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

Reverse Polish Lisp/2

— Manuel de référence —

Bertrand Joël

8 février 2010

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

2

RPL/2

Copyright © 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010 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.

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

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

4

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

Table des matières

Avant-propos, de la machine de Turing au RPL/2 **13**

Genèse **15**

Historique **16**
RPL 6502, version 1.0 17
RPL 6809, version 2.0 17
RPL 386, version 3.0 17
RPL 387, version 3.x 18
RPL/2 ou RPL 4.x, une renaissance 18

Des avantages du RPL/2 **18**

I Concepts fondamentaux **21**

1 Notations **23**
1.1 Notation polonaise inverse 23
1.2 Notation algébrique 24
1.3 Notation infixe 24
1.4 Commentaires 24

2 Types de données **25**
2.1 Scalaires 25
2.1.1 Booléens 25
2.1.2 Entiers 25
2.1.3 Réels 26
2.1.4 Complexes 26
2.2 Vecteurs 26
2.3 Matrices 26
2.4 Listes 26
2.5 Tables 27
2.6 Expressions 27

2.6.1 Expressions algébriques . . . 27
2.6.2 Expressions RPN . . . 27
2.7 Noms . . . 27
2.8 Chaînes de caractères . . . 28
2.9 Binaires . . . 28
2.10 Fichiers . . . 28
2.11 Sockets . . . 29
2.12 Bibliothèques . . . 29
2.13 Processus . . . 29
2.14 Connecteurs SQL . . . 30
2.15 Mutexes . . . 30
2.16 Sémaphores nommés . . . 30

3 Variables **31**
3.1 Définitions . . . 32
3.1.1 Définitions intrinsèques . . . 33
3.1.2 Définitions extrinsèques . . . 33
3.1.3 Définitions utilisateur . . . 33
3.2 Variables globales . . . 33
3.3 Variables locales . . . 34
3.3.1 Variables volatiles . . . 35
3.3.2 Variables statiques . . . 35
3.3.3 Variables partagées . . . 36
3.4 Variables virtuelles . . . 36
3.5 Verrouillage . . . 36
3.6 Héritage . . . 36

II Appel du RPL/2 **37**

4 Ligne de commande **39**
4.1 Options de la ligne de commande . . . 40
4.2 Fonctionnement interactif . . . 41

5 Exécution de programmes **43**
5.1 Préprocesseur . . . 43
5.1.1 `#define x y#define` . . . 43
5.1.2 `#defeval x y#defeval` . . . 44
5.1.3 `#undef x#undef` . . . 44
5.1.4 `#ifdef x#ifdef` . . . 44
5.1.5 `#ifndef x#ifndef` . . . 45
5.1.6 `#ifeq x y#ifeq` . . . 45
5.1.7 `#ifneq x y#ifneq` . . . 45
5.1.8 `#else#else` . . . 45
5.1.9 `#endif#endif` . . . 45
5.1.10 `#include "file"#include` . . . 45
5.1.11 `#exec command#exec` . . . 45
5.1.12 `#eval expr#eval` . . . 46
5.1.13 `#if expr#if` . . . 46
5.1.14 `#elif expr#elif` . . . 46

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

5.1.15 `#mode keyword#mode` . . . 46
5.1.16 `#line#line` . . . 47
5.1.17 `#file#file` . . . 47
5.1.18 `#date fmt#date` . . . 47
5.1.19 `#error msg#error` . . . 47
5.1.20 `#warning msg#warning` . . . 47
5.2 Organisation des programmes . . . 47
5.3 Extension . . . 48

III Manipulations de la pile **49**

6 Modifications automatiques **51**
6.1 Fonctionnement des routines d'évaluation . . . 52
6.1.1 Cas d'un programme interprété . . . 53
6.1.2 Cas d'un programme compilé . . . 55
6.2 Opérations implicites . . . 55
6.3 Opérations explicites . . . 55

7 Manipulation des objets **57**
7.1 Gestion de la pile . . . 57
7.1.1 Clear . . . 57
7.1.2 Depth . . . 57
7.1.3 Last . . . 57
7.2 Duplication d'éléments . . . 57
7.2.1 Dup . . . 57
7.2.2 Dup2 . . . 58
7.2.3 Dupn . . . 58
7.2.4 Over . . . 58
7.2.5 Pick . . . 58
7.3 Supression d'éléments . . . 59
7.3.1 Drop . . . 59
7.3.2 Drop2 . . . 59
7.3.3 Dropn . . . 59
7.4 Modification de la hiérarchie . . . 59
7.4.1 Swap . . . 59
7.4.2 Rot . . . 59
7.4.3 Roll . . . 60
7.4.4 Rolld . . . 60
7.5 Gestion des contextes . . . 60
7.5.1 Pshcntxt . . . 60
7.5.2 Pulcntxt . . . 60
7.5.3 Dupcntxt . . . 61
7.5.4 Dropcntxt . . . 61
7.5.5 Swapcntxt . . . 61
7.5.6 Exemple d'utilisation . . . 61

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

8 Entrées et sorties **63**
8.1 Sorties . . . 63
8.1.1 Disp . . . 63
8.1.2 Format . . . 63
8.1.3 Clmf . . . 66
8.2 Entrées . . . 66
8.2.1 Input . . . 66
8.2.2 Prompt . . . 66
8.2.3 Key . . . 66
8.3 Beep . . . 67

IV Manipulation des variables **69**

9 Accessibilité des variables **71**
9.1 Niveaux d'exécution . . . 71
9.2 Évaluation implicite . . . 72
9.3 Évaluation explicite . . . 72

10 Variables globales **75**
10.1 Création . . . 75
10.2 Accès . . . 76
10.3 Modification . . . 76
10.4 Libération . . . 76
10.5 Verrouillage . . . 76

11 Variables locales **79**
11.1 Création . . . 79
11.2 Portée et visibilité . . . 82
11.3 Modification . . . 84

12 Arithmétique directe **85**
12.1 Les quatre opérations . . . 85
12.1.1 Addition . . . 85
12.1.2 Soustraction . . . 85
12.1.3 Multiplication . . . 86
12.1.4 Division . . . 86
12.2 Autres opérations . . . 86
12.2.1 Inversion . . . 86
12.2.2 Opposition . . . 87
12.2.3 Conjugaison . . . 87

V Contrôle **89**

13 Conditions et tests **91**
13.1 Tests simples . . . 91
13.1.1 If...then...(else)...end . . . 91
13.1.2 Ift . . . 92
13.1.3 Ifte . . . 92
13.2 Reprise sur erreur . . . 93

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

13.2.1 Types d'erreurs . . . 93
13.2.2 Errn . . . 93
13.2.3 Errm . . . 94
13.2.4 Clerr . . . 94
13.2.5 Iferr...then...(else)...end . . . 94
13.3 Tests multiples . . . 96
13.3.1 If...then...elseif...then...(else)...end . . . 96
13.3.2 Select...case...then...end...(default)...end . . . 96

14 Boucles **99**
14.1 Boucles définies . . . 99
14.1.1 Boucle sans compteur . . . 100
14.1.2 Boucle avec compteur . . . 101
14.1.3 Exemples . . . 102
14.2 Boucles indéfinies . . . 103
14.2.1 While...repeat...end . . . 103
14.2.2 Do...until...end . . . 103
14.3 Instruction exit . . . 103

15 Contrôle de l'exécution **105**
15.1 Mode de fonctionnement . . . 105
15.1.1 Indicateurs . . . 105
15.1.2 Manipulation . . . 105
15.2 Exécution normale . . . 107
15.2.1 Retour anticipé . . . 107
15.2.2 Abandon . . . 108
15.3 Débogage . . . 109
15.3.1 Point d'arrêt . . . 109
15.3.2 Exécution pas à pas . . . 110
15.3.3 Retour en exécution normale . . . 110

VI Arithmétique **111**

16 Les opérations de base **113**
16.1 Notations . . . 113
16.2 Addition . . . 113
16.3 Soustraction . . . 114
16.4 Multiplication . . . 114
16.5 Division . . . 114
16.5.1 Division standard . . . 114
16.5.2 Inversion . . . 116
16.6 Puissance . . . 116
16.6.1 Puissance standard . . . 116
16.6.2 Carré . . . 116
16.6.3 Racine carrée . . . 116
16.6.4 Racine $n^{\text{ième}}$ . . . 116

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

17 Arithmétique générale **119**
17.1 Proportions . . . 119
17.1.1 Instruction % . . . 119
17.1.2 Instruction %CH . . . 119
17.1.3 Instruction %T . . . 119

VII Trigonométrie et logarithmes **121**

18 Fonctions trigonométriques **123**
18.1 Cosinus . . . 123
18.2 Sinus . . . 123
18.3 Tangente . . . 123

19 Fonctions hyperboliques **125**
19.1 Cosinus hyperbolique . . . 125
19.2 Sinus hyperbolique . . . 125
19.3 Tangente hyperboliques . . . 125

20 Fonctions logarithmiques **127**
20.1 Logarithme naturel . . . 127
20.2 Logarithme vulgaire . . . 127

VIII Algèbre linéaire **129**

21 Vecteurs et matrices **131**

22 Résolution **133**
22.1 Inversion . . . 133
22.2 Système linéaire . . . 133
22.3 Moindres carrés . . . 133
22.4 Moindres carrés généralisés . . . 133

23 Décompositions **135**
23.1 Vecteurs propres . . . 135
23.2 Vecteurs propres généralisés . . . 135
23.3 Décomposition de Cholesky . . . 135
23.4 Décomposition LDU . . . 135
23.5 Décomposition LU de Crout . . . 135
23.6 Décomposition LQ . . . 135
23.7 Décomposition QR . . . 135
23.8 Décomposition de Schur . . . 135

IX Statistiques **137**

24 Description **139**
24.1 Matrice de statistique . . . 139
24.2 Moyenne . . . 139
24.3 Variance et écart-type . . . 139

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

25 Dénombrement **141**

26 Lois de probabilité cumulées **143**
26.1 Distribution de Laplace-Gauß dite normale 143
26.2 Distribution du χ^2 143
26.3 Distribution de Fisher 143
26.4 Distribution de Student 143

X Listes et tables 145

XI Fichiers 147

27 Variable virtuelle **149**

28 Fichiers à accès séquentiel **151**

29 Fichiers à accès direct **153**

XII Accès au réseau 155

XIII Processus 157

30 Processus détachés **159**

31 processus légers **161**

XIV Graphisme 163

32 Dessin **165**

33 Graphiques à deux dimensions **167**

34 Graphiques à trois dimensions **169**

35 Graphiques de statistiques **171**
35.1 Nuages de points 171
35.2 Histogrammes 171

XV Impression 173

XVI Optimisations 175

36 Du bon usage des variables **177**
36.1 Création de variables 177
36.2 Utilisation de la pile 177

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

37 Bibliothèques partagées **179**
37.1 Extensions compilées au RPL/2 179
37.2 Interfaces 179
37.3 Fonctions accessibles 179

XVII Exemples **181**

38 Programmes RPL/2 **183**
38.1 Premier exemple simple 183
38.2 Programme complexe 185

39 Bibliothèques **201**

Index **204**
Instructions 205
Index général 207

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

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

Avant-propos
De la machine de Turing au RPL/2

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

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

Genèse

Lorsqu'au début des années 1990, j'ai commencé à écrire des algorithmes de traitement du signal, je me suis aperçu rapidement qu'il n'existait aucun langage informatique de formalisme mathématique capable de les implanter efficacement. La mode était aux langages comme le C++ ou Java, voire aux outils comme Matlab. Le point commun de tous ces outils est leur ajout de nombreuses couches d'abstractions entre le matériel et le programme exécuté. La gestion des ressources est alors particulièrement inefficace. En effet, lorsqu'on travaille sur des matrices carrées de cent mille par cent mille éléments complexes, chaque élément étant codé sur cent vingt-huit bits, on ne peut pas se permettre de laisser à la discrétion du langage des opérations aussi sensible que la libération de la mémoire, celle-ci étant faite par un ramasse-miette asynchrone aux processus de calcul. D'autres langages comme les Fortran 90 ou 95 étaient prometteurs, mais les compilateurs suivant la norme n'étaient ni matures ni disponibles aisément, et rendaient les programmes non portables. J'avais alors sur mon bureau une station *Alpha* sans aucun outil pour l'utiliser pleinement.

