Reverse Polish Lisp/2

— Reference Manual —

Joël BERTRAND[1]

October 16, 2001

[1]Translated by David HARCOMBE

RPL/2

Copyright © 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001 BERTRAND Joël.

This reference manual is maintained by the author. All rights reserved worldwide.

Contents

Foreword: About a Turing Machine in RPL/2 **7**

Genesis **9**

RPL/2, A History **9**
RPL 6502, the beginning . . . 11
RPL 6809, an uncompleted test . . . 11
RPL 386, version 3.00 . . . 11
RPL 387, version 3.0x . . . 11
RPL/2, a new beginning . . . 11

The advantages of RPL/2 **12**

I Fundamental concepts **15**

1 Notations **17**
1.1 Agebraic notation . . . 17
1.2 Reverse Polish Notation . . . 17
1.3 Internal Notation . . . 18

2 Variable Types **19**
2.1 Scalars . . . 19
2.1.1 Booleans . . . 19
2.1.2 Integers . . . 19
2.1.3 Reals . . . 19
2.1.4 Complex numbers . . . 20
2.2 Vectors . . . 20
2.3 Matrices . . . 20
2.4 Lists . . . 20
2.5 Expressions . . . 20
2.6 Algebraic expressions . . . 21
2.6.1 RPN expressions . . . 21
2.7 Names . . . 21
2.8 Character strings . . . 21
2.9 Binaries . . . 21

3 Variables **23**
3.1 Definitions 23
3.1.1 Intrinsic definitions 24
3.1.2 External definitions 24
3.1.3 User definitions 25
3.2 Global variables 25
3.3 Local variables 25
3.4 Inheritance 26

Index **28**
Instructions 29
Index 31

Preface
About a Turing Machine in RPL/2

Genesis

Reverse Polish Lisp/2 – or RPL/2 – came about from the need to be able to program formal algorithms quickly, reliably and with a way of minimizing the computational errors inherent in the numerical drifts and overflows. Not having found a programming language which met these criteria, and having a little spare time, I began to write a language which represented all of the existing concepts[1] whilst minimizing computational errors and adding functions required in everyday use. After several years, the language (although still a little experimental) has now developed to the point where it is an easily used, semi-compiled object language.

[1] RPL for the Hewlett-Packard HP-23S calculator

RPL/2, A History

RPL 6502, the beginning

The first operational version of this language was written in 1989 on a 48Kb Oric Atmos in Tangerine Basic v1.1. It only worked in strict reverse polish notation and on real, single precision numbers. The restrictions of the available memory and the modest power of the machine made it impossible to deal with complex objects such as matrices. Nevertheless, it was usable and became the foundation of all of the other programs.

RPL 6809, an uncompleted test

The second version, written in 1990, was never finished. It was supposed to run on an SMT-Goupil G3/6809 with 320Kb of memory, but all of the memory above 64Kb was only accessible through a set of virtual tables on a system of memory files... This workstation ran on TSC-Flex9, and the program was coded in SBASIC. The great novelty lay in the use of disks and the introduction of file management, as well as the appearance of symbolic calculations.

RPL 386, version 3.00

1991 saw a third version of the program, running on an IBM PS/2 P70, a 16MHz 80386DX - a very reasonable processor at the time. Graphing commands also made their first appearance. The program was still written in BASIC, but this time in TurboBasic on IBM-DOS 5.0...

RPL 387, version 3.0x

Something happened in the innards of the IBM; it received a maths co-processor (an 80387DX). Due to the speed gains from that component, it was possible to carry out calculations on complex sets. In the end, to make life simpler, all calculations were done this way...

RPL/2, a new beginning

At the start of 1998, the program was hindered by decisions made 10 years earlier and motivated by the calculation workstations now at my disposal. The processing power

now available made obsolete the choice of performing the calculations in static memory and allowed all of the calculations to be done as sets, even if it were only a matter of adding 1 and 1...

The program has now been rewritten from scratch in C and FORTRAN under a POSIX system. A large part is clearly thanks to the appearance of dynamic memory management making possible the use of strongly typed data. The language also makes use of a preprocessor, and comments can appear in RPL/2 code. In 2001, the program included the Lapack 3.0 library, allowing a sophisticated treatment of matrix calculations, entry-points to allow for the use of pre-compiled shared libraries, as well as management of arithmetic overflows and minimizing cumulative floating point errors. The first complex program passed the validation tests successfully. The exports have also evolved, using gnuplot as a peripheral program to perform graphics and LaTeXto transform the data for printing.

The advantages of RPL/2

RPL/2 is a very high level semi-compiled language allowing the user to constantly work at the maximum precision available, whilst avoiding the numeric overflows inherent in the different representations used in calculators. It is very fast, and thanks to the usage of reverse polish notation, it is capable of performing – in addition to complex arithmetic – symbolic and vector calculations.

