RLisp: User Manual

1 Introduction

RLisp is a Java package to create Java objects and execute its methods in runtime. It is, then, a Java interpreter. But, because this interpreter implements a whole Lisp machine, we can also say that RLisp is a Lisp interpreter running on the Java semantics.

2 License

3 The Files

RLisp is distributed in four (or three) files:

- RLisp.jar is the program, and the only one required.
- RLispManE.pdf is this document that you are reading.
  - RLispManS.pdf is the Spanish version of this document.
- RLispCode.pdf is a document where you can find the whole source code in a human readable form, and it is, therefore, RLisp conclusive specification.

If you want to access the other three documents from the RLisp program in runtime, then the four files should be located in the same directory. If you want to hack RLisp, you will find the source code files, files with java extension, inside the RLisp.jar file.

4 The Machine

To run RLisp you will need a computer with a Java virtual machine (JVM) installed.

RLisp works with the JVM that is part of Sun's Java Runtime Environment (JRE) version 1.4.2. You can download the JRE from its official home in Internet. It will surelly work with newer versions, and possibly work with older versions, but I am not sure.

5 Starting

To start RLisp you will have to call the Java virtual machine and instruct it to run the code in file RLisp.jar. That's all.

In Windows, the command is:

```
java.exe -jar RLisp.jar
```

6 The Main Window (The Yellow Window)

If everything goes right, then the RLisp main window, also known as the yellow window, will be open. There are three parts in the main window that are, from top down: a toolbar, that we will explain in the next sections; a big yellow text area where every action and reaction will be noted, that we will explain in this section; and a status line, where RLisp will show some additional information.

In the big yellow text area, every command to and every result from RLisp will be written, one in each line. Command lines will begin with the pair of characters "<<", and lines showing the results will begin with ">>". For example:

```
<< (new RLisp.RPair (string (1 2 3)))
>> (1 2 3)
```

The first line in the example shows that we have asked RLisp to build a new object of class RLisp.RPair using the constructor RPair(String) and, as argument, the String (1 2 3), including parentheses. The second line is the answer from RLisp showing the new object. You can give the new object a name to add it to the object dictionary so it will be available later on. We will see that there are several ways to do this, but in any case, if we choose list123 as its name, this action will be set down in the main window as shown:

```
<< (def list123 @)
>> list123
```

If we now want, for example, to know the class of the named object, then the note would be:

```
<< (method list123 'getClass)
>> class RLisp.RPair
```

7 The Toolbar

The toolbar has four sections:

- `File` to open data sources and sinks, as the log files, the tree, and the keyboard.
- `Edit` to cut, copy, and paste.
- `Action` to execute Java access operations (path), or operations to read and execute Lisp code files (Lisp code), or operations related with the object dictionary (name, unname, list).
- `Help` to access the documents.

The toolbar whole structure is as follows:

- Tool bar
  - File
    - Load
    - Load from
    - Save
    - Save to
    - —
    - Tree
    - Keyboard
    - Session
    - —
    - Close
  - Edit
    - Cut
    - Copy
    - Paste
  - Action
    - Path
    - Lisp code
    - —
    - Name
    - Unname
    - List
  - Help
    - `Manual` (User Manual)
    - `Spanish Manual` (Spanish Version)
    - `Code` (`RLisp` Source Code)
    - `About RLisp`

8 The Log Files

The log is a book in which the record of a ship's speed, its progress, and any shipboard events of navigation is kept. To record the events happening when using `RLisp` we employ the log files. For convenience, log files should have a log extension, though the code does not requires it nor verifies it, but only makes using it easy.

`RLisp` logs are text files. This means that they can be created with any text editor, such as Unix' `emacs` or Windows' `notepad`. If you use instead a word processor, for example MS Word, then you will have to save it as text, that is, without any control code. But, the easiest way to create a log file, is to let `RLisp` to do it.

If you stroke `File-Save to`, that is, if you choose the option `Save to` after having selected `File` in the toolbar, then `RLisp` starts an explorer to select a file name and a location in your computer to be used as a log file. From this point in time on, every piece of information shown in the main window will be also written to the selected log file.

The default log file is file `RLisp.log` in the current directory. This means that if you execute `File-Save`, then `RLisp` assumes that the log file is `RLisp.log` in the current directory, and it does not requires any other data.