D'autre part, la plupart des langages existants génèrent des dérives numériques plus ou moins importantes qui deviennent d'autant plus problématiques que les algorithmes sont itératifs. Ceux qui ne provoquent pas de telles dérives ne sont pas efficaces soit parce que les temps d'exécution deviennent prohibitifs — l'outil embarque plusieurs algorithmes différents et effectue un choix du meilleur algorithme à utiliser en fonction de la structure des arguments —, soit parce que la syntaxe devient rapidement complexe en raison de l'optimisation des routines internes.

Ces deux raisons m'ont conduit à concevoir un langage de haut niveau qui devait au moins satisfaire aux conditions suivantes :

- une portabilité sur tout système POSIX ;
- l'utilisation simultanée de tous les processeurs d'un calculateur ;
- la manipulation des objets mathématiques de base (matrices, listes, expressions alébriques...) ;
- une utilisation raisonnée de la mémoire ;
- une mécanisme d'interface avec d'autres langages ;
- une vitesse conséquente d'exécution.

Je désirais aussi que ce langage contienne un mécanisme de gestion des erreurs ainsi qu'une gestion dynamique de la mémoire sans que l'utilisateur final n'ait à sa disposition des primitives d'allocation ou de libération de cette mémoire.

L'ensemble de ces conditions conduit à une complexité que n'ont pas les langages traditionnels. Son coût 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 RPL [1] développé par Hewlett-Packard pour son calculateur HP-28S, mais il a été considérablement amélioré pour permettre une écriture aisée de programmes massivement parallèles. Ce langage s'apparente à la fois au Forth et au Lisp, mais il garde le meilleur des deux mondes. Il utilise par exemple la notation infixe pour ses structures de contrôle contrairement au Forth, ce qui rend les programmes écrits bien plus lisibles.

Les principaux avantages de ce langage sont sa portabilité, car celui-ci fonctionne sur tout calculateur ou grappe de calculateurs fonctionnant sous un système d'exploitation de type POSIX, sa compacité et sa rapidité d'exécution, l'absence de toute gestion de la mémoire par l'utilisateur, celle-ci étant laissée à la discrétion de l'interpréteur mais restant synchrone, ce qui permet de se concentrer sur l'algorithme et non sur sa transcription informatique, ses possibilités d'extensions par des fonctions écrites dans un langage proche du C, et ses capacités de manipulation d'objets mathématiques. Il supporte la parallélisation des algorithmes et la transmission de données au travers de sockets locales de type unix ou réseau de type IPv4 ou IPv6.

1. pour Reverse Polish Lisp

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

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 plus une curiosité informatique, un langage ésotérique ou une prouesse de programmation sans grand intérêt qu'un outil réellement utilisable. En effet, le but des premières versions étaient plus de copier le RPL originel que d'y apporter les fonctionnalités manquantes à une utilisation efficace.

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. Il ne travaillait 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.

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 mémoire au delà de 64 Ko n'étaient accessibles qu'au travers de tableaux virtuels sur un système de fichiers en mémoire. Cette station fonctionnait sous TSC-Flex9 et le séquenceur 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 u débuté l'écriture d'une troisième version du séquenceur 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.

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

Le séquenceur reste toujours écrit en Basic, mais cette fois-ci en Turbo-Basic V1 sous IBM-DOS 5.00.

RPL 387, version 3.x

L'ajout d'un i387DX dans la machine de développement permet l'adjonction de fonctions utilisant le coprocesseur arithmétique. Au regard du gain apporté par ce composant, il est possible d'effectuer les calculs en double précision sur l'ensemble des complexes. Au reste, pour simplifier la gestion des différents objets, tous les calculs se font sur des nombres complexes au détriment des performances. Il n'est toujours pas possible d'utiliser dynamiquement la mémoire et tous les objets sont traités sous forme de chaîne de caractères. Le séquenceur est porté sous OS/2 Warp 3 et 4.

RPL/2 ou RPL 4.x, une renaissance

Début 1998, le séquenceur se trouve entravé par des décisions prises une dizaine d'années plus tôt et motivées par les stations de calcul alors disponibles. Les puissances maintenant utilisables rendaient caducs le choix de travailler en mémoire statique et le fait d'effectuer tous les calculs sur des nombres complexes. Par ailleurs, le projet GNU mettant à ma disposition une collection de compilateurs C et Fortran portables et efficace, je décide de réécrire totalement l'interprète de manière à obtenir un outil portable et sans aucune limite autre que les limites des calculateurs hôtes en terme de mémoire. Toute la mémoire est gérée dynamiquement et surtout de façon synchrone. Les niveaux de variables, qui peuvent être volatiles, statiques ou partagées, verrouillées ou non, font leur apparition. Un préprocesseur est ajouté ainsi qu'un compilateur, des routines d'impressions et une gestion des threads et des processus.

Un programme RPL/2 peut maintenant être appelé à partir d'un autre langage pour peu qu'il soit possible d'avoir une convention d'appel de type C, et il est aussi possible d'ajouter de nouvelles fonctions au langage sous la forme de bibliothèques partagées.

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 compilés. Cette 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.

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

Des avantages du RPL/2

Le RPL/2 est un langage interprété ou compilé, Turing equivalent, de haut niveau et permettant de constamment travailler à la précision maximale disponible tout en évitant les débordements numériques inhérents aux différentes représentations utilisées dans les calculateurs. Très véloce, grâce à l'usage de la notation polonaise inverse, il est capable d'effectuer — en plus de l'arithmétique complexe — des calculs symboliques, vectoriels et matriciels.

Ce langage est à la fois faiblement et fortement typé. Fortement, car il existe une foultitude de types de données distincts, mais faiblement, car l'association de ces types aux diverses données se fait implicitement. En effet, un objet acquiert un type lors de sa première manipulation. De la même façon, une variable n'existe qu'après sa première affectation et n'a pas besoin d'être déclarée. Avant sa première affectation, elle n'existe simplement pas, ce qui permet de mémoriser sous le même nom des objets de types différents au cours d'un programme. Il est aussi faiblement typé, car une donnée peut changer de type au cours d'une opération pour que la cohérence des calculs soit maintenue, par exemple passer d'une représentation entière vers une représentation réelle si le besoin s'en fait sentir. Ce transtypage est implicite et empêche tout débordement de capacité. Il faut noter que les calculs sont effectués au moindre coût et à la précision maximale, ce qui revient à dire que le type associé à une donnée est toujours le type minimal permettant de la représenter correctement.

Le séquenceur RPL/2 se veut d'usage le plus général possible. Il possède 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 généré par un moteur TEX, des instructions de gestion de fichiers et de sockets, 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.

Le RPL/2 évite aussi tous les problèmes d'allocation de mémoire. Cette gestion — dynamique et laborieuse dans la plupart des langages — est laissée à la discrétion du séquenceur. Elle est totalement transparente et assujettie à un

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

mécanisme de protection des accès mémoire, les seules erreurs pouvant alors survenir étant des erreurs de conception des programmes. Ces erreurs ne provoquent pas obligatoirement un arrêt du programme — sauf s'il s'agit d'une erreur de type système — car il est possible de les gérer grâce à un mécanisme de reprise sur erreur. Si une erreur n'est pas masquée, elle provoque un arrêt anormal de l'exécution et peut entraîner la création d'un fichier de trace contenant un certain nombre d'informations relative à cette erreur. Un mode de fonctionnement en pas à pas peut alors aider à la correction de ce dysfonctionnement.

Le RPL/2 gère nativement des bases de données de type SQL ainsi que des fichiers de type formatés ou non formatés (fichiers embarquant pour chaque enregistrement ses informations de type rendant impossible une mauvaise lecture), et des fichiers de type flux (données brutes sans information annexe) accessibles de manière séquentielle, directe our par index.

Des fonctions de haut niveau permettent de gérer différentes tâches parallèles au sein d'un même programme, chaque tâche se déroulant dans un contexte qui lui est propre. La parallélisation des algorithmes sur un calculateur n'est alors plus qu'un problème intrinsèque à l'algorithme et non plus un problème de transciption dans un langage donné. L'utilisation de fonctions réseau de haut niveau — les sockets sont vues comme des fichiers — permet d'étendre ces capacités de parallélisation à des grappes de calculateurs.

Enfin, il est possible d'utiliser le séquenceur 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.

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

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

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

1 Notations

Le fonctionnement interne du RPL/2 se fait exclusivement en notation polonaise inversée. Pour des raisons d'efficacité, cette notation sera la logique principale utilisée par les programmes écrits en RPL/2. Néanmoins, pour des raisons de lisibilités, deux autres notations sont utilisées : 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 Łukasiwiecz (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 Łukasiwiecz 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, où ils peuvent à leur tour être utilisés comme arguments d'autres commandes.

Le RPL/2 utilise 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. 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. Celà 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 formel, il reste

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

possible d'utiliser des expressions en notation algébrique mais au détriment de la vélocité du langage. L'un des avantages essentiel résidant 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 expressions algébriques classiques et manipulations en notation polonaise inverse.

La pile utilisée par le RPL/2 n'est pas limitée en taille. Les entrées et sorties de la pile se font exclusivement par le niveau 1. 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 lors de l'empilage initial de l'objet.

1.3 Notation infixe

Certaines fonctions du langage travaillent en notation infixe 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.

1.4 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.

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

2 Types de données

Les données manipulées par le RPL/2 sont typées. Cependant, l'affectation d'un type à un objet se fait de manière automatique et non par déclaration préalable. Il faut aussi 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 passera en réel voire en complexe.

Pour des raisons de performances, les objets ne sont copiés que lors de modifications. 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.

2.1 Scalaires

2.1.1 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.

2.1.2 Entiers

La longueur nominale des entiers manipulés par le séquenceur est de 64 bits signés. Cette longueur ne peut varier en fonction de l'architecture de la machine utilisée. Néanmoins, il existe un mécanisme de traitement des débordements entiers rendant les algorithmes portables quelle que soit l'architecture de la machine. Si un calcul ne peut se poursuivre en entier du fait de l'impossibilité de représenter une grandeur en entier, celle-ci est automatiquement convertie en réel.

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

2.1.3 Réels

Les réels utilisés sont par défaut des flottants codés sur 64 bits. Cette longueur ne dépend pas de l'architecture du système hôte. Trois constantes symboliques — sur lequelles 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 $\pm$`(MANT)E`$\pm$`(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.

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.

2.1.4 Complexes

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

2.2 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 `...` s_n `]`.

2.3 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} `...` s_{1n} `]...[`s_{m1} s_{m2} `...` 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.

2.4 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 `{` $objet_1$ $objet_2$ `...` $objet_n$ `}`.

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

2.5 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 l'accolade, ce qui fait qu'une table sera notée `<[ objet1 objet2 ... objetn ]>`.