The language is at once both strongly and weakly typed. Strongly because there exists many distinct types of data, but weakly because the association of these types with others is done implicitly. In effect, an object acquires a type at the time of its first use. In the same way, a variable doesn't exist until after its first use and doesn't need to be declared. Before use, it simply doesn't exist, allowing the storage of different types under the same name throughout the course of a program. It is also weakly typed because a piece of data can change its type during the course of an operation as provided the integrity of the calculation is maintained. It should be noted that the calculations are carried out at the highest precision and the minimum cost, which is to say that the type given to a piece of data is always the smallest type that can represent it correctly.

The RPL/2 compiler is designed to be used as generally as possible. It has functions for dealing with statistical data, graphing possibilities, functions for creating output as PostScript files generated by a TEXengine and file management facilities to name but a few. In addition, it is extensible by the user as it can be seamlessly integrated with all sorts of compiled routines (written in C, FORTRAN or Ada...).

RPL/2 also avoids all of the problems of memory allocation. This function – dynamic and laborious in most languages – is left to the compiler. It is completely transparent and since it is subject to a memory access protection system, the only errors possible are those errors in the programs themselves. These errors do not necessarily force the program to stop – unless they are a system error – as it is possible to manage them via an error recovery mechanism. Nevertheless, is an error is not masked, it will cause an abnormal termination of the program's execution and can create a trace file containing information about the error. A step-by-step mode can also help with debugging and error correction.

Finally, it is possible to use the compiler in an interactive mode by means of a command line. This means you can perform simple calculations as if they were run on a standard reverse polish notation calculator.

Part I

Fundamental concepts

Chapter 1

Notations

1.1 Agebraic notation

This notation allows the creation an expression in its normal algebraic form, then to calculate the result at the time of evaluation. This approach has the benefit of conserving the correspondance between the expression as they are written on paper and their keyboard input, but has the general disadvantage of not producing intermediate results. You also need to know the complete form of an expression before beginning to input it, which can sometimes be difficult when you are moving towards a solution involving the change of the calculation based on intermediate results. Note also that this notation requires a complex interpreteation before the expression can be evaluated.

1.2 Reverse Polish Notation

The principal logic used by the RPL/2 compiler is founded upon a mathematical logic known as "Reverse Polish Notation", developed by the Polish logician Jean Łukasiwiecz (1878-1956). Whereas conventional agebraic notation places the operators *between* the numbers or variables of an expression at the time of its evaluation, the notations introduced by Łukasiwiecz specifies the operators *before* the variables. A variant of this logic places the operators *after* the variables, and is also called "Reverse Polish Notation"

The main idea of RPN is that the numbers and other objects are keyed in first, and afterwards a command which acts on these numbers or objects or "arguments" of the command. The "operational stack" is a set of these objects which are waiting to be used. The majority of commands remove objects from the stack and replace them with their results, where they can in turn be used as arguments to other commands.

RPL/2 uses this logic for many reasons, the first being that it avoids the costly interpretation necessary for algebraic notation, but also because it gives the flexibility necessary for the uniform execution of calculations permitted by the language. All operations, even those which cannot be expressed in an algebraic form, are executed in the same way - the arguments placed on the stack and the results placed back there.

This said, the use of RPN remains the major obstacle for most users of RPL/2, since, although it is very efficient, it requires a mental rearrangement of the instruction sequences of an expression before being able to calculate the result. Notwithstanding,

RPL/2 remains capable of performing formal calculations; it is possible to use expressions in algebraic notation, but to the detriment of the speed of the language. One of its essential advantages lies in the use of these two logics and that the end user doesn't have to worry about whether RPN is better of worse than algebraic notation. They can choose the system that is best adapted to their needs, and mix classic algebraic notation and RPN.

1.3 Internal Notation

Certain language functions are carried out in an internal format for reasons of clarity. These are principally to do with loops and conditions. In these cases, the instruction is placed between its arguments.

Chapter 2

Variable Types

The variables manipulated by RPL/2 are strongly typed. Nevertheless, the assignment of a type to an object is performed automatically. It should also be noted that an inheritance mechanism exists between certain types, to permit coherent calculation results. In this way, if a calculation cannot be completed in integers, it will move from real to complex numbers.

2.1 Scalars

2.1.1 Booleans

Booleans have two values, true or false. False corresponds to 0, and true to all non-zero scalars. The following two symbolic constants are used:

- `true` value -1 as RPL/2 treats logic operations bitwise.
- `false` value 0

2.1.2 Integers

The nominal length of integers is 64 bits, signed. This length can, however, vary as a consquence of the architecture of the computer on which RPL/2 is running. Nevertheless, there is a way of dealing with integer overflows thrown as a result of the machine's architecture. In effect, if a calculation cannot be completed in integers as a due to the size of the integer, it is automatically converted to a real.

