Reverse Polish Lisp/2

— Manuel de référence —

BERTRAND Joël

18 janvier 2007

RPL/2

Table des matières

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

Genèse

Le *Reverse Polish Lisp/2* — ou RPL/2 — est né de la nécessité de pouvoir coder des algorithmes formels rapidement, de façon fiable et de sorte à minimiser les erreurs de calculs inhérentes aux dérives numériques et débordements. N'ayant pas trouvé de langage de programmation satisfaisant à ces critères et ayant un peu de temps libre à consacrer, j'ai débuté l'écriture d'un langage reprennant des concepts existants[1] tout en minimisant toutes les erreurs de calcul possibles, et enrichi de toutes les fonctions nécessaires à son usage courant. Au fil des années, ce langage, tout d'abord un peu expérimental et il faut bien dire rudimentaire, s'est développé au point de devenir un langage objet semi-compilé assez facilement utilisable.

[1]Le langage RPL du calculateur HP-28S de Hewlett-Packard

Historique

RPL 6502, les limbes

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. L'exiguïté de la mémoire disponible ainsi que la modeste puissance de la machine rendait impossible les traitements des objets complexes comme les matrices. Néanmoins, il était utilisable et a permis de poser les fondements d'une longue lignée.

RPL 6809, un essai non transformé

La deuxième version millésimée 1990 n'a jamais été finalisée. Elle était censée fonctionner sur un SMT-Goupil G3/6809 muni de 320 Ko de mémoire, mais toute mémoire au delà de 64 Ko étant accessible 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 et en l'introduction de commandes de gestion des fichiers, ainsi qu'en l'apparition des calculs symboliques.

RPL 386, version 3.00

L'année 1991 a vu naître 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. Le séquenceur reste toujours écrit en Basic, mais cette fois-ci en Turbo-Basic sous IBM-DOS 5.00...

RPL 387, version 3.0x

Une grande révolution dans les entrailles profondes de cet IBM qui reçoit un coprocesseur arithmétique i387DX. Au regard du gain apporté par ce composant, il est possible d'effectuer les calculs sur l'ensemble des complexes. Au reste, pour simplifier la gestion, tous les calculs se font en complexe...

RPL/2, 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 à ma disposition. Les puissances maintenant utilisables rendaient caducs le choix de travailler en mémoire statique et le fait d'effectuer tous les calculs en complexe, même lorsqu'il s'agissait d'additionner un et un...

Le séquenceur est alors réécrit à partir de rien, en C et Fortran sous système POSIX. Un grand pas est franchi grâce à l'apparition de la gestion dynamique de la mémoire rendant possible l'utilisation de données fortement typées. Le langage s'adjoint l'usage d'un préprocesseur et les commentaires apparaissent dans le code RPL/2. En 2001, le séquenceur inclut la bibliothèque Lapack 3.0 permettant un traitement sophistiqué des calculs matriciels, des points d'entrée autorisant l'usage de bibliothèques partagées compilées, ainsi que des routines de gestion des débordements arithmétiques et de minimisation des erreurs de cumul en virgule flottante. Le premier programme complexe passe les tests de validation avec succès. Les sorties ont aussi fortement évolué, utilisant gnuplot comme périphérique graphique, et LaTeX comme formateur des données à imprimer.

Des avantages du RPL/2

Le RPL/2 est un langage semi-compilé de très haut niveau 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 et vectoriels.

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. Faiblement aussi, car une donnée peut changer de type au cours d'une opération pour que la cohérence des calculs soit maintenue. 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.

Ce 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, cette liste n'étant de loin pas exhaustive. Par ailleurs, il est extensible par l'utilisateur, car il s'interface naturellement à toutes sortes de routines compilées (C, Fortran, Ada...).

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 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 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.

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.

Première partie

Concepts fondamentaux

Chapitre 1

Notations

1.1 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 le désavantage 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. Il faut aussi noter qu'une telle notation demande une interprétation complexe lors de l'évaluation de l'expression

1.2 Notation polonaise inverse

La logique principalement utilisée par le séquenceur RPL/2 est fondée sur une logique mathématique connue sous le nom de « notation polonaise inverse » 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 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 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. Néanmoins, le RPL/2 étant capable d'effectuer des calculs formel, il reste 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.

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 les boucles et les conditions. Dans ce cas, l'instruction est mise entre ses arguments.

Chapitre 2

Types de données

Les données manipulées par le RPL/2 sont fortement typées. Néanmoins, l'affectation d'un type à un objet se fait de manière automatique. 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.

2.1 Scalaires

2.1.1 Booléens

Les booléens prennent deux valeurs, vrai ou faux. La valeur fausse correspond à 0 et la valeur vraie, à tout scalaire non nul. Néanmoins, 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 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. En effet, si un calcul ne peut se poursuivre en entier du fait de l'impossibilité de représenter cette grandeur en entier, celle-ci est automatiquement convertie en réel.

2.1.3 Réels

Les réels utilisés sont par défaut des flottants codés sur 64 bits. Cette longueur dépend fortement 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 ±(MANT)E±(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 { $\texttt{objet}_1$ $\texttt{objet}_2$ ... $\texttt{objet}_n$ }.

2.5 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.

2.5.1 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.

2.5.2 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.

2.6 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'`.

2.7 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.

2.8 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.

Chapitre 3

Variables

Si les définitions intrinsèques ne sont pas sensibles à la casse, les variables le sont. Par ailleurs, le concept de variable RPL/2 est assez particulier. En effet, une variable n'est pas définie lors du lancement d'un programme et prendra automatique le type de la donnée qui y sera remisée.

D'autre part, il peut cohabiter simultanément plusieurs variables portant le même nom sans que cela soit préjudiciable au bon fonctionnement du système et sans perte d'information. En effet, chaque variable est associée à un niveau affecté par le séquenceur et la dotant d'une certaine visibilité :

- le niveau 0 correspond aux diverses définitions introduites dans le programme. Une variable de niveau 0 n'est pas directement accessible à l'utilisateur et ne comporte que des noms de définition ;
- le niveau 1 indique une variable globale, visible dans tout le programme ;
- un niveau strictement supérieur à 1 rend compte d'une variable locale soumise à certaines règles.

A priori, il est possible de nommer des définitions ou des variables par tout ce qui peut être représenté sous la forme d'un nom, mais sachant que le séquenceur donne la priorité aux définitions intrinsèques et non aux variables lors de l'exécution d'un programme. Ainsi, il est tout à fait possible de créer une définition s'appellant `SIN`, mais celle-ci sera masquée par la définition intrinsèque portant le même nom.

3.1 Définitions

Une définition est une expression en notation polonaise inverse prenant ses arguments sur la pile est y renvoyant ses résultats, et associé à un nom l'identifiant. Ce nom, sensible à la casse, doit être unique.

```
Ceci_est_ma_première_définition
<<
    "C'est un rêve modeste et fou" disp
    "Il aurait mieux valu le taire" disp
    "Vous me mettrez avec en terre" disp
    "Comme une étoile au fond d'un trou" disp
    "" disp "                    Aragon" disp
>>
```

La notion de définition est plus large que celle des routines ou fonctions Fortran ou C, car une définition peut prendre un nombre quelconque d'objets — voire un nombre variable — comme argument et en renvoyer un nombre quelconque sur la pile opérationnelle.

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.

```
CALCUL_DE_FACTORIELLE
<<
    "Calcul de n!" disp "n = " prompt str->
    if
        dup dup ip same
    then
        FACTORIELLE disp
    else
        drop
        "ERREUR : argument n non entier !" disp
    end
>>

/*
===============================================================
  Calcul récursif de factorielle
===============================================================
*/

FACTORIELLE
<<
    -> N
    <<
        if
            N 1 >
        then
            N dup 1 - FACTORIELLE *
        else
            1
        end
    >>
>>
```

Il existe trois types de définitions se distinguant par leur position vis à vis du langage.

3.1.1 Définitions intrinsèques

Les définitions intrinsèques correspondent aux instructions internes du langage. Elles s'opposent aux définitions extrinsèques et sont directement exécutables.

