 étant capable d'effectuer des calculs formels, il reste possible d'utiliser des expressions en notation algébrique au détriment de la vitesse du langage. L'un des avantages essentiel résultant de l'utilisation de ces deux logiques est que l'utilisateur final n'a pas à se préoccuper de savoir si la notation polonaise inverse est pire ou meilleure que la notation algébrique. Il peut choisir le système de plus adapté à ses besoins et mélanger

CHAPITRE 1. NOTATIONS

expressions algébriques classiques et manipulations en notation polonaise inverse.

La pile opérationnelle utilisée par le RPL/2 n'est pas limitée en taille. Les entrées et sorties de la pile se font exclusivement par le niveau le plus bas. Pour des raisons d'efficacité et afin de réduire les coûts des opérations d'ajout et de suppression d'éléments, cette pile n'est pas implantée sous la forme d'un tableau en mémoire mais d'une double liste chaînée, l'une pour les éléments visibles, la seconde comme cache. Pour les mêmes raisons, les objets contenus dans la pile ne sont réellement copiés ou dupliqués que lors de leurs modifications.

1.2 Notation algébrique

Cette notation permet de saisir une expression sous sa forme algébrique ordinaire, puis de calculer le résultat lors de son évaluation. Cette approche a le mérite de conserver la correspondance entre les expressions telles qu'elles sont écrites sur le papier et leur saisie au clavier, mais en général l'inconvénient de ne pas fournir de résultats intermédiaires. Par ailleurs, il est nécessaire de connaître la forme complète d'une expression avant de commencer à la saisir, ce qui rend difficile dans ces conditions, de s'acheminer vers une solution en modifiant les calculs en fonction des résultats intermédiaires.

L'évaluation d'une expression algébrique est une opération complexe et coûteuse en temps de calcul. Pour réduire ce coût au maximum, une conversion de toute expression algébrique en notation polonaise inversée est effectuée une fois pour toute lors de l'empilement initial de l'objet. Réciproquement, lors de l'affichage de l'objet, celui-ci est automatiquement converti en notation algébrique.

1.3 Notation infix

Certaines fonctions du langage travaillent en notation infix pour des raisons de clarté. Il s'agit principalement des instructions portant sur des blocs de programmes comme les boucles et les structures de contrôle et de reprise sur erreur. Dans ce cas, l'instruction est mise entre ses arguments.

2

Types de données

Les données manipulées par le RPL/2 sont fortement typées. Cependant, l'affectation d'un type à un objet se fait de manière automatique et non par déclaration préalable. Il faut noter qu'un mécanisme d'héritage existe entre certains types, permettant une cohérence des résultats de calcul. Ainsi, si un calcul ne peut être achevé en entier, il s'effectuera en réel voire en complexe. Cette inférence de type n'a aucun impact sur la gestion des variables. En effet, une variable, contrairement à la majorité des autres langages, n'est pas associée à un type d'objet, elle n'est qu'un objet de type nom permettant d'accéder aisément par référence à un autre objet. Elle n'a donc nul besoin d'être déclarée explicitement.

Pour des raisons de performances, les objets ne sont copiés que lors de modifications ou lors de copie de contexte comme par exemple lors de la création d'un nouveau fils d'exécution. Dans tous les autres cas, seule une référence est copiée et un même objet peut être utilisé de nombreuses fois sans jamais être copié à partir du moment où il n'est accessible qu'en lecture seule.

ICI Les instructions `type` et `kind` permettent à tout instant de connaître le type d'un objet. Ces deux instructions prennent un objet au niveau 1 de la pile et renvoient un entier. La signification de l'entier renvoyé est donné aux tableaux 2.1 et 2.2.

2.1 Types exportables

Un type de donnée exportable est un type de donnée gardant une signification non ambiguë et indépendante de l'état courant du RPL/2. Les types exportables sont les seuls types qui peuvent être utilisés directement dans une définition, dans un fichier ou sur une socket réseau car ils ne font aucune hypothèse sur le contexte d'exécution courant.

2.1.1 Scalaires

Les définitions des types de données utilisées ne varient pas en fonction de l'architecture du calculateur sur lequel fonctionne le RPL/2. Ainsi, un programme fonctionnant une machine fonctionnera exactement de la même façon sur une autre machine.

Par ailleurs, toutes les opérations assurent des conversions implicites d'un type vers un autre lorsque le résultat ne peut être représenté sans erreur dans le type de résultat par défaut. En particulier, le RPL/2 sait que l'ensemble des entiers est inclus dans celui des réels, lui-même inclus dans l'ensemble des complexes.

CHAPITRE 2. TYPES DE DONNÉES

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Argument	Résultat
Entier	0
Réel	0
Complexe	1
Chaîne de caractères	2
Vecteur d'entiers	3
Vecteur de réels	3
Matrice d'entiers	3
Matrice de réels	3
Vecteur de complexes	4
Matrice de complexes	4
Liste	5
Adresse	6
Nom	7
Expression RPN	8
Expression algébrique	9
Entier binaire	10
Fichiers	11
Bibliothèque partagée	12
Socket	13
Processus	14
Fonction	15
Table	16
Connecteur SQL	17
Mutex	18
Sémaphore	19

TABLE 2.1 : Valeurs renvoyées par type

Argument	Résultat
Entier	0
Vecteur d'entiers	0
Matrice d'entiers	0
Réel	1
Vecteur de réels	1
Matrice de réels	1
Complexe	2
Vecteur de complexes	2
Matrice de complexes	2

TABLE 2.2 : Valeurs renvoyées par kind

2.1. TYPES EXPORTABLES

2.1.1.1 Entiers

Les entiers manipulés par le RPL/2 sont des entiers signés de longueur 64 bits. À l'exception des instructions INCR et DECR, toutes les opérations effectuées sur des entiers s'assurent qu'il n'y a aucun débordement. Si le résultat d'une opération ne peut être représenté sans erreur en entier, il sera automatiquement converti dans un type permettant de le représenter sans erreur.

Lorsqu'un nombre ne comporte que des chiffres, précédé ou non par un signe + ou -, il est considéré comme un entier. Si sa valeur est trop grande pour pouvoir le représenter, il est converti automatiquement en réel.

2.1.1.2 Booléens

Les booléens ne forment pas un type à part. Il s'agit d'entiers qui ne prennent que deux valeurs, vrai ou faux. La valeur fausse correspond à 0 et la valeur vraie, à tout entier non nul. Deux constantes symboliques sont introduits par le langage :

- `true` valant -1 car le RPL/2 traite les opérations logiques bit à bit;
- `false` valant 0.

2.1.1.3 Réels

Les réels utilisés sont des flottants codés sur 64 bits en représentation IEEE. Trois constantes symboliques — sur lesquelles il est possible d'effectuer des opérations arithmétiques — peuvent apparaître :

- `nan` ou *not a number* représentant survenant lors d'un calcul dont le résultat est indéterminé comme 0/0;
- `-inf` ou `+inf` représentant les deux infinis.

Un réel se note sous la forme $\pm (\text{MANT})E\pm (\text{EXP})$, la mantisse étant réelle, l'exposant entier, les signes optionnels et la casse indifférente. Le séparateur décimal varie en fonction du mode de fonctionnement et sera soit le point, soit la virgule. Pour indiquer au RPL/2 lorsqu'il y a ambiguïté qu'une valeur doit être stockée en réel plutôt qu'en entier, il suffit d'indiquer le séparateur décimal. Ainsi 184 est un entier alors que `-37 .` est un flottant.

Lorsqu'une opération n'admet pas de résultat dans le corps des réels, mais un résultat unique dans le corps des complexes, elle retourne un résultat complexe. Ainsi, `<< -1 SQRT >>` donne un résultat complexe, mais `<< -1 3 INV ^ >>` provoque une erreur car le résultat de l'opération n'est pas unique.

2.1.1.4 Complexes

Un complexe est pour le séquenceur un groupe de deux réels représentant respectivement sa partie réelle et sa partie imaginaire. Il est donc codé sur deux fois 64 bits. Il se note `(Re , Im)` lorsque le séparateur décimal est le point, et `(Re . Im)` lorsque celui-ci est la virgule.

2.1.2 Vecteurs

Un vecteur est un tableau de scalaires à une seule dimension. Selon les scalaires le composant, un vecteur peut être déclaré comme entier, réel ou complexe. Il faut noter que tous les scalaires composant le vecteur sont du même type. Si un vecteur apparaît dans une expression matricielle, celui-ci correspond généralement à une matrice colonne. Le délimiteur utilisé pour un vecteur est le crochet. Ainsi, un vecteur sera noté $[s_1 \ s_2 \ \dots \ s_n]$.

2.1.3 Matrices

Une matrice est un tableau de scalaires à deux dimensions, même si cette matrice ne comporte qu'une seule ligne. À l'instar des vecteurs, elle peut être entière, réelle ou complexe. Comme il s'agit d'un assemblage de vecteurs, elle est notée $[[s_{11} \ s_{12} \ \dots \ s_{1n}] \ \dots \ [s_{m1} \ s_{m2} \ \dots \ s_{mn}]]$.

Il est impossible de créer des tableaux à plus de deux dimensions sous la forme de matrices. Néanmoins, il est possible d'utiliser des tableaux de dimensions quelconques, voire de section non régulière, par le biais de listes ou de tables.

2.1.4 Listes

Les listes sont formées de collections d'objets hétérogènes ou non. Elles peuvent être incluses les unes dans les autres sans limitation aucune si ce n'est la mémoire disponible sur le calculateur. Le délimiteur utilisé est l'accolade, ce qui fait qu'une liste sera notée $\{ \text{objet}_1 \ \text{objet}_2 \ \dots \ \text{objet}_n \}$.

2.1.5 Tables

Les tables sont formées de collections d'objets hétérogènes ou non. Elles peuvent être incluses les unes dans les autres sans limitation aucune si ce n'est la mémoire disponible sur le calculateur. Contrairement aux listes qui peuvent contenir un nombre variable d'éléments au cours de son utilisation, une table en contient un nombre fixe rendant sa représentation interne possible sous forme de tableau. L'accès aux derniers éléments d'une table est ainsi plus rapide que dans le cas de la liste équivalente. Le délimiteur utilisé est le chevron suivi d'un crochet, ce qui fait qu'une table sera notée $<[\text{objet}_1 \ \text{objet}_2 \ \dots \ \text{objet}_n]>$.

2.1.6 Expressions

Toute suite d'instructions est considérée comme expression au sens large. Ainsi, un programme est une expression, généralement en notation polonaise inverse, composée d'un certain nombre d'expressions plus petites pouvant être chacune soit en notation polonaise inverse, soit en notation algébrique.

2.1.6.1 Expressions algébriques

La notation algébrique correspond plus ou moins à l'écriture naturelle des expressions. Néanmoins, l'opérateur de multiplication de deux termes ne peut être implicite. Le délimiteur utilisé pour ce type d'expression est l'apostrophe. Aussi l'équation

$$\frac{\sin(\pi x)}{\pi x}$$

sera-t-elle notée $'\text{SIN}(\text{PI} * x) / (\text{PI} * x)'$. Ces expressions, dont l'évaluation est coûteuse en terme de calcul, sont transformées en notation polonaise inverse par le RPL/2 lors de l'opération consistant à affecter un type à la donnée. Ainsi, il est préférable d'utiliser ces expressions avec parcimonie.

2.1.6.2 Expressions RPN

Les expressions en notation polonaise inverse sont des expressions directement compréhensibles par le séquenceur. Elles peuvent être incluses les unes dans les autres et sont délimitées

2.1. TYPES EXPORTABLES

par des guillemets français. L'expression précédente pourra dès lors être notée `<< PI x * dup sin swap / >>`. Il faut noter que cette écriture n'est pas unique. En effet, il aurait été possible de noter par exemple `<< PI x * sin PI x * / >>`, mais cette nouvelle notation est moins efficace que la précédente puisque comportant plus d'opérations arithmétiques et d'appel de variables.

2.1.7 Noms

Les noms sont des objets particuliers représentant une variable de façon symbolique, cette variable pouvant le cas échéant ne pas exister. Ils sont délimités grâce aux apostrophes. La variable *x* sera ainsi représentée par `'x'`. Lorsqu'un programme RPL/2 rencontre un nom muni de ses délimiteurs, ce nom est empilé. Lorsqu'il rencontre un nom sans délimiteurs, le RPL/2 tente d'évaluer le contenu de la variable si elle existe.

2.1.8 Chaînes de caractères

Une chaîne de caractères est un objet formé d'une succession de caractères délimitée par des guillemets anglais comme `"cette chaîne"`. Le nombre de caractères de la chaîne peut varier au cours du temps car cet objet est associé à un descripteur de chaîne. Une chaîne de caractères peut contenir tous les caractères ASCII entre 0 et 255 inclus, les caractères non imprimables étant représentés grâce à une séquence d'échappement.

Les caractères d'échappement autorisés dans les chaînes sont indiqués dans le tableau 2.3. Ces séquences d'échappement ne sont converties que lors de leur affichage sur la sortie standard ou d'opérations sur des flux. La seule exception concerne l'affichage de la pile où ses séquences apparaissent toujours non converties. Les séquences d'échappement inconnues sont ignorées et provoquent un message d'avertissement.

La séquence d'échappement spéciale `\x` demande à être suivie de deux chiffres hexadécimaux en majuscule permettant de coder tous les caractères entre 0 et 255.

Séquences	Signification
<code>\\</code>	<code>\</code>
<code>\"</code>	<code>"</code>
<code>\b</code>	backspace
<code>\n</code>	retour à la ligne
<code>\t</code>	tabulation
<code>\x??</code>	caractère ASCII

TABLE 2.3 : Séquences d'échappement

2.1.9 Entiers Binaires

Les entiers binaires sont des entiers non signés d'une longueur maximale de 64 bits. Cette longueur est définie par l'instruction `STWS`. Ils sont manipulés par des instructions logiques et arithmétiques et sont notés sous la forme `# (ENTIER)b` où *b* représente la base de l'entier binaire, à savoir

- *b* pour binaire;
- *o* pour octale;
- *d* pour décimale;

- h pour hexadécimale.

Contrairement aux autres types qui peuvent être convertis automatiquement en des types de portées plus grandes lors d'une opération, les entiers binaires ne sont jamais convertis implicitement. Un mécanisme de conversion explicite des entiers binaires vers les entiers naturels et réciproquement existe néanmoins grâce aux instructions `r->b` et `b->r`. Une conséquence de cette absence de conversion implicite est l'absence de tout mécanisme de contrôle de débordement sur les opérations manipulant les entiers binaires.

2.2 Types non exportables

Un type de donnée non exportable est associé à une donnée qui perd toute signification en dehors du contexte d'exécution courant. Étant dépendants de l'état interne du RPL/2, il est impossible d'utiliser ces types de données entre deux processus s'exécutant dans des contextes différents. Il n'existe donc aucun mécanisme permettant de les utiliser dans des fichiers, sur des sockets réseau ou tout simplement pour écrire des programmes.

2.2.1 Fichiers

Le type fichier est un type composite qui ne peut être créé que par l'instruction `open`. Les fichiers utilisés par le RPL/2 sont de trois types :

- fichiers formatés : le fichier est éditable par l'utilisateur. Chaque enregistrement apparaît sur une ligne d'un fichier texte sous la forme d'une liste d'éléments. Seul le format d'écriture est imposé, le format de lecture étant déduit automatiquement de la structure du fichier ;
- fichiers non formatés : le fichier n'est pas éditable par l'utilisateur. Chaque enregistrement est composé de données binaires structurées. Ces données ne peuvent être garanties indépendantes de l'architecture de la machine hôte. Les fichiers non formatés, à l'instar des fichiers formatés, ne nécessitent aucune indication de format en lecture ;
- fichiers de type flux : les fichiers de type flux sont à format libre. Ils ne contiennent aucune information de structure d'enregistrement pour leur lecture. Contrairement aux fichiers formatés et non formatés, un fichier de type flux suppose que le processus de lecture connaisse *a priori* la structure des enregistrements.

L'accès aux fichiers est séquentiel, direct ou indexé pour les fichiers formatés ou non formatés. Pour les fichiers de type flux, seuls les accès séquentiel et direct existent. Le mécanisme de gestion des fichiers à accès direct ou indexé n'impose pas une longueur fixe des enregistrements. Enfin, le RPL/2 possède un mécanisme fermant automatiquement les fichiers qui n'ont pas été fermés à la fin du fil d'exécution qui les ont ouverts.

Un fichier n'est accessible que depuis le fil d'exécution qui l'a ouvert. Si un programme tente d'accéder à un fichier depuis un autre fil d'exécution que celui qui l'a ouvert, la tentative d'accès échouera avec une erreur de contexte.

2.2.2 Sockets

Le type socket est un objet composite permettant la gestion des sockets locales, IPv4 et IPv6 en TCP ou UDP. Comme le type fichier, une socket ne peut être créée que par l'instruction `open`. Les sockets non fermées à la fin du fil d'exécution qui les a créées sont autoritairement closes par le RPL/2.

Une socket n'est accessible que depuis le fil d'exécution qui l'a ouvert. Si un programme tente d'accéder à une socket depuis un autre fil d'exécution que celui qui l'a ouvert, la tentative d'accès échouera avec une erreur de contexte.

2.2.3 Bibliothèques

Le RPL/C est un langage de macroinstructions permettant d'étendre les fonctions intrinsèques du RPL/2. Toute bibliothèque écrite en RPL/C devient part intégrante des fonctions intrinsèques du langage et se comporte comme elles. Le type bibliothèque contient la structure de description d'une bibliothèque chargée en mémoire et permet le cas échéant de la retirer du RPL/2. Un objet de type bibliothèque ne peut être créé que par l'instruction `useuse@use`. Les bibliothèques sont automatiquement fermées à la fin du fil d'exécution qui les ont ouvertes.

Les différentes instructions ajoutées par une bibliothèque externe sont exportées vers tout fil d'exécution créé par `detach` ou `spawn` créé après le chargement de la bibliothèque. En cas de retrait de la bibliothèque par `remove`, il convient de s'assurer qu'aucun des fils d'exécution fil n'appellera plus les instructions de cette bibliothèque.

2.2.4 Processus

Le RPL/2 gère nativement les processus et les processus légers. L'objet processus contient toutes les informations nécessaires à leur description :

- plan d'adressage;
- variables globales et locales;
- pointeurs sur les variables partagées;
- processus père et fils;
- canaux de communication vers les processus père et fils;
- interruptions du processus père...

Un objet de type processus ne peut être créé que par deux instructions :

- `detach`, qui crée un processus détaché s'exécutant dans un environnement distinct du processus père;
- `spawn`, qui lance un processus léger s'exécutant dans le même espace que son père, mais sur une copie de son environnement.

Du point de vue de l'utilisateur, un processus, léger ou non, est un fil d'exécution qui possède un contexte d'exécution propre qui prend sa place dans un arbre hiérarchique. Dans cet arbre, chaque processus ou processus léger possède deux canaux de communication, l'un pour émettre des données à destination de son père, l'autre pour envoyer des données à destination de ses fils. Un processus fils peut envoyer une interruption à son père.

Un processus s'achève normalement lorsque son fil d'exécution ne contient plus aucune définition à exécuter et que toutes les données envoyées vers son père ont été acquittées. En cas d'erreur, un processus fils renvoie un signal d'erreur à son père. Charge au père de le gérer par un mécanisme de reprise sur erreur de type `iferr`. Si le processus père ne gère pas cette erreur, l'erreur remonte l'arbre des processus jusqu'à trouver un processus capable de gérer

cette erreur, le processus à la base de l'arbre, ou une racine secondaire des processus installée par l'instruction `nrprocnrproc`.

Lorsqu'un processus s'achève, tous les processus fils reçoivent un signal d'arrêt qu'ils peuvent honorer immédiatement ou de façon différée s'ils sont dans un bloc de programme compris entre les instructions `cstop` et `rstop`. Le processus en cours d'achèvement attend la fin de tous ses fils avant de libérer ses propres ressources.

2.2.5 Connecteurs SQL

Un programme RPL/2 peut interroger directement des bases de données locales ou distantes de type SQL. L'interrogation se fait au travers d'un connecteur à la base et d'une requête SQL passée sous la forme d'une chaîne de caractères. Toutes les bases de données sont traitées de la même manière. Un mécanisme de transcodage des requêtes et des résultats permet de s'affranchir de la différence d'encodage existant entre l'encodage interne du RPL/2 et celui de la base de données cible.

Ce type de donnée ne peut être créé que par l'instruction `sqlconnect`. Tout comme les fichiers, les connecteurs SQL ne sont valables qu'au sein du fil d'exécution qui les a créés. Ils sont automatiquement clos à la fin du fil d'exécution.

2.2.6 Mutexes

Un programme partageant plusieurs processus légers peut requérir un mécanisme de synchronisation, en particulier lors d'accès à des variables partagées. Le type mutex contient une structure permettant de poser des verrous atomiques. Il ne peut être créé que par l'instruction `crmtx`. Tous les mutexes sont automatiquement relâchés et libérés à la fin du fil d'exécution qui les ont créés.

Il faut noter que les mutexes ne sont partagés qu'entre des processus légers. En aucun cas il ne sont partagés entre des processus normaux pour lesquels il convient d'utiliser des sémaphores.

Par défaut, les mutexes sont des mutexes récursifs. Un même fil d'exécution peut donc demander un mutex plusieurs fois, charge à lui de le libérer autant de fois qu'il ne l'a demandé. Un mutex ne peut être libéré que par le fil d'exécution qui l'a verrouillé. Si un autre fil d'exécution tente de libérer le mutex, cette tentative provoquera une erreur de contexte.

2.2.7 Sémaphores nommés

Un sémaphore nommé est un entier sur lequel il est possible d'effectuer des opérations atomiques permettant de poser des verrous. Le sémaphore nommé peut être partagé entre plusieurs processus même si ceux-ci ne sont pas sur le même arbre de processus. Ils sont créés par l'instruction `crsmphr` mais ne sont pas détruits à la fin du fil d'exécution. Le sémaphore nommé apparaît comme un fichier dans le système de fichiers de l'hôte et est soumis à des droits d'accès.

2.3 Types internes

Les types internes sont des types qui sont inaccessibles à l'utilisateur. Typiquement, il s'agit d'adresses sur 64 bits contenant des adresses de saut — tests, boucles — ou de retour de définition.

3

Niveaux et fils d'exécution

3.1 Niveau d'exécution

Un programme RPL/2 est constitué d'une collection de blocs logiques imbriqués les uns dans les autres à la façon de poupées russes. Lors de l'analyse d'un programme, ce niveau d'exécution est initialisé à zéro. À chaque entrée dans un bloc de programme, qu'il s'agisse d'une boucle, d'une expression ou d'un appel à une définition utilisateur, ce niveau est incrémenté. Il est automatiquement décrémenté à sa sortie. Lorsque le niveau d'exécution devient nul, l'exécution s'arrête.

Sont considérés comme blocs :

- les évaluations d'objets de type expressions;
- les appels explicites ou implicites à des définitions utilisateurs;
- les blocs créant des variables, que ces blocs soient définis comme des expressions algébriques ou RPN;
- les boucles définies ou indéfinies.

En aucun cas les appels à des définitions intrinsèques ou extrinsèques ne forment implicitement des blocs de programme. En tout état de cause, même si une définition intrinsèque ou extrinsèque utilise un bloc interne pour créer des variables locales — cas de `draw` par exemple —, le niveau d'exécution courant au retour de cette définition sera le même que le niveau courant lors de l'appel à cette définition.

Ce niveau d'exécution n'est pas associé à un bloc de programme mais au niveau d'exécution courant à l'entrée de ce bloc. Ainsi, dans l'exemple suivant :

0001	NIVEAU			
0002	<<			-\ 1
0003	1 -> A		-\ 2	
0004	<<			
0005	A 1 +			
0006	-> B		-\ 3	
0007	<<			
0008	1 10 for A		-\ 4	
0009	// Traitement			
0010	next		-/	
0011				
0012	NIVEAU2		-- 4	
0013	>>		-/	

CHAPITRE 3. NIVEAUX ET FILS D'EXÉCUTION

0014	>>	-/	
0015			
0016	NIVEAU2	-- 2	
0017	>>		-/
0018			
0019	NIVEAU2		
0020	<<		- \ n+1
0021	do	- \ n+2	
0022	// Traitement		
0023	until		
0024	// Condition		
0025	end	-/	
0026	>>		-/

le niveau de la définition utilisateur NIVEAU2 dépend de son point d'appel. Lors du premier appel, elle est de niveau 5 alors qu'elle ne sera plus que de niveau 3 lors de son second appel. En aucun cas l'utilisateur ne peut modifier autoritairement le niveau d'exécution courant.

Avant d'aller plus loin, il est nécessaire de s'assurer d'avoir bien compris ce concept de niveau car il conditionne le comportement des variables ainsi que l'évaluation des objets et les mécanismes de reprise sur erreur. Le niveau d'exécution est une originalité du RPL/2 qu'il ne faut pas confondre avec le nombre d'éléments dans une pile système comme celle du langage C. Il n'y a en effet aucune relation directe entre ce niveau d'exécution et le nombre d'objets dans la pile système.

3.2 Contexte

Un contexte correspond à l'ensemble des données nécessaires à l'évaluation d'un bloc. Il contient entre autre :

- un pointeur sur la définition en cours;
- les différentes piles (opérationnelle, sauvegardes de la pile opérationnelle, système...);
- les arbres des variables et des définitions;
- les définitions extrinsèques;
- l'état du système...

Le contexte permet une isolation parfaite entre deux fils d'exécution même si ceux-ci se partagent le même espace mémoire.

3.3 Fil d'exécution

Un fil d'exécution correspond à une séquence de définitions qui s'exécute dans un contexte particulier. Chaque fil d'exécution prend place dans un arbre. À l'exception du fil d'exécution racine, chaque fil d'exécution possède un unique parent et un nombre quelconque d'enfants. Lorsqu'un fil d'exécution s'achève, il arrête autoritairement tous ses enfants. En d'autres termes, les enfants doivent se protéger s'ils ne veulent pas mourir en même temps que leur parent.

Des canaux de communications sont établis entre un fil d'exécution et son parent immédiat. Ils permettent à ces deux fils d'exécution d'échanger des données, l'enfant pouvant même interrompre l'exécution du parent grâce à un mécanisme d'interruption logique.

3.3. FIL D'EXÉCUTION

L'évaluation d'un fil d'exécution s'achève naturellement lorsque son niveau d'exécution devient égal au niveau d'exécution courant lors de sa création. Il envoie un signal d'extinction à tous ses enfants puis libère son contexte d'exécution avant d'avertir son père de son achèvement. À tout moment, un parent peut envoyer un signal d'arrêt à l'un de ses enfants.

Il existe en RPL/2 deux types de fils d'exécution, les processus et les processus légers. Le choix du type de fil d'exécution est conditionné par sa création. Il est impossible de changer le type d'un fil d'exécution une fois que celui-ci est actif.

3.3.1 Processus

Un processus est un fil d'exécution qui est créé seul dans un espace mémoire qui lui est propre par l'instruction `detach`.

3.3.2 Processus léger

Un processus léger est un processus qui est créé dans l'espace mémoire du parent. Ce processus parent peut être un processus ou un processus léger. Deux processus légers peuvent accéder aux mêmes variables partagées. Il est créé par l'instruction `spawn`.

CHAPITRE 3. NIVEAUX ET FILS D'EXÉCUTION

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

4

Variables

Contrairement à bon nombre de langages, un objet peut être utilisé en RPL/2 en dehors de toute référence à une variable. À la limite, il est possible au détriment de la lisibilité du code d'effectuer des manipulations complexes sur des données sans jamais n'utiliser une seule variable.

En effet, en RPL/2, une variable n'est qu'un contenant référencé par un nom. Seul son contenu, l'objet qu'elle référence, possède un type. Ainsi, une même variable peut successivement contenir des objets de types différents. L'appel à une variable se fait par l'évaluation de son nom. Comme un objet de type nom est sensible à la casse, les noms des variables le sont aussi. Comme les variables ne sont que des contenants, elles n'ont pas besoin d'être explicitement déclarées.

4.1 Niveau de variable

À chaque variable est associé un niveau de variable. Ce niveau de variable est un entier positif ou nul initialisé automatiquement lors de la création de la variable et conditionnant sa portée et sa visibilité. Seules les variables de niveau strictement positif sont modifiables par un programme au cours de son exécution. Les variables de niveau 0 ne sont modifiables que lors de l'analyse structurelle d'un programme, soit avant le début de l'exécution. Comme elles sont, contrairement à toutes les autres variables, non modifiables, elles sont verrouillées et ne sont pas dupliquées lors de la création d'un processus fils. En aucun cas ce niveau ne peut être modifié une fois la variable créée.

Par défaut, si à un point donné d'un programme existent simultanément plusieurs variables de même nom, seule la variable de plus haut niveau sera visible. Une dérogation à cette règle est autorisée pour les variables globales.

4.2 Classes de variables

Si une variable ne contient aucune information quant au type des données qu'elle contient, elle est en revanche associée à une classe de variable par le contexte de sa création. La classe de la variable conditionne à la fois la visibilité de cette variable et sa durée de vie. Les différentes classes de variables sont indiquées dans les paragraphes suivants.

CHAPITRE 4. VARIABLES

4.2.1 Variable interne

Une variable interne est une variable créée lors de l'analyse d'un programme RPL/2. Elle contient une définition utilisateur soit sous la forme de son adresse — cas d'un programme interprété —, soit sous la forme d'une expression algébrique ou RPN — cas d'un programme compilé. Elle ne peut être modifiée ultérieurement et est de niveau nul. Une variable interne est partagée par tous les fils d'exécution.

4.2.2 Variable privée

Une variable privée est une variable propre à un fil d'exécution. Sauf à créer explicitement une variable partagée, toutes les variables sont des variables privées.

4.2.2.1 Variable globale

Une variable globale est une variable privée qui est accessible depuis tout point du fil d'exécution courant. Elle reste accessible, le cas échéant, depuis l'évaluation d'une définition enregistrée par `atexit`. Une variable globale est de niveau égal à 1 et est copiée vers les fils d'exécution créés à partir du fil d'exécution courant.

4.2.2.2 Variable automatique ou locale

Une variable locale ou automatique est une variable créée à l'entrée d'un bloc et à l'usage exclusif de celui-ci. Elle est accessible depuis tous les blocs de la même définition utilisateur inclus dans ce bloc. Elle est toujours créée par l'instruction `->` précédée de la valeur initiale de la variable et suivie de nom de cette variable ainsi que d'une expression algébrique ou RPN. En aucun cas cette variable n'est accessible depuis une autre définition utilisateur.

Il existe deux types de variables automatiques, les variables volatiles et les variables statiques. La différence entre les deux types de variables réside dans le fait qu'une variable volatile est perdue à la sortie du bloc alors qu'une variable statique est sauvegardée pour une utilisation ultérieure. Le niveau des variables automatiques est strictement supérieur à 1.

Les variables automatiques sont créées par l'instruction `->`. Si cette instruction est précédée par le mot-clef `static`, les variables créées ne seront plus volatiles — comportement par défaut — mais seront statiques. Dans le cas où la même variable statique a déjà été initialisée, la valeur d'initialisation est silencieusement ignorée. À tout moment, il est possible de rendre volatile une variable statique.

Plusieurs variables locales peuvent être créées ensemble comme le montre l'exemple suivant où `X` prend la valeur 1 et `Y`, la valeur 2.

```
0001 VARIABLE_LOCALE
0002 <<
0003   1 2
0004   -> X Y
0005   <<
0006           // X et Y sont des variables locales au bloc
0007   >>
0008 >>
```

4.2.2.3 Variable implicite

Certaines définitions créent implicitement une ou plusieurs variables locales. Il s'agit principalement des boucles avec compteur `for` et `forall`. Les variables locales ainsi créées se comportent comme des variables volatiles. Elles cessent d'exister à la fin du bloc et son modifiables comme une variable volatile classique.

4.3. MASQUE DE VARIABLES

Ainsi, l'exemple suivant où la variable implicite est modifiée dans la boucle est parfaitement valide. La variable I contient successivement les valeurs 1, 3, 7, 15, 31 et 63.

```
0001 VARIABLE_IMPLICITE
0002 <<
0003     1 100 for I
0004         I disp 2 'I' sto*
0005     next
0006 >>
```

Les définitions susceptibles de créer des variables implicites incrémentent toutes le niveau d'exécution. Ainsi, une variable implicite ne peut pas écraser une autre variable locale. Tout au plus, une variable locale sera masquée temporairement par une variable implicite.

4.2.3 Variable partagée

Une variable partagée est une variable statique qui n'est plus locale au contexte d'un fil d'exécution mais partagée entre plusieurs processus légers. Elle est créée par l'instruction `->` précédée du mot-clé `shared`. À tout moment, il est possible de transformer une variable partagée en variable privée.