2.6 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.

2.6.1 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 `'SIN(PI*x)/(PI*x)'`. Ces expressions dont l'évaluation est coûteuse en terme de calcul sont transformées en notation polonaise inverse par le séquenceur. Ainsi, il est préférable d'utiliser ces expressions avec parcimonie, ou de les remiser dans des variables si ces expressions apparaissent dans des boucles, la transformation en notation polonaise inverse ne s'effectuant qu'une seule fois lors de la mémorisation de l'objet dans la variable.

2.6.2 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 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 véloce que la précédente puisque comportant plus d'opérations aritmétiques et d'appel de variables.

2.7 Noms

Les noms sont des objets particuliers représentant une variable de façon symbolique, cette variable pouvant le cas échéant ne pas exister. Ils sont délimité grâce aux apostrophes. La variable x sera ainsi représentée par `'x'`.

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

2.8 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.

2.9 Binaires

Les binaires sont des entiers non signés d'une longueur de 64 bits quelle que soit l'architecture du système hôte. 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.

Il faut noter qu'aucun mécanisme de contrôle de débordement ne vérifie les calculs portant sur des entiers binaires.

2.10 Fichiers

Le type fichier est un type composite contenant une structure de données décrivant le fichier, son type, son état et le format de lecture ou d'écriture d'un enregistrement. Tout objet de type fichier est créé par l'instruction `OPEN`. Le format est associé au descripteur de fichier par l'instruction `FORMAT`. Il est impossible de créer un objet de type fichier d'une autre façon.

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 est déduit 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 sont 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 pour relire un enregistrement. Ils ne sont donc plus portables d'une architecture à une autre et surtout, ils requièrent un format en écriture et en lecture.

Les fichiers disponibles pour les types formatés et non formatés sont à accès séquentiel, direct ou indexé sur une clef. Pour les fichiers de type flux, ils ne sont qu'à accès séquentiel ou direct. Il faut noter que 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.

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

2.11 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`. Le cas échéant, un format lui est associé par `FORMAT`. Les sockets non fermées à la fin du fil d'exécution qui les ont créées sont autoritairement closes par le RPL/2.

2.12 Bibliothèques

Le RPL/C est un langage de macroinstructions du langage C 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 comportent 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 `USE`. Les bibliothèques sont automatiquement fermées à la fin du fil d'exécution qui les ont ouvertes.

2.13 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.

Les processus forment un arbre, chaque processus ou processus léger ayant deux canaux de communication, l'un pour émettre des données à destination de son père, l'autre pour envoyer des données à destination de son fils. Un processus fils peut envoyer une interruption à son père.

Un processus d'achève normalement lorsque son fil d'exécution est vide 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 `NRPROC`.

Lorsqu'un processus s'achève, tous les processus fils recoivent 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.

2.14 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 caractère. 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 ne peut être créé que par l'instruction `SQLCONNECT`. Tous les connecteurs de bases sont autoritairement clos à la fin du fil d'exécution les ayant créés.

2.15 Mutexes

Un programme partageant plusieurs processus légers peut demander 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 ls mutexes sont partageables entre des processus légers. Ils ne sont pas partageables entre des processus qui s'exécutent dans des environnements séparés. Lorsqu'un mécanisme de synchronisation entre processus est requis, il convient d'utiliser des sémaphores nommés.

2.16 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.

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

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

3 Variables

Si les définitions intrinsèques ne sont pas sensibles à la casse, les variables le sont. En RPL/2, tout ce qui n'est pas une instruction intrinsèque est un objet soumis à évaluation et pouvant être sauvegardé et rappelé symboliquement par un objet de type nom. Les variables ne sont ni déclarées ni typées, elles sont créées automatiquement par un programme et ne contiennent qu'une référence sur l'objet à sauvegarder et un niveau qui lui donnera une visibilité et une durée de vie.

Une variable est locale, ou virtuelle ou globale. Lorsqu'elle est locale, elle peut être partagée, statique ou volatile. Les variables sont verrouillables, ce qui permet entre autre la création de constantes et de se prémunir contre toute modification ultérieure. Toute variable locale est visible à l'intérieur du bloc de programme qui l'a créée et automatiquement masquée à sa sortie. Si la variable était volatile, elle est détruite. Une variable globale est accessible depuis l'ensemble du programme mais n'est pas partagée entre deux processus même légers. La seule façon de partager une variable entre plusieurs processus légers est l'utilisation d'une variable partagée. Entre plusieurs processus, il convient d'utiliser une variable virtuelle encadrée par des sémaphore.

Les variables sont par défaut volatiles. Elles sont effacées à la fin du bloc de programme qui les ont créés. Cependant, il est possible de créer des variables statiques ou partagées. Une variable statique est sauvegardée à la fin de l'exécution du bloc et non effacée pour reprendre sa dernière valeur lors de la nouvelle exécution du bloc. Elle n'est pas visible à l'extérieur du bloc et disparaît à la fin du fil d'exécution courant. Une variable partagée est une variable statique partagée entre différents processus légers.

Il peut cohabiter simultanément plusieurs variables portant le même nom sans que cela ne compromette le bon fonctionnement du système et sans perte d'information. Chaque variable est associée à un niveau affecté par le séquenceur lors de sa création. Lorsqu'une variable est appelée par son nom, le RPL/2 renvoie la variable visible de plus haut niveau. Ces niveaux correspondent à :

- 0 pour les diverses définitions introduites dans le programme. Une variable de niveau 0 n'est pas modifiable par l'utilisateur et n'est donc pas copiée lors de la création d'un processus léger ;
- 1 pour les variables globales ;

– un niveau strictement supérieur à 1 pour toute variable variable locale.

Une variable virtuelle est une variable qui apparaît dans le système de fichiers de la machine hôte. Elle est accessible à tout programme RPL/2 à l'instar des sémaphores nommés et n'est pas détruite à la fin du fil d'exécution courant.

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.

3.1 Définitions

Une définition est une expression en notation polonaise inversée prenant ses arguments sur la pile et y renvoyant ses résultats. Elle est associé à un nom sensible à la casse et l'identifiant de façon unique.

```
0001 Ceci_est_ma_premiere_definition
0002 <<
0003     "C'est un rêve modeste et fou" disp
0004     "Il aurait mieux valu le taire" disp
0005     "Vous me mettrez avec en terre" disp
0006     "Comme une étoile au fond d'un trou" disp
0007     "" disp "                    Aragon" disp
0008 >>
```

La notion de définition est plus large que celle des routines ou fonctions des langages impératifs comme le Fortran ou le C car une définition prend un nombre quelconque d'objets — voire un nombre variable — comme arguments depuis la pile opérationnelle et en renvoyer un nombre quelconque — voire variable — dans cette pile.

Par ailleurs, une définition se comporte de la même façon qu'une définition intrinsèque du langage et peut être appelée par son nom à partir de n'importe quelle autre définition, en particulier d'elle-même, ce qui permet d'utiliser des fonctions récursives comme le calcul de factorielle suivant.

```
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
```


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

```
0025            N 1 >
0026        then
0027            N dup 1 - FACTORIELLE *
0028        else
0029            1
0030        end
0031    >>
0032 >>
```

Il existe trois types de définitions se distinguant par leur position vis à vis du langage : les définitions intrinsèques, extrinsèques et utilisateur.

3.1.1 Définitions intrinsèques

Les définitions intrinsèques correspondent aux instructions internes du langage. Elles s'opposent aux définitions extrinsèques et sont directement exécutables. Elles ne peuvent être surchargées et sont insensibles à la casse. Dans la suite de ce document, le terme « instruction » sera souvent utilisé à la place de « définition intrinsèque ».

3.1.2 Définitions extrinsèques

Les définitions extrinsèques sont quant à elles des définitions externes au RPL/2, écrites en RPL/C et disponibles sous la forme de bibliothèques dynamiques compilées. Ces définitions sont trop spécifiques pour justifier une intégration en tant que définitions intrinsèques. Elles ne peuvent surcharger les définitions intrinsèques mais contrairement aux définitions intrinsèques, elles sont sensibles à la casse.

3.1.3 Définitions utilisateur

Les définitions utilisateurs sont les définitions présentes dans le code source du programme. Celui-ci peut comporter un nombre quelconque de définitions, la seule contrainte étant que l'exécution du programme commencera toujours par la première définition présente dans le code quel qu'en soit son nom. Elles ne peuvent surcharger ni les définitions intrinsèques ni les définitions extrinsèques. Elles sont gérées par le système comme des variables de niveau 0 et sont sensibles à la casse.

3.2 Variables globales

Une variable globale est par définition une variable visible de tous les points d'un programme. Il existe un certain nombre de variables globales utilisées par le séquenceur comme `EQ` et `SDAT`, mais celles-ci ne sont pas réservées. Une variable globale peut être créée par un petit nombre d'instructions intrinsèques dont la plus courante est `SAVE`. Les variables globales ne sont partagées ni par les processus légers ni par les processus. Le seul moyen d'effacer une variable globale est le recours à l'instruction `PURGE`.

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

3.3 Variables locales

Contrairement aux variables globales, une variable locale n'est visible que dans un bloc de programme, généralement une fonction utilisateur ou une expression issue de celle-ci, que cette expression soit algébrique ou en notation polonaise inversée. À l'extérieur de ce bloc, la variable locale n'est plus définie et le nom ne peut être évalué. Ainsi, le petit programme suivant

```
0001 Petit_programme_sans_pretention
0002 <<
0003     1
0004     // Visibilité de X |
0005     -> X               |
0006     <<                 |
0007         X disp         |
0008     >>                 V
0009
0010     // Visibilité de X |
0011     1 -> X 'X+1' disp  V
0012
0013     X disp
0014 >>
```

provoquera une sortie sous la forme

```
1
2
'X'
```

En effet, la variable `X` est une variable locale à l'expression centrale et est détuite à la sortie de celle-ci pour être recréée lors du traitement de l'expression algébrique suivante et à nouveau détruite. Il faut noter que ce mécanisme peut se reproduire au sein d'une seule et même définition. Le petit programme suivant

```
0001 Petit_programme_sans_prétention_mais_plus_compliqué
0002 <<
0003     1
0004     // Visibilité de X ----------------------+
0005     -> X                                     |
0006     <<                                       |
0007         X disp                               |
0008                                              |
0009         // Visibilité de X -------------+    |---+ Masque
0010         2                               |    |   |
0011         -> X 'X+1' disp                 V    V   V
0012                                              |
0013         // Visibilité de X -------------+    |---+ Masque
0014         1 3 for X                       |    |   |
0015             X disp                      |    |   |
0016         next                            V    V   V
0017                                              |
0018         X disp                               V
0019     >>
0020
0021     X disp
0022 >>
```

a pour sortie

```
1
```


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

```
3
1
2
3
1
'X'
```

montrant par là que les variables locales ne s'écrasent pas mutuellement. Il faut noter que si les variables locales ne sont pas perdues lors de la création de variables locales de même noms mais de niveau supérieur, elles sont inaccessibles. La seule exception à cette règle est le fonctionnement particulier des variables globales qui restent toujours accessible par un jeu de fonctions particulier (`RCL`, `SAVE` et quelques autres fonctions spécifiques comme les fonctions de verrouillage.).

3.3.1 Variables volatiles

Toute variable définie par l'utilisateur est par défaut une variable volatile. Celle-ci est détruite dès que l'exécution du bloc d'instructions dans lequel est a été définie est terminé. Il est possible de contourner ce problème en déclarant explicitement une variable comme statique.