3.1.2 Définitions extrinsèques

Les définitions extrinsèques sont quant à elles des définitions externes au RPL/2 disponibles sous la forme de bibliothèques dynamiques compilées. Ces définitions sont trop spécifiques pour justifier une intégration en tant que définitions intrinsèques.

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, et ce quel que soit son nom.

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 `SIGMA`, 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 `STO`.

3.3 Variables locales

Contrairement aux variables globales, une variable locale n'est visible que dans une partie d'un programme, généralement une expression, que celle-ci soit algébrique ou en notation polonaise inverse. Ainsi, le petit programme suivant

```
Petit_programme_sans_prétention
<<
    1
    -> X
    <<
        X disp
    >>

    1 -> X 'X+1' disp X disp
>>
```

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

```
Petit_programme_sans_prétention_mais_plus_compliqué
<<
    1
    -> X
    <<
        X disp
        2 -> X 'X+1' disp

        1 3 for X
            X disp
        next

        X disp
    >>

    X disp
>>
```

a pour sortie

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

montrant par là que les variables locales ne s'écrasent pas mutuellement. Il faut néanmoins noter que si les variables locales ne sont pas perdues lors de la création de variables locales de même noms mais de niveau supérieur, il n'est pas possible d'y accéder. La seule exception à cette règle est le fonctionnement particulier des variables globales qui restent toujours accessible par un jeu de fonctions particulier (`RCL` et fonctions similaires).

3.4 Héritage

Le système des variables locales diffère sensiblement de celui qu'il est possible de rencontrer dans d'autres langages car une définition hérite — en lecture seule, les objets renvoyés par la définition devant impérativement passer par la pile — de toutes les variables locales de la définition appelante. Le système de masque de variable présenté précédemment permet alors d'utiliser n'importe quel nom de variable dans les définitions filles sans modifier les variables des définitions mères. Un petit exemple valant généralement mieux qu'un grand discours abscons, voici un petit programme mettant en lumière ce mécanisme d'héritage.

```
Programme_principal
<<
    1
```

```
    -> X
    <<
        X disp
        Définition_1
        X disp
    >>
>>

Définition_1
<<
    X disp X 1 +
    -> X
    <<
        X disp
    >>
>>
```

La sortie de ce programme est

```
1
1
2
1
```

montrant par là, la robustesse de la variable locale `X` de niveau 2 lors de la création de la nouvelle variable locale ainsi que l'exportation de cette variable dans la routine fille. Il faut néanmoins se défier des effets de bords que peut entraîner ce mécanisme. En effet, si l'instruction `sto` permet de créer une variable globale, elle se contente de modifier le contenu d'une variable locale si celle-ci préexiste.

Deuxième partie

Appel du RPL/2

Chapitre 4

Ligne de commande

Le séquenceur RPL/2 peut se charger de deux façons, soit explicitement par une commande système, soit par l'exécution d'un programme RPL/2. La première manière permettant de charger le séquenceur en mémoire consiste en son appel sur la ligne de commande du système d'exploitation. Cette instruction de lancement peut prendre diverses formes allant de `rpl` sous les shells Unix à `run rpl` sous OpenVMS en passant par un icone...

4.1 Options de la ligne de commande

Si le séquenceur est appelé d'une ligne de commande, il est possible de lui passer un nom de programme à exécuter ainsi qu'un certain nombre d'options sous la forme `rpl [-acdhilstv] [fichier_rpl/2]` avec

- `-a` donnant les coordonnées de l'auteur à toutes fins utiles, pour l'encourager, lui faire des remarques, lui soumettre de nouvelles fonctions intrinsèques ou modules externes ou lui envoyer du chocolat ;
- `-c` autorise lors d'une erreur la génération d'un fichier rpl-core contenant la pile, l'image du programme ainsi que la position de cette erreur ;
- `-d` permet de déverminer les allocations mémoire internes au RPL/2. Cette option n'est à utiliser que pour déverminer le séquenceur ;
- `-h` rappelant les options disponibles ;
- `-i` permettant un fonctionnement interactif. Le RPL/2 ne traite aucun programme, mais offre une invite permettant d'utiliser le séquenceur grâce à des lignes de commande. Ce mode de fonctionnement est incompatible avec la présence d'un fichier exécutable sur la ligne de commande ;
- `-l` rappelle la licence d'utilisation de l'outil ;
- `-s` empêche l'ouverture de l'écran graphique initial ;
- `-t` force la séquenceur à fournir une trace des opérations effectuées ;
- `-v` affichant le numéro de version du séquenceur.

Le RPL/2 est capable de prendre un script sur l'entrée standard et fourni le résultat sur la sortie standard.

```
rayleigh:[~] > echo 1 15.2 + X 3 \* sin - | rpl -is | \
> awk '/1: / {printf("%s\n", $2);}'
'16.2-SIN(X*3)'
rayleigh:[~] >
```

4.2 Fonctionnement interactif

Lors du lancement du séquenceur en mode interactif, celui propose une invite composé de trois signes plus. La ligne de commande est alors traitée lors de sa validation par un appui sur la touche « entrée ». Les instructions `kill`, `exit` et `abort` permettent de quitter le séquenceur.

À noter : la pile last est active par défaut.

Chapitre 5

Exécution de programmes

Si l'option `-i` n'est pas présente sur la ligne de commande, le séquenceur s'attend à trouver un fichier à exécuter.

À noter : la pile last est inactive par défaut.

5.1 Préprocesseur

Avant d'exécuter le contenu du fichier, le séquenceur appelle un préprocesseur chargé de traiter un certain nombre de macroinstructions ainsi que les commentaires.

Les 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 simplement ignorée.

Les macroinstructions comprises par le préprocesseur sont

- `#include` ;
- `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` et `#endif` ;
- `#define` et `#undef`.

5.1.1 Macroinstructions

L'instruction `#define` permet d'une part de créer des macroinstructions, et d'autre part de définir des variables qui pourront être utilisées lors d'exécutions conditionnelles.

5.1.2 Inclusions

L'instruction `#include` suivi d'un nom de fichier permet d'inclure le fichier spécifié au même endroit dans le code source. Le nom du fichier peut être spécifié de deux façons, d'une part par l'usage de crochets

```
#include <filtre.rpl>
```

signifiant que le fichier à inclure est un fichier système se trouvant dans l'arborescence des fichiers standard du séquenceur, et d'autre part par l'usage de guillemets

```
#include "topologie.rpl"
```

lorsqu'il s'agit d'un fichier faisant partie des sources d'un programme.

À noter : le fichier à inclure doit être un fichier source RPL/2 à l'exclusion de tout autre.

5.1.3 Exécutions conditionnelles

Les directives `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` et `#endif` permettent d'exécuter des parties de programme. La décision est prise par le préprocesseur et ne saurait en rien être modifée par le séquenceur, le code indésirable étant purement éliminé lors de la compilation. Ces directives étant celle du préprocesseur C, elles ne seront pas développées plus avant.

5.2 Organisation des programmes

Un programme RPL/2 est composé d'un ensemble de définitions. Il existe trois types de définitions, les définitions intrinsèques correspondant aux mots-clefs du langage, les définitions extrinsèques, disponibles dans des bibliothèques chargées dynamiquement par un programme, et les définitions utilisateur. Les définitions utilisateur se composent de deux entités :
- un nom de définition sensible à la casse. Ce nom doit être un identifiant unique et permet d'effectuer des branchements à cette définition. Lorsque ce nom survient dans un programme, la définition utilisateur est traité comme le serait une définition intrinsèque ou extrinsèque ;
- une expression en notation polonaise inverse.

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.

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 code source directement exécutable sans lancer explicitement le séquenceur RPL/2. Un tel code débutera par exemple par la ligne `#!/usr/local/bin/rpl`.

Troisième partie

Manipulations de la pile

Chapitre 6

Modifications automatiques

La pile opérationnelle est une liste contenant une collection d'objets de toute sorte. Tout objet remisé dans cette pile se retrouve au bas de celle-ci — au niveau 1 —, tous les autres objets remontant d'un niveau. Réciproquement, tout objet retiré de la pile l'est par le bas. Le nombre d'élément de la pile n'est limité que par la mémoire disponible sur le calculateur.