À l'instar d'une variable statique, elle est créée au début d'un bloc à l'usage exclusif de celui-ci, mais elle peut être accédée en lecture et en écriture depuis plusieurs processus légers distincts. Dans la plupart des cas, il convient de protéger les accès à cette variable par un mutex pour éviter toute modification de cette variable par un fil d'exécution concurrent.

Il n'existe pas de mécanisme pour partager directement une variable entre deux processus. Il est toutefois possible d'émuler un tel comportement en utilisant une variable virtuelle protégée par un sémaphore.

4.2.4 Variable virtuelle

Une variable virtuelle n'est pas une variable à proprement parler. Contrairement à toutes les autres classes de variable, une variable virtuelle ne possède pas de niveau. Il s'agit de la simple transcription d'un objet dans un fichier.

Une variable virtuelle ne peut être utilisée directement dans une expression. Il convient tout d'abord de la rappeler grâce à l'instruction `recall` puis de la sauvegarder grâce à `store`.

4.3 Masque de variables

À l'exception des variables globales qui sont toujours créées avec un niveau égal à 1, les variables locales ou implicites de mêmes noms ne peuvent être créées avec des niveaux identiques. Ainsi, plusieurs variables de mêmes noms ne peuvent s'écraser mutuellement car il s'agit de variables différentes. À un point donné d'un programme, seule la variable de plus haut niveau — ou la variable globale au travers d'une instruction spécifique — est accessible et modifiable.

```
0001 MASQUE
0002 <<
0003     1
0004     -> X
0005     <<
0006         -> X 'X+1' disp    // X masque la variable de même nom
0007
0008     1 3 for X              // X masque la variable de même nom
0009         // Traitement
```

CHAPITRE 4. VARIABLES

```
0010      next
0011  >>
0012 >>
```

4.4 Portée de variables

La portée d'une est la partie d'un programme dans laquelle cette variable est définie. Cette portée est une zone continue. Dans l'exemple suivant, les portées des différentes variables sont indiquées à droite du code.

```
0001 NIVEAU
0002 <<
0003     1 -> A
0004     <<
0005         A 1 +
0006         -> B
0007         <<
0008             1 10 for A
0009             // Traitement
0010             next
0011         >>
0012     >>
0013 >>
```

-\ 1

-\ 2

|

|

-\ 3

|

|

-\ 4

|

|

-/

|

-/

-/

|

-/

4.5 Visibilité de variables

Il faut distinguer la portée d'une variable de sa visibilité. La visibilité d'une variable est la partie d'un programme dans laquelle cette variable est accessible. La visibilité est donc incluse dans la portée d'une variable. Contrairement à la portée, la visibilité d'une variable peut être une zone discontinue.

```
0001 NIVEAU
0002 <<
0003     1 -> A
0004     <<
0005         A 1 +
0006         -> B
0007         <<
0008             1 10 for A
0009             // Traitement
0010             next
0011         >>
0012     >>
0013 >>
```

-\ 1

-\ 2

|

|

-\ 3

|

|

-\ 4

|

|

-/

|

-/

-/

|

-/

Une variable interne possède une portée égale à sa visibilité valant l'intégralité du programme qui la définit. En aucun cas, elle ne peut être masquée par une autre variable. Une variable globale peut, quant à elle, être masquée par une variable locale ou partagée mais sa visibilité est aussi égale à l'intégralité du programme définissant cette variable. Il existe en effet des mécanismes spécifiques permettant d'accéder à une variable globale même si elle était masquée par une variable de même nom et de niveau supérieur.

4.6 Protection des variables

Une variable peut être verrouillée pour en interdire toute modification accidentelle. Toute tentative d'écriture dans une variable verrouillée provoque une erreur. Le verrouillage d'une variable ne change ni la portée ni la visibilité d'une variable.

4.7 Héritage

Chaque fil d'exécution possède un contexte qui lui est propre. Lors de son initialisation, ce fil d'exécution hérite des variables courantes du fil d'exécution qui le crée.

Comme le nouveau fil d'exécution débute par l'évaluation d'une expression, les variables locales ou implicites du fil père sont masquées dans le fils. Ainsi, seules les variables internes — de niveau 0 — et globales — de niveau 1 — sont copiées du père vers le fils.

Le sort des variables partagées dépend du type de fil d'exécution créé. Dans le cas d'un processus léger, les variables partagées sont partagées entre le fils et le père ou entre le fils et un processus léger frère. S'il s'agit d'un processus, les variables partagées sont copiées et une modification d'une variable partagée dans le processus fils ne modifie plus la variable partagée dans le père car ce dernier s'exécute dans un espace mémoire distinct.

4.8 Noms de variable

Les noms de variables sont quelconques, sensibles à la casse, et uniquement limités par ce qui peut être représenté dans un objet de type nom. Il n'est pas possible de surcharger une instruction intrinsèque par une variable.

CHAPITRE 4. VARIABLES

5

Définitions

Un programme RPL/2 est constitué d'un ensemble de définitions imbriquées les unes dans les autres. Chaque définition, quel que soit son type ou la manière dont elle est définie, prend des arguments depuis la pile opérationnelle ou depuis des variables. Elle renvoie ses résultats dans la pile opérationnelle ou plus rarement dans des variables.

Il existe trois types de définitions, selon que ces définitions sont des mots-clefs du langage, des extensions au langage ou des programmes.

5.1 Définitions intrinsèques

Une définition ou instruction intrinsèque est un mot clef du langage insensible à la casse. Elle est irréductible et ne peut être masquée par un nom de variable. Sauf exception notoire des instructions utilisant des appels système lents, les définitions intrinsèques sont ininterrompibles.

Les définitions intrinsèques s'opposent aux définitions extrinsèques et sont directement exécutables. Dans la suite de ce document, le terme « instruction » sera souvent utilisé à la place de « définition intrinsèque ».

5.2 Définitions extrinsèques

Une définition extrinsèque est une extension du langage RPL/2. Elle permet d'ajouter au langage de nouvelles instructions trop spécifiques pour justifier un ajout dans le cœur du langage. Pour utiliser une définition extrinsèque, un programme RPL/2 doit charger une bibliothèque d'extension écrite en RPL/C au travers de l'instruction `use`.

Une définition extrinsèque permet d'encapsuler en une seule instruction un traitement complexe qui ne pourrait être remplacé que par une longue séquence d'instructions intrinsèques. Comme le RPL/C est un langage compilé, une seule instruction extrinsèque remplace avantageusement cette séquence d'instructions intrinsèques. Contrairement aux définitions intrinsèques, les définitions extrinsèques sont sensibles à la casse.

5.3 Définitions utilisateur

Une définition utilisateur est une suite d'instructions en notation polonaise inverse, chaque instruction étant au choix une donnée à empiler ou une définition à exécuter, que cette dé-

finition soit intrinsèque, extrinsèque ou elle-même utilisateur. Une définition utilisateur ne peut masquer ni une définition intrinsèque ni une définition extrinsèque.

Les variables utilisées dans une définition utilisateur sont par défaut automatiques et locales. En d'autres termes, les variables utilisées dans cette définition sont créées à l'intérieur de celle-ci et sont détruites avant d'en sortir. De plus, les variables locales étant attachées à un niveau de variable dépendant entre autres du nombre d'appels imbriqué à des définitions utilisateurs, une telle définition peut être appelée récursivement sans aucune déclaration spécifique.

Une définition utilisateur est associée à une variable interne — de niveau 0 donc sensible à la casse, non modifiable et de visibilité maximale — qui contient le nom de cette définition. Ce nom précède le bloc de programme qui la définit.

```

0001 CALCUL_DE_FACTORIELLE
0002 <<
0003   "Calcul de n!" disp "n = " prompt str->
0004   if
0005     dup dup ip same
0006   then
0007     FACTORIELLE disp
0008   else
0009     drop
0010     "ERREUR : argument n non entier !" disp
0011   end
0012 >>
0013
0014 /*
0015 =====
0016   Calcul récursif de factorielle
0017 =====
0018 */
0019
0020 FACTORIELLE
0021 <<
0022   -> N
0023   <<
0024     if
0025       N 1 >
0026     then
0027       N dup 1 - FACTORIELLE *
0028     else
0029       1
0030     end
0031   >>
0032 >>

```

L'exemple précédent contient deux définitions utilisateur qui sont d'une part `CALCUL_DE_FACTORIELLE` à la ligne 1, et d'autre part `FACTORIELLE` à la ligne 20. Si la première ne prend aucun argument dans la pile et n'en renvoie aucun, la seconde prend un argument entier (`N`) et renvoie un argument entier. Cette dernière est de plus une définition utilisateur récursive puisqu'elle peut s'appeler elle-même.

Deuxième partie

Appel

6

Différentes utilisations

À l'invite de commande du shell, l'interprète RPL/2 est appelé par la commande `rpl` suivie d'un ou plusieurs paramètres.

6.1 Fonctionnement interactif

Le RPL/2 offre un mode de fonctionnement interactif. Au lieu d'exécuter un programme, le logiciel affiche une invite de commande permettant d'exécuter des calculs de manière interactive. Ce mode de fonctionnement est obtenu en utilisant l'option `-i` de la ligne de commande.

```
schroedinger:[~] > rpl -i
+++RPL/2 (R) version 4.1.14 (Wednesday 04/17/13, 15:46:06 CEST)
+++Copyright (C) 1989 à 2012, 2013 BERTRAND Joël

+++Ce logiciel est un logiciel libre sans aucune garantie de fonctionnement.
+++Pour plus de détails, utilisez la commande 'warranty'.

RPL/2> 4 X +

1: '4+X'
RPL/2>
RPL/2> [[ 1 2 ]
+ 001> [ 4 3 ]
+ 002> [ 5 6 ]]

2: '4+X'
1: [[ 1 2 ]
   [ 4 3 ]
   [ 5 6 ]]
RPL/2> abort
schroedinger:[~] >
```

L'invite de commande par défaut du RPL/2 est `RPL/2>`. Pour revenir au système d'exploitation, il suffit d'utiliser la combinaison de touches `CTRL+D` ou les commandes `abort`, `kill` ou `exit`. Les flèches haut et bas rappellent les anciennes commandes enregistrées.

Un appui sur la touche **entrée** force l'évaluation de la ligne de commande en cours. Si cette ligne de commande est vide et que la pile opérationnelle contient au moins un objet, un appui sur **entrée** est équivalent à l'exécution de la commande `dup`. Dans le cas où l'évaluation de la ligne de commande ne peut se faire car elle n'est pas syntaxiquement correcte,

l'invite se transforme en + 001>, puis + 002> et ainsi de suite jusqu'à ce que l'entrée forme un objet dont la syntaxe est correcte. Pour sortir de ce mode en perdant la commande en cours, il suffit d'utiliser la combinaison de touches CTRL+G.

À tout moment, en fonctionnement interactif, il est possible de passer dans un mode spécial permettant d'obtenir une aide sur toutes les définitions intrinsèques et extrinsèques. Pour cela, il faut utiliser la commande `help`. Dans ce mode de fonctionnement, l'aide d'une définition est obtenue en tapant à l'invite `HELP>` le nom de cette définition. Le RPL/2 repasse en mode de fonctionnement normal à la première ligne vide entrée après l'invite `HELP>`.

6.2 Exécution de programmes

L'interprète tente d'exécuter tout fichier dont le nom est passé sur sa ligne de commande. La présence d'un nom de fichier sur la ligne de commande est incompatible avec l'option `-i` et provoque une erreur. Traditionnellement, les programmes RPL/2 ont une extension en `.rpl`. Ce n'est en aucun cas une nécessité, aucun contrôle n'étant effectué sur le nom du fichier passé en argument.

6.2.1 Programmes interprétés

Le fonctionnement par défaut consiste à interpréter le programme contenu dans le fichier passé en argument. Bien que lent, ce mode permet d'accéder au *debugger* interne.

```
riemann:[~] > cat hello.rpl
HELLO
<<
    "Hello, World!" disp
>>
riemann:[~] > rpl hello.rpl
+++RPL/2 (R) version 4.1.14 (Friday 04/19/13, 19:48:10 CEST)
+++Copyright (C) 1989 à 2012, 2013 BERTRAND Joël
Hello, World!
riemann:[~] >
```

6.2.2 Programmes compilés

Si l'interprète est lancé avec l'option `-p`, le programme contenu dans le fichier passé en argument est compilé en mémoire avant d'être exécuté. Le déroulement des programmes est bien plus rapide mais les instructions permettant d'accéder au *debugger* interne sont inactivées.

```
riemann:[~] > rpl -p hello.rpl
+++RPL/2 (R) version 4.1.14 (Friday 04/19/13, 19:48:10 CEST)
+++Copyright (C) 1989 à 2012, 2013 BERTRAND Joël
Hello, World!
riemann:[~] >
```

6.2.3 Utilisation du *sha bang*

Les *shells* des systèmes Unix permettent d'indiquer à la première ligne d'un fichier le programme qui sera utilisé pour interpréter le contenu du fichier. Le RPL/2 peut utiliser cette fonctionnalité pour produire des programmes autonomes.

```

riemann:[~] > cat hello.rpl
#!/usr/local/bin/rpl -p

HELLO
<<
    "Hello, World!" disp
>>
riemann:[~] > ls -l hello.rpl
-rwxr-xr-x 1 bertrand bertrand 61 avril 19 20:33 hello.rpl
riemann:[~/tmp] > ./hello.rpl
+++RPL/2 (R) version 4.1.14 (Friday 04/19/13, 19:48:10 CEST)
+++Copyright (C) 1989 à 2012, 2013 BERTRAND Joël
Hello, World!
riemann:[~] >

```

6.3 Passage d'arguments

6.3.1 Arguments

Le RPL/2 ne possède aucune fonction lui permettant d'accéder directement aux options de la ligne de commande. Il serait en effet difficile de distinguer les options du programme de celles de l'interprète. Néanmoins, l'option `-A` permet sous certaines conditions de passer des objets en arguments, objets qui seront utilisés pour initialiser la pile opérationnelle avant le lancement du programme.

```

riemann:[~] > cat arguments.rpl
ARGUMENTS
<<
    clmf
>>
riemann:[~] > rpl -p arguments.rpl -A '"Hello" "World!'"
+++RPL/2 (R) version 4.1.14 (Friday 04/19/13, 19:48:10 CEST)
+++Copyright (C) 1989 à 2012, 2013 BERTRAND Joël
2: "Hello"
1: "World!"
riemann:[~] >

```

Les arguments permettant d'initialiser la pile opérationnelle sont passés à l'interprète sous la forme d'une unique chaîne de caractère du point de vue du *shell*, ce qui explique les apostrophes encadrant les deux arguments suivant l'option `-A`.

6.3.2 Programmes

De la même manière que l'option `-A` permet de passer des arguments à un programme au cours de son initialisation, il est possible grâce à l'option `-S` de passer le programme à exécuter sur la ligne de commande.

```

riemann:[~] > rpl -pS 'HELLO << "Hello, World!" disp >>'
+++RPL/2 (R) version 4.1.14 (Friday 04/19/13, 19:48:10 CEST)
+++Copyright (C) 1989 à 2012, 2013 BERTRAND Joël
Hello, World!
riemann:[~] >

```

6.4 Option de débogage

6.4.1 Débogage des programmes

Par défaut, si un programme rencontre une erreur, il s'achève en affichant un message correspondant à l'erreur sur la sortie standard. L'option `-c` force l'interprète à écrire en cas d'erreur un fichier texte appelé `rpl-core` contenant toutes les informations nécessaires au débogage :

- les piles opérationnelles et, le cas échéant, *last*;
- la pile des processus;
- la pile système;
- un pointeur sur l'instruction fautive;
- les variables.

En cas d'erreur non récupérable, chaque fil d'exécution produit son propre fichier `rpl-core`.

6.4.2 Débogage de l'interprète

L'interprète utilise une pile alternative pour détecter des dépassements de pile ainsi que des segments de mémoire partagée qui ne sont pas libérés en cas d'arrêt brutal. Ainsi, par défaut, l'interprète traite les signaux de type *segmentation fault* et *bus error* de telle sorte que cette mémoire partagée puisse être libérée.

L'option `-d` évite d'installer ce mécanisme de protection. Elle ne devrait jamais être utilisée en fonctionnement normal. Son seul intérêt est de pouvoir déboguer l'interprète qui peut alors générer un fichier *core* analysable *post mortem*. L'utilisation de cette option est déconseillée sauf à fin de tests et mise au point.

6.4.3 Contrôles

-t

6.5 Interaction avec le système d'exploitation

-n -D

6.6 Autres options

-a -h -l -P -s -v
 === ICI ===

La ligne de commande de l'interprète RPL/2 peut contenir des options conditionnant son fonctionnement. Les options disponibles sont les suivantes :

- `-a` renvoie les coordonnées de contact de l'auteur, l'adresse de la liste de diffusion du langage ainsi que l'adresse du site web officiel;
- `-A` permet d'envoyer des arguments au programme principal :

```

rayleigh: [~] > rpl -A '4 5' -sS "DEF << clmf >>"
+++RPL/2 (R) version 4.0.10 (lundi 01/02/2010, 11:21:30 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
2: 4
1: 5
rayleigh: [~] >

```

- -c autorise la création d'un fichier rpl-core nécessaire à une analyse *post mortem*. Ce fichier contient toutes les informations nécessaires au déverminage d'un programme (état de la pile opérationnelle, variables, processus, instruction fautive, pile last...). Ce fichier est créé dans le répertoire courant et porte le nom `rpl-core-xxx-yyy` ou `xxx` est l'identifiant du processus et `yyy` l'identifiant du processus léger;
- -d permet de déverminer les allocations mémoire internes au RPL/2. En effet, le RPL/2 utilise une pile alternative pour récupérer les erreurs de type violation d'accès et les dépassements de pile. Utiliser cette option revient à interdire l'utilisation de cette pile alternative et la gestion de certains signaux de récupération d'erreur. Le RPL/2 peut alors générer un *core* analysable *post mortem*. L'utilisation de cette option est déconseillée sauf à fin de tests et mise au point; Cette option n'est à utiliser que pour déverminer le séquenceur;
- -D lance le séquenceur sous la forme d'un daemon¹. Le processus est détaché du terminal courant et rattaché au processus init dans le cas des systèmes Unix. Ses entrées et sorties standard ne peuvent plus être utilisées. Une définition intrinsèque permet de basculer sous certaines conditions un processus standard en daemon. Il est par contre impossible de basculer un processus fonctionnant en daemon en processus standard;
- -h retourne une aide sommaire sur la ligne de commande;
- -i lance le séquenceur en mode interactif. Le RPL/2 ne traite aucun programme mais offre une invite de commande permettant de l'utiliser directement. Cette option est incompatible avec la présence d'un fichier exécutable sur la ligne de commande;
- -l rappelle la licence d'utilisation du RPL/2;
- -n rend le RPL/2 insensible au signal HUP donc à la destruction de terminal de contrôle. Cette option est principalement destinée à la surveillance de programmes distants lancés dans un terminal. Lorsque le signal HUP est récupéré par le RPL/2 à la suite par exemple d'une rupture de session ssh, il ne tue pas le processus mais le transforme en daemon. Les sorties standard sont redirigées dans un fichier créé dans le répertoire courant. Le nom de ce fichier est de la forme `rpl-out-xxx-yyy` ou `xxx` est l'identifiant du processus et `yyy` l'identifiant du processus léger;
- -p précompile les programmes avant de les exécuter. Un programme précompilé tourne plus vite qu'un programme interprété, mais il ne peut appeler les fonctions du debugger interne;
- -P génère à la fin de chaque fil d'exécution un fichier profilage contenant toutes les informations nécessaires à l'optimisation d'un programme (nombre d'appels de variables, temps passé dans les différentes fonctions, temps processeur consommé...). Le fichier de profilage est créé dans le répertoire courant et porte le nom `rpl-profile-xxx-yyy` ou `xxx` est l'identifiant du processus et `yyy` l'identifiant du processus léger;

¹Disk And Extension MONitor

CHAPITRE 6. DIFFÉRENTES UTILISATIONS

- -s empêche l'affichage de l'écran graphique initial;
- -S exécute le script passé en ligne de commande;
- -t trace le fonctionnement interne du RPL/2. Cette option attend un argument sous la forme d'un nombre hexadécimal défini comme un ou logique entre les drapeaux suivants :
 - 0000 : rien;
 - 0001 : opérations sur la pile opérationnelle;
 - 0002 : opérations sur la pile système;
 - 0004 : appels de fonctions;
 - 0008 : gestion des processus;
 - 0010 : surveillance des routines d'analyse;
 - 0020 : surveillance des processus fusibles;
 - 0040 : gestion des variables;
 - 0080 : appels des fonctions intrinsèques;
 - 0100 : surveillance des niveaux d'exécution;
 - 0200 : conversions de la notation algébrique en notation polonaise inversée;
 - 0400 : supervision des interruptions;
 - 0800 : supervision des signaux.

Cette fonction de trace est activable ou modifiable à partir d'un programme en appelant la fonction `itrace`;

- -v renvoie la version du RPL/2.

7

Ligne de commande

Selon ses besoins, l'invocation du RPL/2 peut se faire de plusieurs manières. Tout d'abord, il est possible de l'utiliser de manière interactive en le lançant directement depuis une invite du système d'exploitation — typiquement un shell Unix — et en entrant les commandes directement depuis l'invite.

```
schroedinger:[~] > rpl -is
+++RPL/2 (R) version 4.1.14 (Wednesday 04/17/13, 15:46:06 CEST)
+++Copyright (C) 1989 à 2012, 2013 BERTRAND Joël

+++Ce logiciel est un logiciel libre sans aucune garantie de fonctionnement.
+++Pour plus de détails, utilisez la commande 'warranty'.

RPL/2> 4 X +

1: '4+X'
RPL/2>
RPL/2> [[ 1 2 ]
+ 001> [ 4 3 ]
+ 002> [ 5 6 ]]

2: '4+X'
1: [[ 1 2 ]
    [ 4 3 ]
    [ 5 6 ]]
RPL/2> abort
schroedinger:[~] >
```

L'invite de commande par défaut du RPL/2 est RPL/2>. Pour revenir au système d'exploitation, il suffit d'utiliser la combinaison de touches CTRL+D ou les commandes `abort`, `kill` ou `exit`. Les flèches haut et bas rappellent les anciennes commandes enregistrées.

Un appui sur la touche **entrée** force l'évaluation de la ligne de commande en cours. Si cette ligne de commande est vide et que la pile opérationnelle contient au moins un objet, un appui sur **entrée** est équivalent à l'exécution de la commande `dup`. Dans le cas où l'évaluation de la ligne de commande ne peut se faire car elle n'est pas syntaxiquement correcte, l'invite se transforme en `+ 001>`, puis `+ 002>` et ainsi de suite jusqu'à ce que l'entrée forme un objet dont la syntaxe est correcte. Pour sortir de ce mode en perdant la commande en cours, il suffit d'utiliser la combinaison de touches CTRL+G.

À tout moment, en fonctionnement interactif, il est possible de passer dans un mode spécial permettant d'obtenir une aide sur toutes les définitions intrinsèques et extrinsèques.

CHAPITRE 7. LIGNE DE COMMANDE

Pour cela, il faut utiliser la commande `help`. Dans ce mode de fonctionnement, l'aide d'une définition est obtenue en tapant à l'invite `HELP>` le nom de cette définition. Le RPL/2 repasse en mode de fonctionnement normal à la première ligne vide entrée après l'invite `HELP>`.

=== ICI ===

Une autre manière d'appeler le RPL/2 est le passage d'arguments ou de script sur l'entrée standard et fournit le résultat sur la sortie standard. Les erreurs passent sur la sortie d'erreur standard. Pour toute information, se reporter au manuel de votre shell favori.

```
rayleigh:[~] > echo 1 15.2 + X 3 \* sin - disp | rpl -i 2> /dev/null
RPL/2> 1 15.2 + X 3 * sin - disp
'16.2-SIN(X*3)'
RPL/2> abort
rayleigh:[~] > rpl -S "DEF << 1 15.2 + X 3 * sin - disp >>" 2> /dev/null
'16.2-SIN(X*3)'
rayleigh:[~] >
```

Ces deux appels ne permettent pas l'exécution de programmes de façon simple. Pour lancer un programme RPL/2, il convient de charger l'interprète et de lui fournir en argument le nom du fichier contenant le programme principal.

```
rayleigh:[~] > rpl mon_programme.rpl
```

Sous Unix, l'interprète peut être appelé directement depuis le fichier source au travers du *sha-bang* et former un fichier exécutable autonome. Pour de plus amples informations, se reporter au manuel du shell.

```
rayleigh:[~] > head -n5 mon_programme.rpl
#!/usr/local/bin/rpl -csp

MAIN
<<
// Main program
rayleigh:[~] >
```

7.1 Fonctionnement interactif

Lors du lancement du séquenceur en mode interactif, celui affiche un bandeau et propose une invite.

```
rayleigh:[~] > rpl -is
+++RPL/2 (R) version 4.0.10 (lundi 01/02/2010, 11:21:30 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël

+++Ce logiciel est un logiciel libre sans aucune garantie de fonctionnement.
+++Pour plus de détails, utilisez la commande 'warranty'.

RPL/2>
```

Toute séquence d'instructions entrée à cette invite est immédiatement évaluée lors de l'appui sur la touche « entrée ». Ses arguments sont pris dans la pile et les résultats renvoyés immédiatement dans cette pile. Par défaut, la pile last est active.

7.1. FONCTIONNEMENT INTERACTIF

```
rayleigh:[~] > rpl -is
+++RPL/2 (R) version 4.0.10 (lundi 01/02/2010, 11:21:30 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël

+++Ce logiciel est un logiciel libre sans aucune garantie de fonctionnement.
+++Pour plus de détails, utilisez la commande 'warranty'.

RPL/2> 1 X + sin

1: 'SIN(1+X)'
RPL/2> disp
'SIN(1+X)'
RPL/2>
```

Les instructions `kill`, `exit` et `abort` permettent de quitter le séquenceur. S'il existe des processus fils, il essaiera de tuer ces processeurs fils. Si pour une raison ou pour une autre un processus fils n'honore pas le signal stop — par exemple en bouclant indéfiniment dans une structure protégée par un bloc `CSTOP/RSTOP` —, il sera nécessaire de le tuer à la main à partir de l'invite du système d'exploitation.

CHAPITRE 7. LIGNE DE COMMANDE

8

Exécution de programmes

Dans le cas où les options `-i` et `-S` ne sont pas spécifiées, le séquenceur s'attend à trouver sur la ligne de commande le nom d'un fichier contenant le programme principal de l'application à exécuter. Dans ce cas, la pile `last` est par défaut inactive. Il est possible de modifier ce comportement en touchant à l'indicateur `31`.

L'exécution d'un programme commence par un appel à un préprocesseur qui forme le fichier réellement chargé en mémoire. Ces données subissent alors une analyse syntaxique puis logique avant d'être soit interprétées, soit compilées. Ces opérations sont détaillées dans les points suivants.

8.1 Préprocesseur

Avant d'exécuter le contenu du fichier, le RPL/2 appelle un préprocesseur spécialisé (`rplpp`) chargé de traiter un certain nombre de macroinstructions ainsi que d'éliminer les commentaires. Ces commentaires sont des zones du code source délimités par les symboles `« /* »` d'une part, et `« */ »` d'autre part. Si le préprocesseur rencontre les symboles `« // »`, le reste de la ligne est silencieusement ignoré. Les macroinstructions traitées par le préprocesseurs sont détaillées dans les points suivants.

8.1.1 Commentaires

Les programmes RPL/2 peuvent contenir deux sortes de commentaires. Tout ce qui est compris entre les balises `/*` et `*/` est ignoré. De la même manière, tout ce qui suit une balise `//` est ignoré jusqu'à la fin de la ligne.

8.1.2 `#define x y`

À partir de cette primitive, la macro `x` est définie comme étant `y`. `y` peut être n'importe quelle entrée valide ou contenant une autre macro. `x` doit être un identifiant, c'est-à-dire une suite de caractères alphanumériques et de soulignements. Si `x` était déjà défini, l'ancienne définition est écrasée silencieusement. Il faut noter que ni `x` ni `y` ne sont évalués lors de la définition, la macro n'étant évaluée que lors de son utilisation.

Un cas particulier d'utilisation consiste à omettre `y`. Dans ce cas, `x` est défini comme une macro qui ne contient rien. Un autre cas consiste à utiliser des macros à arguments comme

CHAPITRE 8. EXÉCUTION DE PROGRAMMES

celles de cpp. L'exemple suivant est un peu plus parlant.

```
0001 #!/usr/local/bin/rpl -sp
0002
0003 #define macro(X) X disp
0004
0005 MAIN
0006 <<
0007     macro(5)
0008 >>
```

8.1.3 #defeval x y

Cette primitive est similaire à `#define`, mais à la différence de cette dernière, le second argument de `#defeval` est immédiatement évalué.

```
0001 #!/usr/local/bin/rpl -sp
0002
0003 #mode standard default
0004 #define RETRAIT_PARENTHESSES(p) p
0005 #define APPLY(f,x) RETRAIT_PARENTHESSES(#defeval FONCTION f
0006     FONCTION(x))
0007 #define sinus(x) x sin
0008
0009 MAIN
0010 <<
0011     rad APPLY(\sinus,'2*pi') disp
0012 >>
```

Le résultat de l'exécution de ce code est :

```
rayleigh:[~] > macro.rpl
+++RPL/2 (R) version 4.0.10 (lundi 01/02/2010, 11:21:30 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
'SIN(2*PI)'
rayleigh:[~] >
```

Cet exemple utilise la primitive `#mode standard default` qui sera documentée plus bas. Par défaut, le préprocesseur fonctionne en mode `cpp` qui interdit les retours à la ligne dans la définition d'une macro et des primitives débutant ailleurs qu'en colonne 1.

8.1.4 #undef x

Cette primitive efface la définition de la macro `x` pour la suite du programme.

8.1.5 #ifdef x

La primitive `#ifdef` commence un bloc conditionnel. Si `x` est défini, tout ce qui suit `#ifdef` est évalué jusqu'à `#else` ou `#endif`. Dans le cas où la fin du bloc est signalé par une primitive `#else`, les lignes comprises entre `#else` et la primitive `#endif` correspondante sont ignorées. Cependant, le texte ignoré est analysé par le préprocesseur et doit être valide. En particulier, l'expansion des macros est effectuée.

8.1.6 #ifndef x

À l'instar de `#ifdef`, la primitive `#ifndef` débute un bloc conditionnel. Tout ce qui suit est évalué si la macro `x` n'est pas définie.

8.1.7 #ifeq x y

La primitive `#ifeq` commence un bloc conditionnel. Tout ce qui suit est évalué si les résultats des évaluations des macros `x` et `y` sont identiques. La comparaison se fait en terme de chaînes de caractères débarassées des espaces initiaux et finaux. Pour des raisons de compatibilité, lorsque le préprocesseur fonctionne en mode `cpp`, les espaces suivant le premier argument ne sont pas ignorés.

8.1.8 #ifneq x y

Cette primitive commence un bloc conditionnel. Tout ce qui suit est évalué dès que les résultats des évaluations des macros `x` et `y` sont différents.

8.1.9 #else

La primitive `#else` inverse la condition logique d'un bloc conditionnel. Les lignes qui suivent sont évaluées si et seulement si les lignes qui précèdent ne l'ont pas été.

8.1.10 #endif

Cette primitive termine un bloc conditionnel débutant par `#if...`

8.1.11 #include "file"

Cette primitive demande au préprocesseur l'ouverture du fichier `file`, l'évaluation de son contenu et l'insertion du résultat dans le flot courant. Toutes les macros définies sont aussi définies dans le fichier inclus et, réciproquement, toutes les macros définies dans le fichier inclus sont définies dans tout ce qui suit. Par défaut, si aucune indication de répertoire n'est fournie, le préprocesseur commence par chercher le fichier à inclure dans le répertoire courant et, s'il ne le trouve pas, il cherchera ce fichier dans un répertoire spécifié par l'option `-I` de la ligne de commande ou `/usr/include` si aucun autre répertoire n'est indiqué. Pour des raisons de compatibilités, il est possible d'utiliser indifféremment `"file"` ou `<file>`.

8.1.12 #exec command

Cette primitive permet au préprocesseur d'exécuter la commande `command` et de l'inclure dans le flot courant. Pour des raisons de sécurité, cette primitive est désactivée par défaut et provoque d'un message d'avertissement. Pour réactiver cette primitive, il convient d'appeler le préprocesseur avec l'option `-x`. Sa sortie est laissée telle quelle. Toute évaluation de cette sortie se fait en utilisant `#defeval`.

8.1.13 #eval expr

Cette primitive permet l'évaluation de l'expression `expr` et la tenue de calculs arithmétiques ou d'expressions régulières. La syntaxe et les priorités des différents opérateurs sont les mêmes qu'en C, les seuls opérateurs manquants étant les opérateurs `<<`, `>>`, l'opérateur ternaire et les opérateurs d'assignation.