3.3.2 Variables statiques

Une variable déclarée comme statique lors de sa création ne sera visible que dans le bloc d'instructions pour lequel elle a été définie. Cependant, elle n'est que masquée à la fin de l'exécution de ce bloc et son contenu n'est pas détruit. Lors d'un appel ultérieur au même bloc d'instructions, la variable sera initialisée à l'aide de la valeur sauvegardée précédemment. Une variable statique n'est pas partagée entre deux processus même légers.

```
0001 Variable_statique
0002 <<
0003     def1 disp
0004     def1 disp
0005 >>
0006
0007 def1
0008 <<
0009     static 1
0010     -> I
0011     <<
0012         I 'I' incr
0013     >>
0014 >>
```

La valeur passée lors de la création d'une variable statique ne sert que pour l'initialisation de cette variable. Si cette variable est déjà initialisée, la valeur est silencieusement ignorée et l'objet associé détruit.

Une variable statique peut être rendue volatile par un appel à la fonction `VOLATILE`. Elle disparaît alors à la fin du bloc et sera réinitialisée lors de sa prochaine utilisation.

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

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

3.3.3 Variables partagées

Une variable partagée est une variable statique commune à plusieurs processus légers. Elle est déclarée par la fonction `shared` en lieu et place de `static`. Il convient souvent d'encadrer son utilisation par des mutexes pour éviter des accès concurrents. À l'instar des variables statiques, une variable partagée peut être rendue volatile par l'utilisation de la fonction `PRIVATE`.

3.4 Variables virtuelles

Une variable virtuelle est une variable sauvegardée dans un fichier sur la machine hôte. De ce fait, elle n'est pas référencée par un objet de type nom, mais par un objet de type chaîne de caractères. Elle n'est pas effacée à la fin du fil d'exécution.

Une variable virtuelle est ainsi utile pour sauvegarder des données externes à un programme ou pour partager des données entre processus. Il convient alors d'encadrer l'utilisation d'une telle variable par des sémaphores nommés.

3.5 Verrouillage

Toutes les variables, dès lors qu'elles ne sont pas virtuelles, peuvent être verrouillées en écriture. La modification d'une variable verrouillée provoque une erreur d'accès. Il existe deux instructions de verrouillage `PROTECT` et `PARAMETER` selon qu'elles portent sur des variables générales ou globales, et deux instructions de déverrouillsge `UNPROTECT` et `VARIABLE`. Par défaut, aucune variable n'est verrouillée lors de sa création.

3.6 Héritage

Les variables locales, statiques et partagées sont locales au bloc d'instruction dans lequel elles ont été définies. Elles ne sont visibles que depuis ce bloc. Si une définition utilisateur est appelée depuis ce bloc, aucune des variables définies dans le bloc appelant ne sera visible dans la fonction appelée. Il y a complète isolation des variables entre les différentes définitions utilisateur. Les seules variables visibles sur l'ensemble d'un programme sont les variables globales.

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

Deuxième partie

Appel du RPL/2

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

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

4

Ligne de commande

L'invocation du RPL/2 peut se faire de plusieurs manières. Tout d'abord, il est possible d'utiliser le RPL/2 de manière interactive en le lançant directement depuis une ligne de commande et en entrant les commandes RPL/2 directement depuis l'invite de commande.

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:[~] >
```

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

4.1 Options de la ligne de commande

La ligne de commande du séquenceur 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...) ;
- `-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 daemon [1]. 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 ;
- `-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 rpl-profile 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...) ;

1. Disk And Extension MONitor

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

- `-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.

4.2 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.

```
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 essayera 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 protogée par un bloc `CSTOP/RSTOP` —, il sera nécessaire de le tuer à la main à partir de l'invite du système d'exploitation.

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

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

5 Exécution de programmes

Dans le cas où les options `-i` et `-S` ne sont pas spécifiées, le séquenceur s'attends 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 preprocesseur 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.

5.1 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.

5.1.1 `#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 celles de `cpp`. L'exemple suivant est un peu plus

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

parlant.

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

5.1.2 #defeval x y

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

```
0000 #!/usr/local/bin/rpl -sp
0001
0002 #mode standard default
0003 #define RETRAIT_PARENTHESES(p) p
0004 #define APPLY(f,x) RETRAIT_PARENTHESES(#defeval FONCTION f
0005     FONCTION(x))
0006 #define sinus(x) x sin
0007
0008 MAIN
0009 <<
0010     rad APPLY(\sinus,'2*pi') disp
0011 >>
```

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.

5.1.3 #undef x

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

5.1.4 #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 la préprocesseur et doit être valide. En particulier, l'expansion des macros est effectuée.

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

5.1.5 #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.

5.1.6 #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.

5.1.7 #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.

5.1.8 #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.

5.1.9 #endif

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

5.1.10 #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>`.

5.1.11 #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`.

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

5.1.12 #eval expr

Cette primitive permet l'évaluation de l'expression `expr` et la tenue de calculs arithmétiques ou d'expressions régulières. La syntaxe et 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 intervalile 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.

5.1.13 #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 débuter un bloc conditionnel. En particulier, la valeur logique de `expr` est toujours vraie si cette expression ne peut être évaluée numériquement.

5.1.14 #elif expr

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

5.1.15 #mode keyword

La primitive `#mode` contrôle le fonctionne 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` : selectionne 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.

5.1.16 #line

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

5.1.17 #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`.

5.1.18 #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.

5.1.19 #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.

5.1.20 #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.

5.2 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 `rplpp` fonctionnant par défaut en mode `cpp`. La syntaxe et la structure du préprocesseur sont analysées avant la phase d'interprétation

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

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 évalution prend fin pour. Le programme s'achève à la fin de l'exécution de la définition principale.

5.3 Extension

Sous certains *shells*, en particulier sous Unix, il est possible de débuter 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/rpl`.

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

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

Troisième partie

Manipulations de la pile

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

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

6

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.

4			objet_1
3	objet_1		objet_2
2	objet_2	$\longrightarrow$	objet_3
1	objet_3	insertion de l'objet_4	objet_4

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 traité 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.

4	objet_1		
3	objet_2		objet_1
2	objet_3	$\longrightarrow$	objet_2
1	objet_4	retrait de l'objet_4	objet_3

Ainsi, la pile opérationnelle se comporte comme une file LIFO [1] 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`.

1. Last In First Out

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

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

Ainsi, pour qu'un objet soit évalué, il est réduit à un arbre [2] 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 6.1 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

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

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

nécessaire de déclarer une fonction récursive car elles ne s'exécutent pas dans le même plan mémoire.

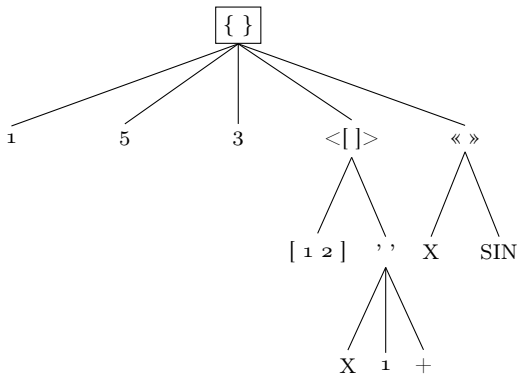
L'exemple précédent est volontairement simpliste. Il n'utilise que des types de données atomiques et ne doit pas faire accroire qu'il existe une quelconque équivalence entre les atomes et les types d'objets du RPL/2. Les atomes sont les définitions intrinsèques, les définitions extrinsèques et les objets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

- scalaires (entiers, réels, complexes) ;
- 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



Les manipulations des objets atomiques sont immédiates. En revanche, les opérations portant sur des objets non atomiques requièrent toujours des opérations complexes d'accès aux atomes. En cas de modification d'un objet, il faut accéder au moins une fois à chacun de ses atomes, ce qui est coûteux en terme de temps de calcul. Il s'agit donc d'utiliser les objets non atomiques à bon escient.

6.1.1 Cas d'un programme interprété

Les différents arbres sont créés dynamiquement lors de l'interprétation du programme. Chaque arbre est parcouru de façon à évaluer séquentiellement tous les atomes qui le composent. Chaque atome est traité selon l'algorithme 6.2. Les traitements des exceptions comme les erreurs de syntaxe n'apparaissent pas sur la figure. L'emplacement du debugger dans la boucle principale n'est pas indiquée.

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

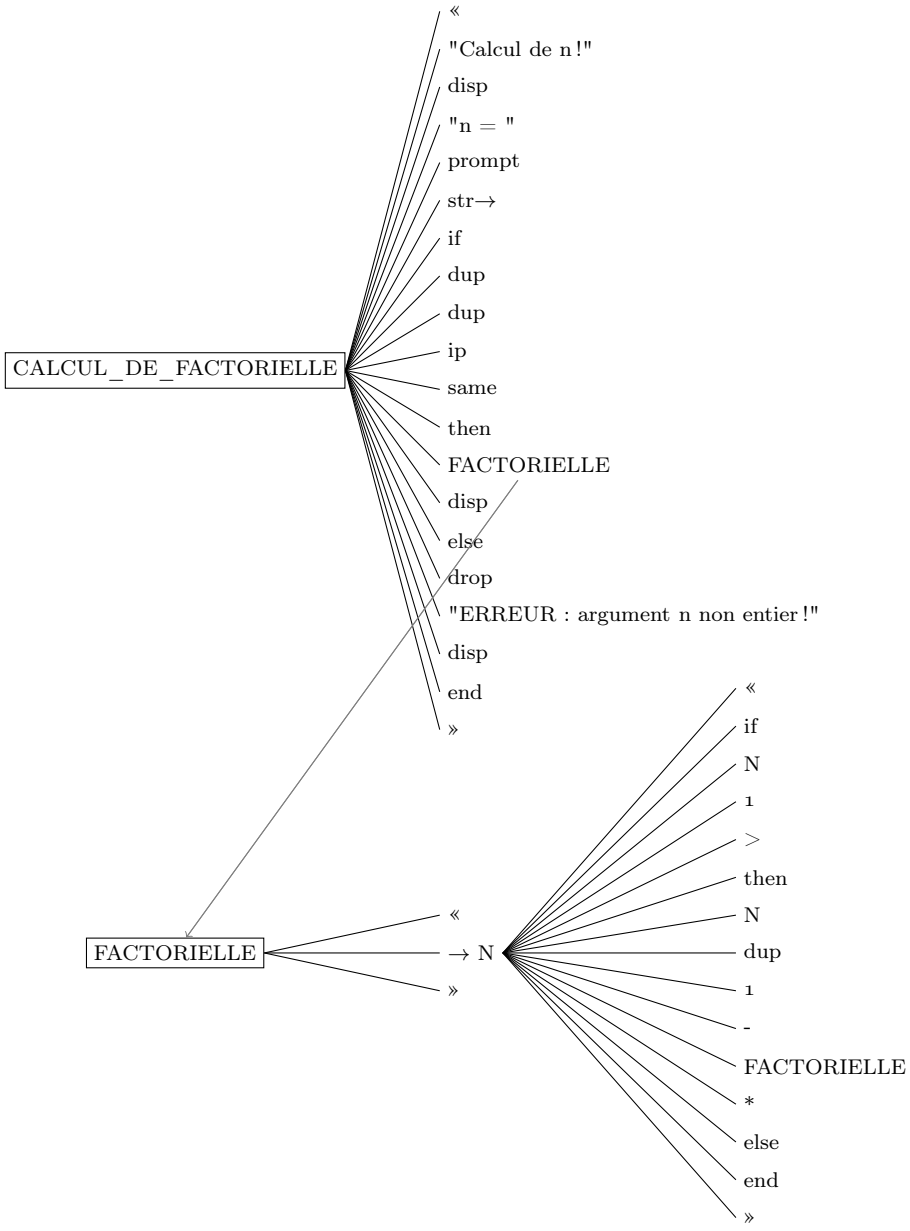


FIGURE 6.1 – Arbre du programme `CALCUL_DE_FACTORIELLE`

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

6.1.2 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.