6.1 Empilement explicite

Le séquenceur scinde le code source en atomes à fin d'analyse. Tout d'abord, cet atome est comparé — cette opération n'est pas sensible à la casse — aux diverses définitions intrinsèques que comporte le langage. S'il s'agit d'un mot clef, l'opération correspondante est effectuée. Dans le cas contraire, l'analyse devient dépendante de la casse, et le séquenceur recherche s'il existe une définition extrinsèque — présente dans une bibliothèque chargée dynamiquement — correspondant à l'atome et l'exécute le cas échéant.

Si l'atome ne correspont ni à une définition intrinsèque ni à une définition externe, le séquenceur recherche une correspondance parmi les variables courantes. Cette opération reste dépendante de la casse de l'atome. S'il trouve une variable correspondant à l'atome, son contenu est empilé, sauf s'il s'agit d'un nom de définition utilisateur, auquel cas l'exécution du programme se poursuit par un branchement à ladite définition.

Toute donnée qui ne correspond à aucun des cas précédents est empilée. Cette opération provoque l'attribution d'un type à cette donnée.

6.2 Empilement implicite

Toute définition, intrinsèque ou non, est susceptible de renvoyer un certain nombre d'objet dans la pile. 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, mais leur position. Néanmoins et dans la mesure du possible, la pile retrouve un état cohérent même après tel arrêt.

6.3 Dépilement

Toute définition peut prendre ses arguments dans la pile. Ces objets sont retirés par le bas de la pile dans leur ordre d'apparition. S'il n'y a pas assez d'objets dans la pile pour satisfaire au nombre d'arguments requis par une définition, celle-ci renvoie une erreur.

Chapitre 7

Instructions spécifiques

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` effectue une *vraie* copie de l'objet, et crée un objet totalement indépendant de l'original.

$$\begin{array}{r|ccc|l} 2 & \text{objet}_2 & \longrightarrow & & \text{objet}_1 \\ 1 & \text{objet}_1 & \texttt{dup} & & \text{objet}_1 \end{array}$$

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.

$$\begin{array}{r|cc|l} 4 & & & \text{objet}_2 \\ 3 & & & \text{objet}_1 \\ 2 & \text{objet}_2 & \longrightarrow & \text{objet}_2 \\ 1 & \text{objet}_1 & \texttt{dup2} & \text{objet}_1 \end{array}$$

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 $n+1$.

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

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.

$$\begin{array}{r|cc|l} 3 & & & \text{objet}_2 \\ 2 & \text{objet}_2 & \longrightarrow & \text{objet}_1 \\ 1 & \text{objet}_1 & \texttt{over} & \text{objet}_2 \end{array}$$

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`.

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

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.

$$\begin{array}{r|cc|}
 & & \longrightarrow \\
1 & \text{objet}_1 & \texttt{drop}
\end{array}$$

7.3.2 Drop2

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

$$\begin{array}{r|cc|}
2 & \text{objet}_2 & \longrightarrow \\
1 & \text{objet}_1 & \texttt{drop2}
\end{array}$$

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.

$$\begin{array}{r|cc|}
n+1 & \text{objet}_n & \\
\vdots & \vdots & \\
2 & \text{objet}_1 & \longrightarrow \\
1 & n & \texttt{dropn}
\end{array}$$

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.

$$\begin{array}{r|cc|l}
2 & \text{objet}_2 & \longrightarrow & \text{objet}_1 \\
1 & \text{objet}_1 & \texttt{swap} & \text{objet}_2
\end{array}$$

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`.

$$\begin{array}{r|cc|l}
3 & \text{objet}_3 & & \text{objet}_2 \\
2 & \text{objet}_2 & \longrightarrow & \text{objet}_1 \\
1 & \text{objet}_1 & \texttt{rot} & \text{objet}_3
\end{array}$$

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`.

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

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.

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

Quatrième partie

Manipulation des variables

Chapitre 8

Accessibilité des variables

Il existe plusieurs sortes de variables, chacune associée à un niveau d'exécution conditionnant leurs porté et visibilité.

8.1 Niveaux d'exécution

Les niveaux d'exécution correspondent au nombre d'expressions en notation polonaise inverse imbriquées les unes dans les autres vues du points courant.

```
NIVEAU
<<
    'X+1' 'EQ' sto
    1 -> A
    <<
        A 1 +
        -> B
        <<
            B disp
        >>
    >>
>>
```

Les noms de définitions correspondant à leurs points d'entrée sont des variables de niveau 0. Ces variables ne sont pas modifiables par l'utilisateur au cours de l'exécution d'un programme. Les variables globales, accessibles en lecture et en écriture de n'importe quel point d'un programme sont des variables de niveau 1. Toutes les variables de niveau strictement supérieur à 1 sont des variables locales. Ainsi, dans la définition précédente

- `NIVEAU` est une variable de niveau 0 ;
- `EQ`, une variable globale de niveau 1 ;
- `A`, une variable locale de niveau 2 ;
- `B`, une variable locale de niveau 3.

Le niveau d'exécution d'une variable conditionne sa portée. En effet, à chaque point du programme correspond un niveau donné, toutes les variables existantes à ce point étant de niveau inférieur ou égal au niveau d'exécution courant. Les autres variables de niveau supérieur sont détruites.

8.2 Évaluation implicite

L'évaluation — ou l'accès en lecture — des variables ne peut se conduire que sur l'ensemble des variables. Cependant, l'action associée aux différents niveaux de variable diffère. En effet, l'évaluation d'une variable de niveau 0, correspondant à un point d'entrée d'une définition occasionne un branchement à cette définition, alors que l'évaluation d'une variable de niveau supérieur ou égal à 1 provoque un empilement de la donnée contenue dans cette variable.

Il faut noter que toute variable de niveau n est automatiquement masquée par une variable de niveau supérieur portant le même nom. Ainsi, l'évaluation d'une variable retourne l'objet contenu dans la variable de plus haut niveau disponible.

À noter : les points d'entrée des définitions étant des variables comme les autres, elles sont susceptibles d'être masquées par des variables globales ou locales.

Chapitre 9

Variables globales

Une variable globale est une variable de niveau 1. Elle est accessible en lecture de chaque point du programme et en écriture si elle n'est pas masquée. Sa portée est la totalité du programme. Sa visibilité est par défaut celle d'une variable locale. Elle peut ainsi être masquée, mais grâce à une instruction spécifique, elle reste toujours accessible en lecture.

9.1 Création

La création d'une variable globale par l'instruction `sto` est soumise à la non préexistence dans la définition courante d'une variable locale de même nom. Une variable globale est inconditionnellement créée par l'instruction `save`.

L'instruction `sto` prend deux objets dans la pile, un objet à mémoriser au niveau 2 ainsi qu'un nom de variable présent au niveau 1.

$$\begin{array}{r|cc|} 2 & \text{objet} & \longrightarrow \\ 1 & \text{'nom global'} & \texttt{sto} \end{array}$$

L'instruction `save` demande les mêmes arguments.

9.2 Modification

La modification d'une variable globale est permise grâce à la même instruction `sto`. Il faut noter que la modification d'une telle variable est soumise aux règles de visibilité des variables. La modification d'une variable globale n'est ainsi possible que si elle est accessible en écriture, donc non masquée par une variable locale.

Dans le cas où cette variable est masquée, il n'est pas possible d'accéder directement à l'objet qu'elle contient. Néanmoins, la définition intrinsèque `rcl` permet de contourner le phénomène de masque. L'instruction `save` permet elle aussi de contourner le mécanisme de masquage et d'écrire directement dans une variable globale.

$$\begin{array}{r|cc|l} & & \longrightarrow & \\ 1 & \text{'nom global'} & \texttt{rcl} & \text{objet} \end{array}$$

À noter : `rcl` n'agit que sur les variables globales. Un appel sur un nom local provoquerait une erreur.