Les expressions régulières POSIX ne sont disponibles que sur les systèmes POSIX et peuvent être invoqués grâce à l'opérateur `=`. Rapidement, un `?` remplace un caractère unique quelconque et `*` une suite quelconque de caractères incluant une suite de longueur nulle. La classe `[...]` correspond à l'un des caractères entre crochets. Le complémentaire d'une

classe est donné lorsque le premier caractère entre crochets est un `!`. Les caractères entre crochets peuvent être définis par un intervalle à l'aide du signe `-`. Ainsi `[F-N]` est équivalent à `[FGHIJKLMN]`.

Si la primitive est incapable de renvoyer une valeur numérique, elle retourne une chaîne de caractères qui est le résultat de l'expansion des macros dans aucune évaluation arithmétique. La seule exception à cette règle passe par l'usage des opérateurs de comparaison `==`, `!=`, `<`, `>`, `<=` et `>=` qui, si l'un de leurs arguments n'est pas numérique, effectue une comparaison de chaînes ignorant les espaces initiaux et finaux des deux chaînes. Un opérateur `length(macro)` renvoie la longueur de la chaîne résultant de l'évaluation de `macro`.

La primitive spéciale `defined(...)` est disponible. Elle renvoie 1 si le nom passé en argument est une macro, 0 sinon. Cette primitive n'est utilisable que dans une évaluation d'expression.

8.1.14 `#if expr`

Cette primitive invoque la fonction d'évaluation de la même manière que la primitive `#eval` et compare le résultat de cette évaluation avec la chaîne de caractères `"0"` (chaîne contenant le caractère zéro) avant de débiter un bloc conditionnel. En particulier, la valeur logique de `expr` est toujours vraie si cette expression ne peut être évaluée numériquement.

8.1.15 `#elif expr`

La primitive `#elif expr` remplace les primitives imbriquées `#else`, `#if expr` et `#endif`.

8.1.16 `#mode keyword`

La primitive `#mode` contrôle le fonctionnement interne du préprocesseur. Elle demande une commande obligatoire parmi les commandes suivantes :

- `save` ou `push` : sauvegarde un état du préprocesseur dans sa pile;
- `restore` ou `pop` : restaure un état du préprocesseur depuis sa pile;
- `standard` : sélectionne l'un des cinq standard de fonctionnement parmi `default`, `cpp` ou `C`, `tex` ou `TeX`, `html` ou `HTML`, `xhtml` ou `XHTML`, `prolog` ou `Prolog`. Le mode doit être donné directement, non sous la forme d'une chaîne de caractères. Par défaut, le préprocesseur fonctionne en mode `cpp`;
- `user`, `meta`, `quote`, `[no] comment`, `[no] string`, `preservelf`, `charset` : ces options permettent de modifier le fonctionnement du préprocesseur. Sauf utilisation spéciale du préprocesseur, elles ne sont pas nécessaires au fonctionnement du RPL/2. De plus amples informations sont disponibles dans le manuel du préprocesseur.

8.1.17 `#line`

L'expansion de cette primitive renvoie le numéro de ligne du fichier courant.

8.1.18 `#file`

La primitive `#file` renvoie le nom du fichier courant tel qu'il apparaît dans la ligne de commande ou dans une primitive `#include`. Si le fichier d'entrée du préprocesseur est l'entrée standard, elle est évaluée en `stdin`.

8.2. ORGANISATION DES PROGRAMMES

8.1.19 #date fmt

Cette primitive renvoie l'horodatage courant en fonction du format spécifié. De plus amples informations sur le format sont disponibles dans le manuel du préprocesseur.

8.1.20 #error msg

Cette primitive envoie un message d'erreur dans le flot. Ce message comprend le fichier courant, le numéro de la ligne et la chaîne `msg`. Le préprocesseur s'arrête.

8.1.21 #warning msg

Cette primitive envoie un message d'information dans le flot. Ce message comprend le fichier courant, le numéro de la ligne et la chaîne `msg`. Le préprocesseur continue le traitement.

8.2 Organisation des programmes

Un programme RPL/2 se compose d'un ensemble de définitions pouvant se trouver dans un ou plusieurs fichiers. Chaque définition comporte un nom sensible à la casse et une séquence d'instructions dans un bloc délimité par `<<` et `>>`.

```
0001 DEFINITION_1
0002 <<
0003     DEFINITION_2 disp
0004 >>
0005
0006 DEFINITION_2
0007 <<
0008     "Hello, world !"
0009 >>
```

Le fichier principal, passé sur la ligne de commande lors de l'appel, est traité par le préprocesseur `rp1pp` fonctionnant par défaut en mode `cpp`. La syntaxe et la structure du préprocesseur sont analysées avant la phase d'interprétation ou de compilation. L'ordre d'apparition de ces définitions dans le code source est indifférent, sauf en ce qui concerne la définition principale — par laquelle débute l'interprétation du programme — devant être la première à apparaître dans le code source une fois que celui-ci a été traité par le préprocesseur.

Une définition peut être appelée depuis n'importe quelle autre définition par son nom. La définition appelante effectue alors un branchement à la définition appelée, laquelle retourne à l'instruction de la définition appelante suivant le branchement lorsque son évaluation prend fin pour. Le programme s'achève à la fin de l'exécution de la définition principale.

8.3 Extension

Sous certains *shells*, en particulier sous Unix, il est possible de débiter le code source par un appel au séquenceur. Cela permet d'obtenir un fichier directement exécutable sans lancer explicitement

le séquenceur RPL/2. Un tel code débutera par exemple par la ligne `#!/usr/local/bin/rp1`.

CHAPITRE 8. EXÉCUTION DE PROGRAMMES

9

Convention d'appel librpl

CHAPITRE 9. CONVENTION D'APPEL LIBRPL

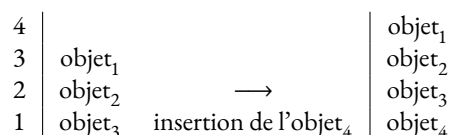
Troisième partie

Objets, pile et variables

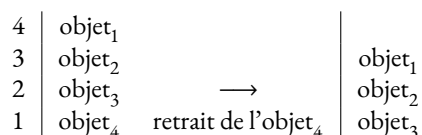
10

Modifications automatiques

La pile opérationnelle est une liste contenant une collection d'objets de tout type. Tout objet inséré dans cette pile se retrouve au bas de celle-ci — au niveau 1 —, les autres objets remontant automatiquement d'un niveau.



Réciproquement, tout objet retiré de la pile l'est par le bas. Le nombre d'élément de la pile n'est limité que par la mémoire disponible sur le calculateur. Cette pile est traitée comme une liste chaînée et l'ajout ou la suppression d'un objet de provoque que le chaînage ou la suppression d'un maillon. Les maillons supprimés entretiennent une liste cache pour réduire au maximum les allocations de mémoire.



Ainsi, la pile opérationnelle se comporte comme une file LIFO¹ à extension non limitée. Seules deux erreurs peuvent être provoquées par des opérations directes sur cette pile :

- une erreur de dépassement mémoire, de type système et non récupérable, lorsqu'il ne reste plus de mémoire allouable sur la machine hôte pour chaîner un maillon;
- une erreur signalant une pile vide en cas de tentative de dépilement d'un objet. Cette erreur est une erreur d'exécution pouvant être récupérée grâce à une structure de contrôle `iferr` ou à une racine secondaire de processus `nrproc`.

10.1 Fonctionnement des routines d'évaluation

Avant d'aborder les modifications implicites de la pile, il convient de comprendre le fonctionnement interne du RPL/2 et en particulier les mécanismes conduisant à l'évaluation d'une expression. Pour le séquenceur, tout est objet, et en tant qu'objet, soumis à évaluation. Les

¹Last In First Out

objets qui ne peuvent pas être scindés en objets plus petits sont appelés des atomes et sont les seuls objets directement évaluable.

Ainsi, pour qu'un objet soit évalué, il est réduit à un arbre² d'atomes évalué séquentiellement. Un programme est composé de plusieurs arbres, chaque arbre contenant une définition. Le branchement d'une définition à une autre revient à greffer un arbre sur un autre. Lors de l'évaluation séquentielle d'un arbre, le séquenceur exécute les atomes qui correspondent à des instructions et pousse dans la pile tous les autres.

La représentation interne du programme de calcul de factorielle déjà évoqué dans cet ouvrage

```

0001 CALCUL_DE_FACTORIELLE
0002 <<
0003   "Calcul de n!" disp "n = " prompt str->
0004   if
0005     dup dup ip same
0006   then
0007     FACTORIELLE disp
0008   else
0009     drop
0010     "ERREUR : argument n non entier !" disp
0011   end
0012 >>
0013
0014 /*
0015 =====
0016   Calcul recursif de factorielle
0017 =====
0018 */
0019
0020 FACTORIELLE
0021 <<
0022   -> N
0023   <<
0024     if
0025       N 1 >
0026     then
0027       N dup 1 - FACTORIELLE *
0028     else
0029       1
0030     end
0031   >>
0032 >>

```

est faite à la figure 10.1 sous la forme de deux arbres atomiques et d'une greffe, apparaissant en gris, et reliant le programme principal `CALCUL_DE_FACTORIELLE` à la fonction `FACTORIELLE`. La fonction `FACTORIELLE` est récursive et s'appelle elle-même. Le séquenceur accepte les fonctions récursives sans limitation de profondeur et contrairement à d'autres langages comme le Fortran, il n'est pas nécessaire de déclarer une fonction récursive car elles ne s'exécutent pas dans le même plan mémoire.

L'exemple précédent est volontairement simpliste. Il n'utilise que des types de données atomiques et ne doit pas faire accroire qu'il existe une quelconque équivalence entre les atomes et les types d'objets du RPL/2. Les atomes sont les définitions intrinsèques, les définitions extrinsèques et les objets des types suivants :

- scalaires (entiers, réels, complexes);

²Il s'agit d'un arbre et non d'une séquence car les objets peuvent contenir non seulement des atomes mais aussi d'autres objets.

"Calcul de n!"

10.1. FONCTIONNEMENT DES ROUTINES D'ÉVALUATION

«

CALCUL_DE_FACTORIELLE

CHAPITRE 10. MODIFICATIONS AUTOMATIQUES

- vecteurs;
- matrices;
- entiers binaires;
- nom;
- chaîne de caractères;
- descripteurs de fichier et de socket;
- descripteur de bibliothèque;
- descripteur de processus;
- connecteur SQL;
- mutex et sémaphore.

Les objets de type liste, table et expressions algébrique ou en notation polonaise inversée ne sont pas atomiques. Ils se composent d'une fonction et d'une collection d'objets atomiques ou non. L'arbre associé à l'objet

{ 1 5 3 <[[1 2] 'X+1']> << X sin >> }

est

{ }

1

5

3

<[]>

10.1.2 Cas d'un programme compilé

La compilation d'un programme RPL/2 consiste à le traduire sous forme d'une forêt d'arbres avant de commencer son exécution. En particulier, tous les atomes formant les arbres sont associés à leur type une fois pour toute. De la même façon, les définitions intrinsèques on non sont remplacées par un objet interne contenant un pointeur sur la fonction réellement effectuée. La routine d'évaluation se réduit donc à sa plus simple expression : balayer séquentiellement un arbre en évaluant tous les atomes rencontrés.

10.2 Opérations implicites

Toute séquence d'instructions est transformée pour être évaluée en une suite d'atomes. Chaque atome est soit une définition intrinsèque ou extrinsèque, soit une donnée à empiler. Dans les deux cas, il modifie le contenu de la pile. Plus généralement, toutes ces définitions prennent leurs arguments dans la pile — marginalement dans des variables globales — et renvoient leurs arguments dans cette pile — marginalement dans des variables globale.

$n + m$	objet _{<i>m</i>}		$p + m$	objet _{<i>m</i>}
$\vdots$	$\vdots$		$\vdots$	$\vdots$
$n + 2$	objet ₂		$p + 2$	objet ₂
$n + 1$	objet ₁		$p + 1$	objet ₁
n	argument _{<i>n</i>}		p	résultat _{<i>p</i>}
$n - 1$	argument _{<i>n-1</i>}		$p - 1$	résultat _{<i>p-1</i>}
$\vdots$	$\vdots$		$\vdots$	$\vdots$
2	argument ₂	→	2	résultat ₂
1	argument ₁	instruction	1	résultat ₁

Il faut noter que cette pile est utilisée par le séquenceur pour certains de ses calculs internes. Un arrêt anormal dans l'exécution d'une définition peut ainsi corrompre la pile — non les objets contenus dans cette pile. Néanmoins et dans la mesure du possible, la pile retrouve un état cohérent même après tel arrêt.

10.3 Opérations explicites

Un certain nombre d'opérations spécifiques permet de gérer directement les données contenues dans la pile opérationnelle. Il s'agit principalement de duplication, de destruction d'objets, de modification d'ordre dans la pile et de changement de contextes. Contrairement aux opérations implicites qui imposent l'évaluation des objets traités, les opérations explicites n'évaluent jamais leurs arguments.

oui

Programme interprété

Recherche de
l'atome suivant

Évaluation

Instruction?

Empilement

Variable?

Création d'un arbre et
affectation d'un type

non

II.1 Gestion de la pile

Cette instruction retire tous les objets de la pile et vide la pile LAST. Il n'est pas possible de récupérer les objets effacés.

L'instruction `depth` renvoie un entier au niveau 1 de la pile donnant le nombre d'objets de la pile *avant* l'appel de cette instruction.

$$\begin{array}{ccc|ccc}
n+1 & & & & & \text{obj}_{n-1} \\
n & \text{obj}_n & & & & \text{obj}_{n-1} \\
n-1 & \text{obj}_{n-1} & & & & \text{obj}_{n-2} \\
\vdots & \vdots & & & & \vdots \\
2 & \text{obj}_2 & \longrightarrow & & & \text{obj}_1 \\
1 & \text{obj}_1 & \text{depth} & & & n
\end{array}$$

`last` renvoie dans la pile opérationnelle le contenu de la pile `last` qui contient les arguments de la dernière définition intrinsèque exécutée. Le drapeau 31 conditionne la validation de cette sauvegarde et la pile `last` n'est pas directement modifiable par l'utilisateur.

II.2.1 Dup

L'instruction `dup` duplique l'objet présent au niveau 1 de la pile opérationnelle. `dup` n'effectue pas de copie de l'objet mais ajoute simplement une référence sur le premier objet et sur tous les atomes le composant.

$$\begin{array}{c|cc|c} 2 & \text{objet}_2 & \longrightarrow & \text{objet}_1 \\ 1 & \text{objet}_1 & \text{dup} & \text{objet}_1 \end{array}$$

II.2.2 Copy

L'instruction `copy` renvoie une copie de l'objet présent au niveau 1 de la pile opérationnelle. Contrairement à l'instruction `dup`, `copy` effectue une copie de l'objet passé en argument atome par atome. L'utilité de cette fonction est limitée et un programme standard ne devrait jamais l'appeler.

$$\begin{array}{c|c|c|c} 2 & \text{objet}_2 & \longrightarrow & \text{objet}_1 \\ 1 & \text{objet}_1 & \text{copy} & \text{objet}_1 \end{array}$$

II.2.3 Dup2

Cette instruction est identique à `dup`, si ce n'est que les deux objets situés aux niveaux 1 et 2 de la pile sont dupliqués.

$$\begin{array}{c|c|c|c} 4 & & & \text{objet}_2 \\ 3 & & & \text{objet}_1 \\ 2 & \text{objet}_2 & \longrightarrow & \text{objet}_2 \\ 1 & \text{objet}_1 & \text{dup2} & \text{objet}_1 \end{array}$$

II.2.4 Dupn

`dupn` permet de dupliquer en une seule fois n éléments de la pile. Cette instruction prend un entier positif présent au niveau 1 comme argument et duplique les objets des niveaux 2 à $n + 1$.

$$\begin{array}{c|c|c|c} 2n & & & \text{objet}_n \\ \vdots & & & \vdots \\ n+1 & \text{objet}_n & & \text{objet}_1 \\ n & \text{objet}_{n-1} & & \text{objet}_n \\ \vdots & \vdots & & \vdots \\ 2 & \text{objet}_1 & \longrightarrow & \text{objet}_2 \\ 1 & n & \text{dupn2} & \text{objet}_1 \end{array}$$

II.2.5 Over

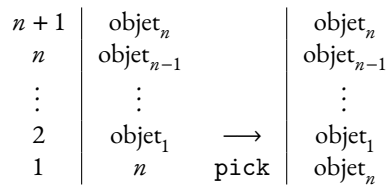
L'instruction `over` duplique non l'objet présent au niveau 1 de la pile, mais celui occupant de niveau 2. Il s'agit de l'une des rares instructions du RPL/2 n'opérant pas sur le niveau 1.

$$\begin{array}{c|c|c|c} 3 & & & \text{objet}_2 \\ 2 & \text{objet}_2 & \longrightarrow & \text{objet}_1 \\ 1 & \text{objet}_1 & \text{over} & \text{objet}_2 \end{array}$$

II.2.6 Pick

`pick` est une extension de l'instruction `dup`. En effet, cette dernière duplique l'objet occupant la première position dans la pile. Contrairement à `dup`, `pick` prend un argument entier positif n au niveau 1 de la pile et duplique le $n^{\text{ième}}$ objet de la pile *avant* l'empilement de l'argument n de `pick`, ou — ce qui est rogneusement identique — le $(n + 1)^{\text{ième}}$ de la pile avant l'appel à `pick`.

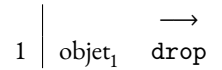
II.3. SUPPRESSION D'ÉLÉMENTS



II.3 Suppression d'éléments

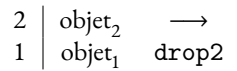
II.3.1 Drop

L'instruction `drop` retire le premier élément de la pile, les autres redescendant d'un niveau. L'objet retiré de la pile peut être retrouvé par un appel à la commande `last` si cette dernière est active. Cette instruction retire une référence de l'objet passé en argument et de tout ses atomes et ne libère la mémoire utilisé par cet objet que lorsque cet objet n'est plus référencé.



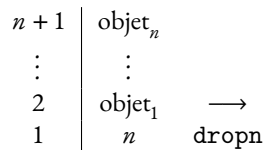
II.3.2 Drop2

`drop2` retire les deux premiers éléments de la pile qui peuvent encore être retrouvés par la commande `last`.



II.3.3 Dropn

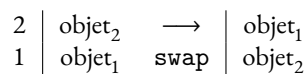
L'instruction `dropn` retire les $(n + 1)^{\text{ième}}$ premiers objets de la pile (les n premiers objets et l'argument n lui-même). L'argument n est sauvegardé sur la pile `last` et peut ainsi être retrouvé grâce à un appel à la commande `last`, mais les autres objets sont définitivement perdus.



II.4 Modification de la hiérarchie

II.4.1 Swap

`swap` permute les deux premiers objets de la pile. Cette opération se fait sans copie d'objet mais les deux arguments sont sauvegardés dans la pile `last` si celle-ci est active.



II.4.2 Rot

`rot` effectue une permutation circulaire des trois premiers éléments de la pile opérationnelle, l'objet présent au niveau 1 de la pile étant celui occupant le niveau 3 avant l'appel de `rot`.

$$\begin{array}{c|c|c} 3 & \text{objet}_3 & \\ 2 & \text{objet}_2 & \longrightarrow \\ 1 & \text{objet}_1 & \text{rot} \end{array} \quad \begin{array}{c} \text{objet}_2 \\ \text{objet}_1 \\ \text{objet}_3 \end{array}$$

II.4.3 Roll

`roll` prend un nombre entier positif n dans la pile opérationnelle, puis « déplace » les n premiers objets restant sur la pile de sorte que l'objet 1 occupant le niveau $n + 1$ de la pile se retrouve au niveau 1, les objets susceptibles d'occuper les niveaux supérieurs à $n + 2$ ne sont pas concernés par cette instruction. Si la pile last est validée, il est possible d'utiliser la séquence de commandes « `last roll` » pour inverser l'effet de `roll`.

$$\begin{array}{c|c|c} n+1 & \text{objet}_n & \\ n & \text{objet}_{n-1} & \\ \vdots & \vdots & \\ 2 & \text{objet}_1 & \longrightarrow \\ 1 & n & \text{roll} \end{array} \quad \begin{array}{c} \text{objet}_{n-1} \\ \vdots \\ \text{objet}_1 \\ \text{objet}_n \end{array}$$

II.4.4 Rolll

`rolll` prend un nombre entier positif n dans la pile est fait défiler vers le bas les n premiers objets restants dans la pile, l'objet présent au niveau 1 se retrouvant alors au niveau n .

$$\begin{array}{c|c|c} n+1 & \text{objet}_n & \\ n & \text{objet}_{n-1} & \\ n-1 & \text{objet}_{n-2} & \\ \vdots & \vdots & \\ 2 & \text{objet}_1 & \longrightarrow \\ 1 & n & \text{rolll} \end{array} \quad \begin{array}{c} \text{objet}_1 \\ \text{objet}_n \\ \vdots \\ \text{objet}_3 \\ \text{objet}_2 \end{array}$$

II.4.5 Edit

L'instruction `edit` appelle l'éditeur `vim` pour modifier l'objet présent au niveau 1. Après la fermeture de l'éditeur, le nouvel objet est soumis à une analyse syntaxique. Si le nouvel objet est valide, il est renvoyé dans la pile. Sinon, l'instruction `edit` provoque une erreur d'exécution.

$$\begin{array}{c|c|c} 2 & & \longrightarrow \\ 1 & \text{objet} & \text{edit} \end{array} \quad \begin{array}{c} \\ \text{objet} \end{array}$$

II.5 Gestion des contextes

Le RPL/2 est un langage de programmation utilisant un adressage par pile. Tous les processus s'exécutent dans des plans différents, que ces processus soient détachés ou légers. En particulier, les variables modifiées dans un fil d'exécution ne sont modifiées que pour ce fil sauf si elles sont explicitement déclarées comme partagées. Ce mode d'adressage par pile impose

II.5. GESTION DES CONTEXTES

donc la création de piles indépendantes, appelées contextes d'exécution, pour traiter les différents fils d'exécution. La création explicite de contextes permet aussi de gérer simplement les exceptions dans les programmes. Tous les contextes sont effacés à la fin du fil d'exécution qui les a créés.

II.5.1 Pshcntxt

L'instruction `pshcntxt` empile le contexte courant et retourne une pile opérationnelle vide.

II.5.2 Pulcntxt

L'instruction `pulcntxt` détruit la pile courante et la remplace par le dernier contexte sauvegardé qui est retiré de la pile des contextes. Si aucun contexte n'a été précédemment sauvegardé, cette instruction renvoie une erreur d'exécution.

II.5.3 Dupcntxt

L'instruction `dupcntxt` sauvegarde le contexte courant pour une réutilisation ultérieure. Contrairement à `pshcntxt`, la pile opérationnelle n'est pas affectée.

II.5.4 Dropcntxt

L'instruction `dropcntxt` efface le dernier contexte sauvegardé. Si la pile des contextes est vide, elle retourne une erreur d'exécution. La pile opérationnelle n'est pas affectée par cette instruction.

II.5.5 Swapcntxt

Cette instruction échange le contexte courant avec le dernier contexte sauvegardé. Elle est prévu pour échanger rapidement deux contextes, par exemple dans le cas de traitement d'interruption rapide.

II.5.6 Clrcntxt

L'instruction `clrcntxt` efface l'ensemble des contextes sauvegardés. Elle ne touche pas au contenu de la pile opérationnelle courante.

II.5.7 Exemple d'utilisation

La fonction suivante illustre une utilisation simple des instructions de changement de contexte. Cette fonction comporte un bloc, encadré par une structure `if err` de reprise sur erreur, susceptible de provoquer une exception ou une erreur d'exécution. Si cette fonction est relativement simple, il est facile d'encadrer les zones critiques par plusieurs mécanismes de reprise sur erreur. Pour un traitement complexe, il peut être plus élégant d'utiliser un seul bloc de reprise sur erreur, mais il est beaucoup plus difficile de maintenir la pile opérationnelle dans un état cohérent pour la suite de l'exécution.

Ainsi, cette fonction sauvegarde le contexte courant à la ligne 3 sans toucher au contenu de la pile opérationnelle. Si une erreur survient, le mécanisme de reprise sur erreur traite l'instruction `pulcntxt` de la ligne 8 qui efface la pile courante et la remplace par le contexte sauvegardé à la ligne 3 et pousse la valeur « faux » dans la pile. S'il n'y a eu aucune erreur, l'instruction `dropcntxt` de la ligne 10 efface silencieusement le contexte sauvegardé à la ligne 3

CHAPITRE II. MANIPULATION DES OBJETS

et empile la valeur « vrai ». Dans tous les cas de figure, la pile reste dans un état cohérent, charge à la fonction appelante de traiter l'erreur en fonction de la valeur logique renvoyée dans le premier niveau de la pile.

```
0001 FONCTION
0002 <<
0003     dupcntxt
0004
0005     iferr
0006         // Traitements
0007     then
0008         pulcntxt false
0009     else
0010         dropcntxt true
0011     end
0012 >>
```


12

Entrées et sorties

Ce point ne traite que des entrées et sorties standard en mode texte. Il ne traite ni des fichiers, ni des graphiques, ni de l'impression. Le RPL/2 ne comporte aucune instruction de gestion évoluée d'un terminal texte ou graphique, mais il est possible de l'enrichir à l'aide des bibliothèques RPL/C Ncurses ou Motif.

12.1 Sorties

L'affichage se fait par défaut dans un terminal orienté caractère, ligne par ligne. Une redirection de la sortie standard dans un fichier est possible. Seules les informations explicitement envoyées par un programme RPL/2 le sont sur la sortie standard. Toutes les autres sorties (bandeaux et messages d'erreur) sont faites sur l'erreur standard.

12.1.1 Disp

L'instruction `disp` prend un objet quelconque au premier niveau de la pile opérationnelle et l'affiche en tenant du format courant des nombres et de celui des objets. Le comportement de cette instruction dépend de l'état des indicateurs 32, 33 et 45. Par défaut, l'indicateur 33 est désarmé et un caractère de retour à la ligne est ajouté à l'objet à afficher. L'indicateur 45 correspondant à un affichage multiligne est armé par défaut. Si l'indicateur 32 est armé, l'instruction `disp` envoie une copie de ce qui est affichée vers la sortie d'impression.

1	objet ₁	→	disp
---	--------------------	---	------

12.1.2 Format

Les quatre instructions de format `std`, `fix`, `sci` et `eng` modifient l'affichage des nombres. En aucun cas, leur représentation interne n'est touchée. L'utilisation de l'une de ces instructions conditionne le fonctionnement de toutes les instructions future utilisant la routine de formatage des nombres. En particulier, elle conditionne le fonctionnement de l'instruction `→str→str` transformant un nombre en chaîne de caractères :

```
cauchy:[~] > rpl -is
+++RPL/2 (R) version 4.0.10 (lundi 08/02/2010, 10:45:34 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
```

```
+++Ce logiciel est un logiciel libre sans aucune garantie de fonctionnement.
```

CHAPITRE 12. ENTRÉES ET SORTIES

```
+++Pour plus de détails, utilisez la commande 'warranty'.

RPL/2> 2 fix 0 ->str

1: "0.00"
RPL/2>
```

12.1.2.1 Séparateur décimal

Le séparateur décimal est au choix le point ou la virgule et dépend de la valeur de l'indicateur 48. Si l'indicateur 48 est armé, le séparateur décimal est le point. Sinon, il s'agit de la virgule. Le nombre complexe $3 + 2i$ peut ainsi être (3 . , 2 .) si l'indicateur 48 est désarmé ou (3 , . 2 ,) s'il est armé. Pour tout ce qui suit, le séparateur décimal sera le point.

Tout nombre sans séparateur décimal et sans exposant est un entier. Ainsi 1 . est un réel alors que 1 est un entier. Tous les nombres représentés avec un séparateur décimal ou un exposant sont des réels. Les complexes sont toujours représentés par un couple de nombres réels.

12.1.2.2 Format standard

Le format standard est le format par défaut d'affichage des nombres et donne les résultats suivants lors de l'affichage ou du traitement d'un nombre :

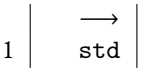
- les nombres pouvant être représentés exactement comme des entiers avec quinze chiffres ou moins sont affichés sans séparateur décimal ni exposant ;
- les nombres pouvant être représentés exactement avec quinze chiffres ou moins, mais qui ne sont pas des entiers, sont affichés avec un séparateur décimal, mais sans exposat. Les zéros de tête à gauche du séparateur décimal et les zéros de queue dans la partie fractionnaire sont omis ;
- tous les autres nombres sont affichés dans le format suivant :

$$(\text{signe}) \text{ mantisse } E (\text{signe}) \text{ exposant}$$

où l'exposant est un nombre de un à trois chiffres et où la valeur x de la mantisse satisfait à

$$1 \leq x < 10$$

L'instruction `std` ne prend aucun argument dans la pile et ne renvoie rien. Elle se contente de modifier les indicateurs 49 et 50.



Le tableau suivant donne des exemples de nombres affichés en format standard :

Nombre	Affiché	Représentable avec 15 chiffres
10^{11}	100000000000	oui
10^{16}	1E16	non
10^{-15}	0.000000000000001	oui
1.2×10^{-15}	1.2E-15	non
12.345	12.345	oui

12.1.2.3 Format fixe

L'instruction `fix` choisit le format fixe comme mode d'affichage numérique. Elle prend un argument entier pour définir le nombre de décimales n — entre 0 et 15 — à afficher. Si la valeur de l'argument n'est pas dans cet intervalle, l'instruction renvoie une erreur d'exécution.

$$1 \mid n \xrightarrow{\quad} \text{fix} \mid$$

En format fixe, les nombres apparaissent sous la forme :

(signe) mantisse

La mantisse apparaît arrondie à n décimales à droite du séparateur décimal. Même lorsque le format fixe est imposé, le RPL/2 utilisera le format scientifique dans l'un des deux cas ci-dessous :

- si le nombre de chiffres à afficher dépasse quinze ;
- si une valeur non nulle arrondie à n décimales est affichées comme un zéro en format fixe.

12.1.2.4 Format scientifique

L'instruction `sci` choisit le mode d'affichage scientifique. Elle prend un argument entier pour définir le nombre de chiffres significatifs n — entre 0 et 15 — à afficher. Si la valeur de l'argument n'est pas dans cet intervalle, l'instruction renvoie une erreur d'exécution.

$$1 \mid n \xrightarrow{\quad} \text{sci} \mid$$

En format scientifique, les nombres sont affichés avec $n + 1$ chiffres significatifs, n étant la valeur de l'argument de la fonction `sci`. Toute valeur apparaît sous la forme :

(signe) mantisse E (signe) exposant

où la mantisse satisfait à

$$1 \leq x < 10$$

12.1.2.5 Format ingénieur

L'instruction `eng` choisit le mode d'affichage scientifique. Elle prend un argument entier pour définir le nombre de chiffres significatifs n — entre 0 et 15 — à afficher. Si la valeur de l'argument n'est pas dans cet intervalle, l'instruction renvoie une erreur d'exécution.

$$1 \mid n \xrightarrow{\quad} \text{eng} \mid$$

En format ingénieur, les nombres sont affichés avec $n + 1$ chiffres significatifs, n étant la valeur de l'argument de la fonction `eng`. Toute valeur apparaît sous la forme :

(signe) mantisse E (signe) exposant

où l'exposant est un multiple de 3 et où la mantisse satisfait à

$$1 \leq x < 1000$$

CHAPITRE 12. ENTRÉES ET SORTIES

12.1.2.6 Arrondi

L'instruction `rnd` arrondit l'objet pris au niveau 1 de la pile en fonction du mode d'affichage des nombres.

```
RPL/2> pi ->num 3 fix rnd std
1: 3.142
RPL/2>
```

12.1.2.7 Troncature

Cette instruction prend comme argument un scalaire, un vecteur ou un tableau et renvoie dans la pile un objet de même type.

Elle prend deux arguments dans la pile, un objet de type scalaire, vecteur ou matrice et un entier n qui conditionne la troncature. Si n est positif, la troncature est faite selon le format `n sci`. Si n est négatif, cette troncature est faite selon le format `n fix`. Contrairement à l'instruction `rnd`, aucune hypothèse n'est faite sur le format courant.

$$1 \mid \begin{array}{c} \text{Objet} \\ n \end{array} \xrightarrow{\text{trnc}} \mid \text{Objet}$$

12.1.3 Clmf

`clmf`

L'instruction `clmf` affiche le contenu de la pile. Elle ne prend aucun argument et ne renvoie rien dans la pile. Tous les objets sont affichés conformément au format courant.

12.2 Entrées

Par défaut, toute entrée est faite depuis l'entrée standard. Cette entrée standard peut prendre la forme d'une redirection depuis un fichier.