6.2 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_m		$p+m$	objet_m
$\vdots$	$\vdots$		$\vdots$	$\vdots$
$n+2$	objet_2		$p+2$	objet_2
$n+1$	objet_1		$p+1$	objet_1
n	argument_n		p	résultat_p
$n-1$	argument_{n-1}		$p-1$	résultat_{p-1}
$\vdots$	$\vdots$		$\vdots$	$\vdots$
2	argument_2	$\longrightarrow$	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.

6.3 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.

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

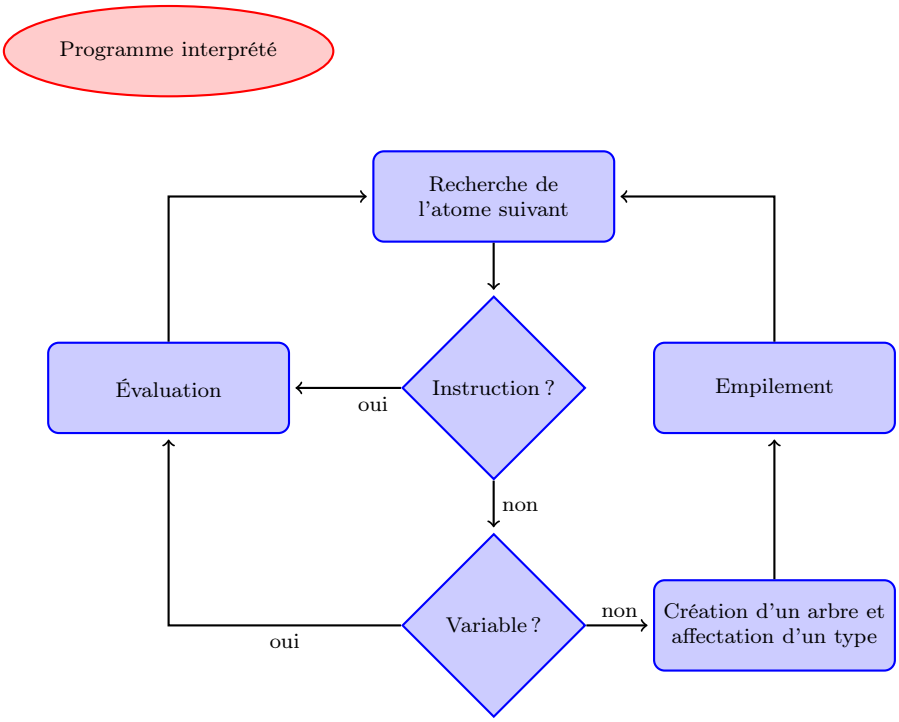


FIGURE 6.2 – Boucle d'analyse d'un programme interprété

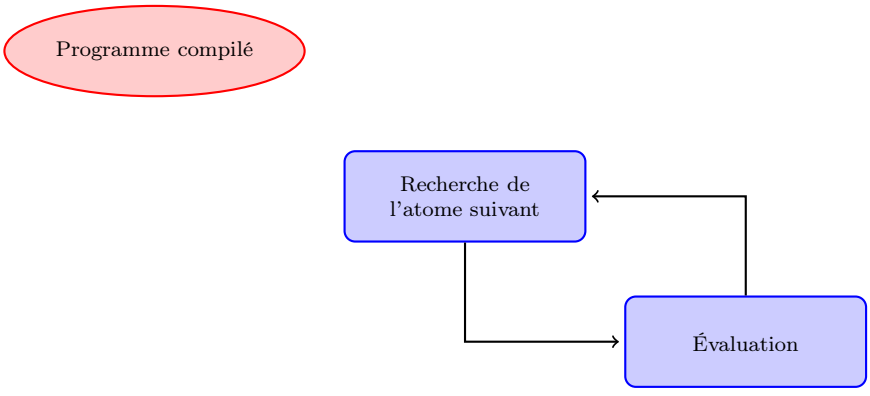


FIGURE 6.3 – Boucle d'analyse d'un programme compilé

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

7

Manipulation des objets

7.1 Gestion de la pile

7.1.1 Clear

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

7.1.2 Depth

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.

$n+1$			objet_n
n	objet_n		objet_{n-1}
$n-1$	objet_{n-1}		objet_{n-2}
$\vdots$	$\vdots$		$\vdots$
2	objet_2	$\longrightarrow$	objet_1
1	objet_1	`depth`	n

7.1.3 Last

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

7.2 Duplication d'éléments

7.2.1 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.

2	objet_2	$\longrightarrow$	objet_1
1	objet_1	`dup`	objet_1

7.2.2 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.

4			objet_2
3			objet_1
2	objet_2	$\longrightarrow$	objet_2
1	objet_1	`dup2`	objet_1

7.2.3 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 $n+1$.

$2n$			objet_n
$\vdots$			$\vdots$
$n+1$	objet_n		objet_1
n	objet_{n-1}		objet_n
$\vdots$	$\vdots$		$\vdots$
2	objet_1	$\longrightarrow$	objet_2
1	n	`dupn2`	objet_1

7.2.4 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.

3			objet_2
2	objet_2	$\longrightarrow$	objet_1
1	objet_1	`over`	objet_2

7.2.5 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 rogoureusement identique — le $(n+1)^{\text{ième}}$ de la pile avant l'appel à `pick`.

$n+1$	objet_n		objet_n
n	objet_{n-1}		objet_{n-1}
$\vdots$	$\vdots$		$\vdots$
2	objet_1	$\longrightarrow$	objet_1
1	n	`pick`	objet_n

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

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

7.3 Supression d'éléments

7.3.1 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.

		$\longrightarrow$
1	objet_1	`drop`

7.3.2 Drop2

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

2	objet_2	$\longrightarrow$
1	objet_1	`drop2`

7.3.3 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.

$n+1$	objet_n	
$\vdots$	$\vdots$	
2	objet_1	$\longrightarrow$
1	n	`dropn`

7.4 Modification de la hiérarchie

7.4.1 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.

2	objet_2	$\longrightarrow$	objet_1
1	objet_1	`swap`	objet_2

7.4.2 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`.

3	objet_3		objet_2
2	objet_2	$\longrightarrow$	objet_1
1	objet_1	`rot`	objet_3

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

7.4.3 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 rolld »` pour inverser l'effet de `roll`.

$n+1$	objet_n		
n	objet_{n-1}		objet_{n-1}
$\vdots$	$\vdots$		$\vdots$
2	objet_1	$\longrightarrow$	objet_1
1	n	`roll`	objet_n

7.4.4 Rolld

`rolld` 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.

$n+1$	objet_n		
n	objet_{n-1}		objet_1
$n-1$	objet_{n-2}		objet_n
$\vdots$	$\vdots$		$\vdots$
2	objet_1	$\longrightarrow$	objet_3
1	n	`rolld`	objet_2

7.5 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 modifiés 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 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.

7.5.1 Pshcntxt

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

7.5.2 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.

7.5.3 Dupcntxt

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

7.5.4 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.

7.5.5 Swapcntxt

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

7.5.6 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 `iferr` 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 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 >>
```

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

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

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

8 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.

8.1 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.

8.1.1 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	$\xrightarrow{\texttt{disp}}$

8.1.2 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` transformant un nombre en chaîne de caractères :

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


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

RPL/2> 2 fix 0 ->str

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

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.

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 :

  `(signe) mantisse E (signe) exposant`

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

  $$1 \leqslant x < 10$$

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

1	$\longrightarrow$ `std`	

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

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

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.

		$\longrightarrow$
1	n	`fix`

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.

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.

		$\longrightarrow$
1	n	`sci`

En format scientifique, les nombres sont affichés avec $n+1$ chiffres significatis, 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

$$1 \leqslant x < 10$$

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.

		$\longrightarrow$
1	n	`eng`

En format ingénieur, les nombres sont affichés avec $n+1$ chiffres significatis, 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

$$1 \leqslant x < 1000$$

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

8.1.3 Clmf

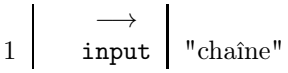
L'instruction `clmf` affiche le contenu de la pile. Elle ne prend aucun argument et ne renvoie rien dans la pile. Tous les objets sont affichés conformément au format courant.

8.2 Entrées

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

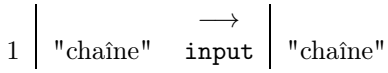
8.2.1 Input

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.



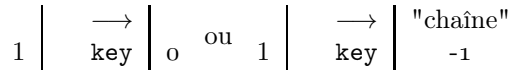
8.2.2 Prompt

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.



8.2.3 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 lorssue l'utilisateur appuit 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 >>
```

8.3 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.

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

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

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

Quatrième partie

Manipulation des variables

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

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

9

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 soumis 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 31. 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.

9.1 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  |
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 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 structurelle d'un programme, soit avant le début de l'exécution. Comme elles sont, contrairement à toutes les autres variables, non modifiables, elles 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.

9.2 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 9.1.

9.3 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 évaluables, sont traités comme des noms évaluables. 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 :

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

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

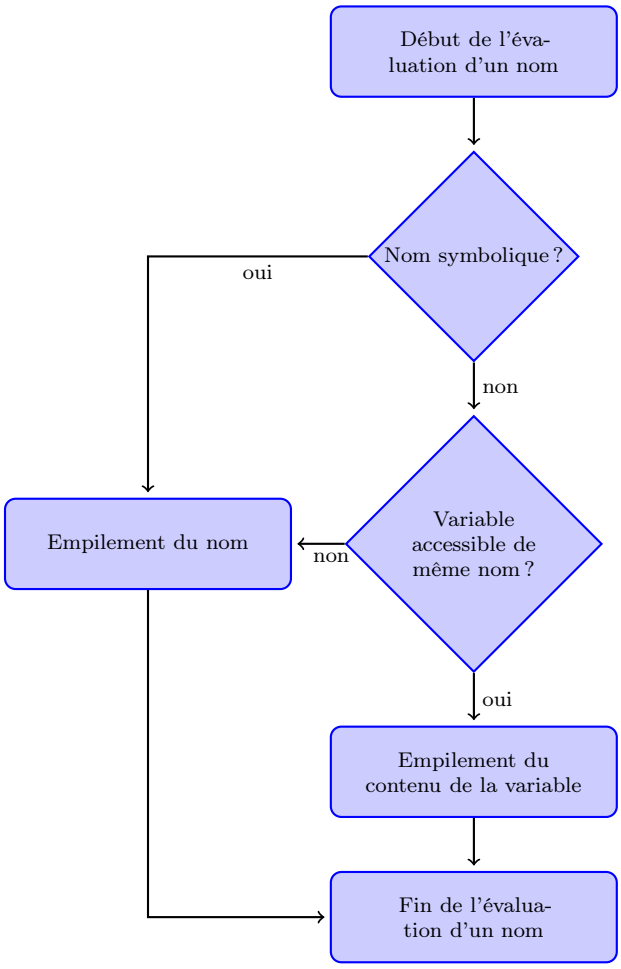


FIGURE 9.1 – Traitement des noms

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

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

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

10 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` et `save`.

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.