9.3 Libération

Les variables globales n'étant pas libérées automatiquement par une décroissance du niveau d'exécution, il existe un mécanisme spécifique propre de destruction. La commande `purge` prend dans la pile un nom de variable globale ou une liste contenant plusieurs de ces noms et efface les variables globales correspondantes.

$$1 \left| \text{'nom global'} \quad \overset{\longrightarrow}{\texttt{purge}} \right|$$

$$1 \left| \{ \text{ 'nom global}_1\text{' } \ldots \text{ 'nom global}_n\text{' } \} \quad \overset{\longrightarrow}{\texttt{purge}} \right|$$

À noter : le mécanisme de libération des variables locales étant fortement différent de celui des variables globales, il n'est pas permis d'utiliser l'instruction `purge` sur ces dernières. Dans le cas contraire, `purge` retourne une erreur.

Chapitre 10

Variables locales

Une variable locale est une variable de niveau strictement supérieure à 1. Elle est accessible du bloc de programme (définition ou partie de définition) qui la définit et est détruite à la sortie de ce bloc. Elle peut néanmoins être statique.

10.1 Création

Une variable locale est créée par l'instruction « `->` ». Cette instruction ne peut se concevoir seule, mais en association avec une expression que celle-ci soit algébrique ou en notation polonaise inverse. Elle permet de créer simultanément une ou plusieurs variables, les variables étant ainsi créées pour la durée de l'expression suivant leur création.

$$
\begin{array}{c|cc|}
2n & \text{objet}_1 & \\
\vdots & \vdots & \\
n+1 & \text{objet}_n & \\
n & \text{'nom local}_1\text{'} & \\
\vdots & \vdots & \\
2 & \text{'nom local}_{n-1}\text{'} & \longrightarrow \\
1 & \text{'nom local}_n\text{'} & \texttt{->}
\end{array}
$$

10.2 Modifications

L'instruction `sto` permet de modifier le contenu d'une variable locale. S'il existe plusieurs variables de même nom, la variable effectivement modifiée est celle de niveau maximal.

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

À noter : contrairement aux variables globales, l'instruction `sto` ne permet pas de créer des variables locales.

10.3 Libération

La portée d'un variable locale est celle de l'expression — algébrique ou en notation polonaise inverse — suivant sa création. Lors de la sortie de cette expression, les variables locales créées au commencement de cette expression sont détruites. Il n'est ainsi pas nécessaire de recourir à un autre mécanisme de libération.

Chapitre 11

Arithmétique directe

Il est possible de travailler sur le contenu des variables en les évaluant puis, après modifications, en les sauvegardant à nouveau. Néanmoins, pour un certain nombre d'opérations simples, il est possible de recourir à des instructions d'arithmétique portant directement sur les variables, que celles-ci soient locales ou globales.

Cette écriture d'une part simplifie l'écriture des programmes, mais d'autre part permet une manipulation d'objets volumineux dans avoir à les copier dans la pile. Elles sont donc efficientes en terme de vélocité.

11.1 Les quatre opérations

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

11.1.1 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.

$$\begin{array}{r|ll|} 2 & \text{objet} & \longrightarrow \\ 1 & \text{'nom'} & \texttt{sto+} \end{array}$$

$$\begin{array}{r|ll|} 2 & \text{'nom'} & \longrightarrow \\ 1 & \text{objet} & \texttt{sto+} \end{array}$$

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.

11.1.2 Soustraction

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

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

de

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

En effet, la première écriture soustrait le contenu de la variable 'nom' à l'objet au niveau 2 avant que 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.

11.1.3 Multiplication

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

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

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

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

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

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

11.1.4 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 remise le résultat dans la variable. La règle dictant l'ordre des arguments est similaire à celle énoncée dans le cas de la soustraction directe.

11.2 Autres opération

Les autres opérations d'arithmétique directe se font sans limitation de type. Elles ont trait aux inversions, conjugaisons, et opposition.

11.2.1 Inversion

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

$$\begin{array}{r|ll|} & & \longrightarrow \\ 1 & \text{'nom'} & \texttt{sinv} \end{array}$$

11.2.2 Opposition

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

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

11.2.3 Conjugaison

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

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

Cinquième partie

Contrôle

Chapitre 12

Conditions et tests

Les différentes commandes de contrôle peuvent être combinées en un grand nombre de structure de décision et de traitement d'erreur. Il faut cependant noter que certains tests ont un comportement inclusif alors que d'autres ont un comportement exclusif.

12.1 Tests simples

12.1.1 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.

```
if X 0 > then "Positif" end disp
if X 0 >= then "Positif" else "Négatif" end disp
```

À noter : Il est possible d'imbriquer plusieurs tests sans perte de cohérence du résultat.

```
if
    X dup 0 >
then
    drop "Positif"
else
    if 0 < then "Négatif" else "Nul" end
end disp
```

12.1.2 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*`)`'

12.1.3 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`

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

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

12.2 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.

12.2.1 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.

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].

12.2.2 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`

Ainsi, la clause erreur est évaluée s'il est survenue une erreur durant l'exécution de la clause piège. Lorsque la clause piège est évaluée, les éléments de la clause sont successivement évalués, sauf si une erreur survient. Dans ce cas, l'exécution saute à la clause erreur, le reste de la clause piège étant ignoré. Par exemple

```
iferr while true repeat + end then "Fin" disp end
```

additionne tous les nombres présents dans la pile. La fonction `+` est répétée jusques à ce qu'une erreur se produise indiquant alors que la pile est vide ou qu'un type d'objet inadéquat a été trouvé. La clause erreur affiche alors `Fin`.

La structure complète `iferr/then/else/end` diffère de la structure précédente car elle permet de spécifier non seulement qu'une clause erreur doit être évaluée lorsque survient une erreur, mais aussi qu'une clause normale doit être exécutée s'il ne se produit aucune erreur.

Portée : la reprise sur erreur ne se fait que sur l'exécution de la clause piège et non sur l'ensemble du test.

À noter : lors de l'écriture des clauses d'erreur, il faut garder à l'esprit 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.

12.3 Tests multiples

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

Cette structure de contrôle permet d'imbriquer plusieurs tests mutuellement exclusifs — contrairement à la structure case — 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`.

[1] *cf.* indicateur 59

Les différentes conditions sont ici mutuellement exclusives. 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`

À noter : les clauses test ne sont évaluées que jusques à concurrence de la première occurence vraie.

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

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

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

Le mot clef `select` attend l'objet qui servira dans les différents tests de la structure de contrôle. Chaque instruction `case` empile l'objet retenu par `case` et évalue la séquence d'instructions jusques au mot clef `then`. Si le résultat de la clause est vrai, la clause vrai est évaluée. Dans le cas contraire, l'exécution reprend après l'instruction `end` correspondante au `then`. Toutes les clauses test sont évaluées, et toutes les clauses vraie sont susceptibles d'être exécutées. En revanche, la clause défaut n'est évaluée que si aucune clause vraie n'a été exécutée.

Chapitre 13

Boucles

13.1 Boucles définies

13.1.1 Remarque importante

Le séquenceur ne connaît pas *a priori* le sens de variation de l'indice de boucle définie, qu'il s'agisse de boucle `for` ou `start`. Celui-ci peut même être variable. De ce fait, le séquenceur effectuera au moins une fois la boucle, quitte à ce que cette dernière soit abrégée par un appel à la fonction `cycle`.

L'instruction `cycle` permet d'abréger une boucle. L'exécution normale de la boucle s'interrompt, le programme sautant à l'instruction `step` ou `next` cloturant la boucle en cours.

À noter : 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.

13.1.2 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 égale à la valeur *fin*, la clause boucle est à nouveau évaluée. Ce processus se poursuit jusques à ce que le compteur dépasse la valeur *fin*. L'exécution continue alors après l'instruction `next`.

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.

13.1.3 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. Néanmoins, 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* ;
- 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.

13.2 Boucles indéfinies

13.2.1 While...repeat...end

Cette structure évalue itérativement une clause test et une clause boucle tant que l'indicateur binaire renvoyé par la clause test est vrai (non nul).

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

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

13.2.2 Do...until...end

Cette structure évalue de manière répétée une clause boucle et une clause test jusques à ce que l'indicateur renvoyé par la clause test soit vrai (non nul).

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

13.3 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.

Chapitre 14

Contrôle de l'exécution

14.1 Mode de fonctionnement

L'état du système est assujetti à un certain nombre d'indicateurs binaires. Tous ces indicateurs sont accessibles et modifiables au travers d'un certain nombre d'instructions spécifiques.