12.2.1 Input

L'instruction `input` ne prend aucun argument. Elle attend une entrée terminée par un retour à la ligne depuis l'entrée standard et renvoie la chaîne de caractère correspondante. Contrairement à l'instruction `prompt`, elle n'affiche aucune invite.

$$1 \mid \begin{array}{c} \rightarrow \\ \text{input} \end{array} \mid \text{"chaîne"}$$

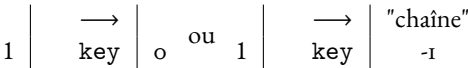
12.2.2 Prompt

Le fonctionnement de l'instruction `prompt` très similaire à celui de l'instruction `input` puisque qu'elle attend aussi une entrée depuis l'entrée standard. Contrairement à l'instruction `input`, `prompt` affiche l'invite passée en argument.

$$1 \mid \text{"chaîne"} \mid \begin{array}{c} \rightarrow \\ \text{input} \end{array} \mid \text{"chaîne"}$$

12.2.3 Key

L'instruction `key` permet de saisir une touche au vol. Dans le cas où une saisie est effective, elle renvoie cette touche sous la forme d'une chaîne de caractère et la valeur binaire *vrai*. Dans tous les autres cas, elle renvoie la valeur *faux*.



Le programme suivant est constitué d'une boucle qui s'achève lorsque l'utilisateur appuie sur une touche. La valeur de cette touche est affichée par l'instruction `disp`.

```
0001 MAIN
0002 <<
0003     do
0004     until
0005         key
0006     end
0007
0008     disp
0009 >>
```

12.3 Beep

Cette instruction envoie le caractère *bell* à la sortie standard. Elle est soumise à l'état de l'indicateur `51`. Si cet indicateur est désarmé, l'instruction `beep` est silencieusement ignorée.

13

Accessibilité des variables

La notion de variable est en RPL/2 assez différente de ce qui peut se rencontrer dans d'autres langages. En effet, une variable n'est que l'association d'un nom à un objet préexistant. Elle ne contient aucune information quant au type *a priori* de l'objet associé.

Cette association entre un nom et un objet est soumise à des règles précises de visibilité ou de portée conditionnées par les niveaux d'exécution et le type de variable (globale, locale, volatile, statique ou partagée). La définition de ces types de variables figure page 55. Une variable volatile est soumise à une portée car l'association entre le nom et la donnée ne survit pas à la fin du bloc dans lequel elle a été définie. Une variable statique ou partagée relève de règles de visibilité car cette association survit tant que cette variable n'est pas transformée en une variable volatile. Même partagée ou statique, cette variable n'est accessible que dans le bloc d'instructions qui l'a définie.

13.1 Niveaux d'exécution

Les niveaux d'exécution conditionnent la visibilité d'une variable ou sa portée dans le cas d'une variable volatile. Ces niveaux correspondent aux différentes structures imbriquées, qu'ils s'agisse d'expressions en notation polonaise inversée, d'expressions algébriques ou de structures de boucle créant un compteur. Les variables à l'intérieur d'une même fonction se masquent les unes les autres et seules sont accessibles les variables de plus haut niveau.

0001	NIVEAU				
0002	<<			-\ 1	
0003	1 -> A			-\ 2	1
0004	<<				
0005	A 1 +				
0006	-> B		-\ 3		
0007	<<				
0008	1 10 for A	-\ 4			
0009	// Traitement				
0010	next	-/			
0011	>>		-/		
0012	>>			-/	
0013	>>				-/

En d'autres termes, la variable A accessible ligne 9 est la variable locale et volatile définie par la boucle ligne 8 car son niveau 4 est supérieur au niveau 1 de la variable locale A créée ligne 3. Toute variable créée dans le bloc de niveau 1 ne peut l'être que par une instruction `sto` ou `save` et devient une variable globale. La variable NIVEAU contenant le nom de la

CHAPITRE 13. ACCESSIBILITÉ DES VARIABLES

définition est créée hors du bloc de niveau 1. Elle est de niveau 0.

Seules les variables de niveau strictement positif sont modifiables par un programme au cours de son exécution. Les variables de niveau 0 ne sont modifiables que lors de l'analyse structurale d'un programme, soit avant le début de l'exécution. Comme elles sont, contrairement à toutes les autres variables, non modifiables, elles sont verrouillées et ne sont pas dupliquées lors de la création d'un processus fils.

À chaque point d'un programme, les seules variables accessibles sont les variables de niveaux 0 et 1 et les variables définies dans la fonction courante. Toutes les variables définies dans toutes les autres fonctions sont inaccessibles.

13.2 Évaluation implicite

Une variable est une association entre un nom et un objet. L'évaluation de ce nom permet dans certain cas d'accéder au contenu de la variable. En effet, un objet de type nom peut être symbolique ou non. Un nom symbolique est un nom apparaissant avec ses délimiteurs de type

'NOM_SYMBOLIQUE'

Si les délimiteurs de type sont absents, le nom est dit évaluable

NOM_ÉVALUABLE

Le traitement d'un atome reconnu comme un nom par la routine d'évaluation dépend du caractère évaluable ou symbolique de ce nom. Le traitement d'un nom est expliqué à la figure 13.1.

13.3 Évaluation explicite

Les instructions EVAL ou ->NUM forcent l'évaluation d'un nom symbolique. Ces deux instructions ne sont pas identiques. En effet, le traitement des noms symboliques rencontrés lors de l'évaluation diffèrent. Dans le cas d'une évaluation par EVAL, les noms symboliques sont traités comme tels. Si l'évaluation est faite par ->NUM, tous les noms, qu'ils soient symboliques ou évaluable, sont traités comme des noms évaluable. Ainsi, le programme suivant définit une variable évaluée explicitement de deux manières différentes :

```
0001 MAIN
0002 <<
0003 << 0 -> X << 'X' sin >> >> 'EVALUATION' sto
0004 'EVALUATION' eval disp
0005 'EVALUATION' ->num disp
0006 >>
```

dont la sortie est :

```
+++RPL/2 (R) version 4.0.10 (lundi 01/02/2010, 11:21:30 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
'SIN(X)'
0.
```

De la même manière, les instructions eval et ->num fonctionnent avec des définitions qui ne sont que des variables particulières de niveau 0.

```
0001 MAIN
```


13.3. ÉVALUATION EXPLICITE

Début de l'évaluation d'un nom

Nom symbolique?

oui Empilement du nom Variable accessible de même nom?

Empilement du contenu de la variable

Fin de l'évaluation d'un nom

CHAPITRE 13. ACCESSIBILITÉ DES VARIABLES

```

0002 <<
0003   'EVALUATION' eval disp
0004   'EVALUATION' ->num disp
0005 >>
0006
0007 EVALUATION
0008 <<
0009   0 -> X
0010   << 'X' sin >>
0011 >>

```

13.4 Liste des variables

À tout moment, l'instruction `vars` renvoie la liste des variables connues dans la pile.

1	$\rightarrow$	liste
	<code>vars</code>	

La liste retournée par l'instruction `vars` contient l'état de chaque variable et son contenu. Chaque variable est décrite par une liste contenant dans l'ordre :

- le nom de la variable;
- le niveau de la variable;
- un indicateur de portée parmi "VOLATILE" et "STATIC";
- un indicateur de verrou parmi "LOCKED" et "UNLOCKED";
- un indicateur de visibilité parmi "PRIVATE" et "PUBLIC".

Les indicateurs sont modifiés instantanément par les instructions `volatile` et `private`. Le résultat de l'instruction `vars` est écrite dans le fichier d'analyse *post mortem* `rpl-core`.

13.4.1 Exemple

L'exemple suivant illustre l'utilisation de la commande `vars`. Lorsque le RPL/2 est lancé en mode interactif, il exécute un programme interprété dans l'outil de débogage. Ce programme appelé `MODE_INTERACTIF` consiste en la boucle infinie suivante :

```

0001 MODE_INTERACTIF
0002 << DO HALT UNTIL FALSE END >>

```

Ainsi, le démarrage en mode interactif du RPL/2 induit la création d'une entrée dans la table des variables correspondant au point d'entrée de la définition principale. Cette variable est de niveau 0, volatile et verrouillée. L'adresse du point d'entrée de la définition principale est @ 000000000000000F. La variable globale 'X' est initialisée avec une chaîne de caractères. Elle est par défaut volatile et non verrouillée.

```

cauchy:[~] > rpl -is
+++RPL/2 (R) version 4.0.10 (lundi 08/02/2010, 20:16:30 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël

```

```

+++Ce logiciel est un logiciel libre sans aucune garantie de fonctionnement.
+++Pour plus de détails, utilisez la commande 'warranty'.

```

```

RPL/2> "Chaîne de caractère" X sto

```

13.4. LISTE DES VARIABLES

```
RPL/2> vars
1: { { 'MODE_INTERACTIF'
      0
      @ 000000000000000F
      "VOLATILE"
      "LOCKED"
      "PRIVATE" }
  { 'X'
    1
    "Chaîne de caractère"
    "VOLATILE"
    "UNLOCKED"
    "PRIVATE" } }
```


14

Variables globales

Une variable globale ne peut être ni statique ni partagée. Elle est de niveau 1 et accessible de tout point du programme. Elle peut être masquée par une variable locale. Dans ce cas, son accès en lecture et en écriture se fait au travers des instructions spécifiques `rcl`, `save` et `visit`.

Une variable globale n'est pas partagée entre différents processus. Un processus fils hérite cependant de l'ensemble des variables de son père, en particulier de ses variables globales. Une variable globale dans un processus est ainsi globale dans tous ses fils même si elle n'est pas partagée par aucun de ces processus.

14.1 Création

Le moyen canonique de création d'une variable globale passe par l'utilisation de l'instruction `save`. Contrairement à l'instruction `sto` qui donne accès à la variable de plus haut niveau au point courant et ne crée une variable globale que si aucune variable locale de même nom n'est accessible, l'instruction `save` garantit l'accès à la variable globale.

```
0001 MAIN
0002 <<
0003     0
0004     -> X
0005     <<
0006         5 'X' sto    // Modification de la variable locale X
0007         3 'X' save   // Création de la variable globale X
0008         1 'Y' sto    // Création de la variable globale Y
0009     >>
0010 >>
```

Les instructions `save` et `sto` prennent deux arguments dans la pile. La seule différence entre ces deux instructions est la garantie donnée par `save` de travailler sur une variable globale.

2	objet	→	
1	'nom global'	save ou sto	

14.2 Accès

Deux mécanismes distincts permettent d'accéder à une variable globale. Par défaut, l'évaluation d'un nom correspondant à une variable renvoie dans la pile le contenu de cette variable.

CHAPITRE 14. VARIABLES GLOBALES

En absence de toute variable locale de même nom, le contenu de la variable globale est empilé. Cependant, pour éviter toute ambiguïté et permettre l'accès en toute circonstance à une variable globale, il est possible de la rappeler grâce à l'instruction `rcl`. Cette instruction ne porte que sur une variable globale et retourne une erreur d'exécution s'il n'existe aucune variable globale correspondant au nom passé en argument.

$$1 \mid \text{'nom global'} \quad \xrightarrow{\text{rcl}} \mid \text{objet}$$

14.3 Modification

Les fonctions arithmétiques directes fonctionnent avec des variables locales. Il convient donc de faire attention lors de leur utilisation avec des variables globales qui peuvent être masquées par des variables locales. Le seul mécanisme garantissant une modification de variable globale est une utilisation séquentielle des instructions `rcl` et `save`.

L'instruction `visit` permet la modification interactive d'une variable globale dans un éditeur. Le contenu de la variable est remplacé par le contenu de l'éditeur si ce contenu est syntaxiquement correct. L'instruction `visit` prend le nom de la variable globale dans la pile et ne renvoie rien.

$$1 \mid \text{'nom global'} \quad \xrightarrow{\text{visit}} \mid$$

14.4 Libération

Les variables globales ont une portée égale à celle du programme. Elles ne sont jamais détruites comme le sont les variables locales. L'instruction `purge` détruit une variable ou un ensemble de variables globales.

$$1 \mid \text{'nom global'} \quad \xrightarrow{\text{purge}} \mid$$

$$1 \mid \{ \text{'nom global}_1' \dots \text{'nom global}_n' \} \quad \xrightarrow{\text{purge}} \mid$$

L'instruction `purge` ne fonctionne que sur des variables globales. Une exécution de cette instruction sur une variable locale provoque une erreur.

L'instruction `clusr` permet quant à elle d'effacer toutes les variables globales du processus courant. Elle ne prend aucun argument.

14.5 Verrouillage

Les variables globales peuvent être verrouillées pour éviter toute modification non intentionnelle. Ce verrouillage n'est pas définitif et toute variable verrouillée peut être déverrouillée. En revanche, toute tentative d'accès en lecture à une variable verrouillée provoquera sur une erreur d'accès. Les instructions de verrouillage et de déverrouillage des variables sont `protect`, `unprotect`, `parameter` et `variable`.

Les deux instructions `protect` et `parameter` verrouillent une variable ou un groupe de variables, alors que `unprotect` et `variable` déverrouillent une variable ou un groupe de variables. Il existe la même différence entre `protect` et `parameter`, `unprotect` et `variable` qu'entre `sto` et `save`. En effet, seules les instructions `parameter` et `variable` garantissent un verrouillage ou un déverrouillage d'une variable globale.

$$\begin{array}{c}
1 \mid \begin{array}{c} \xrightarrow{\hspace{1cm}} \\ \text{'nom global' } \text{parameter ou variable} \end{array} \mid \\
1 \mid \begin{array}{c} \xrightarrow{\hspace{1cm}} \\ \{ \text{'nom global}_1' \dots \text{'nom global}_n' \} \text{ parameter ou variable} \end{array} \mid \\
1 \mid \begin{array}{c} \xrightarrow{\hspace{1cm}} \\ \text{'nom global' } \text{protect ou unprotect} \end{array} \mid \\
1 \mid \begin{array}{c} \xrightarrow{\hspace{1cm}} \\ \{ \text{'nom global}_1' \dots \text{'nom global}_n' \} \text{ protect ou unprotect} \end{array} \mid
\end{array}$$

Les variables globales ne sont pas partagées entre différents processus mais un processus fils, quel que soit son type, hérite de toutes les variables de son père et de leurs caractéristiques au moment de sa création. En particulier, une variable verrouillée dans un processus au moment où celui-ci lance un processus fils reste verrouillée dans le processus fils. Elle peut par la suite être déverrouillée de façon indépendante dans les deux processus.

15

Variables locales

Une variable locale est une variable de niveau strictement supérieur à 1. Elle est accessible du bloc de programme, définition ou partie de définition, qui la définit et est par défaut volatile, donc détruite à la sortie de ce bloc. Contrairement aux variables globales, elle peut être statique ou partagée.

15.1 Création

Une variable locale est créée par l'instruction « -> » associée à une expression, quelle soit en notation algébrique ou en notation polonaise inversée, ou par une instruction de boucle avec compteur. Ainsi, les expressions suivantes définissent toutes les trois une variable locale X susceptible de masquer une autre variable X de plus bas niveau :

-> X << 1 X + >>

-> X 'X+1'

1 10 for X next

Le détail du fonctionnement de l'instruction `for` se trouve au chapitre traitant des boucles. La seule chose à retenir ici est que l'instruction `for` crée toujours une variable locale contenant le compteur de boucle dont la portée se limite à cette boucle.

L'instruction `->` ne peut quant à elle se concevoir seule. Par défaut, elle crée des variables locales volatiles et fait partie d'un groupe formé par `->`, par l'expression suivante — que celle-ci soit en notation algébrique ou en notation polonaise inversée —, et par l'ensemble des atomes compris entre `->` et cette expression. Tous ces atomes doivent être des noms et reconnus en tant que tels car ils serviront de noms aux variables locales créées. La portée des variables se limite à l'expression. Plusieurs variables locales peuvent être créées simultanément.

$2n$	objet ₁	
$\vdots$	$\vdots$	
$n+1$	objet _n	
n	'nom local ₁ '	
$\vdots$	$\vdots$	
2	'nom local _{n-1} '	→
1	'nom local _n '	->

CHAPITRE 15. VARIABLES LOCALES

L'instruction `->` crée des variables statiques si elle est précédée de l'instruction `static`. Le mot-clef `static` peut apparaître n'importe où avant l'instruction `->`.

```

0001 MAIN
0002 <<
0003     FONCTION disp
0004     FONCTION disp
0005 >>
0006
0007 FONCTION
0008 <<
0009     static
0010     0 -> I
0011     <<
0012     I dup 'I' incr
0013     >>
0014 >>

```

Si la variable statique n'existe pas, elle est créée et initialisée à la valeur indiquée. Si elle pré-existe, la valeur indiquée, ici zéro, est silencieusement ignorée. De la même manière, l'instruction `->` permet la création de variables partagée si elle est précédée de l'instruction `shared`. Le mot-clef `shared` peut apparaître n'importe où avant l'instruction `->`. Comme le montre l'exemple suivant, une variable partagée est une variable statique partagée entre plusieurs processus légers.

```

0001 MAIN
0002 <<
0003     'FONCTION' spawn wfproc
0004     'FONCTION' spawn wfproc
0005 >>
0006
0007 FONCTION
0008 <<
0009     shared
0010     0
0011     -> I
0012     <<
0013     I disp 'I' incr
0014     >>
0015 >>

```

L'instruction `spawnspawn` crée un processus léger. La variable n'est pas perdue d'un processus à l'autre comme le montre le résultat de l'exécution du programme précédent.

+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)

+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël

0

1

En revanche, elle ne peut être partagée entre deux processus.

```

0001 MAIN
0002 <<
0003     'FONCTION' detach wfproc
0004     'FONCTION' detach wfproc
0005 >>
0006
0007 FONCTION
0008 <<
0009     shared

```

```

0010      0
0011      -> I
0012      <<
0013      I disp 'I' incr
0014      >>
0015 >>

```

car l'instruction `detach` crée des processus détachés qui ne partagent pas le même plan de mémoire comme le montre le résultat de l'exécution :

```

+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
0
0

```

Dans ce dernier cas, la variable déclarée comme partagée devient une simple variable statique déclarée dans chaque processus comme le montre l'exemple suivant :

```

0001 MAIN
0002 <<
0003      'FONCTION' detach wfproc
0004      'FONCTION' detach wfproc
0005 >>
0006
0007 FONCTION
0008 <<
0009      FONCTION2
0010      FONCTION2
0011 >>
0012
0013 FONCTION2
0014 <<
0015      shared
0016      0
0017      -> I
0018      <<
0019      I disp 'I' incr
0020      >>
0021 >>

```

qui renvoie :

```

+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
0
1
0
1

```

Néanmoins, toute variable déclarée comme `shared` reste partageable entre plusieurs processus légers et gérée comme telle indépendamment de son utilisation. En particulier, ce n'est pas parce que le contexte de sa création la transforme en variable statique qu'elle est gérée comme une variable statique. Elle reste partagée et soumise à des mécanismes d'accès concurrents.

15.2 Portée et visibilité

Toute variable locale, quelle soit volatile, statique ou partagée est créée au début d'une expression ou d'une boucle avec compteur. En l'absence de toute autre indication, ces variables sont toutes volatiles et leur portée se limite à la fin de l'expression courante ou de la boucle avec compteur. Dans le cas d'une boucle avec compteur, la variable contenant l'indice de boucle est toujours une variable locale volatile. Il n'existe aucun mécanisme de libération des variables locales, celles-ci étant automatiquement détruites à la fin du bloc courant.

Les variables locales statiques ou partagées ne peuvent être créées qu'au début d'une expression et non par une structure de boucle avec compteur. Leur portée est celle du processus courant et, à l'instar de toutes autres variables, elles sont disponibles dans les processus fils. En revanche, leur visibilité est restreinte à l'expression dans laquelle elles ont été créées et elles ne sont pas détruites à la fin de cette expression mais masquées et non accessibles jusqu'à une nouvelle utilisation dans la même structure de programme. Les variables statiques et partagées sont définies de manière unique et non ambiguë par leur nom et par leur position de création dans un programme. Le programme suivant définit donc deux variables statiques portant le même nom mais indépendantes

```

0001 MAIN
0002 <<
0003     1 2 start
0004         FONCTION1
0005         FONCTION2
0006     next
0007 >>
0008
0009 FONCTION1
0010 <<
0011     static
0012     1
0013     -> I
0014     <<
0015         "F1 " I ->str + disp 'I' incr
0016     >>
0017 >>
0018
0019 FONCTION2
0020 <<
0021     static
0022     5
0023     -> I
0024     <<
0025         "F2 " I ->str + disp 'I' incr
0026     >>
0027 >>

```

ce qui est montré par le résultat de son exécution

```

+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
F1 1
F2 5
F1 2
F2 6

```

Un effet de bord de l'association de la position de création d'une variable statique ou partagée et de son nom est l'impossibilité de définir une telle variable dans une expression

qui n'est pas une partie fixe d'une définition utilisateur comme le montre l'exemple suivant

```
0001 MAIN
0002 <<
0003   << static 1 -> I << I disp 'I' incr >> >> 'FONCTION' sto
0004   FONCTION eval
0005   FONCTION eval
0006
0007   << static 1 -> I << I disp 'I' incr >> >> dup
0008   eval
0009   eval
0010 >>
```

dont l'évaluation donne

```
+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
1
1
1
1
```

En effet, les deux expressions définies aux lignes 3 et 7 de ce programme ne sont pas évaluées directement mais empilées car il s'agit d'un objet et non d'une expression à évaluer qui serait précédée d'une instruction de création de variable locale `->`. Il n'y a aucune raison que l'évaluation de l'expression définie ligne 3 se fasse deux fois à la même adresse. En revanche, la définition de la ligne 7 est évaluée deux fois à la même adresse car elle est dupliquée par l'instruction `dup` et il semblerait logique que la variable statique ne soit pas perdue d'une évaluation à l'autre. Pour que le comportement soit comparable au comportement de l'expression ligne 3 et parce que rien ne garantit que l'évaluation d'une telle expression se fasse toujours à la même adresse, les variables déclarées comme statiques ou partagées dans des expressions mobiles sont silencieusement détruites à la fin de celles-ci. Cela évite aussi les fuites de mémoire lors de l'évaluation successive d'expressions mobiles contenant des variables statiques ou partagées et un fonctionnement imprévu lorsque deux expressions différentes sont évaluées à la même adresse avec des noms de variables statiques ou partagée indéniables.

Par défaut, la portée des variables statiques et partagées est celle du processus courant pour une variable statique et du groupe de processus légers concurrents pour une variable partagée. Il est possible n'en restreindre les portées grâce aux instructions `private` et `volatile`. L'instruction `private` transforme immédiatement une variable partagée en variable privée volatile tandis que `volatile` transforme immédiatement une variable statique en variable volatile. La portée de la variable statique ou partagée est alors réduite à sa visibilité et la variable est détruite à la fin de l'expression courante comme le montre le programme suivant.

```
0001 MAIN
0002 <<
0003   1 4 start FONCTION next
0004 >>
0005
0006 FONCTION
0007 <<
0008   static
0009   1 -> I
0010   <<
0011     I disp 'I' incr
0012     if I 2 > then 'I' volatile end
0013   >>
```

CHAPITRE 15. VARIABLES LOCALES

0014 >>

Lorsque la valeur de la variable statique `I` devient strictement supérieure à 2, cette variable est transformée immédiatement ligne 12 en une variable volatile qui est détruite à la ligne 13. L'itération suivante crée une nouvelle variable statique et l'initialise à sa valeur par défaut.

```
+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
1
2
1
2
```

15.3 Modification

La modification d'une variable locale peut se faire soit par lecture puis enregistrement, soit par des opérations arithmétiques directes détaillées au chapitre suivant.

Les variables locales sont directement accessibles en lecture au travers de leur nom. Ce nom peut être évaluable ou symbolique (entouré par les délimiteurs du type nom). Dans le cas où ce nom est évaluable, il est directement évalué pour renvoyer le contenu de la variable. S'il est symbolique, il convient de forcer l'évaluation de la variable par la commande `eval` ou `->num`. L'instruction `sto` permet ensuite d'enregistrer le contenu d'une variable locale. S'il existe plusieurs variables de même nom, la variable effectivement modifiée est celle de plus haut niveau.

2	objet	→
1	'nom local'	sto

S'il n'existe aucune variable locale accessible, l'instruction `sto` se comporte comme `save` et crée une variable globale. En aucun cas, l'instruction `sto` permet la création d'une variable locale.

16

Arithmétique directe

Il est possible de travailler sur le contenu des variables en les évaluant puis, après modification, en les sauvegardant à nouveau dans la même variable. Pour un certain nombre d'opérations simples, il est aussi possible de recourir à des instructions arithmétiques portant directement sur les variables, que celles-ci soient locales ou globales.

D'autres instructions peuvent prendre des variables comme arguments, mais seules sont détaillées dans ce chapitre les instructions arithmétiques qui portent exclusivement sur des variables.

16.1 Les quatre opérations

Les quatre opérations arithmétiques que sont les addition, soustraction, multiplication et division peuvent être traitées par les instructions d'arithmétique directe dès que leurs arguments sont des scalaires, des vecteurs ou des matrices.

16.1.1 Addition

La définitions intrinsèque `sto+` permet d'ajouter une quantité à une variable. Les deux syntaxes d'appel suivantes sont identiques car l'addition est commutative pour tout objet.

2	objet	→	
1	'nom'	<code>sto+</code>	

2	'nom'	→	
1	objet	<code>sto+</code>	

Au sortir de l'exécution de `sto+`, le contenu de la variable est son contenu initial auquel s'est ajouté l'objet présent au niveau 1 ou 2 de la pile.

16.1.2 Soustraction

L'arithmétique directe permise par l'instruction `sto-` n'est pas commutative. Il faut ainsi distinguer l'écriture

2	objet	→	
1	'nom'	<code>sto-</code>	

CHAPITRE 16. ARITHMÉTIQUE DIRECTE

de

$$\begin{array}{c|c|c} 2 & \text{'nom'} & \longrightarrow \\ 1 & \text{objet} & \text{sto-} \end{array}$$

car la première écriture soustrait le contenu de la variable 'nom' à l'objet au niveau 2 avant de remettre le résultat dans la variable 'nom', alors que la seconde écriture soustrait l'objet au niveau 1 à la variable 'nom'. Cette distinction est valable pour toutes les opérations d'arithmétique directe non commutatives.

16.1.3 Multiplication

L'instruction `sto*` permet d'effectuer une multiplication entre un objet et une variable.

$$\begin{array}{c|c|c} 2 & \text{objet} & \longrightarrow \\ 1 & \text{'nom'} & \text{sto*} \end{array}$$

$$\begin{array}{c|c|c} 2 & \text{'nom'} & \longrightarrow \\ 1 & \text{objet} & \text{sto*} \end{array}$$

L'opération de multiplication n'est pas commutative pour les objets de type vecteur et matrice. L'ordre des arguments de l'instruction `sto+` est dicté par les mêmes règles que celles énoncées pour la soustraction directe.

$$\begin{array}{c|c|c} 2 & \text{objet} & \longrightarrow \\ 1 & \text{'nom'} & \text{sto/} \end{array}$$

$$\begin{array}{c|c|c} 2 & \text{'nom'} & \longrightarrow \\ 1 & \text{objet} & \text{sto/} \end{array}$$

16.1.4 Division

La définition intrinsèque `sto/` effectue la division entre un objet et le contenu d'une variable — ou l'opération inverse selon l'ordre d'apparition des arguments dans la pile — et enregistre le résultat dans la variable. La règle dictant l'ordre des arguments est similaire à celle énoncée dans le cas de la soustraction directe.

16.2 Autres opérations

Les autres opérations d'arithmétique directe se font sans limitation de type par rapport aux opérations atomiques sous-jacentes. Elles ont trait aux inversions, conjugaisons, opposition.

16.2.1 Inversion

L'instruction `sinv` calcule l'inverse du contenu de la variable 'nom'. L'objet contenu dans la variable doit être un objet sur lequel il est licite de calculer un inverse. Si cela n'est pas le cas, l'opération retourne une erreur.

$$\begin{array}{c|c|c} 1 & \text{'nom'} & \longrightarrow \\ & & \text{sinv} \end{array}$$

16.2. AUTRES OPÉRATIONS

16.2.2 Opposition

L'instruction `sneg` transforme un objet contenu dans une variable référencée par '`nom`' en son opposé. Cet objet doit être un objet sur lequel l'opposition possède un sens. Dans le cas contraire, une erreur est renvoyée.

$$1 \left| \begin{array}{cc} & \longrightarrow \\ \text{'nom'} & \text{sneg} \end{array} \right|$$
16.2.3 Conjugaison

L'instruction `sconj` calcule le conjugué de l'objet contenu dans la variable '`nom`'. Si ce calcul ne peut être rendu à son terme, une erreur est retournée.

$$1 \left| \begin{array}{cc} & \longrightarrow \\ \text{'nom'} & \text{sconj} \end{array} \right|$$

Quatrième partie

Contrôle

17

Conditions et tests

Les différentes commandes de contrôle peuvent être combinées en un grand nombre de structure de décision et de traitement d'erreur. Certaines structures comme celles qui sont formées autour des instructions `if` ou `iferr` sont exclusives — la première clause vraie de la structure est exécutée, toutes les suivantes, même vraies, sont ignorées — alors que d'autres, formées autour de `select` sont inclusives — un nombre indéterminé de clauses allant de zéro au nombre total de clauses peut être exécuté, toutes les clauses étant testées les unes après les autres.

17.1 Tests simples

17.1.1 If...then...(else)...end

L'instruction `else` étant optionnelle, il existe deux formes de cette commande :

- `if clause test then clause vraie end;`
- `if clause test then clause vraie else clause fausse end.`

La commande `then` prend un indicateur binaire dans la pile. Cet indicateur doit être un scalaire. S'il est non nul, la clause de test est considérée comme vraie et la clause vraie est évaluée, l'exécution normale se poursuivant après l'instruction `end`. Dans le cas contraire — indicateur nul —, le séquenceur saute à l'instruction `else` si celle-ci existe ou à l'instruction `end` correspondante.

```

0001 <<
0002   0 -> X
0003   <<
0004       if
0005           X 0 >
0006       then
0007           "X est strictement positif." disp
0008       end
0009
0010       if
0011           X 0 >
0012       then
0013           "X est strictement positif." disp
0014       else
0015           "X est négatif ou nul." disp
0016       end

```

CHAPITRE 17. CONDITIONS ET TESTS

```
0017    >>
0018 >>
```

L'instruction `if` crée un bloc de programme. Il est possible de les imbriquer sans perte de cohérence.

```
0001 <<
0002     0 -> X
0003 <<
0004     if
0005         X 0 >
0006     then
0007         "X est strictement positif." disp
0008     else
0009         if
0010             X 0 <
0011         then
0012             "X est négatif." disp
0013         else
0014             "X est nul." disp
0015         end
0016     end
0017 >>
0018 >>
```

Ce dernier exemple peut être écrit de façon plus synthétique en utilisation l'instruction `elseif` détaillée plus loin.

17.1.2 Ift

Cette commande est une forme de structure `if/then/end` à une seule commande. `ift` prend un indicateur binaire dans la pile, au niveau 2 puis s'il est non nul, évalue l'objet au niveau 1. Dans le cas contraire, cet objet est supprimé.

2	Indicateur binaire	→
1	objet	<code>ift</code>

Cette instruction peut être utilisée dans une expression algébrique sous la forme

```
'ift (expression test, expression vraie) '
```

17.1.3 Ifte

L'instruction `ifte` est une forme de `if/then/else/end` à une seule commande. Elle prend un indicateur binaire au niveau 3 de la pile et deux objets aux niveaux 1 et 2. Si l'indicateur est vrai, l'objet de niveau 2 est évalué et celui de niveau 1 supprimé. S'il est faux, l'objet de niveau 1 est évalué et celui de niveau 2 supprimé.

3	Indicateur binaire	
2	Objet vrai	→
1	Objet faux	<code>ifte</code>

Cette instruction est aussi utilisable dans les expression algébriques grâce à sa variante

```
'ifte (expression test, expression vraie, expression fausse) '
```

17.2 Reprise sur erreur

Le mécanisme de reprise sur erreur du RPL/2 est aussi puissant qu'il est particulier. En effet, ce mécanisme distingue plusieurs types d'erreur, certaines récupérables, d'autres non.

17.2.1 Types d'erreurs

17.2.1.1 Erreurs système

Les erreurs systèmes proviennent pour leur plus grand nombre du système d'exploitation. Ces erreurs ne sont pas imputables directement au programme exécuté par le séquenceur et ne sont donc pas récupérables. Ce type d'erreur provoque toujours un arrêt du programme. Toute erreur système est indiquée au processus racine primaire par un signal spécifique. Le processus racine primaire est alors interrompu et envoie une requête d'arrêt récursive à tous ses fils.