2.1.3 Reals

Reals, by default, use floats of 64 bits length. The actual size depends greatly on the host system architecture. Three symbolic constants – with which is is possible to perform arithmetical operations – can appear:

- `nan` or `not a number` this is the result of a calculation whose result is indeterminate, such as $0/0$.
- `-inf` or `+inf` represent the two infinities.

A real is shown in the form ±(MAN)E±(EXP), where the mantissa is real, the exponent an integer, the signs are optional and the case is unimportant. The decimal point changes its behaviour with the mode in which it is running, and will be either the point or the comma.

When an operation does not produce a result which lies within the set of real numbers (reals), but can return a single complex result, it will return a complex number result. Thus, « -1 SQRT » will give a complex result, but « -1 3 INV ^ » throws an error as the result of the operation is not unique.

2.1.4 Complex numbers

A complex number is, to the compiler, a group of two reals representing the real part and the imaginary part of the number. It is shown as (Re,Im), if the decimal separator is the point, and (Re.Im) if it is the comma.

2.2 Vectors

A vector is a table of scalars of a single dimension. According to how the scalars are composed, a vector can be declared as an integer, a real or a complex. It should be noted that all of the scalars making up a vector must be of the same type. Is a vector appears in a matrix expression, it will generally correspond to a matrix column. The delimiter used for a vector is the square bracket ([]). Thus, a vector would be written as [s_1 s_2 ... s_n].

2.3 Matrices

A matrix is a two-dimensional table of scalars, even if the matrix comprises only one line. Like vectors, they can be integer, real or complex. As it is a group of vectors, is is shown as [[s_{11} s_{12} ... s_{1n}]...[s_{m1} s_{m2} ... s_{mn}]].

It is impossible to create tables with more than two dimensions in the form of matrices. It is possible, however, to use tables of any dimension by means of lists.

2.4 Lists

Lists are made up of collections of objects, whether heterogenous or not. They can be included inside each other without limit, up to the maximium memory available. The delimiter is the brace ({}), which means a list is shown as { $objet_1$ $objet_2$ ... $objet_n$ }.

2.5 Expressions

Any set of instructions is considered to be an expresson in the larger sense. Thus, a program in RPN is an expression, generally comprised of a number of smaller expressions which can each be in reverse polish or algebraic notation.

2.6 Algebraic expressions

Algebraic notation corresponds more or less to the way you would normally write equations, however the implicit multiplication of two adjacent terms is not allowed. The delimiter used for this type of expression is the apostrophe. It thus follows that the equation

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

should be written `'SIN(PI*x)/(PI*x)'`. These expressions are costly in terms of calculation as they must first be transformed into RPN by the compiler. It is better to use these expressions sparingly, or to place them in variables if they are used in loops; in this way, the translation into RPN is only carried out once at the time of storage.

2.6.1 RPN expressions

Reverse Polish Notation expressions are directly understandable by the compiler. They can be included inside each other, and are delimited by french quotation marks (« and »). The previous algebraic expression could be written in RPN as « `PI x * dup sin swap /` ». Note that this form is not unique. It is also possible, for example, to write this expression as « `PI x * sin PI x * /` », but this second form is slower than the previous one as it contains more arithmetical operations.

2.7 Names

Names are particular objects which symbolically represent a variable, which need not exist yet. They are delimited by apostrophes. Variable x would be shown as `'x'`.

2.8 Character strings

A character string is an object made up of a suceession of characters delimited by double (or english) quotation marks, for example `"this string"`. The number of characters in the string can vary, since the object is associated with a string descriptor.

2.9 Binaries

Binaries are unsigned 64 bit integers, whatever the the host system architecture. They are manipulated by both logical and arithmetical opeartions and are written in the form `# (INTEGER)b` where `b` represents the base binary.

- `b` for binary;
- `o` for octal;
- `d` for decimal;
- `h` for hexadecimal;

IMPORTANT NOTE: there is no overflow control for calculations cantaining binary integers.

Chapter 3

Variables

If the built in variable types are not appropriate for your particular case, then variable are the solution. In RPL/2, the concept of a variable is somewhat peculiar; a variable is not defined at program launch-time, and is typed automatically according to the data type that is stored in it.

Additionally, several variables with the same name can exist without affecting the functioning of the system or loss of information. In effect, each variable is associated with a functional level by the compiler an is granted a certain visibility:

- Level 0 is composed of the definitions introduced in to programs. A variable at level 0 is not directly accesible by the user, and only makes up definitions;
- Level 1 indicates a global variable, accessible throughout the entire program;
- Levels higher than 1 denote a local variable subject to cetrain rules.

It is possible to give a variable or a definition any name you want, but it should be noted that the compiler gives priority to the intrinsic keywords rather than to the variable during program execution. Thus, it is possible to create a definition called `SIN`, but it will be overridden by the intrinsic definition carrying the same name.