14.1.1 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. Lors du lancement du séquenceur, ils prennent les valeurs par défaut suivantes :

- indicateur 31 armé : activation de la pile last ;
- indicateur 35 armé : utilisation des constantes symboliques ;
- indicateurs 37 à 42 : taille des mots utilisés lors de la représentation des entiers binaires, l'indicateur 42 étant le bit de poids fort. Ils sont la représentation de $n-1$, la longueur des mots binaires étant n. Par défaut, cette longueur est de soixante-quatre bits ;
- indicateurs 43 et 44 : base des entiers binaires. La base décimale correspond aux deux indicateurs désarmé, la base hexadécimale, aux deux indicateurs armés. L'indicateur 43 armé et le 44 désarmé indique un mode de fonctionnement en octal alors que la configuration contraire signifie un fonctionnement binaire. La base par défaut est la base décimale ;
- indicateur 45 armé : affichage des objets sur plusieurs lignes ;
- indicateur 48 désarmé : utilisation du point comme séparateur décimal. Si cet indicateur est armé, le séparateur décimal devient la virgule ;
- indicateurs 49 et 50 : format des nombre réels. Les deux indicateurs désarmés indiquent un format standard, les deux armés indiquent un format « ingénieur ». La virgule fixe est donnée par l'indicateur 49 armé et le 50 désarmé, la notation scientifique étant obtenue par la combinaison contraire. La notation par défaut est la notation standard ;
- indicateur 51 désarmé : autorisation d'avertissement sonore ;
- indicateurs 53 à 56 : nombre de chiffres utilisés dans les diverses représentations numériques, l'indicateur 53 correspondant au bit de poids fort ;

- indicateur 59 armé : traitement normal des dépassement de capacité par le report d'une erreur. Dans le cas contraire, le système considère une exception et est conduit à travailler sur une représentation de l'infini ;
- indicateur 60 armé : utilisation de radians lors de l'appel des diverses fonctions trigonométriques. Lorsque l'indicateur est désarmé, le séquenceur considère les arguments de ces fonctions exprimés en degrés.

14.1.2 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.

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

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

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.

14.2 Exécution normale

14.2.1 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. Il est néanmoins possible de forcer un retour anticipé — à la définition utilisateur appelante ou au système d'explotation — par l'instruction `return`. Cette instruction peut apparaître à n'importe quel endroit du code exécutable.

14.2.2 Abandon

Un programme s'arrête à la fin de la définition principale. Cependant, il est possible de forcer l'arrêt — définitif — de l'exécution du programme grâce à l'instruction `abort`. À l'instar de la fonction `return`, cette fonction peut apparaître n'importe où dans le programme.

À noter : les instructions `abort` et `return` sont équivalentes lorsqu'elles surviennent dans la définition principale.

14.3 Débogage

Le séquenceur possède un outil de débogage intégré. Il 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 et de modifier le contenu des variables ou les éléments de la pile.

14.3.1 Passage en mode débogage

Le passage en mode débogage se fait au travers de l'instruction `halt`. L'exécution normale du programme s'interrompt pour laisser la place à une exécution interactive. Il est possible d'arrêter définitivement l'exécution grâce à l'instruction `kill` ou de relancer l'exécution normale grâce à la commande `cont`.

14.3.2 Retour en exécution normale

L'instruction `cont` permet de relancer l'exécution normale d'un programme se déroulant en mode débogage. Si cela n'est pas le cas, cette instruction est ignorée sans renvoyer d'erreur.

14.3.3 Exécution pas à pas

L'exécution du programme en pas à pas est rendu possible par l'instruction `sst`. Chaque instruction `sst` entrée à l'invite de commande du mode débogage permet d'exécuter une définitions intrinsèque ou extrinsèque. Cette instruction entre dans toutes les définitions utilisateur rencontrées.

14.3.4 Abandon en cours de débogage

L'instruction `kill` abandonne un programme en cours de débogage. L'exécution est arrêté et provoque la sortie du séquenceur.

Sixième partie

Arithmétique

Chapitre 15

Les quatre opérations

15.1 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.

15.2 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 15.1 page 77.

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.

15.3 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 15.2 page suivante.

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.

15.4 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 15.3 page 78.

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.

15.5 Division

15.5.1 Division standard

L'instruction « / » calcul le quotient — l'obet 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.

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

15.5.2 Inversion

15.6 Puissance

15.6.1 Puissance standard

15.6.2 Carré

15.6.3 Racine carrée

15.6.4 Racine $n^{\text{ième}}$

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

TAB. 15.1 – 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

TAB. 15.2 – Soustraction

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

TAB. 15.3 – Multiplication

Chapitre 16

Arithmétique générale

16.1 Proportions

16.1.1 Instruction %

16.1.2 Instruction %CH

16.1.3 Instruction %T

Septième partie

Trigonométrie et logarithmes

Chapitre 17

Fonctions trigonométriques

17.1 Cosinus

17.2 Sinus

17.3 Tangente

Chapitre 18

Fonctions hyperboliques

18.1 Cosinus hyperbolique

18.2 Sinus hyperbolique

18.3 Tangente hyperboliques

Chapitre 19

Fonctions logarithmiques

19.1 Logarithme naturel

19.2 Logarithme vulgaire

Huitième partie

Algèbre linéaire

Chapitre 20

Vecteurs et matrices

Chapitre 21

Résolution

21.1 Inversion

21.2 Système linéaire

21.3 Moindres carrés

21.4 Moindres carrés généralisés

Chapitre 22

Décompositions

22.1 Vecteurs propres

22.2 Vecteurs propres généralisés

22.3 Décomposition de Cholesky

22.4 Décomposition LDU

22.5 Décomposition LU de Crout

22.6 Décomposition LQ

22.7 Décomposition QR

22.8 Décomposition de Schur

Neuvième partie

Statistiques

Chapitre 23

Description

23.1 Matrice de statistique

23.2 Moyenne

23.3 Variance et écart-type

Chapitre 24

Dénombrement

Chapitre 25

Lois de probabilité cumulées

25.1 Distribution de Laplace-Gauß dite normale

25.2 Distribution du χ^2

25.3 Distribution de Fisher

25.4 Distribution de Student

Dixième partie

Listes et tableaux

Onzième partie

Fichiers

Chapitre 26

Variable virtuelle

Chapitre 27

Fichiers à accès séquentiel

Chapitre 28

Fichiers à accès direct

Douzième partie

Graphisme

Chapitre 29

Dessin

Chapitre 30

Graphiques à deux dimensions

Chapitre 31

Graphiques à trois dimensions

Chapitre 32

Graphiques de statistiques

32.1 Nuages de points

32.2 Histogrammes

Treizième partie

Impression

Quatorzième partie

Optimisations

Chapitre 33

Du bon usage des variables

33.1 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.

33.2 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.

Chapitre 34

Bibliothèques partagées

34.1 Extensions compilées au RPL/2

34.2 Interfaces

34.3 Fonctions accessibles

Quinzième partie

Exemples

Chapitre 35

Programmes RPL/2

35.1 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

```
#!/usr/local/bin/rpl -c
// Script de calcul bestial de la la décomposition A=L*D*Lt

DECOMPOSITION
<<
    [[ 10 20 30 ][ 20 45 80 ][ 30 80 171 ]] // Matrice A

    dup size dup 1 get 1 ->list 0 con
    -> R S V
    <<
        1 S 1 get for J // Boucle de 1 à n
            1 J 1 - for I // Boucle de 1 à J-1
                if
                    J 1 same
                then
                    cycle
                end

                'V' I 1 ->list 'R' J I 2 ->list get 'R' I I 2 ->list get * put
            next

            'V' J 1 ->list
            'R' J J 2 ->list get

            // Produit scalaire à calculer

            0 1 J 1 - for K
                if
                    J 1 same
                then
```

```
                cycle
            end

            'R' J K 2 ->list get
            'V' K 1 ->list get * +
        next

        - put 'R' J dup 2 ->list 'V' J 1 ->list get put

        J 1 + S 1 get for K

            // Une boucle est toujours effectuée au moins une fois,
            // Donc, on saute au « next » si les indices sont trivialement
            // faux.

            if
                K S 1 get >
            then
                cycle
            end

            'R' K J 2 ->list dup2 get

            0 1 J 1 - for L
                if
                    J 1 same
                then
                    cycle
                end

                'R' K L 2 ->list get
                'V' L 1 ->list get * +
            next

            - 'V' J 1 ->list get / put
        next
    next

    R
  >>
>>
```

35.2 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.