La gestion des erreurs système ne tient pas compte des racines secondaires créées par l'instruction `nrproc` car les erreurs systèmes ne proviennent pas directement du programme courant.

17.2.1.2 Erreurs d'exécution

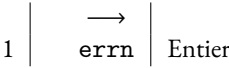
Une erreur d'exécution est une erreur intrinsèque au programme exécuté. Elle peut être récupérée si elle se trouve dans une structure `iferr/then/end`. Dans le cas contraire, elle provoque un arrêt du programme, suivi le cas échéant de la création d'une image du processus fautif.

17.2.1.3 Exceptions mathématiques

Les exceptions mathématiques sont dues principalement à des dépassements de capacité. Elles se distinguent des erreurs d'exécutions, car elles peuvent être masquées par un indicateur système. Ainsi, une division par zéro générera une exception si la séquenceur travaille sur le corps de réels et un infini ou une indétermination s'il travaille sur la droite achevée¹.

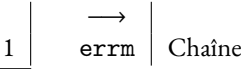
17.2.2 Errn

L'instruction `errn` ne prend aucun argument et renvoie un entier correspondant au numéro d'erreur identifiant la dernière erreur traitée. Dans le cas où aucune erreur n'a été traité précédemment ou que l'information a été effacée par l'instruction `clerr`, elle renvoie une valeur nulle.



17.2.3 Errm

L'instruction `errm` renvoie une chaîne de caractères correspondant à la dernière erreur traitée. Si aucune erreur n'a été traitée précédemment ou que cette information a été effacée par `clerr`, elle renvoie une chaîne vide. La chaîne de caractère est renvoyée dans la localisation courante.



¹cf. indicateur 59

CHAPITRE 17. CONDITIONS ET TESTS

17.2.3.1 Exemple

```
cauchy:[~] > repl -is
+++RPL/2 (R) version 4.0.10 (samedi 06/02/2010, 19:48:18 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël

+++Ce logiciel est un logiciel libre sans aucune garantie de fonctionnement.
+++Pour plus de détails, utilisez la commande 'warranty'.

RPL/2> iferr 0 inv then end errm disp
+++Exception : Division par zéro
RPL/2> abort
cauchy:[~] > LANG=C repl -is
+++RPL/2 (R) version 4.0.10 (Saturday 02/06/10, 19:48:18 CET)
+++Copyright (C) 1989 to 2009, 2010 BERTRAND Joël

+++This is a free software with absolutely no warranty.
+++For details, type 'warranty'.

RPL/2> iferr 0 inv then end errm disp
+++Exception : Division by zero
RPL/2> abort
cauchy:[~] >
```

17.2.4 Clerr

L'instruction `clerr` efface les indications concernant les dernières erreurs. En particulier, `errn` et `errm` renvoient respectivement une valeur nulle et une chaîne vide après tout appel à `clerr` jusqu'à ce qu'une nouvelle erreur soit traitée.

17.2.5 Iferr...then...(else)...end

La syntaxe de cette structure de contrôle est similaire à celle utilisée pour les tests simples.

```
iferr clause piège then clause erreur end
iferr clause piège then clause erreur else clause normale end
```

L'évaluation de la clause piège se fait normalement jusqu'à ce que survienne soit une erreur récupérable, soit l'instruction `then` signalant sa fin. Dans le cas d'une erreur, la clause erreur est évaluée. S'il n'y a pas eu d'erreur et si la clause normale existe, elle est évaluée. Dans tous les cas, le programme continue après l'instruction `end`. Aucune erreur n'est reprise par défaut dans les clauses normale ou erreur.

Le programme suivant illustre l'utilisation des blocs de reprises sur erreur en calculant et affichant la factorielle de N . Ce programme n'est qu'un exemple académique et particulièrement peu performant de l'utilisation de ces blocs (nombreux changements de contextes, deux boucles...) mais il en montre bien le fonctionnement.

```
0001 FACTORIELLE
0002 <<
0003   6 -> N
0004   <<
0005       pshcntxt
0006
0007       2 N for I I next
0008
0009       do
0010           dupcntxt
0011       until
0012           iferr
```



```

0013          *
0014      then
0015          pulcnttxt true
0016      else
0017          dropcnttxt false
0018      end
0019  end
0020
0021      disp
0022      pulcnttxt
0023  >>
0024  >>

```

Ligne 5, l'instruction `pshcnttxt` crée un nouveau contexte de pile. La pile opérationnelle est sauvegardée et une nouvelle pile vide la remplace. La boucle ligne 7 empile les entiers successifs compris entre 2 et N , montrant au passage qu'il faudrait ajouter un test au programme pour être sûr que l'argument soit supérieur ou égal à 2. La boucle comprise entre les lignes 9 et 19 est effectuée tant que programme ne rencontre aucun erreur donc que la multiplication de la ligne 13 est possible. En l'absence d'erreur, le contexte sauvegardé est silencieusement détruit à la ligne 17 et le programme empile le drapeau faux qui sera traité par l'instruction de fin de boucle de la ligne 19. Lorsque cette multiplication échoue parce que la pile ne contient plus qu'un seul objet, le contexte courant est détruit par la restauration de l'ancienne pile à la ligne 15 et le programme sort de la boucle. Le résultat de l'exécution de ce programme est alors :

```

+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
720

```

Il ne faut pas perdre de vue lors de l'écriture des clauses d'erreur que l'état de la pile après une erreur peut dépendre du fait que la pile *last* soit validée ou non. Si la pile *last* est validée, les commandes donnant une erreur renvoient leurs arguments dans la pile. Dans le cas contraire, ces arguments sont supprimés. Un moyen de contourner ce problème est d'utiliser les instructions de gestion des contextes d'exécution. Une autre façon de s'affranchir du problème est de forcer l'utilisation de la pile *last* au travers de l'indicateur 31. Cependant, la gestion de la pile *last* peut s'avérer coûteuse en terme de temps de calcul car elle requière la duplication de tous les atomes des objets passés en argument à toutes les instructions.

17.3 Tests multiples

17.3.1 If...then...elseif...then...(else)...end

Cette structure de contrôle permet d'imbriquer plusieurs tests mutuellement exclusifs — contrairement à la structure *case* — et pouvant porter sur des conditions différentes. Il est possible de lancer une action par défaut lorsque toutes les conditions examinées sont fausses grâce au mot clef optionnel `else`. Seule la première clause test vraie engendre l'évaluation de la clause vraie correspondante puis un saut à l'instruction `end` cloturant la structure.

```

if clause test 1 then clause vraie 1
elseif clause test 2 then clause vraie 2
...
elseif clause test n then clause vraie n
else clause défaut
end

```

Le programme de test de la valeur de *X* montrant des structures de test simple imbriquées *if/then/(else/)* *end* peut s'écrire simplement sous la forme :

```

0001 <<
0002     0 -> X
0003     <<
0004         if
0005             X 0 >
0006         then
0007             "X est strictement positif." disp
0008         elseif
0009             X 0 <
0010         then
0011             "X est négatif." disp
0012         else
0013             "X est nul." disp
0014         end
0015     >>
0016 >>

```

Les différentes clauses test étant mutuellement exclusives, elles ne sont évaluées que jusqu'à trouver la première clause test vraie. Toutes les clauses test suivantes sont ignorées et non évaluées. Il est donc plus intéressant d'utiliser une structure contenant *elseif* qu'une structure à base de *select* et de *case* dans le cas où les différentes conditions de test sont mutuellement exclusives. D'un autre côté, il convient de porter une grande attention aux différentes clauses d'une structure contenant *elseif* car l'ordre des différentes clauses influe directement sur le résultat.

17.3.2 Select...case...then...end...(default)...end

La structure *select/case* permet de contrôler l'exécution d'une séquence d'instructions dépendant d'un paramètre unique. Les tests ainsi écrits ne sont pas mutuellement exclusifs.

```

select
    objet
    case clause test 1 then clause vraie 1 end
    case clause test 2 then clause vraie 2 end
    ...
    case clause test n then clause vraie n end
default
    clause défaut
end

```

Le mot clef *select* crée la structure de contrôle allant jusqu'à l'instruction *end* correspondant. Cette structure peut contenir une instruction optionnelle *default* qui ne peut apparaître qu'après toutes les structures *case/then/end* incluses. L'instruction *default* correspond à une clause spéciale détaillée plus bas.

La première instruction *case* rencontrée es dans la structure de contrôle est spéciale et prend un objet quelconque en argument. Cet objet est évalué par chaque instruction *case* avant l'exécution de la clause test. Hormi cette spécificité, toutes les structures *case/then/end* fonctionnent de la même façon. L'instruction *case* évalue l'objet qui sera utilisé dans la clause test. Si le résultat de cette évaluation est vrai, la clause vraie correspondante est exécutée. Dans tous les cas, le programme continue par l'évaluation de la structure *case/then/end* suivante. Si la clause spéciale *default* existe, deux cas se présentent :

- au moins une clause test était vraie et le programme ignore la clause `default` pour continuer après l'instruction `end` correspondante;
- aucune clause test n'était vraie et le programme évalue la clause défaut.

```

0001 MAIN
0002 <<
0003     4
0004     -> X
0005     <<
0006         select
0007             X
0008             case 0 >= then "X est positif ou nul." disp end
0009             case 1 > then "X est strictement supérieur à 1." disp end
0010             case -1 < then "X est strictement inférieur à -1." disp end
0011         default
0012             "X est supérieur ou égal à -1 et strictement négatif." disp
0013         end
0014     >>
0015 >>

```

L'exemple précédent montre que l'ensemble des clauses test sont évaluées dans leur ordre d'apparition et que les différents tests ne sont pas mutuellement exclusifs.

```

+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
X est positif ou nul.
X est strictement supérieur à 1.

```


18

Boucles

Le langage RPL/2 prévoit un certain nombre de structures de boucles différentes. Lorsque le nombre d'itérations à effectuer dans la boucle est connu *a priori*, les boucles sont dites définies. Dans tous les autres cas, les boucles sont dites indéfinies.

L'instruction `exit` rencontré dans une boucle force le programme à quitter la boucle courante et à reprendre une exécution normale à la fin de celle-ci. Elle est valable quelle que soit le type de boucle.

18.1 Boucles définies

Certains langages considèrent comme obligatoire la présence d'une variable de boucle. D'autres ne permettent d'effectuer des boucles que sur des entiers. Le RPL/2 s'affranchit de ces deux limites.

En revanche, il est impossible de connaître *a priori* le sens de variation d'une boucle, celui-ci étant donné par l'instruction de cloture de la structure et pouvant même être variable. Toute boucle définie est ainsi au moins effectuée une fois comme le montre l'exemple suivant :

```
0001 MAIN
0002 <<
0003   1 0 for I
0004       I disp
0005   next
0006 >>
```

dont la sortie est

```
+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
1
```

Une boucle définie avec compteur débute à la ligne 3. La variable locale `I`, créée par cette boucle, est initialisée à la valeur 1. Le contenu de la boucle est effectuée jusqu'à trouver l'instruction `next` incrémentant la variable `I` qui vaut maintenant 2. L'instruction `next` impose un sens de variation croissant de la variable de boucle. La valeur courante de la variable étant strictement supérieure à la limite fixée ligne 3, en l'occurrence à zéro, le programme continue son exécution normale après l'instruction `next`. Pour corriger ce problème, il faut soit tester les limites et le sens de variation avant le début de la boucle en utilisant une structure conditionnelle, soit utiliser l'instruction `cycle` dans le corps de la boucle. Il faut noter que le comportement de l'instruction `cycle` diffère selon le type de boucle. En effet, dans

CHAPITRE 18. BOUCLES

une structure `for/next`, `cycle` se comporte comme `next`, alors que dans une structure `for/step`, elle se comporte comme `step` et prend ainsi un argument entier ou réel dans la pile.

18.1.1 Boucle sans compteur**18.1.1.1 Start...(cycle)...next**

Cette boucle est la boucle la plus simple qu'il est possible d'imaginer. Elle ne comporte aucun indice de boucle.

début fin start clause boucle next

La commande `start` prend deux nombres entiers ou réels, *début* et *fin*, dans la pile et les utilise comme valeurs de début et fin d'un compteur de boucle. L'exécution se poursuit par l'évaluation de la séquence d'instructions *clause boucle*. La commande `next` incrémente le compteur de boucle d'une unité. Si ce compteur est inférieur ou égal à la valeur *fin*, la *clause boucle* est à nouveau évaluée. Ce processus se poursuit jusqu'à ce que le compteur devienne strictement supérieur à la valeur *fin*. L'exécution continue alors normalement après l'instruction `next`.

L'instruction `cycle` permet d'abandonner l'itération courante et de boucler immédiatement sur l'itération suivante.

18.1.1.2 Start...(cycle)...step

Cette structure est similaire à la structure précédente au détail près que l'instruction `step` incrémente le compteur de boucle d'une quantité variable prise dans la pile alors que `next` l'incrémente toujours d'une unité.

début fin start clause boucle pas step

Si le *pas* est positif et si le compteur de boucle est inférieur ou égal à la valeur *fin*, la *clause boucle* est à nouveau évaluée, ceci se poursuivant jusqu'à ce que le compteur de boucle dépasse *fin*, à la suite de quoi l'exécution continue normalement après l'instruction `step`.

Si le *pas* est négatif et si le compteur de boucle est supérieur ou égal à *fin*, la *clause boucle* est à nouveau évaluée, ceci se poursuivant jusqu'à ce que le compteur de boucle soit inférieur à *fin*, à la suite de quoi l'exécution continue normalement après l'instruction `step`.

À noter : la grandeur *pas* est prise dans la pile et peut ainsi résulter de l'évaluation d'une expression quelconque. En particulier, il est possible d'avoir un *pas* tantôt positif, tantôt négatif lors de l'exécution d'une même boucle. Dans ce cas, le processus de contrôle alternera les tests de fins de boucle en fonction du signe de l'incrément courant.

À noter : l'instruction `cycle` prend les mêmes arguments que l'instruction de la fin de la boucle courante. En particulier, si la boucle s'achève sur une instruction `next`, `cycle` ne prend aucun argument. Si la boucle s'achève sur `step`, `cycle` attend un argument qui a la même signification que celui de `step`. L'instruction `cycle` effectue le test de fin de boucle avant et ne boucle sur l'itération suivante que si la fin de la boucle n'est pas atteinte.

18.1. BOUCLES DÉFINIES

18.1.2 Boucle avec compteur

Ce type de boucle utilise un compteur accessible directement par une variable locale. Cette variable peut être modifiée comme toute autre variable, aucune protection n'assurant son intégrité.

À noter : l'instruction `for` créant une variable locale, il est possible d'utiliser comme compteur, une variable utilisée par ailleurs, cette variable n'étant pas écrasée, mais simplement masquée. À la sortie de la boucle, le compteur est perdu et la variable masquée retrouve la valeur qu'elle avait avant l'initialisation de la boucle.

Effet de bord : il est tout à fait possible d'imbriquer plusieurs boucles utilisant le même compteur. Cela ne provoque aucun dysfonctionnement, chaque compteur étant masqué par un autre. Seul le compteur courant sera accessible à un instant donné.

18.1.2.1 For...(cycle)...next

Cette structure est une boucle définie dans laquelle le compteur de boucle *nom* est une variable locale qui peut être évaluée dans la boucle.

début fin for nom clause boucle next

En séquence :

- `for` prend deux nombres, entiers ou réels, *début* et *fin* dans la pile et crée une variable locale *nom* initialisée à la valeur *début*;
- la séquence d'objets *clause boucle* est évaluée. Si *nom* est évalué dans cette séquence, il donne la valeur en cours du compteur de boucle;
- `next` incrémente de compteur de boucle d'une unité. Si la valeur du compteur dépasse *fin*, l'exécution se poursuit après `next`, la variable locale *nom* étant alors supprimée. Dans le cas contraire, l'étape précédente se répète.

18.1.2.2 For...(cycle)...step

À l'instar de la structure `for/next`, cette boucle possède un compteur. La différence réside dans le fait que l'incrément du compteur peut être quelconque.

début fin for nom clause boucle pas step

En séquence :

- `for` prend deux nombres, entiers ou réels, *début* et *fin* dans la pile et crée une variable locale *nom* initialisée à la valeur *début*;
- la séquence d'objets *clause boucle* est évaluée. Si *nom* est évalué dans cette séquence, il donne la valeur en cours du compteur de boucle;
- `step` prend un argument *pas* — entier ou réel — dans la pile et incrémente le compteur de boucle de la quantité représentée par *pas*. Si la valeur du compteur est supérieure à *fin* pour un incrément positif, ou inférieure à *fin* pour un incrément négatif, l'exécution se poursuit après `step`, la variable locale *nom* étant alors supprimée. Dans le cas contraire, l'étape précédente se répète.

CHAPITRE 18. BOUCLES

18.1.3 Exemples

Le programme suivant illustre le fonctionnement des fonctions `cycle` et `exit` dans des boucles.

```
0001 MAIN
0002 <<
0003     1 10 for I
0004         if I 5 same then cycle end
0005         if I 7 > then exit end
0006
0007         I disp
0008     next
0009
0010     "FIN" disp
0011 >>
```

Une boucle définie avec compteur est déclarée à la ligne 3. Cette boucle crée une variable locale `I` contenant le compteur de boucle. Le pas de la boucle, imposé par l'instruction `next` de la ligne 8, est unitaire. Dans le corps de la boucle se trouvent deux structures de test. La première, ligne 4, interrompt la boucle en la faisant passer à l'itération suivante. La seconde, ligne 5, force un le programme à quitter la boucle.

```
+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
1
2
3
4
6
7
FIN
```

En remplaçant l'instruction `next` par une instruction `step`, il faut penser à rajouter un argument devant toutes les occurrences de l'instruction `cycle` portant sur cette boucle.

```
0001 MAIN
0002 <<
0003     1 10 for I
0004         // cycle prend maintenant un argument car elle renvoie à step
0005         if I 5 same then 1 cycle end
0006         if I 7 > then exit end
0007
0008         I disp
0009     1 step
0010
0011     "FIN" disp
0012 >>
```

18.2 Boucles indéfinies

18.2.1 While...repeat...end

Cette structure évalue itérativement une clause test et une clause boucle tant que la valeur renvoyée par la clause test est vraie (non nulle).

```
while clause test repeat clause boucle end
```


18.3. INSTRUCTION EXIT

Lorsque la clause test renvoie un indicateur binaire faux, la clause boucle est ignorée et l'instruction reprend normalement après l'instruction `end`.

18.2.2 Do...until...end

Cette structure évalue de manière répétée une clause boucle et une clause test jusqu'à ce que la valeur renvoyée par la clause test soit vraie (non nulle).

```
do clause boucle until clause test end
```

18.3 Instruction exit

L'instruction `exit` contraint le programme à quitter la boucle en cours d'exécution, quel que soit son type. Elle peut apparaître dans n'importe quelle structure de la boucle ou dans une fonction appelée depuis le corps de la boucle. L'exécution du programme continue normalement après l'instruction de cloture de la boucle.

19

Contrôle de l'exécution

19.1 Mode de fonctionnement

L'état du système est assujéti à un certain nombre d'indicateurs binaires. Tous ces indicateurs sont accessibles et modifiables soit au travers d'un certain nombre d'instructions (formats, mode angulaire...) soit directement au travers des instructions **cf** et **sf**.

19.1.1 Indicateurs

Le RPL/2 possède soixante-quatre indicateurs binaires — numéroté de 1 à 64 — dont la plupart sont laissés à la discrétion de l'utilisateur. Cependant, certains d'entre eux conditionnent le fonctionnement du séquenceur. La signification des indicateurs binaires et leurs valeurs par défaut se trouvent au tableau 19.1.

19.1.2 Manipulation

19.1.2.1 Affectations

Il existe deux instructions simples permettant d'affecter un état — armé ou désarmé — à un indicateur binaire :

- **cf** : cette instruction permet de désarmer un indicateur binaire. Elle prend un argument entier dans la pile correspondant au numéro de l'indicateur à désarmer.
- **sf** : le fonctionnement de cette instruction est en tout point comparable au fonctionnement de l'instruction **cf** si ce n'est qu'elle permet d'armer un indicateur.

$$1 \mid n \xrightarrow{\text{sf}} \quad \text{ou} \quad 1 \mid n \xrightarrow{\text{cf}} \mid$$

Il est possible de manipuler simultanément plusieurs indicateurs, voire la totalité, grâce aux instructions **rclf** et **stof**. L'instruction **rclf** renvoie un entier binaire de soixante-quatre bits représentant les états des soixante-quatre indicateurs binaires, l'indicateur binaire 1 correspondant au bit de plus faible de l'entier binaire. Réciproquement, l'instruction **stof** fait correspondre l'état des soixante-quatre indicateurs binaires avec les bits d'un entier binaire pris à la base de la pile. Un bit de valeur « 0 » désarme l'indicateur correspondant alors qu'un bit « 1 » l'arme. Le bit de poids le plus faible correspond à l'indicateur 1.

Ainsi, il est aisé de sauvegarder l'état du système par un appel à la fonction **rclf** puis de le rétablir ultérieurement grâce à **stof**.

Indicateurs	Signification	Valeurs par défaut
1 à 30	aucune signification	désarmés
31	pile <i>last</i> active	armé en mode interactif, désarmé sinon
32	impression automatique	désarmé
33	retour à la ligne automatique invalidé	désarmé
34	réservé	désarmé
35	évaluation symbolique des constantes	armé
36	évaluation symbolique des fonctions	armé
37 à 42	taille des entiers binaires, de 1 à 64, bit de poids faible en tête	armés
43 à 44	base de numérotation binaire	désarmés
45	affichage multiligne	armé
46 à 47	réservés	désarmés
48	virgule comme séparateur décimal	désarmé
49 à 50	format des nombres	désarmés
51	tonalité	désarmé
52	mise à jour automatique des graphiques désactivée	désactivé
53 à 56	nombre de chiffres significatifs, bit de poids faible en tête	désarmés
57 à 59	réservés	désarmés
60	radian comme unité d'angle pour les calculs en réels	armé
61 à 64	réservés	désarmés

TABLE 19.1 : Signification des indicateurs binaires

19.2. EXÉCUTION NORMALE

$$1 \mid \begin{array}{c} \longrightarrow \\ \text{rclf} \end{array} \mid \#n \text{ ou } 1 \mid \begin{array}{c} \longrightarrow \\ \text{stof} \end{array} \mid$$

À noter : la longueur des entiers binaires est soumise à un paramètre spécifié par la commande `stws`. La représentation interne de l'entier comporte toujours soixante-quatre bits, même si un certain nombre d'entre eux ne sont pas affichés.

19.1.2.2 Tests

Plusieurs instructions permettent de tester et de modifier si nécessaire les indicateurs binaires :

- `fc?` renvoie la valeur vraie si l'indicateur est désarmé et faux dans le cas contraire;
- `fs?` renvoie la valeur vraie si l'indicateur est armé et faux dans le cas contraire;
- `fc?c` renvoie la valeur vraie si l'indicateur est désarmé et faux dans le cas contraire. L'indicateur est désarmé dans le même temps;
- `fc?s` renvoie la valeur vraie si l'indicateur est désarmé et faux dans le cas contraire. L'indicateur est armé dans le même temps;
- `fs?c` renvoie la valeur vraie si l'indicateur est armé et faux dans le cas contraire. L'indicateur est désarmé dans le même temps;
- `fs?s` renvoie la valeur vraie si l'indicateur est armé et faux dans le cas contraire. L'indicateur est armé dans le même temps.

$$\begin{array}{ccc} 1 \mid \begin{array}{c} \longrightarrow \\ \text{fc?} \end{array} \mid \#n & 1 \mid \begin{array}{c} \longrightarrow \\ \text{fs?} \end{array} \mid \#n \\ 1 \mid \begin{array}{c} \longrightarrow \\ \text{fc?c} \end{array} \mid \#n & 1 \mid \begin{array}{c} \longrightarrow \\ \text{fs?c} \end{array} \mid \#n \\ 1 \mid \begin{array}{c} \longrightarrow \\ \text{fc?s} \end{array} \mid \#n & 1 \mid \begin{array}{c} \longrightarrow \\ \text{fs?s} \end{array} \mid \#n \end{array}$$

19.2 Exécution normale

19.2.1 Retour anticipé

Une définition utilisateur est une séquence d'instructions regroupée dans une expression en notation polonaise inverse. Une définition se termine naturellement à la fin de l'expression et retourne à l'exécution de l'instruction suivant son appel. Il est néanmoins possible d'anticiper ce retour par l'instruction `return`. Cette instruction renvoie à la routine appelante ou, s'il s'agit de la définition principal d'un programme, au système d'exploitation. Cette instruction ne prend aucun argument et peut figurer à n'importe quel endroit du code exécutable.

$$1 \mid \begin{array}{c} \longrightarrow \\ \text{return} \end{array} \mid$$

L'instruction de retour anticipé provoque une analyse sans exécution de tous les atomes figurant entre `return` et la fin de la définition courante car elle doit rendre la main à la définition appelante et garder la pile système dans un état cohérent. L'utilisation de `return` n'est donc pas aussi efficace que l'utilisation d'une structure de contrôle de type `if/then/end`.

CHAPITRE 19. CONTRÔLE DE L'EXÉCUTION

19.2.2 Abandon

Un programme s'arrête naturellement lorsqu'il atteint la fin de sa définition principale. À la fin de celle-ci, il signale son achèvement à tous ses processus fils, attend leur fin et rend la main au système d'exploitation. En aucun cas, il n'attend la fin normale de ses fils.

Il existe deux instructions pour déroger à cette règle. La première, `abort` provoque un arrêt du programme en cours. Quel que soit le processus du programme RPL/2 exécutant l'instruction `abort`, chacun des processus du programme reçoit une requête d'arrêt. Ces requêtes d'arrêt sont honorées immédiatement ou en temps différé si un bloc de contrôle de requête d'arrêt est installé dans l'un des processus à l'aide des instructions `cstop` et `rstop`. Lorsque toutes les requêtes d'arrêt sont traitées, le programme courant s'achève. À l'instar de l'instruction `return`, la requête d'arrêt peut apparaître n'importe où dans un programme.

Lorsque le RPL/2 reçoit le signal `SIGINT` — généralement contrôle+C sous Unix —, il appelle l'instruction `abort` à la fin de l'évaluation de l'atome en cours. L'exécution de l'instruction `abort` est soumis aux mêmes règles de traitement.

1	→	abort
---	---	-------

À noter : comme `abort` envoie une requête d'arrêt à tous les processus du programme et attend leur achèvement, cette instruction peut ne jamais s'achever si l'un des processus du programme n'honore pas la requête d'arrêt. En particulier, le programme suivant ne s'achève jamais car le processus fils n'honore pas la requête d'arrêt qui est temporisée par l'instruction `cstop` de la ligne 10 et jamais traitée.

```

0001 MAIN
0002 <<
0003     'PROCESSUS_FILS' detach
0004     10 wait
0005     abort
0006 >>
0007
0008 PROCESSUS_FILS
0009 <<
0010     cstop
0011     do 10 sleep until false end
0012 >>

```

L'instruction `kill` envoie une requête d'arrêt à tous les fils du processus qui exécute cette instruction et non à l'ensemble des processus du programme. Elle attend la fin de tous les fils du processus courant et peut ne jamais revenir si l'un des processus fils du processus courant n'honore pas la requête d'arrêt.

1	→	kill
---	---	------

19.3 Débogage

Le séquenceur possède un outil de débogage intégré. Cet outil n'est accessible que lorsque le programme à déboguer est interprété. S'il est compilé, les instructions de débogage sont silencieusement ignorées.

Cet outil très simple permet de positionner des points d'arrêt dans un programme afin d'en surveiller le fonctionnement. Lorsqu'un programme est arrêté, la pile s'affiche ainsi que la prochaine instruction à exécuter dans le programme et une invite de commande. Cette

invite de commande permet d'exécuter le programme pas à pas, d'en relancer l'exécution normale, de modifier le contenu des variables ou les éléments de la pile ou d'exécuter des commandes quelconques.

19.3.1 Point d'arrêt

Lorsqu'un programme rencontre l'instruction `halt` et qu'il est interprété, il passe immédiatement en mode débogage. Le contenu de la pile s'affiche ainsi que la prochaine instruction à exécuter dans le processus courant.

1	→
	halt

Lorsqu'un programme reçoit le signal `SIGTSTP` — généralement `contrôle+Z` sous Unix —, il exécute à la fin de l'évaluation de l'atome courant et dans le processus principal l'instruction `halt`. Ce signal est ignoré si le programme est compilé.

19.3.1.1 Exemple

```
0001 DEBOGAGE
0002 <<
0003     1 -> X
0004     <<
0005         X halt disp
0006     >>
0007 >>
```

Un point d'arrêt est fixé à la ligne 5. À cet endroit, la pile contient normalement la valeur contenue dans la variable `X`, ce qui est confirmé par l'outil de débogage. La valeur figurant entre crochets est l'identifiant du processus courant. Cette information est utile lorsqu'il s'agit de déboguer des programmes comportant plusieurs processus concurrents.

```
cauchy:[~] > rpl -s debug.rpl
+++RPL/2 (R) version 4.0.10 (lundi 08/02/2010, 14:31:33 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
```

```
1: 1
[10453] Instruction : disp
RPL/2> sst
1
[10453] Instruction : >>
RPL/2> sst
[10453] Instruction : >>
RPL/2> sst
cauchy:[~] >
```

19.3.2 Exécution pas à pas

L'exécution pas à pas d'un programme en cours de débogage se fait grâce à l'instruction `sst`. Cette instruction évalue l'objet suivant du programme, quel que soit le type de cet objet. Si un objet est composé de plusieurs atomes, l'évaluation ne rend la main qu'à la fin de l'évaluation de l'objet et non à la fin de l'évaluation de chacun des atomes. Si l'objet est un appel à une définition utilisateur, le contenu de cette définition sera aussi exécuté pas à pas.

1		→	
		sst	

19.3.3 Retour en exécution normale

L’instruction `cont` permet de repasser en mode d’exécution normale un programme se déroulant en mode débogage.

1		→	
		cont	

Cinquième partie

Fonctions mathématiques

20

Les opérations de base

Les différentes opérations présentées dans ce chapitre sont les opérations mathématiques de base. Ces opérations sont définies pour un grand nombre de types d'arguments différents.

20.1 Notations

Les différents types de données figurant dans les tableaux suivants sont :

- n , un entier signé;
- x , un réel;
- z , un complexe;
- s , un scalaire de type quelconque, entier, réel ou complexe;
- [vecteur], un vecteur de scalaires;
- [[matrice]], une matrice de scalaires;
- [tableau], un tableau — vecteur ou matrice — de scalaires;
- <[table]>, un tableau — vecteur ou matrice — de scalaires;
- 'symbole', un symbole algébrique qu'il s'agisse d'un nom ou d'une expression;
- « symbole », une expression en notation polonaise inverse;
- { liste }, une liste d'objets hétéroclites;
- "chaîne", une chaîne de caractères;
- #n, un entier binaire.

20.2 Addition : +

L'instruction + renvoie la somme de ses arguments. La nature de cette somme est déterminée par le type de ces arguments. L'ensemble des arguments de cette instruction est porté au tableau 20.1.

L'addition de tableaux ne peut s'effectuer que sur des tableaux de mêmes dimensions. Le type du tableau résultant suit une règle identique en tout point aux règles régissant les scalaires.

À noter : l'addition de deux entiers peut donner résultat de type réel si celui-ci ne peut être représenté en entier du fait d'un dépassement de capacité.

Niveau ₂	Niveau ₁	→	Niveau ₁	Résultat
n_1	n_2	→	$n_1 + n_2$	Entier
n_1	n_2	→	$n_1 + n_2$	Réel
n_1	x_2	→	$n_1 + x_2$	Réel
n_1	z_2	→	$n_1 + z_2$	Complexe
x_1	n_2	→	$x_1 + n_2$	Réel
x_1	x_2	→	$x_1 + x_2$	Réel
x_1	z_2	→	$x_1 + z_2$	Complexe
n_1	z_2	→	$n_1 + z_2$	Complexe
x_1	z_2	→	$x_1 + z_2$	Complexe
z_1	z_2	→	$z_1 + z_2$	Complexe
[tableau ₁]	[tableau ₂]	→	[tableau ₁ + tableau ₂]	
s	'symbole'	→	's+(symbole)'	Expression
'symbole'	s	→	'symbole+s'	Expression
'symbole ₁ '	'symbole ₂ '	→	'symbole ₁ + symbole ₂ '	Expression
s	« symbole »	→	« s symbole + »	Expression
« symbole »	s	→	« symbole s + »	Expression
« symbole ₁ »	« symbole ₂ »	→	« symbole ₁ symbole ₂ + »	Expression
{ liste ₁ }	{ liste ₂ }	→	{ liste ₁ liste ₂ }	Liste
"chaîne ₁ "	"chaîne ₂ "	→	"chaîne ₁ chaîne ₂ "	Chaîne
# n_1	n_2	→	# $n_1 + n_2$	Binaire
n_1	# n_2	→	# $n_1 + n_2$	Binaire
# n_1	# n_2	→	# $n_1 + n_2$	Binaire

TABLE 20.1 : Addition

20.3 Soustraction : -

La soustraction - donne la différence entre ses arguments, le type de cette différence étant fourni pas ceux des arguments. L'objet présent au niveau 1 de la pile est soustrait à celui occupant le niveau 2. L'ensemble des arguments autorisés figurent sur le tableau 20.2.