3.1 Definitions

A definition is an RPN expression taking its arguments off the stack and putting back its results, and is associated with an identifying name. This name must be unique.

```
This_is_my_first_definition
<<
"Here is Edward Bear, coming downstairs" disp
"now, bump, bump, bump, on the back of his" disp
"head, behind Christopher Robin." disp
"" disp "                    A. A. Milne" disp
>>
```

The idea of a definitions is wider than that of routines or functions in FORTRAN or C, as a definition can take a variable number of objects as arguments and replace a variable number of results back on the operational stack.

In addition, a definition behaves in the same way as an intrinsic definition from the RPL/2 language and can be called by name from any other defintion including itself, which permits the construction of recursive functions such as the following example of factorial calculation.

```
FACTORIAL_CALCULATION
<<
    "Calculation of n!" disp "n = " prompt str->
    if
        dup dup ip same
    then
        FACTORIAL disp
    else
        drop
        "ERROR: argument n must be an integer!" disp
    end
>>

/*
================================================================
  Recursive factorial calculation
================================================================
*/

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

There are three types of definition, distinguishable by their place in the language.

3.1.1 Intrinsic definitions

These correspond to the language processing instructions. They are directly executable.

3.1.2 External definitions

External definitions are external to RPL/2 in the form of dynamically compiled libraries. These are too specific to justify integrating into the set of intrinsic defintions.

3.1.3 User definitions

User definitions are the definitions placed in the source code of the program. The code can contain any number of functions, the sole constraint being that the program execution always begins at the first defintion in the code.

3.2 Global variables

A global variable is, by definition, a variable visible at all points of a program. There are a number of global variables used by the compiler such as `EQ` and `SIGMA`, but they are not reserved. A global variable can, perhaps, be created by a small number of intrinsic instructions, of which the fastest is `STO`.

3.3 Local variables

Unlike global variables, a local variable is only visible in a section of a program, generally an expression, which can be in either algebraic or reverse polish notation.

Thus, the following small program

```
Small_program
<<
     1
     -> X
     <<
         X disp
     >>

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

produces an output of

```
1
2
'X'
```

In effect, the variable `X` is a local variable in the central expression and is destroyed on leaving the central exporession, only to be recreated when the following algebraic expression is handled and then destroyed again. Note that this mechanism can be reproduced within the same definition. This is shown in the following small program

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

        1 3 for X
```

```
            X disp
        next

        X disp
    >>

    X disp
>>
```

will produce

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

showing that the local variables don't erase each other. It should be noted, however, that the local variables are not lost when local variables with the same name are created at a higher level; it's just no possible to access them. The sole exception to this rule is the specific functioning of global variables which are always accessible.

3.4 Inheritance

The local variable system is substantially different than that which is found in other languages because of the definition of inheritance – simply put, objects to be sent to definitions **must** go via the stack. The system of masking the previously named variable makes it possible to use the same names for variables in the child definitions, without affecting the parent variables. Given that an example is better than many paragraphs of discussion, here is a small program to demonstrate this inheritance mechanism.

```
Main_progran
<<
    1
    -> X
    <<
        X disp
        Definition_1
        X disp
    >>
>>

Definition_1
<<
    X disp X 1 +
    -> X
    <<
        X disp
```

```
    >>
>>
```

The output of this program is:

```
1
1
2
1
```

demonstrating that the local variable `X` in level 2. One should, nevertheless, watch for boundary conditions; if the `sto` instruction permits the creation of a global variable, it is still possible to modify the contents of a pre-existing local variable of the same name

Instructions

false, 19

true, 19

Index

Binary, *see* Variable types
Boolean, *see* variable types

Character strings, *see* Variable types
Complexes, *see* Variable types

Definitions, 23
External definitions, 24
Intrinsic definitions, 24
principal, 25
User, 25

Algebraic
    expression, *see* variable types
RPN
    expression, *see* variable types

Indeterminates, *see* Indeterminate results
Inheritance, *see* Variables
Integer, *see* Variable types

List, *see* variable types

Matrices, *see* Variable types

Name, *see* Variable types
NaN, 19
Algebraic
    notation, 17
Notation
    internal, 18
    Reverse Polish (RPN), 17

Overflows, 19

Real, *see* Variable types
Indeterminate
    results, 19
RPN, *see* notation

Variable types, 19
    boolean, 19
    complexes, 20
    integers, 19
    matrix, 20
    real, 19
    scalar, 19
    vector, 20
Variable types
    algebraic expressions, 21
    binary, 21
    character strings, 21
    list, 20
    name, 21
    RPN expressions, 21
Global
    variables, 25
Inheritance, 26
Local
    variables, 25
Variables, 23
Vector, *see* Variable types