Code source

```
#!/usr/local/bin/rpl -cs

/*
================================================================================
  Algorithme de l'Obèle
    Copyright 2001, BERTRAND Joël.
================================================================================
```

```
  Entrées : néant
--------------------------------------------------------------------------------
  Sorties : néant
--------------------------------------------------------------------------------
  Effets de bord : néant
================================================================================
*/

OBELE
<<
    rad 31 cf

    erase
    "" disp
    "Algorithme de l'obèle" disp
    "{\Large\sl Algorithme de l'obèle}" pr1 drop

    { "standard*(*)" }

    if
        "lambda" "existence" inquire
    then
        { "lambda" 'sequential' 'replace' 'writeonly' 'formatted' } open
    else
        { "lambda" 'sequential' 'new' 'writeonly' 'formatted' } open
    end

    format

    /* Paramètres d'entrée */

    4               // Nombre d'antennes
    64              // Nombre de mobiles
    64              // Facteur d'étalement
    true            // Présence de bruit
    1               // Seuil (contrainte à tenir en terme de C/I)
    1E-5            // Niveau de bruit
    1E-8            // Critère de convergence
    "Statistique"   // Modèle de canal ("Statistique" ou "Aléatoire")
    .5              // Distance entre les différents capteurs de l'antenne
    { }             // Directions des trajets (simple déclaration de variable)
    true            // Initialisation omnidirectionnelle (ou par un contrôle
                    // de puissance élémentaire)
    true            // Normalisation des diagrammes
    true            // Diagrammes d'antenne en coordonnées polaires
    false           // Impression du résultat
    3               // Nombre de paquets de mobiles (modèle Statistique)
    .02             // Dispersion en fraction de '2*PI' (modèle Statistique)
    true            // Egalité des puissance émises (modèle Statistique)
    true            // Un mobile par paquet tracé dans les diagrammes
    -> UNITE N_ANTENNES N_MOBILES FACTEUR_ETALEMENT ALGORITHME_BRUITE SEUIL
    BRUIT EPS MODELE_CANAL DIST DIRECTIONS INITIALISATION_OMNIDIRECTIONNELLE
    DIAGRAMME_NORMALISE DIAGRAMMES_POLAIRES AUTORISATION_IMPRESSION PAQUETS
    DISPERSION EQUIPUISSANCE TRACE_UN_MOBILE
    <<
        "" disp
        "\vskip 3ex\noindent" pr1 drop
        "Configuration" pr1 drop
        "\hrule\vskip 1ex" pr1 drop
        cr "Nombre d'antennes : " N_ANTENNES ->str + pr1 disp
        cr "Nombre de mobiles : " N_MOBILES ->str + pr1 disp
        cr "Type de canal     : " MODELE_CANAL ->str + pr1 disp
```

```
PAQUETS 1 ->list 0 con
-> REPARTITION
<<
    N_ANTENNES N_MOBILES MODELE_CANAL
    INITIALISATION_R 'DIRECTIONS' sto 'REPARTITION' sto
    if
        ALGORITHME_BRUITE
    then
        SEUIL CONVERSION_ALGORITHME_BRUITE
    end

    if
        INITIALISATION_OMNIDIRECTIONNELLE
    then
        N_ANTENNES 1 2 ->list 0 con { 1 1 } 1 put
        N_MOBILES 1 - dupn N_MOBILES ->list
    else
        dup OPTIMISATION_SIMPLE
    end

    N_MOBILES N_MOBILES 2 ->list 0 con -1 0 0

    -> LISTE_R V_PONDERATION F AVP VP PUISSANCES_INITIALES
    <<
        rclf 2 sci
        1 N_MOBILES for J
            rclf std "  Utilisateur " J ->str + disp stof
            LISTE_R J get disp "" disp
        next
        stof

        /* Calcul des rapports C/I initiaux */

        LISTE_R V_PONDERATION

        'F' { 1 1 }
        1 N_MOBILES for J
            1 N_MOBILES for K
                if
                    J K same
                then
                    0
                else
                    V_PONDERATION K get
                    dup trn
                    LISTE_R J get
                    rot * * { 1 1 } get
                end
                puti
            next
        next
        drop2

        if
            ALGORITHME_BRUITE not
        then
            F regv max swap drop list-> drop

            -> M C
            <<
                1 N_MOBILES for J
```

```
            M J C 2 ->list get re
        next

        N_MOBILES 1 ->list ->array dup abs /
    >>
else
    N_MOBILES idn N_MOBILES 1 2 ->list SEUIL BRUIT * con swap
    F - inv swap * re array-> list-> drop2 1 ->list ->array
end

2 sci
"\vskip 3ex\noindent" pr1 drop
"  Rapports C/I initiaux" pr1 disp
"\hrule\vskip 1ex" pr1 drop
"" disp
dup array-> 1 get ->list 1

true
-> AUTORISATION_CALCUL
<<
    do
        geti

        if
            0 <
        then
            false 'AUTORISATION_CALCUL' sto
        end
    until
        dup 1 same
    end

    drop2

    if
        AUTORISATION_CALCUL
    then
        V_PONDERATION

        over 720 DIAGRAMME

        if
            DIAGRAMMES_POLAIRES not
        then
            { "Azimut" "Puissance" } label
        end

        "Diagrammes avant optimisation" title persist prlcd
        cllcd

        dup
        -> PUISSANCES
        <<
            0
            1 PUISSANCES size 1 get for I
                PUISSANCES I 1 ->list get +
            next
        >>

        'PUISSANCES_INITIALES' sto

        if
```

```
            ALGORITHME_BRUITE not
        then
            0
        else
            BRUIT
        end

        CALCUL_RAPPORTS_SIGNAUX_INTERFERENCE
    else
        3 dropn
        "    Résolution impossible du système"
    end

    pr1 disp
    "" disp
>>

/* Boucle principale */

"\vskip 3ex\noindent" pr1 drop
"  Minimisation de la plus grande valeur propre Lambda" disp
"  Minimisation de la plus grande valeur propre $\lambda$"
pr1 drop
"\hrule\vskip 1ex" pr1 drop
"" disp std

while
    VP AVP - abs EPS >
repeat

    /* Normalisation des pondérations */

    1 N_MOBILES for J
        LISTE_R J get V_PONDERATION J get
        NORMALISATION array-> drop

        N_ANTENNES 1 2 ->list ->array
        'V_PONDERATION' swap J swap put
    next

    /* Calcul de la matrice F */

    'F' { 1 1 }
    1 N_MOBILES for J
        1 N_MOBILES for K
            if
                J K same
            then
                0
            else
                V_PONDERATION K get
                dup trn
                LISTE_R J get
                rot * * { 1 1 } get
            end
            puti
        next
    next
    drop2

    /* Calcul du plus grand vecteur propre gauche de F */
```

```
F legv
-> MATRICE
<<
    /* Réécriture de la fonction max pour éviter les    */
    /* erreurs numériques d'arrondis apparaissant avec    */
    /* les racines doubles du polynôme caractéristique    */

    do
        MATRICE max
    until
        over re 0 >=

        if
            dup not
        then
            swap 'MATRICE' swap 0 put swap drop
        end
    end
>>

list-> drop rot swap
-> COLONNE
<<
    1 N_MOBILES for J
        dup J COLONNE 2 ->list get re swap
    next

    drop
>>

VP 'AVP' sto
N_MOBILES 1 2 ->list ->array
swap re dup 'VP' sto pr1
UNITE over 1 ->list swap write
"Lambda = " swap ->str + disp

/* Normalisation du plus grand vecteur gauche de F */

dup abs /

/* Calcul des matrices T */

-> PG
<<
    1 N_MOBILES for J
        LISTE_R 1 get 0 con

        1 N_MOBILES for K
            if
                J K same not
            then
                LISTE_R K get PG K 1 2 ->list get * +
            end
        next
    next
>>

N_MOBILES ->list

/* Calcul des vecteurs propres généralisés des matrices T */

-> LISTE_T
```

```
    <<
        1 N_MOBILES for J
            LISTE_T J get LISTE_R J get gregv re min swap drop
            list-> drop

            -> COLONNE
            <<
                1 N_ANTENNES for K
                    dup K COLONNE 2 ->list get swap
                next

                drop N_ANTENNES 1 ->list ->array dup abs /
                N_ANTENNES 1 2 ->list rdm
                'V_PONDERATION' J rot put
            >>
        next
    >>
end

/* Normalisation des pondérations */

1 N_MOBILES for J
    LISTE_R J get V_PONDERATION J get
    NORMALISATION array-> drop

    N_ANTENNES 1 2 ->list ->array