10.1 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	$\longrightarrow$
1	'nom global'	`save` ou `sto`

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

10.2 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. En absence de toute variable locale de même nom, le contenu de la variable globale est empilé. Cependant, pour éviter toute ambiguité 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	'nom global'	$\overrightarrow{\texttt{rcl}}$	objet

10.3 Modification

Les fonctions arothmétiques 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`.

10.4 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	'nom global'	$\overrightarrow{\texttt{purge}}$

1	{ 'nom global$_1$' ... 'nom global$_n$' }	$\overrightarrow{\texttt{purge}}$

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.

10.5 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.

$$1 \left| \text{'nom global'} \quad \overrightarrow{\texttt{parameter} \text{ ou } \texttt{variable}} \right|$$

$$1 \left| \{ \text{'nom global}_1\text{'} \ldots \text{'nom global}_n\text{'} \} \quad \overrightarrow{\texttt{parameter} \text{ ou } \texttt{variable}} \right|$$

$$1 \left| \text{'nom global'} \quad \overrightarrow{\texttt{protect} \text{ ou } \texttt{unprotect}} \right|$$

$$1 \left| \{ \text{'nom global}_1\text{'} \ldots \text{'nom global}_n\text{'} \} \quad \overrightarrow{\texttt{protect} \text{ ou } \texttt{unprotect}} \right|$$

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.

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

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

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

11 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 part défault volatile, donc détruite à la sortie de ce bloc. Contrairement aux variables globales, elle peut être statique ou partagée.

11.1 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 defaut, 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$_{n-1}$'	$\longrightarrow$
1	'nom local$_n$'	->

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 préexiste, 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 léger.

```
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 `spawn` crée un processus léger. La variable n'est pas perdu 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
```


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

```
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
```


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

```
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 soumis à des mécanismes d'accès concurrents.

11.2 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
```


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

```
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 eux 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 successsive 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 indentiques.

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

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

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     >>
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
```

11.3 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	$\longrightarrow$
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.

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

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

12 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.

12.1 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.

12.1.1 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	$\longrightarrow$	
1	'nom'	`sto+`	

2	'nom'	$\longrightarrow$	
1	objet	`sto+`	

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.

12.1.2 Soustraction

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

2	objet	⟶
1	'nom'	`sto-`

de

2	'nom'	⟶
1	objet	`sto-`

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.

12.1.3 Multiplication

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

2	objet	⟶
1	'nom'	`sto*`

2	'nom'	⟶
1	objet	`sto*`

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.

2	objet	⟶
1	'nom'	`sto/`

2	'nom'	⟶
1	objet	`sto/`

12.1.4 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.

12.2 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.

12.2.1 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.

		⟶
1	'nom'	`sinv`

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

12.2.2 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.

$$1 \mid \text{'nom'} \quad \overrightarrow{\texttt{sneg}} \mid$$

12.2.3 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.

$$1 \mid \text{'nom'} \quad \overrightarrow{\texttt{sconj}} \mid$$

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

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

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

Cinquième partie

Contrôle

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

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

13 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.

13.1 Tests simples

13.1.1 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
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.

13.1.2 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	$\longrightarrow$
1	objet	`ift`

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

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

13.1.3 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	$\longrightarrow$
1	Objet faux	`ifte`

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

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

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

13.2 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.

13.2.1 Types d'erreurs

Erreurs système

Les erreurs systèmes proviennent pour leur plus grand nombre du système d'explotation. 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.

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.

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 génèrera une exception si la séquenceur travaille sur le corps de réels et un infini ou une indétermination s'il travaille sur la droite achevée [1].

13.2.2 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.

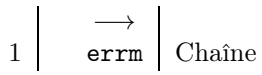
1	$\longrightarrow$ `errn`	Entier

1. *cf.* indicateur 59

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

13.2.3 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.



Exemple

```
cauchy:[~] > rpl -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 rpl -is
+++RPL/2 (R) version 4.0.10 (Saturday 02/06/10, 19:48:18 CET)
+++Copyright (C) 1989 to 2009, 2010 BERTRAND Joel

+++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:[~] >
```

13.2.4 Clerr

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

13.2.5 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.

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

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                 pulcntxt true
0016             else
0017                 dropcntxt false
0018             end
0019         end
0020
0021         disp
0022         pulcntxt
0023     >>
0024 >>
```

Ligne 5, l'instruction `pshcntxt` 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 sucecssifs 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

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

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.

13.3 Tests multiples

13.3.1 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.

13.3.2 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

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

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

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 détaillé plus bas.

Le 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.
```

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

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

14 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.

14.1 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 noteur que le comportement de l'instruction `cycle` diffère selon le type de boucle. En effet, dans 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.

14.1.1 Boucle sans compteur

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.

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 jusques 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 jusques 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.

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

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.

14.1.2 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.

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.

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* ;

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

- 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.

14.1.3 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
```


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

```
0009     1 step
0010
0011     "FIN" disp
0012 >>
```

14.2 Boucles indéfinies

14.2.1 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`

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

14.2.2 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`

14.3 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.

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

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

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

15 Contrôle de l'exécution

15.1 Mode de fonctionnement

L'état du système est assujetti 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`.

15.1.1 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 15.1.

15.1.2 Manipulation

Affectations

Il existe deux instructions simples permettant d'affecter un état — armé ou déasrmé — 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.

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

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

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

Indicateurs	Signification	Valeurs par défaut
1 à 30	aucune signification	désarmés
31	pile *last* 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 15.1 – Signification des indicateurs binaires

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`.

$$1 \;\Big|\; \xrightarrow[\texttt{rclf}]{} \;\Big|\; \#n \quad \text{ou} \quad 1 \;\Big|\; \#n \;\xrightarrow[\texttt{stof}]{}\;\Big|$$

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.

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.

$$1 \;\Big|\; \#n \;\xrightarrow[\texttt{fc?}]{}\;\Big|\; n \qquad 1 \;\Big|\; \#n \;\xrightarrow[\texttt{fs?}]{}\;\Big|\; n$$

$$1 \;\Big|\; \#n \;\xrightarrow[\texttt{fc?c}]{}\;\Big|\; n \qquad 1 \;\Big|\; \#n \;\xrightarrow[\texttt{fs?c}]{}\;\Big|\; n$$

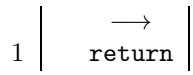
$$1 \;\Big|\; \#n \;\xrightarrow[\texttt{fc?s}]{}\;\Big|\; n \qquad 1 \;\Big|\; \#n \;\xrightarrow[\texttt{fs?s}]{}\;\Big|\; n$$

15.2 Exécution normale

15.2.1 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.

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

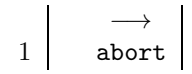


L'instruction de retour anticipé provoque une analyse sans exécution de tous les atomes figurant entre `return` et la fin de la définition courante car elle doit rendre la main à la définition appelante et garder la pile système dans un état cohérent. L'utilisation de `return` n'est donc pas aussi efficace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

15.2.2 Abandon

Un programme s'arrête naturellement lorsqu'il atteint la fin de sa définition principale. À la fin de celle-ci, il signale son achevement 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.

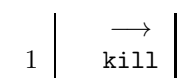
Il existe deux instructions pour déroger à cette règle. La première, `abort` provoque un arrêt du programme en cours. Quel que soit le processus du programme RPL/2 exécutant l'instruction `abort`, chacun des processus du programme reçoit une requête d'arrêt. Ces requêtes d'arrêt sont honorées immédiatement 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.



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.



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

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

15.3 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.

15.3.1 Point d'arrêt

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

1	⟶ `halt`

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:[~] >
```

15.3.2 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'évalution 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 |
```

15.3.3 Retour en exécution normale

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

```
1 | ⟶   |
  | cont |
```

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

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

Sixième partie

Arithmétique

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

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

16 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.

16.1 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 ;
- 'symbole', un symbole algébrique qu'il s'agisse d'un nom ou d'une expression ;
- « expression », une expression en notation polonaise inverse ;
- { liste }, une liste d'objets hétéroclites ;
- "chaîne", une chaîne de caractères ;
- #n, un entier binaire.

16.2 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 16.1.

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.

16.3 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 16.2.

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.

16.4 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 16.3.

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.

16.5 Division

16.5.1 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 determiné par le type de ses arguments. Les différents types acceptés comme arguments ainsi que les types en résultant sont indiqués au tableau 16.3.

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

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

connaissant la matrice **A** et le vecteur b. En effet, en définissant une division d'un vecteur par une matrice, il devriendrait 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: [ 1.66666666666667 -2.33333333333333 1. ]
RPL/2>
```


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

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

Niveau$_2$	**Niveau$_1$**	$\rightarrow$	**Niveau$_1$**	**Résultat**
n_1	n_2	$\rightarrow$	$n_1 + n_2$	Entier
n_1	n_2	$\rightarrow$	$n_1 + n_2$	Réel
n_1	x_2	$\rightarrow$	$n_1 + x_2$	Réel
n_1	z_2	$\rightarrow$	$n_1 + z_2$	Complexe
x_1	n_2	$\rightarrow$	$x_1 + n_2$	Réel
x_1	x_2	$\rightarrow$	$x_1 + x_2$	Réel
x_1	z_2	$\rightarrow$	$x_1 + z_2$	Complexe
n_1	z_2	$\rightarrow$	$n_1 + z_2$	Complexe
x_1	z_2	$\rightarrow$	$x_1 + z_2$	Complexe
z_1	z_2	$\rightarrow$	$z_1 + z_2$	Complexe
[tableau$_1$]	[tableau$_2$]	$\rightarrow$	[tableau$_1$ + tableau$_2$]	
s	'symbole'	$\rightarrow$	's+(symbole)'	Expression
'symbole'	s	$\rightarrow$	'symbole+s'	Expression
'symbole$_1$'	'symbole$_2$'	$\rightarrow$	'symbole$_1$ + symbole$_2$'	Expression
s	« symbole »	$\rightarrow$	« s symbole + »	Expression
« symbole »	s	$\rightarrow$	« symbole s + »	Expression
« symbole$_1$ »	« symbole$_2$ »	$\rightarrow$	« symbole$_1$ symbole$_2$ + »	Expression
{ liste$_1$ }	{ liste$_2$ }	$\rightarrow$	{ liste$_1$ liste$_2$ }	Liste
"chaîne$_1$"	"chaîne$_2$"	$\rightarrow$	"chaîne$_1$chaîne$_2$"	Chaîne
$\#n_1$	n_2	$\rightarrow$	$\#n_1 + n_2$	Binaire
n_1	$\#n_2$	$\rightarrow$	$\#n_1 + n_2$	Binaire
$\#n_1$	$\#n_2$	$\rightarrow$	$\#n_1 + n_2$	Binaire

TABLE 16.1 – Addition

Niveau$_2$	**Niveau$_1$**	$\rightarrow$	**Niveau$_1$**	**Résultat**
n_1	n_2	$\rightarrow$	$n_1 - n_2$	Entier
n_1	n_2	$\rightarrow$	$n_1 - n_2$	Réel
n_1	x_2	$\rightarrow$	$n_1 - x_2$	Réel
n_1	z_2	$\rightarrow$	$n_1 - z_2$	Complexe
x_1	n_2	$\rightarrow$	$x_1 - n_2$	Réel
x_1	x_2	$\rightarrow$	$x_1 - x_2$	Réel
x_1	z_2	$\rightarrow$	$x_1 - z_2$	Complexe
n_1	z_2	$\rightarrow$	$n_1 - z_2$	Complexe
x_1	z_2	$\rightarrow$	$x_1 - z_2$	Complexe
z_1	z_2	$\rightarrow$	$z_1 - z_2$	Complexe
[tableau$_1$]	[tableau$_2$]	$\rightarrow$	[tableau$_1$ − tableau$_2$]	
s	'symbole'	$\rightarrow$	's−(symbole)'	Expression
'symbole'	s	$\rightarrow$	'symbole−s'	Expression
'symbole$_1$'	'symbole$_2$'	$\rightarrow$	'symbole$_1$ − symbole$_2$'	Expression
s	« symbole »	$\rightarrow$	« s symbole − »	Expression
« symbole »	s	$\rightarrow$	« symbole s − »	Expression
« symbole$_1$ »	« symbole$_2$ »	$\rightarrow$	« symbole$_1$ symbole$_2$ − »	Expression
$\#n_1$	n_2	$\rightarrow$	$\#n_1 + n_2$	Binaire
n_1	$\#n_2$	$\rightarrow$	$\#n_1 + n_2$	Binaire
$\#n_1$	$\#n_2$	$\rightarrow$	$\#n_1 + n_2$	Binaire

TABLE 16.2 – Soustraction

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 **A**. La matrice **B** doit posséder le même nombre de lignes que **A**.