À l'instart de l'addition, la soustraction de deux tableaux ne peut s'effectuer que sur des tableaux de mêmes dimensions, le type du tableau résultant suivant une règle identique en tout point aux règles régissant les scalaires.

À noter : la soustraction de deux entiers peut donner résultat de type réel si celui-ci ne peut être représenté en entier du fait d'un dépassement de capacité.

20.4 Multiplication : *

La multiplication * renvoie le produit de ses arguments dans lequel la nature de ce produit est déterminée par le type de ses arguments. Les différents types acceptés comme arguments ainsi que les types en résultant sont indiqués au tableau 20.3.

20.5. DIVISION

À noter : le produit de deux entiers peut donner résultat de type réel si celui-ci ne peut être représenté en entier du fait d'un dépassement de capacité.

20.5 Division

20.5.1 Division standard : /

L'instruction / calcule le quotient — l'objet du niveau 2 étant divisé par celui présent au niveau 1 — de ses arguments. La nature du résultat est déterminé par le type de ses arguments. Les différents types acceptés comme arguments ainsi que les types en résultant sont indiqués au tableau 20.3.

L'instruction / permet, par extension de la division, de résoudre un système linéaire

$$\mathbf{Ax} = \mathbf{b}$$

connaissant la matrice $\mathbf{A}$ et le vecteur $\mathbf{b}$. En effet, en définissant une division d'un vecteur par une matrice, il deviendrait possible d'écrire cette équation sous la forme

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Le système

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix} \times \mathbf{x} = \begin{pmatrix} 0 \\ 1 \\ 3 \end{pmatrix}$$

se résout donc par la séquence d'instructions

```
RPL/2> [ 0 1 3 ] [[ 1 2 3 ] [ 4 5 6 ] [ 7 8 10 ] ] /
```

```
1: [ 1.666666666666667 -2.333333333333333 1. ]
```

```
RPL/2>
```

L'extension de la division reste valable dans le cas général de matrices en calculant

$$\mathbf{X} = \mathbf{A}^{-1}\mathbf{B},$$

l'argument de niveau 1 étant la matrice carrée $\mathbf{A}$. La matrice $\mathbf{B}$ doit posséder le même nombre de lignes que $\mathbf{A}$.

À noter : cette instruction ne provoque le calcul d'une division entière que si l'un au moins des arguments est un entier binaire. Dans tout autre cas, le résultat correspond à une division réelle.

20.5.2 Inversion : inv

L'instruction inv calcule l'inverse son argument. Le fonctionnement de cette instruction est décrit au tableau 20.5.

20.6 Puissance

20.6.1 Puissance standard : ** ou ^

L'élévation à une puissance d'un objet se fait par une instruction existant sous deux formes : ^ et **. Dans la suite de ce manuel, les deux formes seront utilisés indistinctement. Les types d'arguments et de résultats de cette fonction sont décrits dans le tableau 20.6.

Niveau ₂	Niveau ₁	→	Niveau ₁	Résultat
n_1	n_2	→	$n_1 - n_2$	Entier
n_1	n_2	→	$n_1 - n_2$	Réel
n_1	x_2	→	$n_1 - x_2$	Réel
n_1	z_2	→	$n_1 - z_2$	Complexe
x_1	n_2	→	$x_1 - n_2$	Réel
x_1	x_2	→	$x_1 - x_2$	Réel
x_1	z_2	→	$x_1 - z_2$	Complexe
n_1	z_2	→	$n_1 - z_2$	Complexe
x_1	z_2	→	$x_1 - z_2$	Complexe
z_1	z_2	→	$z_1 - z_2$	Complexe
[tableau ₁]	[tableau ₂]	→	[tableau ₁ - tableau ₂]	
s	'symbole'	→	's-(symbole)'	Expression
'symbole'	s	→	'symbole-s'	Expression
'symbole ₁ '	'symbole ₂ '	→	'symbole ₁ - symbole ₂ '	Expression
s	« symbole »	→	« s symbole - »	Expression
« symbole »	s	→	« symbole s - »	Expression
« symbole ₁ »	« symbole ₂ »	→	« symbole ₁ symbole ₂ - »	Expression
# n_1	n_2	→	# $n_1 + n_2$	Binaire
n_1	# n_2	→	# $n_1 + n_2$	Binaire
# n_1	# n_2	→	# $n_1 + n_2$	Binaire

TABLE 20.2 : Soustraction

Niveau ₂	Niveau ₁	→	Niveau ₁	Résultat
n_1	n_2	→	$n_1 n_2$	Entier
n_1	n_2	→	$n_1 n_2$	Réel
n_1	x_2	→	$n_1 x_2$	Réel
n_1	z_2	→	$n_1 z_2$	Complexe
x_1	n_2	→	$x_1 n_2$	Réel
x_1	x_2	→	$x_1 x_2$	Réel
x_1	z_2	→	$x_1 z_2$	Complexe
n_1	z_2	→	$n_1 z_2$	Complexe
x_1	z_2	→	$x_1 z_2$	Complexe
z_1	z_2	→	$z_1 z_2$	Complexe
[[matrice]]	[vecteur]	→	[matrice × vecteur]	
[[matrice]]	[[matrice]]	→	[[matrice × matrice]]	
[matrice]	s	→	[matrice × s]	
s	[tableau]	→	[s × tableau]	
s	'symbole'	→	's*(symbole)'	Expression
'symbole'	s	→	'(symbole)*s'	Expression
'symbole ₁ '	'symbole ₂ '	→	'symbole ₁ * symbole ₂ '	Expression
s	« symbole »	→	« s symbole * »	Expression
« symbole »	s	→	« symbole s * »	Expression
« symbole ₁ »	« symbole ₂ »	→	« symbole ₁ symbole ₂ * »	Expression
# n_1	n_2	→	# $n_1 n_2$	Binaire
n_1	# n_2	→	# $n_1 n_2$	Binaire
# n_1	# n_2	→	# $n_1 n_2$	Binaire

TABLE 20.3 : Multiplication

Niveau ₂	Niveau ₁	→	Niveau ₁	Résultat
n_1	n_2	→	n_1/n_2	Réel
n_1	x_2	→	n_1/x_2	Réel
n_1	z_2	→	n_1/z_2	Complexe
x_1	n_2	→	x_1/n_2	Réel
x_1	x_2	→	x_1/x_2	Réel
x_1	z_2	→	x_1/z_2	Complexe
n_1	z_2	→	n_1/z_2	Complexe
x_1	z_2	→	x_1/z_2	Complexe
z_1	z_2	→	z_1/z_2	Complexe
[tableau]	s	→	[tableau/ s]	
[tableau]	[[m. carrée]]	→	[tableau]	
s	'symbole'	→	' $s/(symbole)$ '	Expression
'symbole'	s	→	' $(symbole)/s$ '	Expression
'symbole ₁ '	'symbole ₂ '	→	'symbole ₁ * symbole ₂ '	Expression
s	« symbole »	→	« s symbole / »	Expression
« symbole »	s	→	« symbole s / »	Expression
« symbole ₁ »	« symbole ₂ »	→	« symbole ₁ symbole ₂ / »	Expression
$\#n_1$	n_2	→	$\#n_1/n_2$	Binaire
n_1	$\#n_2$	→	$\#n_1/n_2$	Binaire
$\#n_1$	$\#n_2$	→	$\#n_1/n_2$	Binaire

TABLE 20.4 : Division

Niveau ₁	→	Niveau ₁	Résultat
n	→	$1/n$	Réel
x	→	$1/x$	Réel
z	→	$1/z$	Complexe
[matrice]	→	[matrice]	
'symbole'	→	'INV(symbole)'	Expression
« symbole »	→	« symbole INV »	Expression

TABLE 20.5 : Inversion

CHAPITRE 20. LES OPÉRATIONS DE BASE

20.6.2 Carré : `sq`

L'instruction `sq` prend un objet dans la pile et retourne le carré de cet objet. Les différents types d'arguments figurent au tableau 20.7.

À noter : l'élévation au carré peut donner résultat de type réel si celui-ci ne peut être représenté en entier du fait d'un dépassement de capacité.

20.6.3 Racine carrée : `sqrt`

L'instruction `sqrt` prend un objet dans la pile et retourne la racine carrée de cet objet. Les différents types d'arguments figurent au tableau 20.8.

20.6.4 Racine $n^{\text{ième}}$: `xroot`

L'instruction `xroot` calcule la racine $n^{\text{ième}}$ d'un objet. Les différents arguments pris par cette instruction figurent au tableau 20.9.

Niveau ₂	Niveau ₁	→	Niveau ₁	Résultat
n_1	n_2	→	$n_1^{n_2}$	Entier
n_1	n_2	→	$n_1^{n_2}$	Réel
n_1	x_2	→	$n_1^{x_2}$	Réel
n_1	z_2	→	$n_1^{z_2}$	Complexe
x_1	n_2	→	$x_1^{n_2}$	Réel
x_1	x_2	→	$x_1^{x_2}$	Réel
x_1	z_2	→	$x_1^{z_2}$	Complexe
n_1	z_2	→	$n_1^{z_2}$	Complexe
x_1	z_2	→	$x_1^{z_2}$	Complexe
z_1	z_2	→	$z_1^{z_2}$	Complexe
s	'symbole'	→	's'(symbole)'	Expression
'symbole'	s	→	'(symbole)^s'	Expression
'symbole ₁ '	'symbole ₂ '	→	'symbole ₁ ^symbole ₂ '	Expression
s	« symbole »	→	« s symbole ^ »	Expression
« symbole »	s	→	« symbole s ^ »	Expression
« symbole ₁ »	« symbole ₂ »	→	« symbole ₁ symbole ₂ ^ »	Expression

TABLE 20.6 : Puissance

Niveau ₁	→	Niveau ₁	Résultat
n	→	n^2	Entier
n	→	n^2	Réel
x	→	x^2	Réel
z	→	z^2	Complexe
[[m. carrée]	→	[m. carrée]	
'symbole'	→	'SQ(symbole)'	Expression
« symbole »	→	« symbole SQ »	Expression

TABLE 20.7 : Élévation au carré

Niveau ₁	→	Niveau ₁	Résultat
n	→	$\sqrt{n}$	Réel
x	→	$\sqrt{x}$	Réel
x	→	$\sqrt{x}$	Complexe
z	→	$\sqrt{z}$	Complexe
'symbole'	→	'SQRT(symbole)'	Expression
« symbole »	→	« symbole SQRT »	Expression

TABLE 20.8 : Racine carrée

Niveau ₂	Niveau ₁	→	Niveau ₁	Résultat
n_1	n_2	→	n_1^{1/n_2}	Réel
x_1	n_2	→	x_1^{1/n_2}	Réel
n_1	n_2	→	n_1^{1/n_2}	Complexe
x_1	n_2	→	x_1^{1/n_2}	Complexe
z_1	n_2	→	z_1^{1/n_2}	Complexe
z_1	n_2	→	z_1^{1/n_2}	Complexe
s	'symbole'	→	$s^{1/(\text{symbole})}$	Expression
'symbole'	s	→	$(\text{symbole})^{1/s}$	Expression
s	« symbole »	→	« s symbole XROOT »	Expression
« symbole »	s	→	« symbole s XROOT »	Expression
« symbole ₁ »	« symbole ₂ »	→	« symbole ₁ symbole ₂ XROOT »	Expression

TABLE 20.9 : Racine $n^{\text{ième}}$

21

Constantes

Les constantes sont des instructions intrinsèques ou non. Contrairement à toutes les autres instructions, leur évaluation peut se faire en deux objets différents, une valeur numérique ou un nom.

Lorsqu'une constante symbolique est évaluée par l'instruction `->num`, elle renvoie toujours sa valeur numérique. Dans le cas où celle-ci est évaluée automatiquement ou par l'instruction `eval`, le résultat dépend de l'indicateur 35. Si l'indicateur 35 est armé, l'évaluation de la constante est symbolique. S'il est désarmé, l'évaluation par défaut est numérique.

21.1 Constantes booléennes : `true` et `false`

Parler de constantes booléennes est un abus de langage car l'évaluation de ces constantes, contrairement à toutes les autres constantes définies dans le RPL/2, est toujours numérique. Les deux constantes `true` et `false` sont évaluées numériquement pour pouvoir écrire des expressions logiques sans les surcharger par l'opérateur d'évaluation numérique `->num`. Cette évaluation autoritaire permet par exemple d'écrire :

```
do until false end
```

Les deux constantes sont des constantes entières. Leurs valeurs sont les suivantes :

$\rightarrow$	$\rightarrow$
false	true
0	-1

21.2 Constantes mathématiques

21.2.1 Base des logarithmes népériens : `e`

La constante symbolique `e` est une constante réelle. Lorsqu'elle est évaluée numériquement, sa valeur est 2,718 281 828 459 05.

21.2.2 Entier de Gauß : `i`

La constante symbolique `i` est une constante complexe. Lorsqu'elle est évaluée numériquement, sa valeur est (0,i) et satisfait à la relation $i^2 = -1$.

21.2.3 Constante d'Archimède : π

La constante symbolique π est une constante réelle. Lorsqu'elle est évaluée numériquement, sa valeur est 3,141 592 653 589 79.

21.3 Constantes physiques

Le RPL/2 ne connaît aucune constante physique. Il est possible d'en créer dans des bibliothèques en utilisant une instruction spécifique du RPL/C. De plus amples informations se trouvent à partir de la page 273 dans le chapitre traitant du RPL/C.

22

Arithmétique générale

22.1 Arithmétique réelle

22.1.1 Incrémentation et décrémentation : `incr` et `decr`

Les deux instructions `incr` et `decr` permettent respectivement d'incrémenter ou de décré-
menter un entier. Dans le cas où l'argument de l'instruction est un entier, elle renvoie dans
la pile le résultat de l'incrémentation ou de la décrémentation. Si l'argument de l'instruction
est un nom de variable, le contenu de la variable est modifié et l'instruction ne retourne rien
dans la pile.

Si l'argument de la fonction n'est ni un nom de variable existante ni un entier, elle ren-
voie une erreur. En particulier, un entier représenté par un objet de type réel ou complexe
provoque une erreur.

22.1.2 Valeur absolue, module et norme de Frobenius : `abs`

L'instruction `abs` renvoie la valeur absolue d'un entier ou d'un réel. Si l'argument est un
complexe, elle renvoie le module de ce complexe. Dans le cas d'un tableau, l'instruction `abs`
calcule la norme de Frobenius (ou norme euclidienne) définie comme la racine carrée de la
somme des carrés des valeurs absolues de tous les éléments. Elle agit aussi sur des expressions.
Les types d'arguments autorisés sont décrits au tableau 22.1.

Niveau ₁	→	Niveau ₁	Résultat
n	→	$ n $	Entier
x	→	$ x $	Réel
z	→	$ z $	Complexe
[tableau]	→	$\ \text{ [tableau] } \ $	
'symbole'	→	'ABS(symbole)'	Expression
« symbole »	→	« symbole ABS »	Expression

TABLE 22.1 : Valeur absolue

22.1.3 Négation : `neg`

L'instruction `neg` renvoie l'opposé de son argument. Les différents types d'arguments auto-
risés figurent au tableau 22.2.

Grâce à l'instruction `relax`, une double négation est traitée comme une identité et non comme le cumul de deux instructions `neg` successives.

L'instruction `neg` ne peut prendre en entrée un nombre binaire car l'arithmétique binaire se fait sur des entiers non signés de longueur variable. Elle n'est donc pas équivalente à l'opération de complément à deux.

22.1.4 Non opération : `relax`

L'instruction `relax` est au premier abord assez étrange puisqu'elle ne fait rien. Non seulement elle ne prend aucun argument, mais elle ne renvoie rien. L'utilité d'une instruction qui ne fait rien peut sembler anecdotique mais ce n'est pas le cas. En effet, si elle n'est pas destinée à être utilisée directement par un programme, elle sert à la simplification de doubles changements de signe comme instruction complémentaire à `NEG`.

Sans l'instruction `relax`, le changement de signe de l'expression '`NEG(PI)`' donnerait '`NEG(NEG(PI))`' au lieu de '`PI`'.

22.1.5 Signe : `sign`

L'instruction `sign` retourne dans le cas général le vecteur unitaire dans la direction de son argument. Pour un argument entier ou réel, le résultat de cette commande est :

- -1 si l'argument est négatif;
- 0 si l'argument est nul;
- $+1$ si l'argument est positif.

Pour un argument complexe non nul, `sign` renvoie

$$(x_2, y_2) = \left(\frac{x_1}{\sqrt{x_1^2 + y_1^2}}, \frac{y_1}{\sqrt{x_1^2 + y_1^2}} \right)$$

Les types d'arguments acceptés par l'instruction `sign` sont indiqués au tableau 22.3

22.1.6 Modulo : `mod`

Appliquée à les arguments entiers ou réels x et y , l'instruction `mod` donne un reste défini par

$$x \bmod y = x - y \times \text{ceil}\left(\frac{x}{y}\right)$$

Cette instruction est périodique en x avec une période y . Son résultat se situe sur l'intervalle $[0, y[$ pour y positif et sur l'intervalle $]y, 0]$ pour y négatif. Ses arguments sont décrits au tableau 22.4.

22.1.7 Minimum et maximum : `min` et `max`

Les deux instructions `min` et `max` retournent respectivement le minimum ou le maximum de leurs arguments. Elles regroupent deux fonctions différentes selon le type de leurs arguments. Si l'argument de niveau 1 est un entier ou un réel, ces deux fonctions s'attendent à trouver au niveau 1 un objet de type entier ou réel et renvoient le plus grand ou le plus petit des deux arguments dans la pile. Si l'argument de niveau 1 est un tableau — vecteur ou matrice —, ces deux fonctions renvoient deux objets :

Niveau ₁	→	Niveau ₁	Résultat
n	→	$-n$	Entier
x	→	$-x$	Réel
z	→	$-z$	Complexe
[tableau]	→	[opposé de chaque élément]	
'symbole'	→	'NEG(symbole)'	Expression
« symbole »	→	« symbole NEG »	Expression

TABLE 22.2 : Négation

Niveau ₁	→	Niveau ₁	Résultat
n	→	signe de n	Entier
x	→	signe de x	Entier
z	→	vecteur unité de z	Complexe
'symbole'	→	'SIGN(symbole)'	Expression
« symbole »	→	« symbole SIGN »	Expression

TABLE 22.3 : Signe

Niveau ₂	Niveau ₁	→	Niveau ₁	Résultat
n_1	n_2	→	$n_1 \bmod n_2$	Entier
n_1	x_2	→	$n_1 \bmod x_2$	Réel
x_1	n_2	→	$x_1 \bmod n_2$	Réel
x_1	x_2	→	$x_1 \bmod x_2$	Réel
'symbole ₁ '	'symbole ₂ '	→	'MOD(symbole ₁ , symbole ₂)'	Expression
« symbole ₁ »	« symbole ₂ »	→	« symbole ₁ symbole ₂ MOD »	Expression

TABLE 22.4 : Modulo

CHAPITRE 22. ARITHMÉTIQUE GÉNÉRALE

- au niveau 2, la plus grande valeur (ou la plus petite) du tableau ;
- au niveau 1, la liste indiquant la position de cet élément dans le tableau.

Dans le cas où le tableau est un tableau de nombres complexes, tous les calculs se font en module comme le montre l'exemple suivant :

RPL/2> [(-1, -25) (4,4)] max

2: (-1. , -25.)

1: { 1 }

RPL/2> 4 -6 min

3: (-1. , -25.)

2: { 1 }

1: -6

RPL/2>

Les types d'objets acceptés par cette fonction sont décrits aux tableaux 22.5 et 22.6. Ces deux fonctions prennent aussi des arguments de type expression.

Niveau ₂	Niveau ₁	→	Niveau ₂	Niveau ₁
n_1	n_2	→		$\min(n_1, n_2)$
n_1	x_2	→		$\min(n_1, x_2)$
x_1	n_2	→		$\min(x_1, n_2)$
x_1	x_2	→		$\min(x_1, x_2)$
	[tableau]	→	element	{ liste }

TABLE 22.5 : Minimum

Niveau ₂	Niveau ₁	→	Niveau ₂	Niveau ₁
n_1	n_2	→		$\max(n_1, n_2)$
n_1	x_2	→		$\max(n_1, x_2)$
x_1	n_2	→		$\max(x_1, n_2)$
x_1	x_2	→		$\max(x_1, x_2)$
	[tableau]	→	element	{ liste }

TABLE 22.6 : Maximum

22.1.8 partie entière : ip

L'instruction **ip** donne la partie entière de son argument. Le signe du résultat est le même que celui de l'argument. Dans le cas d'un résultat numérique, le type du résultat est entier sauf si sa représentation en entier est impossible. Le fonctionnement de l'instruction **ip** est décrit au tableau 22.7.

22.1.9 Partie fractionnaire : fp

L'instruction **fp** donne la partie fractionnaire de son argument. Le résultat a le même signe que l'argument. Les types d'arguments attendus par l'instruction **fp** sont décrits au tableau 22.8.

22.2. ARITHMÉTIQUE COMPLEXE

22.1.10 Valeur plancher : floor

L'instruction `floor` renvoie le plus grand entier inférieur ou égal à son argument lorsque celui-ci est entier ou réel. Si l'argument est un entier, le résultat est identique à l'argument. Cette instruction peut prendre une expression comme argument.

22.1.11 Valeur plafond : ceil

L'instruction `ceil` renvoie le plus petit entier supérieur ou égal à son argument lorsque celui-ci est entier ou réel. Si l'argument est un entier, le résultat est identique à l'argument. Cette instruction peut prendre une expression comme argument.

22.1.12 Mantisse : mant

L'instruction `mant` retourne la mantisse de son argument lorsque celui-ci est entier ou réel. Cette instruction peut prendre une expression comme argument. La mantisse est réelle et comprise dans l'intervalle $[1, 10[$. Ainsi, `0.12E34 mant` renvoie `1.2`.

22.1.13 Exposant : xpon

L'instruction `xpon` renvoie l'exposant de son argument de telle sorte que tout nombre entier ou réel puisse s'écrire comme :

$$\ll -> X \ll X \text{ mant } 10^X \text{ xpon} ** * \gg \gg$$

Le résultat de cette fonction est soit un entier dans le cas où l'argument est numérique, soit une expression algébrique ou non.

22.2 Arithmétique complexe

Cette section présente les fonctions spécifiques aux complexes. Elle ne reprend pas les fonctions communes aux nombres réels et aux nombres complexes.

22.2.1 Partie réelle : re

L'instruction `re`

22.2.2 Partie imaginaire : im

L'instruction `im`

22.2.3 Conjugaison : conj

La conjugaison d'un complexe est faite

22.2.4 Arg

22.3 Conversions

22.3.1 $P \rightarrow r$

22.3.2 $R \rightarrow p$

22.3.3 $C \rightarrow r$

22.3.4 $R \rightarrow c$

22.3.5 $\rightarrow q$

22.4 Proportions

22.4.1 Instruction %

22.4.2 Instruction %CH

22.4.3 Instruction %T

Niveau _I	→	Niveau _I
n	→	$\text{ip}(n)$
x	→	$\text{ip}(x)$
'symbole'	→	'IP(symbole)'
« symbole »	→	« symbole IP »

TABLE 22.7 : Partie entière

Niveau _I	→	Niveau _I
n	→	$\text{fp}(n)$
x	→	$\text{fp}(x)$
'symbole'	→	'FP(symbole)'
« symbole »	→	« symbole FP »

TABLE 22.8 : Partie fractionnaire

23

Tests

23.1 Comparaisons

23.1.1 Égalité

23.1.2 Différence

23.1.3 Inférieur

23.1.4 Inférieur ou égal

23.1.5 Supérieur

23.1.6 Supérieur ou égal

23.1.7 Parmi

23.2 Opérateurs logiques

23.2.1 Et logique

23.2.2 Ou logique

23.2.3 Ou logique exclusif

23.2.4 Non

24

Arithmétique binaire

24.1 Opérations de conversion

24.1.1 $B \rightarrow r$

24.1.2 $R \rightarrow b$

24.2 Formats

24.2.1 Dec

24.2.2 Bin

24.2.3 Oct

24.2.4 Hex

24.2.5 Stws

24.2.6 Rcws

24.3 Opérations sur des octets

24.3.1 Rlb

24.3.2 Rrb

24.3.3 Slb

24.3.4 Srb

24.4 Opérations sur des bits

24.4.1 Asl

24.4.2 Asr

24.4.3 Rl

24.4.4 Rr

24.4.5 Sl

24.4.6 Sr

24.4.7 Fonction et bit à bit

24.4.8 Fonction ou bit à bit

24.4.9 Fonction ou exclusif bit à bit

24.4.10 Fonction complément à 1

25

Fonctions trigonométriques

Attention aux arguments radian ou degré

25.1 Cosinus

25.1.1 Cos

25.1.2 Acos

25.2 Sinus

25.2.1 Sin

25.2.2 Asin

25.3 Tangente

25.3.1 Tan

25.3.2 Atan

25.4 Fonctions de conversion

25.4.1 Deg

25.4.2 Rad

25.4.3 $\rightarrow$ hms

25.4.4 hms $\rightarrow$

25.4.5 D $\rightarrow$ r

25.4.6 R $\rightarrow$ d

25.5 Arithmétique sexadécimale

25.5.1 Hms+

25.5.2 Hms−

26

Fonctions hyperboliques

26.1 Cosinus hyperbolique

26.1.1 Cosh

26.1.2 Acosh

26.2 Sinus hyperbolique

26.2.1 Sinh

26.2.2 Asinh

26.3 Tangente hyperboliques

26.3.1 Tanh

26.3.2 Atanh

27

Fonctions logarithmiques

27.1 Logarithme naturel

27.1.1 Ln

27.1.2 Lnpi

27.1.3 Exp

27.1.4 Expm

27.2 Logarithme vulgaire

27.2.1 Log

27.2.2 Alog

Sixième partie

Algèbre linéaire

28

Vecteurs et matrices

- 28.1** `→array`
- 28.2** `array→`
- 28.3** `Diag→`
- 28.4** `→diag`
- 28.5** `Size`
- 28.6** `Idn`
- 28.7** `Trn`
- 28.8** `Redimensionnement`
- 28.9** `Con`
- 28.10** `Col+`
- 28.11** `Col-`
- 28.12** `Col→`
- 28.13** `→col`
- 28.14** `Row+`
- 28.15** `Row-`
- 28.16** `Row→`
- 28.17** `→row`
- 28.18** `Échange de colonnes`
- 28.19** `Échange de lignes`
- 28.20** `Rci`
- 28.21** `Rcij`
- 28.22** `Get`
- 28.23** `Geti`
- 28.24** `Getc`
- 28.25** `Getr`

29

Résolution

- 29.1 Inversion**
- 29.2 Système linéaire**
- 29.3 Cond**
- 29.4 Rank**
- 29.5 Déterminant**
- 29.6 Moindres carrés**
- 29.7 Moindres carrés généralisés**
- 29.8 Résidus**

CHAPITRE 29. RÉOLUTION

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

30

Décompositions

- 30.1 Vecteurs propres**
- 30.2 Vecteurs propres généralisés**
- 30.3 Décomposition de Cholesky**
- 30.4 Décomposition LU de Crout**
- 30.5 Décomposition LQ**
- 30.6 Décomposition QR**
- 30.7 Décomposition de Schur**
- 30.8 Décomposition en valeurs singulières**

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Septième partie

Analyse

31

Analyse de Fourier

31.1 Transformée discrète

31.1.1 DFT

31.1.2 IDFT

31.2 Transformée rapide

31.2.1 FFT

31.2.2 IFFT

32

_____ *Calcul différentiel et intégral*

32.1 Dérivation et intégration

32.1.1 Der

32.1.2 Int

32.2 Développements limités

32.2.1 Taylr

32.2.2 Mclrin

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Huitième partie

Fonctions spéciales

33

Fonctions mathématiques

33.1 Fonctions Γ

33.2 Fonction de Bessel

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

34

Conversion d'unités

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

35

Fonctions temporelles

35.1 Horodatage

35.1.1 Date

35.1.2 Jdate

35.1.3 Rdate

35.2 Attente

35.2.1 Alarm

35.2.2 Wait

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

36

_____ *Accès au système d'exploitation*

- 36.1 Syseval**
- 36.2 Temps processeur consommé**
- 36.3 Journalisation**
- 36.4 Répertoire de travail**

37

Informations diverses

37.1 Mémoire utilisée

dans la pile dans et dans les variables

37.2 Version

37.3 Copyright

37.4 Garantie

37.5 Splash

37.6 Aide

37.7 Trace

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

38

Profilage

38.1 Pshprfl

38.2 Pulprfl

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

39

Autres fonctions

39.1 Vérification

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.



Neuvième partie

Probabilités et statistiques

40

Probabilités

40.1 Itinialisation d'un générateur

40.1.1 Rdgn

40.2 Rdz

40.3 Tirages aléatoires

40.4 Loi uniforme

40.5 Loi normale

41

Dénombrement

41.1 Fact

L’instruction `fact` calcule la factorielle d’un entier. Le résultat de cette commande est entier s’il peut être représenté en entier, réel sinon. Cette fonction est définie quel que soit la valeur entière. Si l’argument est négatif, elle renvoie une exception ou une valeur infinie selon que l’indicateur `59` est armé ou non. Les types d’arguments autorisés figurent au tableau 41.1.

Niveau ₁	→	Niveau ₁	Résultat
n	→	$n !$	Entier
n	→	n	Réel
'symbole'	→	'FACT(symbole)'	Expression
« symbole »	→	« symbole FACT »	Expression

TABLE 41.1 : Factorielle

41.2 Arrangements

41.3 Permutations

CHAPITRE 41. DÉNOMBREMENT

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

42

42.1 Matrice de statistique

42.1.1 Destruction

42.1.2 S+

42.1.3 S-

42.1.4 Stos

42.1.5 Rcls

42.1.6 Spar

42.1.7 Xcol

42.1.8 Ycol

42.1.9 Cols

42.1.10 Nombre d'éléments

42.2 Valeurs courantes

42.2.1 Maxs

42.2.2 Mins

42.2.3 Sx

42.2.4 Sx2

42.2.5 Sy

42.2.6 Sy2

42.2.7 Sxy

42.2.8 Tot

42.3 Corrélation

42.4 Moyenne

42.5 Variance et écart-type

42.5.1 Var

42.5.2 Pvar

42.5.3 Sdev

42.5.4 Psdev

42.5.5 Cov

42.5.6 Pcov

42.6 Graphiques

42.6.1 Scls

43

_____ *Lois de probabilité cumulées*

43.1 Distribution de Laplace-Gauß dite normale

43.2 Distribution du χ^2

43.3 Distribution de Fisher

43.4 Distribution de Student

Dixième partie

Accès aux éléments constituant les objets

Traiter des différences entre les listes et les tables.