    'V_PONDERATION' swap J swap put
next

/* Pondérations */

"" disp
"  Pondérations optimales" pr1 disp
"\hrule\vskip 1ex" pr1 drop
"" disp

1 N_MOBILES for J
    "W(" std J ->str + ") = " + 2 sci
    'V_PONDERATION' J get N_ANTENNES 1 ->list
    rdm pr1 ->str + disp
next

/* Calcul des puissances par mobile nécessaires */

"" disp
"\vskip 3ex\noindent" pr1 drop
"  Calcul des puissances par mobile nécessaires" pr1 disp
"\hrule\vskip 1ex" pr1 drop
"" disp

if
    ALGORITHME_BRUITE not
then
    F regv max swap drop list-> drop

    -> M C
    <<
        1 N_MOBILES for J
            M J C 2 ->list get re
        next

        N_MOBILES 1 ->list ->array dup abs /
```

```
    >>

    V_PONDERATION over 720 DIAGRAMME

    if
        DIAGRAMMES_POLAIRES not
    then
        { "Azimut" "Puissance" } label
    end

    "Diagrammes après optimisation" title persist prlcd
else
    if
        VP 1 >
    then
        "    Absence de solution physique !"
    else
        N_MOBILES idn N_MOBILES 1 2 ->list
        SEUIL BRUIT * con swap
        F - inv swap * re array-> list-> drop2
        1 ->list ->array

        V_PONDERATION over 720 DIAGRAMME

        if
            DIAGRAMMES_POLAIRES not
        then
            { "Azimut" "Puissance" } label
        end

        "Diagrammes après optimisation" title persist prlcd
    end
end

V_PONDERATION over 2 ->list "resultat_obele" store
{ "graphique.eps" "postscript eps enhanced monochrome dashed" }
lcd->

dup

if
    dup type 2 same
then
    pr1
else
    dup array-> 1 get ->list pr1 drop
end

if
    dup type 2 same not
then
    "P = " swap ->str + disp "" disp

    LISTE_R V_PONDERATION rot

    dup
    -> PUISSANCES
    <<
        0
        1 PUISSANCES size 1 get for I
            PUISSANCES I 1 ->list get +
        next
```

```
                    >>

                    if
                        PUISSANCES_INITIALES 0 same not
                    then
                        rclf swap 3 fix PUISSANCES_INITIALES swap %ch neg ->str
                        " %" + swap stof
                    else
                        drop "absurde"
                    end

                    "\vskip 3ex\noindent" pr1 drop
                    "  Rapports C/I finaux "
                    "(amélioration de la puissance émise : " +
                    swap ->str + ")" + pr1 disp

                    if
                        ALGORITHME_BRUITE not
                    then
                        0
                    else
                        BRUIT
                    end

                    CALCUL_RAPPORTS_SIGNAUX_INTERFERENCE

                    "\hrule\vskip 1ex" pr1 drop
                    "" disp
                    pr1 disp
                    "" disp

                    cllcd

                    if
                        AUTORISATION_IMPRESSION
                    then
                        print
                    else
                        erase
                    end
                else
                    drop disp "" disp
                    erase
                    cllcd
                end
            >>
        >>

        UNITE close
    >>

    "  Temps CPU utilisé : " disp 2 fix time disp std
>>


/*
================================================================================
  Calcul des rapports C/I pour chaque mobile
================================================================================
  Entrées :
    4: liste contenant les matrices R de chaque mobile
    3: liste contenant les pondérations affectées à chaque mobile
```

```
    2: vecteur contenant les puissances
    1: sigma ** 2
--------------------------------------------------------------------------------
  Sorties :
    1: liste contenant les rapports C/I
--------------------------------------------------------------------------------
  Effets de bord : néant
================================================================================
*/

CALCUL_RAPPORTS_SIGNAUX_INTERFERENCE
<<
    -> R W P SIGMA
    <<
        1 N_MOBILES for I
            P I 1 ->list get
            W I get dup trn swap
            R I get swap * * * { 1 1 } get re

            SIGMA
            1 N_MOBILES for J
                if
                    I J same
                then
                    cycle
                end

                P J 1 ->list get
                W J get dup trn swap
                R I get swap * * * { 1 1 } get re +
            next

            / FACTEUR_ETALEMENT *
        next
    >>

    N_MOBILES ->list
>>


/*
================================================================================
  Fonction de normalisation des vecteurs W de telle sorte que trn(W)*R*W = 1
================================================================================
  Entrées :
    2: matrice R
    1: vecteur W
--------------------------------------------------------------------------------
  Sorties :
    1: vecteur W normalisé
--------------------------------------------------------------------------------
  Effets de bord : néant
================================================================================
*/

NORMALISATION
<<
    -> R W
    <<
        W dup trn R FACTEUR_ETALEMENT *

        if
```

```
            ALGORITHME_BRUITE
        then
            SEUIL /
        end

        W * * abs sqrt /
    >>
>>


/*
================================================================================
  Fonction renvoyant une liste contenant les différentes matrices R
================================================================================
  Entrées :
    3: nombre d'antennes (entier)
    2: nombre de mobiles (entier)
    1: nombre de trajets (entier)
--------------------------------------------------------------------------------
  Sorties :
    2: liste contenant autant de matrices R qu'il y a de mobiles
    1: directions des mobiles
--------------------------------------------------------------------------------
  Effets de bord : néant
================================================================================
*/

INITIALISATION_R
<<
    "" disp
    "  Initialisation des matrices d'autocorrélation du canal" disp
    "" disp

    "\vskip 3ex\noindent" pr1 drop
    "Positions et puissances des différents récepteurs" pr1 drop
    "\hrule\vskip 1ex" pr1 drop

    { } dup
    -> NA NM MODELE DIRECTIONS REPARTITION_INTERNE
    <<
        rclf

        if
            MODELE "Statistique" same
        then
            PAQUETS 1 ->list 0 con 'REPARTITION_INTERNE' sto

            /*
            Génération de matrices de covariance du canal grâce
            au modèle du Statistique
            */

            deg 4 fix { 3 9 } 0 con
            DIST 180 25 0 0 PAQUETS 1 ->list 0 con
            -> COEFF    // Coefficients du modèle :
                        // - ligne 1 : direction du trajet en degrés;
                        // - ligne 2 : puissance du trajet en dB;
                        // - ligne 3 : retard du trajet en ns.
            D           // Distance entre deux capteurs consécutifs comptée en
                        // longueur d'onde
            SECTEUR     // Demi angle d'ouverture d'un secteur (en degrés)
            DTHETA      // Paramètre d'ouverture du modèle (en degrés)
```

```
GISMIN      // Gisement le plus faible vu du secteur
GISMAX      // Gisement le plus grand vu du secteur
ANGLES_MOYENS
<<
    SECTEUR neg DTHETA 4 * - 'GISMIN' sto
    SECTEUR DTHETA 2 * + 'GISMAX' sto

    'COEFF' { 1 1 }

    0 puti
    DTHETA 2 / puti
    DTHETA 2 / neg puti
    DTHETA 2 / 1 - puti
    1 DTHETA 2 / - puti
    2 DTHETA * puti
    -2 DTHETA * puti
    3 DTHETA * puti
    4 DTHETA * puti

    -2 puti
    -7 puti
    -7 puti
    -4 puti
    -4 puti
    -9 puti
    -10 puti
    -15 puti
    -20 puti

    0 puti
    0 puti
    0 puti
    310 puti
    310 puti
    710 puti
    1090 puti
    1730 puti
    2510 puti

    drop2

    0 -> CUMUL
    <<
        1 COEFF size 2 get for I
            'COEFF' 2 I 2 ->list 'COEFF' over get 10 / alog
            dup 'CUMUL' sto+ put