À noter : cette instruction ne provoque le calcule 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.

16.5.2 Inversion

L'instruction `inv` calcule l'inverse son argument. Le fonctionnement de cette instruction est décrit au tableau 16.5.

16.6 Puissance

16.6.1 Puissance standard

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 16.6.

16.6.2 Carré

16.6.3 Racine carrée

16.6.4 Racine $n^{\text{ième}}$

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

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

Niveau$_2$	**Niveau$_1$**	$\rightarrow$	**Niveau$_1$**	**Résultat**
n_1	n_2	$\rightarrow$	$n_1 n_2$	Entier
n_1	n_2	$\rightarrow$	$n_1 n_2$	Réel
n_1	x_2	$\rightarrow$	$n_1 x_2$	Réel
n_1	z_2	$\rightarrow$	$n_1 z_2$	Complexe
x_1	n_2	$\rightarrow$	$x_1 n_2$	Réel
x_1	x_2	$\rightarrow$	$x_1 x_2$	Réel
x_1	z_2	$\rightarrow$	$x_1 z_2$	Complexe
n_1	z_2	$\rightarrow$	$n_1 z_2$	Complexe
x_1	z_2	$\rightarrow$	$x_1 z_2$	Complexe
z_1	z_2	$\rightarrow$	$z_1 z_2$	Complexe
[matrice]	[tableau]	$\rightarrow$	[matrice $\times$ tableau]	
[matrice]	s	$\rightarrow$	[matrice $\times$ s]	
s	[tableau]	$\rightarrow$	[$s \times$ tableau]	
s	'symbole'	$\rightarrow$	's$*$(symbole)'	Expression
'symbole'	s	$\rightarrow$	'(symbole)$*$s'	Expression
'symbole$_1$'	'symbole$_2$'	$\rightarrow$	'symbole$_1$ $*$ symbole$_2$'	Expression
s	« symbole »	$\rightarrow$	« s symbole $*$ »	Expression
« symbole »	s	$\rightarrow$	« symbole s $*$ »	Expression
« symbole$_1$ »	« symbole$_2$ »	$\rightarrow$	« symbole$_1$ symbole$_2$ $*$ »	Expression
$\#n_1$	n_2	$\rightarrow$	$\#n_1 n_2$	Binaire
n_1	$\#n_2$	$\rightarrow$	$\#n_1 n_2$	Binaire
$\#n_1$	$\#n_2$	$\rightarrow$	$\#n_1 n_2$	Binaire

TABLE 16.3 – Multiplication

Niveau$_2$	**Niveau$_1$**	$\rightarrow$	**Niveau$_1$**	**Résultat**
n_1	n_2	$\rightarrow$	n_1/n_2	Réel
n_1	x_2	$\rightarrow$	n_1/x_2	Réel
n_1	z_2	$\rightarrow$	n_1/z_2	Complexe
x_1	n_2	$\rightarrow$	x_1/n_2	Réel
x_1	x_2	$\rightarrow$	x_1/x_2	Réel
x_1	z_2	$\rightarrow$	x_1/z_2	Complexe
n_1	z_2	$\rightarrow$	n_1/z_2	Complexe
x_1	z_2	$\rightarrow$	x_1/z_2	Complexe
z_1	z_2	$\rightarrow$	z_1/z_2	Complexe
[tableau]	s	$\rightarrow$	[tableau/s]	
[tableau]	[matrice]	$\rightarrow$	[matrice]	
s	'symbole'	$\rightarrow$	's/(symbole)'	Expression
'symbole'	s	$\rightarrow$	'(symbole)/s'	Expression
'symbole$_1$'	'symbole$_2$'	$\rightarrow$	'symbole$_1$ $*$ symbole$_2$'	Expression
s	« symbole »	$\rightarrow$	« s symbole / »	Expression
« symbole »	s	$\rightarrow$	« symbole s / »	Expression
« symbole$_1$ »	« symbole$_2$ »	$\rightarrow$	« symbole$_1$ symbole$_2$ / »	Expression
$\#n_1$	n_2	$\rightarrow$	$\#n_1/n_2$	Binaire
n_1	$\#n_2$	$\rightarrow$	$\#n_1/n_2$	Binaire
$\#n_1$	$\#n_2$	$\rightarrow$	$\#n_1/n_2$	Binaire

TABLE 16.4 – Division

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

Niveau$_1$	$\rightarrow$	**Niveau$_1$**	**Résultat**
n	$\rightarrow$	$1/n$	Réel
x	$\rightarrow$	$1/x$	Réel
z	$\rightarrow$	$1/z$	Complexe
[matrice]	$\rightarrow$	[matrice]	
'symbole'	$\rightarrow$	'INV(symbole)'	Expression
« symbole »	$\rightarrow$	« symbole INV »	Expression

TABLE 16.5 – Inversion

Niveau$_2$	**Niveau$_1$**	$\rightarrow$	**Niveau$_1$**	**Résultat**
n_1	n_2	$\rightarrow$	$n_1^{n_2}$	Entier
n_1	n_2	$\rightarrow$	$n_1^{n_2}$	Réel
n_1	x_2	$\rightarrow$	$n_1^{x_2}$	Réel
n_1	z_2	$\rightarrow$	$n_1^{z_2}$	Complexe
x_1	n_2	$\rightarrow$	$x_1^{n_2}$	Réel
x_1	x_2	$\rightarrow$	$x_1^{x_2}$	Réel
x_1	z_2	$\rightarrow$	$x_1^{z_2}$	Complexe
n_1	z_2	$\rightarrow$	$n_1^{z_2}$	Complexe
x_1	z_2	$\rightarrow$	$x_1^{z_2}$	Complexe
z_1	z_2	$\rightarrow$	$z_1^{z_2}$	Complexe
s	'symbole'	$\rightarrow$	's^(symbole)'	Expression
'symbole'	s	$\rightarrow$	'(symbole)^s'	Expression
'symbole$_1$'	'symbole$_2$'	$\rightarrow$	'symbole$_1$^symbole$_2$'	Expression
s	« symbole »	$\rightarrow$	« s symbole ^ »	Expression
« symbole »	s	$\rightarrow$	« symbole s ^ »	Expression
« symbole$_1$ »	« symbole$_2$ »	$\rightarrow$	« symbole$_1$ symbole$_2$ ^ »	Expression

TABLE 16.6 – Puissance

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

17 Arithmétique générale

17.1 Proportions

17.1.1 Instruction %

17.1.2 Instruction %CH

17.1.3 Instruction %T

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

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

Septième partie

Trigonométrie et logarithmes

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

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

18

Fonctions trigonométriques

18.1 Cosinus

18.2 Sinus

18.3 Tangente

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

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

19

Fonctions hyperboliques

19.1 Cosinus hyperbolique

19.2 Sinus hyperbolique

19.3 Tangente hyperboliques

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

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

20

Fonctions logarithmiques

20.1 Logarithme naturel

20.2 Logarithme vulgaire

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

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

Huitième partie

Algèbre linéaire

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

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

21

Vecteurs et matrices

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

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

22 *Résolution*

22.1 Inversion

22.2 Système linéaire

22.3 Moindres carrés

22.4 Moindres carrés généralisés

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

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

23

Décompositions

23.1 Vecteurs propres

23.2 Vecteurs propres généralisés

23.3 Décomposition de Cholesky

23.4 Décomposition LDU

23.5 Décomposition LU de Crout

23.6 Décomposition LQ

23.7 Décomposition QR

23.8 Décomposition de Schur

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

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

Neuvième partie

Statistiques

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

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

24 Description

24.1 Matrice de statistique

24.2 Moyenne

24.3 Variance et écart-type

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

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

25

Dénombrement

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

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

26

Lois de probabilité cumulées

26.1 Distribution de Laplace-Gauß dite normale

26.2 Distribution du χ^2

26.3 Distribution de Fisher

26.4 Distribution de Student

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

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

Dixième partie

Listes et tables

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

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

Onzième partie

Fichiers

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

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

27

Variable virtuelle

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

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

28

Fichiers à accès séquentiel

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

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

29

Fichiers à accès direct

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

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

Douzième partie

Accès au réseau

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

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

Treizième partie

Processus

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

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

30

Processus détachés

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

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

31

processus légers

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

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

Quatorzième partie

Graphisme

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

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

32 Dessin

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

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

33

Graphiques à deux dimensions

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

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

34

___ *Graphiques à trois dimensions*

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

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

35

Graphiques de statistiques

35.1 Nuages de points

35.2 Histogrammes

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

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

Quinzième partie

Impression

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

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

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

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

36

Du bon usage des variables

36.1 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 manipilation 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.

36.2 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 volimineux, il est souvent plus efficace de travailler sur la pile que sur des variables.