44

Chaînes de caractères

- 44.1 →str**
- 44.2 str→**
- 44.3 Size**
- 44.4 Chr**
- 44.5 Num**
- 44.6 Sub**
- 44.7 Repl**
- 44.8 Pos**
- 44.9 Tokenize**
- 44.10 Trim**
- 44.11 Ucase**
- 44.12 Lcase**
- 44.13 Recode**
- 44.14 Localization**
- 44.15 Currenc**

45

--- Vecteurs et matrices

- 45.1 Get**
- 45.2 Put**
- 45.3 Geti**
- 45.4 Puti**
- 45.5 Size**

46

Listes

- 46.1** **→list**
- 46.2** **list→**
- 46.3** **Get**
- 46.4** **Sub**
- 46.5** **Put**
- 46.6** **Geti**
- 46.7** **Puti**
- 46.8** **Size**
- 46.9** **Repl**
- 46.10** **Pos**
- 46.11** **Head**
- 46.12** **Tail**
- 46.13** **Revlist**
- 46.14** **Sort**

47

Expressions

47.1 Evaluation

47.2 Size

47.3 Obget

47.4 Osub

47.5 Exget

47.6 Exsub

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

48

Tables

- 48.1 Crtab**
- 48.2 →table**
- 48.3 table→**
- 48.4 Get**
- 48.5 Put**
- 48.6 Size**
- 48.7 Pos**
- 48.8 Sort**

Onzième partie

Fichiers et sockets

49

Variable virtuelle

49.1 Store

49.2 Recall

50

- 50.1 Open**
- 50.2 Close**
- 50.3 Create**
- 50.4 Delete**
- 50.5 Append**
- 50.6 Rewind**
- 50.7 Seek**
- 50.8 Backspace**
- 50.9 Backspace**
- 50.10 Inquire**
- 50.11 Sync**
- 50.12 Lock**
- 50.13 Unlock**
- 50.14 Wflock**
- 50.15 Fichiers à accès séquentiel**
 - 50.15.1 Read**
 - 50.15.2 Format**
 - 50.15.3 Write**
- 50.16 Fichiers à accès direct**
 - 50.16.1 Read**
 - 50.16.2 Format**
 - 50.16.3 Write**
- 50.17 Fichiers à accès indexé**
 - 50.17.1 Read**
 - 50.17.2 Format**
 - 50.17.3 Write**

51

Bases de données

51.1 Sqlconnect

51.2 Sqldisconnect

51.3 Sqlquery

52

Sockets

52.1 Domaine Unix

52.2 Domaine IP

52.3 Instructions génériques

52.3.1 Open

52.3.2 Close

52.3.3 Format

52.3.4 Read

52.3.5 Write

52.3.6 Wfsock

52.3.7 Target

Douzième partie

Processus

53

Processus courant

53.1 Suspend

53.2 Stop

53.3 Continue

53.4 Nrproc

53.5 Daemonize

53.6 Sched

53.7 Yield

53.8 Wfproc

53.9 Traitement différé des requêtes d'arrêt

53.9.1 cstop

53.9.2 rstop

53.10 Fusibles

53.10.1 Fuse

53.10.2 Rfuse

53.10.3 Clrfuse

54

Processus détachés

54.1 Vie d'un processus détaché

54.2 Detach

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

55

Processus légers

55.1 Vie d'un processus léger

55.2 Spawn

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

56

Mutexes et sémaphores

56.1 Mutexes

56.1.1 Crmtx

56.1.2 Clrmtx

56.1.3 Mtxlock

56.1.4 Mtxtrylock

56.1.5 Mtxunlock

56.1.6 Mtxstatus

56.2 Sémaphores

56.2.1 Crsmphr

56.2.2 Clrsmphr

56.2.3 Smphrdecr

56.2.4 Smphrtrydecr

56.2.5 Smphrincr

56.2.6 Smphrgetv

57

Interruptions

- 57.1 Stoswi**
- 57.2 Rclswi**
- 57.3 Clrswi**
- 57.4 Swi**
- 57.5 Swilock**
- 57.6 Swiunlock**
- 57.7 Swiqueue**
- 57.8 Swistatus**
- 57.9 Isci**
- 57.10 Wfswi**

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

58

_____ *Communication interprocessus*

58.1 Vers le processus père

58.1.1 Par interruption

58.1.2 Par scrutation

58.1.3 Mécanisme de communication

58.1.3.1 Send

58.1.3.2 Recv

58.1.3.3 Wfdata

58.1.3.4 Wfack

58.2 Vers le processus fils

58.2.1 Peek

58.2.2 Poke

58.2.3 Wfpoke

58.2.4 Atpoke

58.2.5 clratpoke

CHAPITRE 58. COMMUNICATION INTERPROCESSUS

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Treizième partie

Graphisme

59

Instructions générales

59.1 Variables

59.1.1 Steq

59.1.2 Rceq

59.2 Paramètres

59.2.1 Indep

59.2.2 Depnd

59.2.3 Pmin

59.2.4 Pmax

59.2.5 Res

59.2.6 Ppar

59.3

59.4 Clcd

59.5 Redraw

59.6 Drax

59.7 Dgtiz

59.8 Persist

59.9 Label

59.10 Title

59.11 Keylabel

59.12 Keytitle

59.13 Dimensions

59.13.1 Autoscale

59.13.2 Scale

59.13.3 Slicescale

59.13.4 Logscale

59.13.5 Centr

59.13.6 Axes

59.13.7 *d

59.13.8 *h

60

_____ Sauvegarde et récupérations

60.1 **Lcd**→

60.2 →**lcd**

CHAPITRE 60. SAUVEGARDE ET RÉCUPÉRATIONS

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

61

Dessin

61.1 Newplane

61.2 Line

61.3 Mark

61.4 Plot

CHAPITRE 61. DESSIN

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

62

Fonctions

62.1 Types de fonction

62.1.1 Function

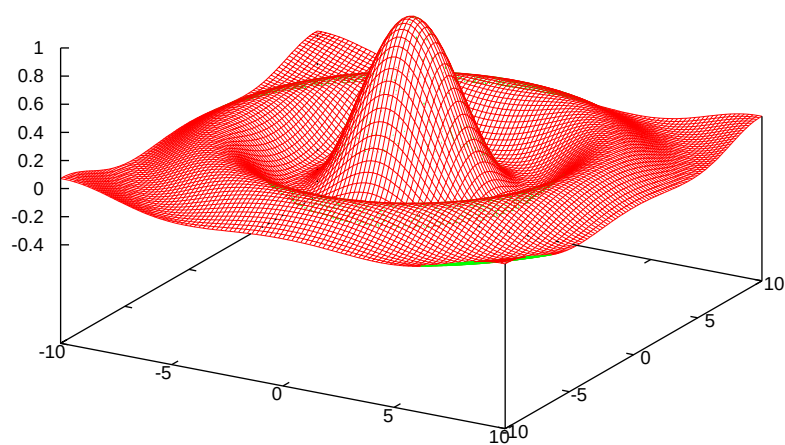
62.1.2 Parametric

62.1.3 Polar

62.1.4 Wireframe

62.1.5 Slice

62.2 Draw



CHAPITRE 62. FONCTIONS

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

63

Statistiques

63.1 Type de tracé

63.1.1 Plotter

63.1.2 Scatter

63.1.3 Histogram

63.2 Drws

63.3 Scls

63.4 Nuages de points

63.5 Histogrammes

CHAPITRE 63. STATISTIQUES

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Quatorzième partie

Impression

64

Gestion de l'impression

T_EX avec le format L^AT_EX 2_ε et gv.

CHAPITRE 64. GESTION DE L'IMPRESSION

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

65

Commandes

65.1 Format du papier

65.2 Effacement des fichiers graphiques

65.3 Print

65.4 Impression de données

65.4.1 pri

65.4.2 prst

65.4.3 prstc

65.4.4 prusr

65.4.5 prvar

65.4.6 prmd

65.5 Impression de graphiques

Quinzième partie

Interfaces externes

66

RPL/C

66.1 Définition du langage

66.2 Utilisation d'une bibliothèque

66.2.1 Use

66.2.2 Remove

66.2.3 Externals

67

_____ *Convention d'appel*

67.1 Depuis une convention C

67.2 Vers une convention C

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Seizième partie

Optimisations

68

Du bon usage des variables

68.1 Création de variables

Par la philosophie intrinsèque de la notation polonaise inverse, il est tout à fait possible de concevoir un programme de plusieurs milliers de lignes n'utilisant aucune variable. Néanmoins, les objets contenus dans la piles sont souvent conduits à être dupliqués, ce qui peut s'avérer préjudiciable à la vélocité du programme voire à son bon fonctionnement — problèmes d'allocation mémoire — lorsque les objets manipulés sont volumineux.

Ainsi, il peut être avantageux de remiser ces objets dans des variables et de travailler directement sur elles, la manipulation d'un nom étant bien plus aisée que celle d'un objet. Cependant, le processus de création d'une variable est complexe et relativement coûteux en terme de temps de calcul. Ainsi, il faut éviter de créer des variables dans une boucle et y préférer la création de ces mêmes variables avant l'initialisation de la boucle.

68.2 Utilisation de la pile

Si la création de variables peut s'avérer intéressante, il faut noter que le processus conduisant à la création d'une variable est coûteux. Ainsi, pour des opérations portant sur des objets peu volumineux, il est souvent plus efficace de travailler sur la pile que sur des variables.

CHAPITRE 68. DU BON USAGE DES VARIABLES

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

69

Bibliothèques partagées

CHAPITRE 69. BIBLIOTHÈQUES PARTAGÉES

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Dix-septième partie

Fonctionnement interne

70

Objets

CHAPITRE 70. OBJETS

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

71

--- *Utilisation de la mémoire*

71.1 Allocateur

71.2 Mémoire cache

CHAPITRE 71. UTILISATION DE LA MÉMOIRE

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

72

Gestion des processus

72.1 Communication interprocessus

72.2 Utilisation des signaux

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Dix-huitième partie

Exemples

73

Programmes RPL/2

73.1 Premier exemple simple

Ce petit programme réalise la décomposition d'une matrice en un produit de matrice. Il faut noter l'utilisation de l'instruction `cycle`. Le résultat de la décomposition est laissé dans la pile sous la forme d'une unique matrice dont la diagonale représente la matrice **D** et la matrice triangulaire inférieure, la matrice **L**, cette dernière comportant des 1 sur sa diagonale non calculée. Le reste de la matrice est demeuré inchangé.

Code source

```

0001 #!/usr/local/bin/rpl -csp
0002 // Script de calcul bestial de la la décomposition A=L*D*Lt
0003
0004 DECOMPOSITION
0005 <<
0006   [[ 10 20 30 ][ 20 45 80 ][ 30 80 171 ]] // Matrice A
0007
0008   dup size dup 1 get 1 ->list 0 con
0009   -> R S V
0010   <<
0011     1 S 1 get for J // Boucle de 1 à n
0012     1 J 1 - for I // Boucle de 1 à J-1
0013       if
0014         J 1 same
0015       then
0016         cycle
0017       end
0018
0019       'V' I 1 ->list 'R' J I 2 ->list get 'R'
0020       I I 2 ->list get * put
0021     next
0022
0023     'V' J 1 ->list
0024     'R' J J 2 ->list get
0025
0026     // Produit scalaire à calculer
0027
0028     0 1 J 1 - for K
0029       if
0030         J 1 same
0031       then

```

CHAPITRE 73. PROGRAMMES RPL/2

```

0032          cycle
0033          end
0034
0035          'R' J K 2 ->list get
0036          'V' K 1 ->list get * +
0037      next
0038
0039      - put 'R' J dup 2 ->list 'V' J 1 ->list get put
0040
0041      J 1 + S 1 get for K
0042
0043          // Une boucle est toujours effectuée au moins une
0044          // fois, Donc, on saute au « next » si les indices
0045          // sont trivialement faux.
0046
0047          if
0048              K S 1 get >
0049          then
0050              cycle
0051          end
0052
0053          'R' K J 2 ->list dup2 get
0054
0055          0 1 J 1 - for L
0056          if
0057              J 1 same
0058          then
0059              cycle
0060          end
0061
0062          'R' K L 2 ->list get
0063          'V' L 1 ->list get * +
0064      next
0065
0066      - 'V' J 1 ->list get / put
0067  next
0068  next
0069
0070      R
0071  >>
0072
0073      clmf
0074  >>

```

Résultat

L'exécution du programme se fait depuis un *shell* Unix en utilisant le *sha-bang*. Pour cela, le programme doit être exécutable.

```

riemann:[~/rpl] > ./decomposition.rpl
+++RPL/2 (R) version 4.0.10 (vendredi 05/02/2010, 15:27:42 CET)
+++Copyright (C) 1989 à 2009, 2010 BERTRAND Joël
1: [[ 10 20 30 ]
    [ 2 5 80 ]
    [ 3 4 1 ]]
riemann:[~/rpl] >

```

73.2 Programme complexe

Le programme présenté ici est un programme d'optimisation d'une valeur propre d'une matrice $\mathbf{F}$ dépendante d'un certain nombre de variables par rapport à ces mêmes variables. Ce programme, relativement compact, retourne ses résultats à la fois sur la sortie standard, dans un fichier PostScript et dans un fichier de données.

Cet algorithme est issu d'un travail de recherche publié dans la revue IEEE TRANSACTIONS ON SIGNAL PROCESSING, pages 1716 à 1721 du volume 51, numéro 7 de juillet 2003.

Code source

```
0001 #!/usr/local/bin/rpl -sp
0002
0003 /*
0004 =====
0005   Algorithme de l'Obèle
0006   Copyright 2001, BERTRAND Joël.
0007 =====
0008   Entrées : néant
0009 -----
0010   Sorties : néant
0011 -----
0012   Effets de bord : néant
0013 =====
0014 */
0015
0016 OBELE
0017 <<
0018   rad 31 sf
0019
0020   erase
0021   "" disp
0022   "Algorithme de l'obèle" disp
0023   "{\\Large\\sl Algorithme de l'obèle}" pr1 drop
0024
0025   { "standard*(*)" }
0026
0027   if
0028     "lambda" "existence" inquire
0029   then
0030     { { "name" "lambda" } "sequential" "replace" "writeonly" "formatted" }
0031     open
0032   else
0033     { { "name" "lambda" } "sequential" "new" "writeonly" "formatted" } open
0034   end
0035
0036   format
0037
0038   /* Paramètres d'entrée */
0039
0040   4           // Nombre d'antennes
0041   64          // Nombre de mobiles
0042   64          // Facteur d'étalement
0043   true        // Présence de bruit
0044   1           // Seuil (contrainte à tenir en terme de C/I)
0045   1E-5        // Niveau de bruit
0046   1E-8        // Critère de convergence
0047   "Statistique" // Modèle de canal ("Statistique" ou "Aléatoire")
0048   .5          // Distance entre les différents capteurs de l'antenne
0049   { }         // Directions des trajets (simple déclaration de variable)
0050   true        // Initialisation omnidirectionnelle (ou par un contrôle
0051              // de puissance élémentaire)
0052   true        // Normalisation des diagrammes
0053   true        // Diagrammes d'antenne en coordonnées polaires
0054   true        // Impression du résultat
0055   3           // Nombre de paquets de mobiles (modèle Statistique)
0056   .02         // Dispersion en fraction de '2*PI' (modèle Statistique)
0057   true        // Egalité des puissance émises (modèle Statistique)
0058   true        // Un mobile par paquet tracé dans les diagrammes
```

CHAPITRE 73. PROGRAMMES RPL/2

```

0059 -> UNITE N_ANTENNES N_MOBILES FACTEUR_ETALEMENT ALGORITHME_BRUTE SEUIL
0060 BRUIT EPS MODELE_CANAL DIST DIRECTIONS INITIALISATION_OMNIDIRECTIONNELLE
0061 DIAGRAMME_NORMALISE DIAGRAMMES_POLAIRES AUTORISATION_IMPRESSION PAQUETS
0062 DISPERSION EQUIPUISSANCE TRACE_UN_MOBILE
0063 <<
0064   "" disp
0065   "\\vskip 3ex\\noindent" pr1 drop
0066   "Configuration" pr1 drop
0067   "\\hrule\\vskip 1ex" pr1 drop
0068   cr "Nombre d'antennes : " N_ANTENNES ->str + pr1 disp
0069   cr "Nombre de mobiles : " N_MOBILES ->str + pr1 disp
0070   cr "Type de canal      : " MODELE_CANAL ->str + pr1 disp
0071
0072   PAQUETS 1 ->list 0 con
0073   -> REPARTITION
0074   <<
0075     PAQUETS DISPERSION DIST EQUIPUISSANCE
0076     N_ANTENNES N_MOBILES MODELE_CANAL
0077     INITIALISATION_R 'DIRECTIONS' sto 'REPARTITION' sto
0078
0079     if
0080       ALGORITHME_BRUTE
0081     then
0082       SEUIL CONVERSION_ALGORITHME_BRUTE
0083     end
0084
0085     if
0086       INITIALISATION_OMNIDIRECTIONNELLE
0087     then
0088       N_ANTENNES 1 2 ->list 0 con { 1 1 } 1 put
0089       1 N_MOBILES 1 - start dup next N_MOBILES ->list
0090     else
0091       dup FACTEUR_ETALEMENT ALGORITHME_BRUTE SEUIL
0092       OPTIMISATION_SIMPLE
0093     end
0094
0095     N_MOBILES N_MOBILES 2 ->list 0 con -1 0 0
0096
0097     -> LISTE_R V_PONDERATION F AVP VP PUISSANCES_INITIALES
0098     <<
0099       rclf 2 sci
0100       1 N_MOBILES for J
0101         rclf std " Utilisateur " J ->str + disp stof
0102         LISTE_R J get disp "" disp
0103       next
0104       stof
0105
0106       /* Calcul des rapports C/I initiaux */
0107
0108       LISTE_R V_PONDERATION
0109
0110       'F' { 1 1 }
0111       1 N_MOBILES for J
0112         1 N_MOBILES for K
0113           if
0114             J K same
0115           then
0116             0
0117           else
0118             V_PONDERATION K get
0119             dup trn
0120             LISTE_R J get
0121             rot * * { 1 1 } get
0122           end
0123           puti
0124         next
0125       next
0126       drop2
0127
0128       if
0129         ALGORITHME_BRUTE not
0130       then
0131         F regv max swap drop list-> drop
0132

```


73.2. PROGRAMME COMPLEXE

```

0133      -> M C
0134      <<
0135          1 N_MOBILES for J
0136              M J C 2 ->list get re
0137          next
0138
0139          N_MOBILES 1 ->list ->array dup abs /
0140      >>
0141  else
0142      N_MOBILES idn N_MOBILES 1 2 ->list SEUIL BRUIT * con swap
0143      F - inv swap * re array-> list-> drop2 1 ->list ->array
0144  end
0145
0146  2 sci
0147  "\\vskip 3ex\\noindent" pr1 drop
0148  " Rapports C/I initiaux" pr1 disp
0149  "\\hrule\\vskip 1ex" pr1 drop
0150  "" disp
0151  dup array-> 1 get ->list 1
0152
0153  true
0154  -> AUTORISATION_CALCUL
0155  <<
0156      do
0157          get1
0158
0159          if
0160              0 <
0161          then
0162              false 'AUTORISATION_CALCUL' sto
0163          end
0164      until
0165          dup 1 same
0166      end
0167
0168      drop2
0169
0170      if
0171          AUTORISATION_CALCUL
0172      then
0173          V_PONDERATION
0174
0175          over 720 DIAGRAMMES_POLAIRES TRACE_UN_MOBILE
0176          N_MOBILES N_ANTENNES DIST DIAGRAMME_NORMALISE
0177          DIRECTIONS PAQUETS REPARTITION DIAGRAMME
0178
0179          if
0180              DIAGRAMMES_POLAIRES not
0181          then
0182              { "Azimut" "Puissance" } label
0183          end
0184
0185          "Diagrammes avant optimisation" title persist prlcd
0186          cllcd
0187
0188          dup
0189          -> PUISSANCES
0190          <<
0191              0
0192              1 PUISSANCES size 1 get for I
0193                  PUISSANCES I 1 ->list get +
0194              next
0195          >>
0196
0197          'PUISSANCES_INITIALES' sto
0198
0199          if
0200              ALGORITHME_BRUTE not
0201          then
0202              0
0203          else
0204              BRUIT
0205          end
0206

```

CHAPITRE 73. PROGRAMMES RPL/2

```

0207             N_MOBILES FACTEUR_ETALEMENT
0208             CALCUL_RAPPORTS_SIGNAUX_INTERFERENCE
0209         else
0210             3 dropn
0211             " Résolution impossible du système"
0212         end
0213
0214         pr1 disp
0215         "" disp
0216     >>
0217
0218     /* Boucle principale */
0219
0220     "\\vskip 3ex\\noindent" pr1 drop
0221     " Minimisation de la plus grande valeur propre Lambda" disp
0222     " Minimisation de la plus grande valeur propre  $\lambda$ " disp
0223     pr1 drop
0224     "\\hrule\\vskip 1ex" pr1 drop
0225     "" disp std
0226
0227     while
0228         VP AVP - abs EPS >
0229     repeat
0230
0231         /* Normalisation des pondérations */
0232
0233         1 N_MOBILES for J
0234             LISTE_R J get V_PONDERATION J get
0235             FACTEUR_ETALEMENT ALGORITHME_BRUIE SEUIL
0236             NORMALISATION array-> drop
0237
0238             N_ANTENNES 1 2 ->list ->array
0239             'V_PONDERATION' swap J swap put
0240         next
0241
0242         /* Calcul de la matrice F */
0243
0244         'F' { 1 1 }
0245         1 N_MOBILES for J
0246             1 N_MOBILES for K
0247                 if
0248                     J K same
0249                 then
0250                     0
0251                 else
0252                     V_PONDERATION K get
0253                     dup trn
0254                     LISTE_R J get
0255                     rot * * { 1 1 } get
0256                 end
0257                 puti
0258             next
0259         next
0260         drop2
0261
0262         /* Calcul du plus grand vecteur propre gauche de F */
0263
0264         F legv
0265         -> MATRICE
0266         <<
0267             /* Réécriture de la fonction max pour éviter les */
0268             /* erreurs numériques d'arrondis apparaissant avec */
0269             /* les racines doubles du polynôme caractéristique */
0270
0271             do
0272                 MATRICE max
0273             until
0274                 over re 0 >=
0275
0276                 if
0277                     dup not
0278                 then
0279                     swap 'MATRICE' swap 0 put swap drop
0280                 end

```

```

0281         end
0282     >>
0283
0284     list-> drop rot swap
0285     -> COLONNE
0286 <<
0287     1 N_MOBILES for J
0288         dup J COLONNE 2 ->list get re swap
0289     next
0290
0291     drop
0292 >>
0293
0294     VP 'AVP' sto
0295     N_MOBILES 1 2 ->list ->array
0296     swap re dup 'VP' sto pr1
0297     UNITE over 1 ->list swap write
0298     "Lambda = " swap ->str + disp
0299
0300     /* Normalisation du plus grand vecteur gauche de F */
0301
0302     dup abs /
0303
0304     /* Calcul des matrices T */
0305
0306     -> PG
0307 <<
0308     1 N_MOBILES for J
0309         LISTE_R 1 get 0 con
0310
0311         1 N_MOBILES for K
0312             if
0313                 J K same not
0314             then
0315                 LISTE_R K get PG K 1 2 ->list get * +
0316             end
0317         next
0318     next
0319 >>
0320
0321     N_MOBILES ->list
0322
0323     /* Calcul des vecteurs propres généralisés des matrices T */
0324
0325     -> LISTE_T
0326 <<
0327     1 N_MOBILES for J
0328         LISTE_T J get LISTE_R J get gregv re min swap drop
0329         list-> drop
0330
0331     -> COLONNE
0332 <<
0333     1 N_ANNENNES for K
0334         dup K COLONNE 2 ->list get swap
0335     next
0336
0337     drop N_ANNENNES 1 ->list ->array dup abs /
0338     N_ANNENNES 1 2 ->list rdm
0339     'V_PONDERATION' J rot put
0340 >>
0341     next
0342 >>
0343 end
0344
0345     /* Normalisation des pondérations */
0346
0347     1 N_MOBILES for J
0348         LISTE_R J get V_PONDERATION J get
0349         FACTEUR_ETALEMENT ALGORITHME_BRUTE SEUIL
0350         NORMALISATION array-> drop
0351
0352     N_ANNENNES 1 2 ->list ->array
0353     'V_PONDERATION' swap J swap put
0354 next

```

CHAPITRE 73. PROGRAMMES RPL/2

```

0355
0356      /* Pondérations */
0357
0358      "" disp
0359      " Pondérations optimales" pr1 disp
0360      "\\hrule\\vskip 1ex" pr1 drop
0361      "" disp
0362
0363      1 N_MOBILES for J
0364      "W(" std J ->str + ") = " + 2 sci
0365      'V_PONDERATION' J get N_ANTENNES 1 ->list
0366      rdm pr1 ->str + disp
0367      next
0368
0369      /* Calcul des puissances par mobile nécessaires */
0370
0371      "" disp
0372      "\\vskip 3ex\\noindent" pr1 drop
0373      " Calcul des puissances par mobile nécessaires" pr1 disp
0374      "\\hrule\\vskip 1ex" pr1 drop
0375      "" disp
0376
0377      if
0378      ALGORITHME_BRUTE not
0379      then
0380      F regv max swap drop list-> drop
0381
0382      -> M C
0383      <<
0384      1 N_MOBILES for J
0385      M J C 2 ->list get re
0386      next
0387
0388      N_MOBILES 1 ->list ->array dup abs /
0389      >>
0390
0391      V_PONDERATION over 720 DIAGRAMMES_POLAIRES
0392      TRACE_UN_MOBILE N_MOBILES N_ANTENNES DIST
0393      DIAGRAMME_NORMALISE DIRECTIONS PAQUETS
0394      REPARTITION DIAGRAMME
0395
0396      if
0397      DIAGRAMMES_POLAIRES not
0398      then
0399      { "Azimut" "Puissance" } label
0400      end
0401
0402      "Diagrammes" title persist prlcd
0403      else
0404      if
0405      VP 1 >
0406      then
0407      " Absence de solution physique !"
0408      else
0409      N_MOBILES idn N_MOBILES 1 2 ->list
0410      SEUIL BRUIT * con swap
0411      F - inv swap * re array-> list-> drop2
0412      1 ->list ->array
0413
0414      V_PONDERATION over 720 DIAGRAMMES_POLAIRES
0415      TRACE_UN_MOBILE N_MOBILES N_ANTENNES DIST
0416      DIAGRAMME_NORMALISE DIRECTIONS PAQUETS
0417      REPARTITION DIAGRAMME
0418
0419      if
0420      DIAGRAMMES_POLAIRES not
0421      then
0422      { "Azimut" "Puissance" } label
0423      end
0424
0425      "Diagrammes" title persist prlcd
0426      end
0427      end
0428

```

73.2. PROGRAMME COMPLEXE

```

0429 V_PONDERATION over 2 ->list "resultat_obeles" store
0430 { "graphique.eps" "postscript eps enhanced color solid" }
0431 lcd->
0432
0433 dup
0434
0435 if
0436     dup type 2 same
0437 then
0438     pr1
0439 else
0440     dup array-> 1 get ->list pr1 drop
0441 end
0442
0443 if
0444     dup type 2 same not
0445 then
0446     "P = " swap ->str + disp "" disp
0447
0448     LISTE_R V_PONDERATION rot
0449
0450     dup
0451     -> PUISSANCES
0452     <<
0453         0
0454         1 PUISSANCES size 1 get for I
0455             PUISSANCES I 1 ->list get +
0456             next
0457     >>
0458
0459     if
0460         PUISSANCES_INITIALES 0 same not
0461     then
0462         rclf swap 3 fix PUISSANCES_INITIALES swap %ch neg ->str
0463         " %" + swap stof
0464     else
0465         drop "absurde"
0466     end
0467
0468     "\\vskip 3ex\\noindent" pr1 drop
0469     " Rapports C/I finaux "
0470     "(amélioration de la puissance émise : " +
0471     swap ->str + ")" + pr1 disp
0472
0473     if
0474         ALGORITHME_BRUTE not
0475     then
0476         0
0477     else
0478         BRUIT
0479     end
0480
0481     N_MOBILES FACTEUR_ETALEMENT
0482     CALCUL_RAPPORTS_SIGNAUX_INTERFERENCE
0483
0484     "\\hrule\\vskip 1ex" pr1 drop
0485     "" disp
0486     pr1 disp
0487     "" disp
0488
0489     cllcd
0490
0491     if
0492         AUTORISATION_IMPRESSION
0493     then
0494         print
0495     else
0496         erase
0497     end
0498 else
0499     drop disp "" disp
0500     erase
0501     cllcd
0502 end

```

CHAPITRE 73. PROGRAMMES RPL/2

```

0503      >>
0504      >>
0505
0506      UNITE close
0507      >>
0508
0509      " Temps CPU utilisé : " disp 2 fix time disp std
0510 >>
0511
0512
0513 /*
0514 =====
0515 Calcul des rapports C/I pour chaque mobile
0516 =====
0517 Entrées :
0518 4: liste contenant les matrices R de chaque mobile
0519 3: liste contenant les pondérations affectées à chaque mobile
0520 2: vecteur contenant les puissances
0521 1: sigma ** 2
0522 -----
0523 Sorties :
0524 1: liste contenant les rapports C/I
0525 -----
0526 Effets de bord : néant
0527 =====
0528 */
0529
0530 CALCUL_RAPPORTS_SIGNAUX_INTERFERENCE
0531 <<
0532 -> R W P SIGMA N_MOBILES FACTEUR_ETALEMENT
0533 <<
0534 1 N_MOBILES for I
0535   P I 1 ->list get
0536   W I get dup trn swap
0537   R I get swap * * * { 1 1 } get re
0538
0539   SIGMA
0540   1 N_MOBILES for J
0541     if
0542       I J same
0543     then
0544       cycle
0545     end
0546
0547     P J 1 ->list get
0548     W J get dup trn swap
0549     R I get swap * * * { 1 1 } get re +
0550   next
0551
0552   / FACTEUR_ETALEMENT *
0553 next
0554
0555 N_MOBILES ->list
0556 >>
0557 >>
0558
0559
0560 /*
0561 =====
0562 Fonction de normalisation des vecteurs W de telle sorte que  $\text{trn}(W)*R*W = 1$ 
0563 =====
0564 Entrées :
0565 2: matrice R
0566 1: vecteur W
0567 -----
0568 Sorties :
0569 1: vecteur W normalisé
0570 -----
0571 Effets de bord : néant
0572 =====
0573 */
0574
0575 NORMALISATION
0576 <<

```

```

0577 -> R W FACTEUR_ETALEMENT ALGORITHME_BRUTE SEUIL
0578 <<
0579   W dup trn R FACTEUR_ETALEMENT *
0580
0581   if
0582     ALGORITHME_BRUTE
0583   then
0584     SEUIL /
0585   end
0586
0587   W * * abs sqrt /
0588 >>
0589 >>
0590
0591
0592 /*
0593 =====
0594 Fonction renvoyant une liste contenant les différentes matrices R
0595 =====
0596 Entrées :
0597   3: nombre d'antennes (entier)
0598   2: nombre de mobiles (entier)
0599   1: nombre de trajets (entier)
0600 -----
0601 Sorties :
0602   2: liste contenant autant de matrices R qu'il y a de mobiles
0603   1: directions des mobiles
0604 -----
0605 Effets de bord : néant
0606 =====
0607 */
0608
0609 INITIALISATION_R
0610 <<
0611   "" disp
0612   " Initialisation des matrices d'autocorrélation du canal" disp
0613   "" disp
0614
0615   "\\vskip 3ex\\noindent" pr1 drop
0616   "Positions et puissances des différents récepteurs" pr1 drop
0617   "\\hrule\\vskip 1ex" pr1 drop
0618
0619   { } dup
0620   -> PAQUETS DISPERSION DIST EQUIPUISSANCE NA NM MODELE DIRECTIONS
0621   REPARTITION_INTERNE
0622   <<
0623     rclf
0624
0625     if
0626       MODELE "Statistique" same
0627     then
0628       PAQUETS 1 ->list 0 con 'REPARTITION_INTERNE' sto
0629
0630       /*
0631       Génération de matrices de covariance du canal grâce
0632       au modèle du Statistique
0633       */
0634
0635       deg 4 fix { 3 9 } 0 con
0636       DIST 180 25 0 0 PAQUETS 1 ->list 0 con
0637       -> COEFF // Coefficients du modèle :
0638               // - ligne 1 : direction du trajet en degrés;
0639               // - ligne 2 : puissance du trajet en dB;
0640               // - ligne 3 : retard du trajet en ns.
0641       D // Distance entre deux capteurs consécutifs comptée en
0642         // longueur d'onde
0643       SECTEUR // Demi angle d'ouverture d'un secteur (en degrés)
0644       DTHETA // Paramètre d'ouverture du modèle (en degrés)
0645       GISMIN // Gisement le plus faible vu du secteur
0646       GISMAX // Gisement le plus grand vu du secteur
0647       ANGLES_MOYENS
0648       <<
0649         SECTEUR neg DTHETA 4 * - 'GISMIN' sto
0650         SECTEUR DTHETA 2 * + 'GISMAX' sto

```

CHAPITRE 73. PROGRAMMES RPL/2

```

0651      'COEFF' { 1 1 }
0652
0653      0 puti
0654      DTHETA 2 / puti
0655      DTHETA 2 / neg puti
0656      DTHETA 2 / 1 - puti
0657      1 DTHETA 2 / - puti
0658      2 DTHETA * puti
0659      -2 DTHETA * puti
0660      3 DTHETA * puti
0661      4 DTHETA * puti
0662
0663      -2 puti
0664      -7 puti
0665      -7 puti
0666      -4 puti
0667      -4 puti
0668      -9 puti
0669      -10 puti
0670      -15 puti
0671      -20 puti
0672
0673      0 puti
0674      0 puti
0675      0 puti
0676      310 puti
0677      310 puti
0678      710 puti
0679      1090 puti
0680      1730 puti
0681      2510 puti
0682
0683      drop2
0684
0685      0 -> CUMUL
0686      <<
0687          1 COEFF size 2 get for I
0688          'COEFF' 2 I 2 ->list 'COEFF' over get 10 / alog
0689          dup 'CUMUL' sto+ put
0690      next
0691
0692          1 COEFF size 2 get for I
0693          'COEFF' 2 I 2 ->list 'COEFF' over get CUMUL / put
0694      next
0695      >>
0696
0697      // Calcul de la répartition des mobiles dans les paquets
0698      'REPARTITION_INTERNE' { 1 }
0699      1 PAQUETS for P
0700          rand puti
0701      next
0702      drop2
0703
0704      REPARTITION_INTERNE array-> 1 get 2 swap for P + next
0705
0706      -> CLEF
0707      <<
0708          1 PAQUETS for P
0709          'REPARTITION_INTERNE' dup P 1 ->list get NM * CLEF /
0710          ip P 1 ->list swap
0711
0712          if
0713              dup 1 <
0714          then
0715              drop 1
0716          end
0717
0718          put
0719      next
0720      >>
0721
0722      0 1 PAQUETS for P
0723          'REPARTITION_INTERNE' P 1 ->list get +
0724