        next

        1 COEFF size 2 get for I
            'COEFF' 2 I 2 ->list 'COEFF' over get CUMUL / put
        next
    >>

    // Calcul de la répartition des mobiles dans les paquets
    'REPARTITION_INTERNE' { 1 }
    1 PAQUETS for P
        rand puti
    next
    drop2

    REPARTITION_INTERNE array-> 1 get 2 swap for P + next
```

```
-> CLEF
<<
    1 PAQUETS for P
        'REPARTITION_INTERNE' dup P 1 ->list get NM * CLEF /
        ip P 1 ->list swap

        if
            dup 1 <
        then
            drop 1
        end

        put
    next
>>

0 1 PAQUETS for P
    'REPARTITION_INTERNE' P 1 ->list get +
next

NM -
-> DIFFERENCE
<<
    if
        DIFFERENCE 0 >
    then
        while
            DIFFERENCE
        repeat
            rand PAQUETS * ip 1 + 1 ->list dup
            if
                'REPARTITION_INTERNE' swap get dup 1 >
            then
                1 - 'REPARTITION_INTERNE' rot rot put
                'DIFFERENCE' 1 sto-
            else
                drop2
            end
        end
    else
        while
            DIFFERENCE
        repeat
            rand PAQUETS * ip 1 + 1 ->list dup
            if
                'REPARTITION_INTERNE' swap get dup NM <
            then
                1 + 'REPARTITION_INTERNE' rot rot put
                'DIFFERENCE' 1 sto+
            else
                drop2
            end
        end
    end
>>

'ANGLES_MOYENS' { 1 }
1 PAQUETS for P
    rand 2 SECTEUR * * SECTEUR - puti
next
drop2
```

```
rclf std
"Répartition : " REPARTITION_INTERNE ->str + pr1
"\hrule\vskip 1ex" pr1 drop
disp "" disp stof

// Boucle sur les paquets de mobiles
1 PAQUETS for P

    // Boucle sur les mobiles
    1 REPARTITION_INTERNE P 1 ->list get for K
        rand 2 SECTEUR * * SECTEUR - DISPERSION *
        ANGLES_MOYENS P 1 ->list get +
        DIRECTIONS over 1 ->list + 'DIRECTIONS' sto
        "Azimut : " over ->hms ->str + " ° (HMS)" + cr pr1 disp

        NA COEFF size 2 get 2 ->list 0 con
        COEFF size 2 get dup 2 ->list 0 con 0
        -> AZIMUT MD P TRAJETS_RETENUS
        <<
            1 COEFF size 2 get for I
                'COEFF' 1 I 2 ->list get AZIMUT +
                -> G
                <<
                    if
                        G SECTEUR <= G SECTEUR neg >= and
                        SECTEUR 180 >= or
                    then
                        1 NA for J
                            'MD' 2 i pi * * ->num
                            D J 1 - * * G sin * exp
                            J I 2 ->list swap put
                        next

                        'P' COEFF 2 I 2 ->list get
                        I dup 2 ->list swap put
                        1 'TRAJETS_RETENUS' sto+
                    end
                >>
            next

            MD P

            if
                EQUIPUISSANCE
            then
                1
            else
                nrand sq
            end

            "Puissance : " over ->str + cr pr1 disp
            * over trn * *

            // Rajout de bruit pour éviter d'avoir une
            // matrice R de rang non plein

            if
                TRAJETS_RETENUS over size 1 get <
            then
                dup idn over abs 1E-6 / * +
            end
        >>
```

```
                next
            next
        >>

        rad
    else

        /* Génération de matrices R aléatoires */

        // Nombre de trajets
        4 -> NT
        <<
            // Boucle sur les mobiles
            1 NM for K
                "  Utilisateur " std K ->str + disp "" disp
                NT NA 2 ->list 0 con

                // Boucle sur les trajets
                1 NT for L
                    L 1 2 ->list
                    rand 2 pi ->num * *
                    DIRECTIONS over r->d 1 ->list + 'DIRECTIONS' sto
                    nrand sq

                    -> D P
                    <<
                        std
                        "    Trajet " L ->str + disp
                        4 sci "      -> Puissance " P ->str + disp
                        4 fix "      -> Azimut    " D r->d ->hms ->str +
                        "° (HMS)" + disp "" disp

                        // Boucle sur les capteurs
                        1 NA for J
                            i ->num D sin J 1 - * * DIST *
                            2 pi ->num * 2E9 * * exp P * puti
                        next
                        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
    <<
        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
            <<
                1 over size 1 get for M
                    dup M 1 2 ->list get 'TABLEAU' swap M 1 2 ->list swap put
                next
                drop LISTE N get TABLEAU NORMALISATION
            >>
        next

        LISTE size ->list
    >>
>>


/*
```

```
===============================================================================
  Calcul du diagramme de rayonnement du réseau d'antennes
===============================================================================
  Entrées :
    3: liste contenant tous les vecteurs de pondération
    2: puissances
    1: nombre de points à calculer par diagramme
-------------------------------------------------------------------------------
  Sorties :
    néant
-------------------------------------------------------------------------------
  Effets de bord : néant
===============================================================================
*/

DIAGRAMME
<<
    0
    -> PONDERATIONS PUISSANCES NB_POINTS MAXIMUM
    <<
        cllcd

        { { 60 "ticsonly" 2 } { "automatic" "ticsonly" 10 } } axes

        if
            DIAGRAMMES_POLAIRES
        then
            1 d->r
            -> PAS
            <<
                0
                1 N_MOBILES for I
                    0 2 pi ->num * for T
                        if
                            FCT_DIAGRAMME dup 3 pick >
                        then
                            swap
                        end
                        drop PAS
                    step
                next
            >>

            dup 'MAXIMUM' sto dup r->c dup pmax neg pmin

            parametric { T 0 'MAXIMUM' } indep MAXIMUM res
            << FCT_DIRECTIONS >> steq

            1 N_MOBILES for I
                draw
            next

            polar { T 0 '2*PI' } indep 2 pi ->num * NB_POINTS / res
            << FCT_DIAGRAMME >> steq

            if
                TRACE_UN_MOBILE
            then
                1
                -> I
                <<
                    1 PAQUETS for POINTEUR
```

```
                        draw
                        'I' REPARTITION POINTEUR 1 ->list get ip sto+
                    next
                >>

                "Et ici" disp
            else
                1 N_MOBILES for I
                    draw
                next
            end
        else
            { X Y } autoscale 'Y' logscale
            parametric { T '-PI' 'PI' } indep 2 pi ->num * NB_POINTS / res
            << T r->d FCT_DIAGRAMME r->c >> steq

            if
                TRACE_UN_MOBILE
            then
                1
                -> I
                <<
                    1 PAQUETS for POINTEUR
                        draw
                        'I' REPARTITION POINTEUR 1 ->list get ip sto+
                    next
                >>
            else
                1 N_MOBILES for I
                    draw
                next
            end
        end
    >>

    drax
>>


FCT_DIRECTIONS
<<
    rclf deg
    DIRECTIONS I get dup cos T * swap sin T * i ->num * +
    swap stof
>>


FCT_DIAGRAMME
<<
    N_ANTENNES 1 ->list i ->num 2 pi ->num * * T sin DIST * * con

    1 N_ANTENNES for J
        dup J 1 ->list get J 1 - * exp
        J 1 ->list swap put
    next

    PONDERATIONS I get array-> 1 get 1 ->list ->array swap dot abs 2 /

    if
        DIAGRAMME_NORMALISE not
    then
        PUISSANCES I 1 ->list get *
```

```
    end
>>
```

Chapitre 36

Bibliothèques

Instructions

Index général