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

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

37

Bibliothèques partagées

37.1 Extensions compilées au RPL/2

37.2 Interfaces

37.3 Fonctions accessibles

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

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

Exemples

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

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

38 Programmes RPL/2

38.1 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
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] >
```

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

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

38.2 Programme complexe

Le programme présenté ici est un programme d'optimisation d'une valeur propre d'une matrice **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'Obêle
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'obêle" disp
0023     "{\\Large\\sl Algorithme de l'obêle}" 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
0059    -> UNITE N_ANTENNES N_MOBILES FACTEUR_ETALEMENT ALGORITHME_BRUITE 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_BRUITE
0081            then
0082                SEUIL CONVERSION_ALGORITHME_BRUITE
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_BRUITE 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
```

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

```
0129                    ALGORITHME_BRUITE not
0130                then
0131                    F regv max swap drop list-> drop
0132
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                        geti
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_BRUITE not
0201                        then
0202                            0
```

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

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

```
0203                    else
0204                        BRUIT
0205                    end
0206
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 $\lambda$"
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_BRUITE 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_ANTENNES for K
0334                                dup K COLONNE 2 ->list get swap
0335                            next
0336
0337                            drop N_ANTENNES 1 ->list ->array dup abs /
0338                            N_ANTENNES 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_BRUITE SEUIL
0350                NORMALISATION array-> drop
```

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

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

```
0351
0352                N_ANTENNES 1 2 ->list ->array
0353                'V_PONDERATION' swap J swap put
0354            next
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_BRUITE 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
0429            V_PONDERATION over 2 ->list "resultat_obele" 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_BRUITE 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
```

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

```
0499                    drop disp "" disp
0500                    erase
0501                    cllcd
0502                end
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 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 ================================================================================
```

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

```
0573 */
0574
0575 NORMALISATION
0576 <<
0577     -> R W FACTEUR_ETALEMENT ALGORITHME_BRUITE SEUIL
0578     <<
0579         W dup trn R FACTEUR_ETALEMENT *
0580
0581         if
0582             ALGORITHME_BRUITE
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
```

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

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


```
0647            ANGLES_MOYENS
0648            <<
0649                SECTEUR neg DTHETA 4 * - 'GISMIN' sto
0650                SECTEUR DTHETA 2 * + 'GISMAX' sto
0651
0652                'COEFF' { 1 1 }
0653
0654                0 puti
0655                DTHETA 2 / puti
0656                DTHETA 2 / neg puti
0657                DTHETA 2 / 1 - puti
0658                1 DTHETA 2 / - puti
0659                2 DTHETA * puti
0660                -2 DTHETA * puti
0661                3 DTHETA * puti
0662                4 DTHETA * puti
0663
0664                -2 puti
0665                -7 puti
0666                -7 puti
0667                -4 puti
0668                -4 puti
0669                -9 puti
0670                -10 puti
0671                -15 puti
0672                -20 puti
0673
0674                0 puti
0675                0 puti
0676                0 puti
0677                310 puti
0678                310 puti
0679                710 puti
0680                1090 puti
0681                1730 puti
0682                2510 puti
0683
0684                drop2
0685
0686                0 -> CUMUL
0687                <<
0688                    1 COEFF size 2 get for I
0689                        'COEFF' 2 I 2 ->list 'COEFF' over get 10 / alog
0690                        dup 'CUMUL' sto+ put
0691                    next
0692
0693                    1 COEFF size 2 get for I
0694                        'COEFF' 2 I 2 ->list 'COEFF' over get CUMUL / put
0695                    next
0696                >>
0697
0698                // Calcul de la répartition des mobiles dans les paquets
0699                'REPARTITION_INTERNE' { 1 }
0700                1 PAQUETS for P
0701                    rand puti
0702                next
0703                drop2
0704
0705                REPARTITION_INTERNE array-> 1 get 2 swap for P + next
0706
0707                -> CLEF
0708                <<
0709                    1 PAQUETS for P
0710                        'REPARTITION_INTERNE' dup P 1 ->list get NM * CLEF /
0711                        ip P 1 ->list swap
0712
0713                        if
0714                            dup 1 <
0715                        then
0716                            drop 1
0717                        end
0718
0719                        put
0720                    next
```

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

```
0721                >>
0722
0723                0 1 PAQUETS for P
0724                    'REPARTITION_INTERNE' P 1 ->list get +
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
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 pr1 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
```

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

```
                            drop
                        >>
                    next
                    trn conj dup trn *
                next
            >>

            NM 1 ->list 1 con 'REPARTITION_INTERNE' sto
        end

        NM ->list
        swap stof
        REPARTITION_INTERNE
        DIRECTIONS
    >>

    "" disp
>>


/*
================================================================================
  Fonction permettant de convertir l'algorithme non bruité en sa version
  bruitée
================================================================================
  Entrées :
    2: liste contenant les différentes matrices R
    1: seuil
--------------------------------------------------------------------------------
  Sorties :
    1: liste contenant les nouvelles matrices R'
--------------------------------------------------------------------------------
  Effets de bord : néant
================================================================================
*/

CONVERSION_ALGORITHME_BRUITE
<<
    -> L_R SEUIL
    <<
        1 L_R size for J
            'L_R' dup J get SEUIL * J swap put
        next

        L_R
    >>
>>


/*
================================================================================
  Calcul simple des pondérations et des puissances
================================================================================
  Entrées :
    1: liste contenant les différentes matrices R
--------------------------------------------------------------------------------
  Sorties :
    2: liste contenant les pondérations et les puissances
    1: valeur propre
--------------------------------------------------------------------------------
  Effets de bord : néant
================================================================================
*/

OPTIMISATION_SIMPLE
<<
    -> LISTE FACTEUR_ETALEMENT ALGORITHME_BRUITE SEUIL
    <<
        1 LISTE size for N
            LISTE N get regv max swap drop
            list-> drop over size 1 get 1 2 ->list 0 con

            -> INDICE TABLEAU
            <<
```

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

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

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

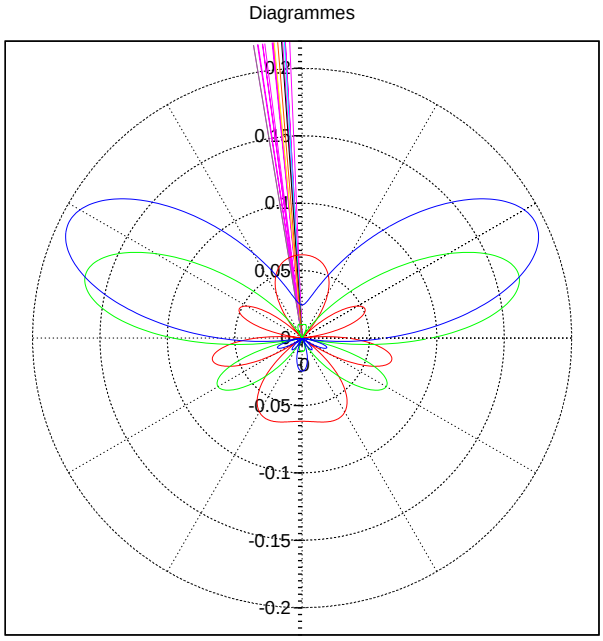
```
1017                TRACE_UN_MOBILE
1018            then
1019                1
1020                -> I
1021                <<
1022                    1 PAQUETS for POINTEUR
1023                        draw
1024                        'I' REPARTITION POINTEUR 1 ->list get ip sto+
1025                    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    >>
```

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

```
1091 >>
1092
1093 // vim: ts=4
```

Graphique généré

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



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

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

39 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
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        elseIf (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
```

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

```
0119         declareHelpString("Levenshtein distance");
0120         numberOfArguments(2);
0121     FUNCTION
0122         pullFromStack(object1, string);
0123         returnOnError(freeObject(object1));
0124         pullFromStack(object2, string);
0125         returnOnError(freeObject(object1); freeObject(object2));
0126
0127         getString(object1, input1);
0128         getString(object2, input2);
0129
0130         createIntegerObject(object3);
0131         setInteger(object3, levenshtein_distance(input1, input2));
0132
0133         pushOnStack(object3);
0134
0135         freeObject(object1);
0136         freeObject(object2);
0137     END
0138 endExternalFunction
```

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

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

204

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

Instructions

*, 114
**, 116
+, 113
−, 114
/, 114, 133
^, 116
→, 79
→num, 84
#date, 47
#defeval, 44
#define, 43
#elif, 46
#else, 45
#endif, 45
#error, 47
#eval, 46
#exec, 45
#file, 47
#if, 46
#ifdef, 44
#ifeq, 45
#ifndef, 45
#ifneq, 45
#include, 45
#line, 47
#mode, 46
#undef, 44
#warning, 47
%, 119
%ch, 119
%t, 119

abort, 42, 108

beep, 67

case, 96
cf, 105
clear, 57
clerr, 94
clusr, 76
cont, 110
cycle, 100, 101

default, 96
depth, 57
detach, 81
disp, 63
do, 103
drop, 59
drop2, 59
dropcntxt, 61
dropn, 59
dup, 57
dup2, 58
dupcntxt, 61
dupn, 58

egv, 135
egvl, 135
else, 91, 94, 96
elseif, 96
end, 91, 94, 96, 103
eng, 65
errm, 94
errn, 93
eval, 84
exit, 42, 99, 103

false, 25
fc ?, 107
fc ?c, 107
fc ?s, 107
fix, 65
for, 79, 101
fs ?, 107
fs ?c, 107
fs ?s, 107

gegv, 135
gegvl, 135
glegv, 135
glsq, 133
gregv, 135

halt, 109

if, 91, 96
iferr, 94
ift, 92
ifte, 92
input, 66
inv, 116, 133

key, 66
kill, 42, 108

last, 57
lchol, 135
legv, 135
lsq, 133
lu, 135

next, 100, 101
nrproc, 93

over, 58

parameter, 76
pick, 58
private, 84
prompt, 66
protect, 76
pshcntxt, 60
pulcntxt, 60
purge, 76

rcl, 76
rclf, 105
regv, 135
repeat, 103
return, 107
roll, 60
rolld, 60

save, 75
schur, 135
sci, 65
sconj, 87
select, 96
sf, 105
shared, 80
sinv, 86
sneg, 87
sq, 116
sqrt, 116
sst, 110
start, 100
static, 80
std, 64
step, 100, 101
sto, 75, 84
sto∗, 86
sto+, 85
sto−, 85
sto/, 86
stof, 107
stws, 107
swap, 59
swapcntxt, 61

then, 91, 94, 96
true, 25

uchol, 135
unprotect, 76
until, 103

variable, 76
volatile, 84

while, 103

xroot, 116

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

Index général

Atome, 52

Bibliothèque, *voir* types de données
Binaire, *voir* types de données
Booléen, *voir* types de données
Boucles
    définies, 99
    indéfinies, 99

Chaîne de caractères, *voir* types de données
Changement de contexte, *voir* contexte
Commentaires, 24
Complexe, *voir* types de données
Connecteur SQL, *voir* types de données
Contexte, 60

Débordement, 25
Définitions, 32
    extrinsèque, 33
    intrinsèque, 33
    principale, 33
    utilisateur, 33

Entier, *voir* types de données
Entrées et sorties, 63
Erreur, *voir* reprise sur erreur
Expression
    algébrique, *voir* types de données
    RPN, *voir* types de données

Fichier, *voir* types de données
Format
    fixe, 65
    ingénieur, 65
    scientifique, 65
    standard, 64

Héritage, *voir* variables

Indétermination, *voir* résultat indéterminé
Infini, *voir* résultat infini

Ligne de commande, 39
Liste, *voir* types de données

Matrice, *voir* types de données
Mutex, *voir* types de données

NaN, 26
Niveau d'exécution, 71
Nom, *voir* types de données
Nom évaluable, *voir* types de données
Nom symbolique, *voir* types de données
Notation
    algébrique, 24
    infixe, 24
    polonaise inverse (RPN), 23

Pile last, 41, 43, 57
Pile opérationnelle, 51
    dépilement, 52
    empilement, 52
Processus, *voir* types de données

Réel, *voir* types de données
Résultat
    indéterminé, 26
    infini, 26
Reprise sur erreur, 93
RPN, *voir* notation

Sémaphore nommé, *voir* types de données
Séparateur décimal, 64
Scalaire, *voir* types de données
Socket, *voir* types de données

Table, *voir* types de données
Types de données, 25
    bibliothèque, 29
    binaire, 28
    booléen, 25
    chaîne de caractères, 28
    complexe, 26
    connecteur SQL, 30
    entier, 25
    expression algébrique, 27
    expression RPN, 27
    fichier, 28
    liste, 26

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

matrice, 26
mutex, 30
nom, 27
évaluable, 72
symbolique, 72
processus, 29
réel, 26
sémaphore nommé, 30
scalaire, 25
socket, 29
table, 27
vecteur, 26

Variables, 31
accessibilité, 71
globales, 33, 75
héritage, 36
locales, 34, 79
partagées, 36, 80
portée, 82
statiques, 35, 80
verrouillage, 36
virtuelles, 36
visibilité, 82
volatiles, 35
Vecteur, *voir* types de données
Verrouillage, *voir* variables

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