```


73.2. PROGRAMME COMPLEXE

```

0725      next
0726
0727      NM -
0728      -> DIFFERENCE
0729      <<
0730          if
0731              DIFFERENCE 0 >
0732          then
0733              while
0734                  DIFFERENCE
0735              repeat
0736                  rand PAQUETS * ip 1 + 1 ->list dup
0737                  if
0738                      'REPARTITION_INTERNE' swap get dup 1 >
0739                  then
0740                      1 - 'REPARTITION_INTERNE' rot rot put
0741                      'DIFFERENCE' 1 sto-
0742                  else
0743                      drop2
0744                  end
0745              end
0746          else
0747              while
0748                  DIFFERENCE
0749              repeat
0750                  rand PAQUETS * ip 1 + 1 ->list dup
0751                  if
0752                      'REPARTITION_INTERNE' swap get dup NM <
0753                  then
0754                      1 + 'REPARTITION_INTERNE' rot rot put
0755                      'DIFFERENCE' 1 sto+
0756                  else
0757                      drop2
0758                  end
0759              end
0760          end
0761      >>
0762
0763      'ANGLES_MOYENS' { 1 }
0764      1 PAQUETS for P
0765          rand 2 SECTEUR * * SECTEUR - puti
0766      next
0767      drop2
0768
0769      rclf std
0770      "Répartition : " REPARTITION_INTERNE ->str + pr1
0771      "\\hrule\\vskip 1ex" pr1 drop
0772      disp "" disp stof
0773
0774      // Boucle sur les paquets de mobiles
0775      1 PAQUETS for P
0776
0777          // Boucle sur les mobiles
0778          1 REPARTITION_INTERNE P 1 ->list get for K
0779              rand 2 SECTEUR * * SECTEUR - DISPERSION *
0780              ANGLES_MOYENS P 1 ->list get +
0781              DIRECTIONS over 1 ->list + 'DIRECTIONS' sto
0782              "Azimut : " over ->hms ->str + dup " ° (HMS)" + disp
0783              "\\degre (HMS)" + cr pr1 drop
0784
0785              NA COEFF size 2 get 2 ->list 0 con
0786              COEFF size 2 get dup 2 ->list 0 con 0
0787              -> AZIMUT MD P TRAJETS_RETENUS
0788              <<
0789                  1 COEFF size 2 get for I
0790                      'COEFF' 1 I 2 ->list get AZIMUT +
0791                      -> G
0792                  <<
0793                      if
0794                          G SECTEUR <= G SECTEUR neg >= and
0795                          SECTEUR 180 >= or
0796                      then
0797                          1 NA for J
0798                              'MD' 2 i pi * * ->num

```

CHAPITRE 73. PROGRAMMES RPL/2

```

0799          D J 1 - * * G sin * exp
0800          J I 2 ->list swap put
0801          next
0802
0803          'P' COEFF 2 I 2 ->list get
0804          I dup 2 ->list swap put
0805          1 'TRAJETS_RETENUS' sto+
0806          end
0807          >>
0808          next
0809
0810          MD P
0811
0812          if
0813              EQUIPUISSANCE
0814          then
0815              1
0816          else
0817              nrand sq
0818          end
0819
0820          "Puissance : " over ->str + cr pri disp
0821          * over trn * *
0822
0823          // Rajout de bruit pour éviter d'avoir une
0824          // matrice R de rang non plein
0825
0826          if
0827              TRAJETS_RETENUS over size 1 get <
0828          then
0829              dup idn over abs 1E-6 / * +
0830          end
0831          >>
0832          next
0833          next
0834          >>
0835
0836          rad
0837      else
0838
0839          /* Génération de matrices R aléatoires */
0840
0841          // Nombre de trajets
0842          4 -> NT
0843          <<
0844              // Boucle sur les mobiles
0845              1 NM for K
0846                  " Utilisateur " std K ->str + disp "" disp
0847                  NT NA 2 ->list 0 con
0848
0849              // Boucle sur les trajets
0850              1 NT for L
0851                  L 1 2 ->list
0852                  rand 2 pi ->num * *
0853                  DIRECTIONS over r->d 1 ->list + 'DIRECTIONS' sto
0854                  nrand sq
0855
0856                  -> D P
0857                  <<
0858                      std
0859                      "   Trajet " L ->str + disp
0860                      4 sci "       -> Puissance " P ->str + disp
0861                      4 fix "       -> Azimut   " D r->d ->hms ->str +
0862                      "° (HMS)" + disp "" disp
0863
0864                  // Boucle sur les capteurs
0865                  1 NA for J
0866                      i ->num D sin J 1 - * * DIST *
0867                      2 pi ->num * 2E9 * * exp P * puti
0868                  next
0869                  drop
0870              >>
0871          next
0872          trn conj dup trn *

```

```

0873         next
0874     >>
0875
0876     NM 1 ->list 1 con 'REPARTITION_INTERNE' sto
0877 end
0878
0879     NM ->list
0880     swap stof
0881     REPARTITION_INTERNE
0882     DIRECTIONS
0883 >>
0884
0885     "" disp
0886 >>
0887
0888
0889 /*
0890 =====
0891 Fonction permettant de convertir l'algorithme non bruité en sa version
0892 bruitée
0893 =====
0894 Entrées :
0895 2: liste contenant les différentes matrices R
0896 1: seuil
0897 -----
0898 Sorties :
0899 1: liste contenant les nouvelles matrices R'
0900 -----
0901 Effets de bord : néant
0902 =====
0903 */
0904
0905 CONVERSION_ALGORITHME_BRUITE
0906 <<
0907     -> L_R SEUIL
0908     <<
0909         1 L_R size for J
0910         'L_R' dup J get SEUIL * J swap put
0911     next
0912
0913     L_R
0914 >>
0915 >>
0916
0917
0918 /*
0919 =====
0920 Calcul simple des pondérations et des puissances
0921 =====
0922 Entrées :
0923 1: liste contenant les différentes matrices R
0924 -----
0925 Sorties :
0926 2: liste contenant les pondérations et les puissances
0927 1: valeur propre
0928 -----
0929 Effets de bord : néant
0930 =====
0931 */
0932
0933 OPTIMISATION_SIMPLE
0934 <<
0935     -> LISTE FACTEUR_ETALEMENT ALGORITHME_BRUITE SEUIL
0936     <<
0937         1 LISTE size for N
0938         LISTE N get regv max swap drop
0939         list-> drop over size 1 get 1 2 ->list 0 con
0940
0941     -> INDICE TABLEAU
0942     <<
0943         1 over size 1 get for M
0944         dup M 1 2 ->list get 'TABLEAU' swap M 1 2 ->list swap put
0945     next
0946     drop LISTE N get TABLEAU FACTEUR_ETALEMENT ALGORITHME_BRUITE

```

CHAPITRE 73. PROGRAMMES RPL/2

```

0947             SEUIL NORMALISATION
0948             >>
0949             next
0950
0951             LISTE size ->list
0952             >>
0953             >>
0954
0955             /*
0956             =====
0957             Calcul du diagramme de rayonnement du réseau d'antennes
0958             =====
0959             Entrées :
0960             3: liste contenant tous les vecteurs de pondération
0961             2: puissances
0962             1: nombre de points à calculer par diagramme
0963             -----
0964             Sorties :
0965             néant
0966             -----
0967             Effets de bord : néant
0968             =====
0969             */
0970             /*
0971             DIAGRAMME
0972             <<
0973             0
0974             -> PONDERATIONS PUISSANCES NB_POINTS DIAGRAMMES_POLAIRES
0975             TRACE_UN_MOBILE N_MOBILES N_ANTENNES DIST DIAGRAMME_NORMALISE
0976             DIRECTIONS PAQUETS REPARTITION MAXIMUM
0977             <<
0978             cllcd
0979
0980             { { 60 "ticsonly" 2 } { "automatic" "ticsonly" 10 } } axes
0981
0982             if
0983             DIAGRAMMES_POLAIRES
0984             then
0985             1 d->r
0986             -> PAS
0987             <<
0988             0
0989             1 N_MOBILES for I
0990             0 2 pi ->num * for T
0991             if
0992             DIST N_ANTENNES T PONDERATIONS I DIAGRAMME_NORMALISE
0993             PUISSANCES FCT_DIAGRAMME dup 3 pick >
0994             then
0995             swap
0996             end
0997             drop PAS
0998             step
0999             next
1000             >>
1001
1002             dup 'MAXIMUM' sto dup r->c dup pmax neg pmin
1003
1004             parametric { T 0 'MAXIMUM' } indep MAXIMUM res
1005             << T I DIRECTIONS FCT_DIRECTIONS >> steq
1006
1007             1 N_MOBILES for I
1008             draw
1009             next
1010
1011             polar { T 0 '2*PI' } indep 2 pi ->num * NB_POINTS / res
1012             << DIST N_ANTENNES T PONDERATIONS I DIAGRAMME_NORMALISE
1013             PUISSANCES FCT_DIAGRAMME >> steq
1014
1015             if
1016             TRACE_UN_MOBILE
1017             then
1018             1
1019             -> I
1020

```

73.2. PROGRAMME COMPLEXE

```

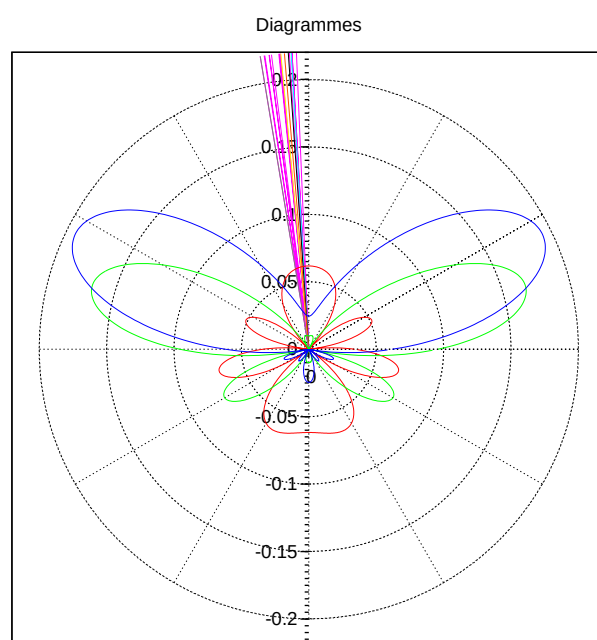
1021         <<
1022             1 PAQUETS for POINTEUR
1023                 draw
1024                 'I' REPARTITION POINTEUR 1 ->list get ip sto+
1025             next
1026         >>
1027     else
1028         1 N_MOBILES for I
1029             draw
1030         next
1031     end
1032 else
1033     { X Y } autoscale 'Y' logscale
1034     parametric { T '-PI' 'PI' } indep 2 pi ->num * NB_POINTS / res
1035     << T r->d DIST N_ANTENNES T PONDERATIONS I DIAGRAMME_NORMALISE
1036     PUISSANCES FCT_DIAGRAMME r->c >> steq
1037
1038     if
1039         TRACE_UN_MOBILE
1040     then
1041         1
1042         -> I
1043         <<
1044             1 PAQUETS for POINTEUR
1045                 draw
1046                 'I' REPARTITION POINTEUR 1 ->list get ip sto+
1047             next
1048         >>
1049     else
1050         1 N_MOBILES for I
1051             draw
1052         next
1053     end
1054 end
1055 >>
1056
1057 drax
1058 >>
1059
1060
1061 FCT_DIRECTIONS
1062 <<
1063     -> T I DIRECTIONS
1064     <<
1065         rclf deg
1066         DIRECTIONS I get dup cos T * swap sin T * i ->num * +
1067         swap stof
1068     >>
1069 >>
1070
1071
1072 FCT_DIAGRAMME
1073 <<
1074     -> DIST N_ANTENNES T PONDERATIONS I DIAGRAMME_NORMALISE PUISSANCES
1075     <<
1076         N_ANTENNES 1 ->list i ->num 2 pi ->num * * T sin DIST * * con
1077
1078         1 N_ANTENNES for J
1079             dup J 1 ->list get J 1 - * exp
1080             J 1 ->list swap put
1081         next
1082
1083         PONDERATIONS I get array-> 1 get 1 ->list ->array swap dot abs 2 /
1084
1085         if
1086             DIAGRAMME_NORMALISE not
1087         then
1088             PUISSANCES I 1 ->list get *
1089         end
1090     >>
1091 >>
1092
1093 // vim: ts=4

```

Toutes copies ou impressions mêmes partielles de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

Graphique généré

Le programme précédent génère une sortie PostScript mise en forme et des diagrammes configurables dans un système de coordonnées cartésiennes ou polaires.



74

Bibliothèques

La bibliothèque présentée ici est une simple interface entre des fonctions écrites en C et le RPL/2. Elle est écrite en RPL/C, porte le nom de `fuzzySearch` et ajoute trois définitions extrinsèques au RPL/2 déclarées à la ligne 7.

```

0001 #include <rplexternals.h>
0002 #include "double_metaphone.h"
0003
0004 double strcmp95(char *ying, char *yang, long y_length, int ind_c[]);
0005
0006 libraryName(fuzzySearch);
0007 exportExternalFunctions(doubleMetaphone, levenshtein, jaroWinkler);
0008
0009 declareSubroutine(onLoading)
0010     notice(stdout, "\nFuzzy search library V1R2 for RPL/2 (C) 2009, 2010 "
0011         "BERTRAND Joel\n");
0012     notice(stdout, "Library successfully loaded.\n\n");
0013 endSubroutine
0014
0015 declareSubroutine(onClosing)
0016     notice(stdout, "\nFuzzy search library unloaded.\n\n");
0017 endSubroutine
0018
0019 declareExternalFunction(jaroWinkler)
0020     declareObject(object1);
0021     declareObject(object2);
0022     declareObject(object3);
0023
0024     string      s1;
0025     string      s2;
0026     string      s3;
0027
0028     int          ind_c[2];
0029
0030     ind_c[0] = 0;
0031     ind_c[1] = 0;
0032
0033     HEADER
0034         declareHelpString("Return Jaro-Winkler distance between two strings");
0035         numberOfArguments(2);
0036     FUNCTION
0037         pullFromStack(object1, string);
0038         returnOnError(freeObject(object1));
0039         pullFromStack(object2, string);
0040         returnOnError(freeObject(object1); freeObject(object2));
0041
0042         getString(object1, s1);
0043         getString(object2, s2);
0044
0045         createRealObject(object3);
0046
0047         if (strlen(s1) gt strlen(s2)) then

```

CHAPITRE 74. BIBLIOTHÈQUES

```

0048         s3 = allocate((strlen(s1) + 1) * size(char));
0049         memset(s3, ' ', strlen(s1));
0050         s3[strlen(s1)] = 0;
0051         strcpy(s3, s2);
0052
0053         setReal(object3, strcmp95(s1, s3, strlen(s1), ind_c));
0054         deallocate(s3);
0055     elsif (strlen(s1) lt strlen(s2)) then
0056         s3 = allocate((strlen(s2) + 1) * size(char));
0057         memset(s3, ' ', strlen(s2));
0058         s3[strlen(s2)] = 0;
0059         strcpy(s3, s1);
0060
0061         setReal(object3, strcmp95(s3, s2, strlen(s2), ind_c));
0062         deallocate(s3);
0063     orElse
0064         setReal(object3, strcmp95(s1, s2, strlen(s1), ind_c));
0065     endif
0066
0067     freeObject(object1);
0068     freeObject(object2);
0069
0070     pushOnStack(object3);
0071 END
0072 endExternalFunction
0073
0074 declareExternalFunction(doubleMetaphone)
0075     declareObject(object1);
0076     declareObject(object2);
0077     declareObject(object3);
0078     declareObject(object4);
0079
0080     char        *output[2];
0081     string      input;
0082
0083     HEADER
0084         declareHelpString("Double Metaphore Algorithm");
0085         numberOfArguments(1);
0086     FUNCTION
0087         pullFromStack(object1, string);
0088         returnOnError(freeObject(object1));
0089         getString(object1, input);
0090
0091         DoubleMetaphone(input, output);
0092
0093         createStringObject(object2);
0094         setString(object2, output[0]);
0095         pushOnStack(object2);
0096
0097         createStringObject(object3);
0098         setString(object3, output[1]);
0099         pushOnStack(object3);
0100
0101         createIntegerObject(object4);
0102         setInteger(object4, 2);
0103         pushOnStack(object4);
0104
0105         intrinsic(fleche_list);
0106         freeObject(object1);
0107     END
0108 endExternalFunction
0109
0110 declareExternalFunction(levenshtein)
0111     declareObject(object1);
0112     declareObject(object2);
0113     declareObject(object3);
0114
0115     string      input1;
0116     string      input2;
0117
0118     HEADER
0119         declareHelpString("Levenshtein distance");
0120         numberOfArguments(2);
0121     FUNCTION

```



```

O122     pullFromStack(object1, string);
O123     returnOnError(freeObject(object1));
O124     pullFromStack(object2, string);
O125     returnOnError(freeObject(object1); freeObject(object2));
O126
O127     getString(object1, input1);
O128     getString(object2, input2);
O129
O130     createIntegerObject(object3);
O131     setInteger(object3, levenshtein_distance(input1, input2));
O132
O133     pushOnStack(object3);
O134
O135     freeObject(object1);
O136     freeObject(object2);
O137     END
O138 endExternalFunction

```


Instructions

<code>*</code> , 148	<code>#elif</code> , 78
<code>**</code> , 149	<code>#else</code> , 77
<code>+</code> , 147	<code>#endif</code> , 77
<code>-</code> , 148	<code>#error</code> , 79
<code>/</code> , 149, 179	<code>#eval</code> , 77
<code>^</code> , 149	<code>#exec</code> , 77
<code>→</code> , 113	<code>#file</code> , 78
<code>→array</code> , 178	<code>#if</code> , 78
<code>→arry</code> , 178	<code>#ifdef</code> , 76
<code>→col</code> , 178	<code>#ifeq</code> , 77
<code>→diag</code> , 178	<code>#ifndef</code> , 76
<code>→hms</code> , 170	<code>#ifneq</code> , 77
<code>→lcd</code> , 257	<code>#include</code> , 77
<code>→list</code> , 223	<code>#line</code> , 78
<code>→num</code> , 118, 155, 225	<code>#mode</code> , 78
<code>→q</code> , 162	<code>#undef</code> , 76
<code>→row</code> , 178	<code>#warning</code> , 79
<code>→str</code> , 219	<code>%</code> , 162
<code>→table</code> , 227	<code>%ch</code> , 162
<code>*d</code> , 256	<code>%t</code> , 162
<code>*h</code> , 256	<code>abort</code> , 73, 142
<code>*s</code> , 256	<code>abs</code> , 157
<code>*w</code> , 256	<code>acos</code> , 170
<code><</code> , 165	<code>acosh</code> , 171
<code><=</code> , 165	<code>alarm</code> , 195
<code><></code> , 165	<code>alog</code> , 173
<code>=<</code> , 165	<code>and</code> , 165, 168
<code>==</code> , 165	<code>append</code> , 234
<code>></code> , 165	<code>arg</code> , 162
<code>>=</code> , 165	<code>array→</code> , 178
<code>#date</code> , 79	<code>arry→</code> , 178
<code>#defeval</code> , 76	<code>asin</code> , 170
<code>#define</code> , 75	

INSTRUCTIONS

asinh, 171
 asl, 168
 asr, 168
 atan, 170
 atanh, 171
 atpoke, 251
 autoscale, 256
 axes, 256

 $b \rightarrow r$, 168
 $b > r$, 48
 backspace, 234
 beep, 101
 bessell, 191
 bin, 168

 $c \rightarrow r$, 162
 case, 130
 ceil, 161
 centr, 256
 cf, 139
 chr, 219
 clear, 91
 clerr, 128
 clld, 256
 close, 234, 237
 clratpoke, 251
 clrcntxt, 95
 clrfuse, 241
 clrmntx, 247
 clrsmphr, 247
 clrswi, 249
 cls, 212
 clusr, 110
 cnrm, 178
 col $\rightarrow$, 178
 col+, 178
 col-, 178
 cols, 212
 comb, 209
 con, 178
 cond, 179
 conj, 161
 cont, 144
 continue, 241
 contrôle+C, 142
 contrôle+Z, 143
 convert, 193
 copy, 92
 copyright, 199

 corr, 212
 cos, 170
 cosh, 171
 cov, 212
 create, 234
 crmtx, 247
 crmtx, 50
 cross, 178
 crsmphr, 247
 crsmphr, 50
 crtab, 227
 cstop, 241
 cstop, 50
 cswp, 178
 currenc, 219
 cycle, 133–135

 $d \rightarrow r$, 170
 daemonize, 241
 date, 195
 dec, 168
 decr, 157
 decr, 45
 default, 130
 deg, 170
 delete, 234
 depnd, 256
 depth, 91
 der, 187
 det, 179
 detach, 115, 243
 detach, 49
 dft, 185
 dgtiz, 256
 diag $\rightarrow$, 178
 disp, 97
 do, 137
 dot, 178
 draw, 261
 drax, 256
 drop, 93
 drop2, 93
 dropcntxt, 95
 dropn, 93
 drws, 212, 263
 dup, 91
 dup2, 92
 dupcntxt, 95
 dupn, 92

INSTRUCTIONS

e, 155
 edit, 94
 egv, 181
 egvl, 181
 else, 125, 128, 129
 elseif, 129
 end, 125, 128–130, 136, 137
 eng, 99
 erase, 269
 errm, 127
 errn, 127
 eval, 118, 155, 225
 exget, 225
 exit, 73, 133, 137
 exp, 173
 expm, 173
 exsub, 225
 externals, 273
 eyept, 256

 fact, 209
 false, 45, 155
 fc?, 141
 fc?c, 141
 fc?s, 141
 fft, 185
 fix, 99
 floor, 161
 for, 113, 135
 format, 234, 237
 fp, 160
 fs?, 141
 fs?c, 141
 fs?s, 141
 function, 261
 fuse, 241

 gamma, 191
 gegv, 181
 gegvl, 181
 get, 178, 221, 223, 227
 getc, 178
 geti, 178, 221, 223
 getr, 178
 glegv, 181
 glsq, 179
 gregv, 181

 halt, 143
 head, 223

 help, 199
 hex, 168
 histogram, 263
 hms+, 170
 hms–, 170
 hms→, 170

 i, 155
 idft, 185
 idn, 178
 if, 125, 129
 iferr, 128
 iferr, 49
 ifft, 185
 ift, 126
 ifte, 126
 im, 161
 in, 165
 incr, 157
 incr, 45
 indep, 256
 input, 100
 inquire, 234
 int, 187
 inv, 149, 179
 ip, 160
 iswi, 249
 itrace, 199

 jdate, 195

 key, 101
 keylabel, 256
 keytitle, 256
 kill, 73, 142
 kind, 43

 label, 256
 last, 91
 lcase, 219
 lcd→, 257
 lchol, 181
 legv, 181
 line, 259
 list→, 223
 ln, 173
 lnpi, 173
 localization, 219
 lock, 234
 log, 173

INSTRUCTIONS

logger, 197
 logscale, 256
 lq, 181
 lsq, 179
 lu, 181

 mant, 161
 mark, 259
 max, 158, 178
 maxs, 212
 Mclrin, 187
 mean, 212
 mem, 199
 min, 158, 178
 mins, 212
 mod, 158
 mtxlock, 247
 mtxstatus, 247
 mtxtrylock, 247
 mtxunlock, 247

 neg, 157
 newplane, 259
 next, 134, 135
 not, 165, 168
 nrand, 207
 nrproc, 50, 127, 241
 ns, 212
 num, 219

 obget, 225
 obsub, 225
 oct, 168
 open, 234, 237
 open, 48
 or, 165, 168
 over, 92

 p→r, 162
 paper, 269
 parameter, 110
 parametric, 261
 pcov, 212
 peek, 251
 perm, 209
 persist, 256
 pi, 156
 pick, 92
 plot, 259
 plotter, 263

 pmax, 256
 pmin, 256
 poke, 251
 pos, 219, 223, 227
 ppar, 256
 pri, 269
 print, 269
 private, 117
 prlcd, 269
 prmd, 269
 prompt, 100
 protect, 110
 prst, 269
 prstc, 269
 prusr, 269
 prvar, 269
 psdev, 212
 pshcntxt, 95
 pshprfl, 201
 pulcntxt, 95
 pulprfl, 201
 purge, 110
 put, 178, 221, 223, 227
 putc, 178
 puti, 178, 221, 223
 putr, 178
 pvar, 212

 qr, 181

 r→b, 168
 r→c, 162
 r→d, 170
 r→p, 162
 r->b, 48
 rad, 170
 rand, 207
 rank, 179
 rceq, 256
 rci, 178
 rcij, 178
 rcl, 110
 rclf, 139
 rcls, 212
 rclswi, 249
 rcws, 168
 rdate, 195
 rdgn, 207
 rdm, 178
 rdz, 207

INSTRUCTIONS

re, 161
 read, 234, 237
 recall, 231
 recode, 219
 recv, 251
 redraw, 256
 regv, 181
 relax, 158
 remove, 273
 remove, 49
 repeat, 136
 repl, 219, 223
 res, 256
 return, 141
 revlist, 223
 rewind, 234
 rfuse, 241
 rl, 168
 rlb, 168
 rnd, 100
 rnrm, 178
 roll, 94
 rolld, 94
 row→, 178
 row+, 178
 row-, 178
 rpl-core, 69
 rpl-out, 69
 rpl-profile, 69
 rr, 168
 rrb, 168
 rsd, 179
 rstop, 241
 rstop, 50
 rswp, 178

 s+, 212
 s-, 212
 same, 165
 save, 109
 scale, 256
 scatter, 263
 sched, 241
 schur, 181
 sci, 99
 scls, 212, 263
 sconj, 121
 sdev, 212
 seek, 234
 select, 130

 send, 251
 sf, 139
 shared, 114
 SIGINT, 142
 sign, 158
 sin, 170
 sinh, 171
 sinv, 120
 size, 178, 219, 221, 223, 225, 227
 sl, 168
 slb, 168
 slice, 261
 slicescale, 256
 smphrdecr, 247
 smphrgetv, 247
 smphrincl, 247
 smphrtrydecr, 247
 sneg, 121
 sort, 223, 227
 spawn, 49
 spar, 212
 spawn, 245
 spawn, 49
 splash, 199
 sq, 152, 178
 sqlconnect, 235
 sqlconnect, 50
 sqldisconnect, 235
 sqlquery, 235
 sqrt, 152
 sr, 168
 srb, 168
 sst, 143
 start, 134
 static, 114
 std, 98
 step, 134, 135
 steq, 256
 STGTSTP, 143
 sto, 109, 118
 sto*, 120
 sto+, 119
 sto-, 119
 sto/, 120
 stof, 139
 stop, 241
 store, 231
 stos, 212
 stoswi, 249
 str→, 219

Toutes copies ou impressions mêmes parties de cette documentation sont formellement interdites. Tous droits réservés à l'auteur sans limitation de durée.

INSTRUCTIONS

stws, 141, 168
 stws, 47
 sub, 219, 223
 suspend, 241
 svd, 181
 svl, 181
 swap, 93, 94
 swapcntxt, 95
 swi, 249
 swilock, 249
 swiqueue, 249
 swistatus, 249
 swiunlock, 249
 sx, 212
 sx2, 212
 sxy, 212
 sy, 212
 sy2, 212
 sync, 234
 syseval, 197

 table→, 227
 tail, 223
 tan, 170
 tanh, 171
 target, 237
 taylr, 187
 then, 125, 128–130
 time, 197
 title, 256
 tokenize, 219
 tot, 212
 trim, 219
 trn, 178
 trnc, 100
 true, 45, 155
 type, 43
 ucase, 219

 uchol, 181
 unlock, 234
 unprotect, 110
 until, 137
 use, 273
 utpc, 213
 utpf, 213
 utpn, 213
 utpt, 213

 var, 212
 variable, 110
 verify, 203
 version, 199
 visit, 110
 volatile, 117

 wait, 195
 warranty, 199
 wfack, 251
 wfdata, 251
 wflock, 234
 wfpoke, 251
 wfproc, 241
 wfsock, 237
 wfswi, 249
 while, 136
 wireframe, 261
 workdir, 197
 write, 234, 237

 xcol, 212
 xor, 165, 168
 xpon, 161
 xroot, 152

 ycol, 212
 yield, 241

Index général

- Atome, 86
- Bibliothèque, *voir* type de donnée
- Binaire, *voir* type de donnée
- Bloc, 51
- Booléen, *voir* type de donnée
- Boucles
 - définies, 133
 - indéfinies, 133
- Changement de contexte, *voir* contexte
- Chaîne de caractères, *voir* type de donnée
- Commentaires, 75
- Complexe, *voir* type de donnée
- Connecteur SQL, *voir* type de donnée
- Contexte, 94
- contexte, seeexécution52
- Définition, 61
 - extrinsèque, 61
 - intrinsèque, 61
 - utilisateur, 61
- Entier, *voir* type de donnée
- Entrées et sorties, 97
- Erreur, *voir* reprise sur erreur
- Expression
 - algébrique, *voir* type de donnée
 - RPN, *voir* type de donnée
- Exécution
 - fil, 52
 - niveau, 51
 - processus, 53
 - processus léger, 53
- Fichier, *voir* type de donnée
- Fil d'exécution, 52
- Format
 - fixe, 99
 - ingénieur, 99
 - scientifique, 99
 - standard, 98
- Indétermination, *voir* résultat
 - indéterminé
- Infini, *voir* résultat infini
- Interruption au clavier, 142, 143
- Ligne de commande, 71
- Liste, *voir* type de donnée
- Matrice, *voir* type de donnée
- Mutex, *voir* type de donnée
- NaN, 45
- Niveau
 - d'exécution, 51, 55
 - de variable, 55
- Niveau d'exécution, 103
- Nom, *voir* type de donnée
- Nom symbolique, *voir* types de données
- Nom évaluable, *voir* types de données
- Norme de Frobenius, 157
- Notation

INDEX GÉNÉRAL

- algébrique, 42
- infixe, 42
- polonaise inverse (RPN), 41
- Pile last, 72, 75, 91
- Pile opérationnelle, 85
 - dépilement, 85
 - empilement, 85
- Processus, *voir* type de donnée
- Processus, 53
 - détaché, 243
 - fusible, 241
 - léger, 53, 245
- Reprise sur erreur, 127
- RPL/2, 49
- RPL/C, 61
- RPN, *voir* notation
- Réel, *voir* type de donnée
- Résultat
 - indéterminé, 45
 - infini, 45
- Scalaire, *voir* types de donnée
- Socket, *voir* type de donnée
- Sémaphore nommé, *voir* type de donnée
- Séparateur décimal, 98
- Table, *voir* type de donnée
- Type de donnée
 - bibliothèque, 49
 - chaîne de caractères, 47
 - complexe, 45
 - connecteur SQL, 50
 - entier, 45
 - entier binaire, 47
 - exportable, 43
 - expression algébrique, 46
 - expression RPN, 46
 - fichier, 48
 - liste, 46
 - matrice, 46
 - mutex, 50
 - nom, 47
 - processus, 49
 - réel, 45
 - scalaire, 43
 - socket, 48
 - sémaphore nommé, 50
 - table, 46
 - vecteur, 45
- type de donnée
 - non exportable, 48
- Types de donnée
 - booléen, 45
- Types de données, 43
 - nom
 - symbolique, 104
 - évaluable, 104
- Variable
 - automatique, 56
 - classe, 55
 - globale, 56
 - héritage, 59
 - implicite, 56
 - interne, 56
 - locale, 56
 - masque, 57
 - niveau, 55
 - nom, 59
 - partagée, 57
 - portée, 58
 - privée, 56
 - statique, 56
 - verrouillage, 59
 - virtuelle, 57
 - visibilité, 58
 - volatile, 56
- Variables, 55
 - accessibilité, 103
 - globales, 109
 - locales, 113
 - partagées, 114
 - portée, 116
 - statiques, 114
 - visibilité, 116
- Vecteur, *voir* type de donnée