The information saved to a log file can also be read. If you execute `File-Load from`, then `RLisp` starts an explorer to choose a file. By default, only directories and files with the log extension are shown, but you can instruct `RLisp` to show all files. If you want to read the file `RLisp.log` in the current directory, then you should execute `File-Load`. In any case, once the file is selected, `RLisp` starts processing the log file.

Log file processing is easy. If the three first characters of a line are not "`<< `" (the third one is a space), then `RLisp` just writes the line to the main window. If, contrarywise, the first three characters of a line are in fact "`<< `", then, `RLisp` writes the whole line in the main window and, in addition, it executes the rest of the line and writes the result in the main window. The result is written in a single line in which the three first characters are "`>< `", and then goes the result. The information written in the main window while processing a log file is not written to any log file, and this is the only exception to the rule.

When you call `RLisp`, you can append the name of a log file to be processed at start. By default none is processed.

```
java.exe -jar RLisp.jar filename.log
```

9 The Tree (The White Window)

If you execute `File-Tree`, then `RLisp` starts an explorer to choose a directory or a jar (Java archive) file. When you select one, `RLisp` opens a window with the tree of all classes (files with the class extension) that are accesible from it.

Please be aware that you should follow the Java access rules, so if a class is defined in a `package`, then the class is not accessible from the directory in where the class is located, but from one which is up in the hierarchy. For example, if class `MyClass` is defined in

```
package MyApplication;
```

then the file `MyClass.class` should be in a subdirectory named `MyApplication` of a certain directory, and it is from that certain directory from which the class `MyClass` is accessible.

For each class there are up to four branches:

- `Array` to create a unidimensional array of objects of this class. This branch is always present.
- `Fields` to access, and modify if it is not `final`, a field of the class.
- `Constructors` to create an object of the class.
- `Methods` to execute a method of the class.

Each of the four main branches, except `Array`, has a number of subbranches. In branch `Fields` there is a subbranch for each and every accesible field of the class, be it `static` or not, be it declared in it or inherited from a superclass. In branch `Constructors` there is a subbranch for each and every constructor in the class. And in branch `Methods` there is one subbranch for each and every method of the class.

Once a subbranch is chosen, you should press the `OK` button to execute the action associated to the subbranch. Depending on the subbranch selected, `RLisp` could ask you for additional data. For example, if you have chosen a non-`static` method subbranch, then `RLisp` will ask you to select an object between the named objects of the chosen class, and a named object of the corresponding class for each of the arguments of the chosen method. In each case, instead of a named object, you can enter a Lisp expression that `RLisp` will evaluate.

The executed command will be shown in the yellow main window *à la* Lisp. The command in one line that starts with "`<<`", and the result with the textual representation (`toString()`) of the resulting Java object, in the next line stating with "`>>`". The resulting object can be named pressing the `Name` button in the tree window, or executing `Action-Name` from the toolbar, or, as we will explain in the next section, writting the command `(def name @)` in the keyboard, where `@` is the `RLisp` way to refer to the last resulting object, and `name` is the name we want give it.

10 The Keyboard (The Green Window)

When you execute `File-Keyboard`, `RLisp` opens a green window to enter text. This way you can ask `RLisp` to execute any command from the keyboard. These commands use the lispian syntax. In the section titled "Lisp", we will explain in detail what commands do obey `RLisp`.

The green window keeps the record of the written parentheses, so when all of them are closed (i.e., paired) and you type the end of line (return key), then it sends the whole expression to the main console where it is interpreted.

You can also ask the execution of an expression from the keyboard using its toolbar buttons. They refer to the cursor current location. For example, when pressing the `Maximum` button, the longest expression surrounding the cursor location is executed. If you press the `Minimum` button, then it takes the shortest expression to execute. The `Word` button selects only the word in which the cursor currently is, so, in general, `RLisp` only task will be to look up the word definition in the dictionary. The `Previous` and `Next` buttons refer to the previous and to the following expression. The only button not asking to execute an expression is `Nesting`. This button is used to ask `RLisp` to calculate the cursor position nesting level, which is shown in the keyboard status bar located in the lower part of the green window.

11 The Session

If you execute the command `File-Session`, then `RLisp` will only accept commands written in the system console. To return to the normal mode of operation, you will have to write `quit` in the system console.

12 To Close

Executing `File-Close` in the toolbar, you close any open file managed by `RLisp`, and finish `RLisp`. Another way to quit `RLisp` is closing the main window. And yet another one is to execute from the keyboard the command:

```
quit
```

Lastly, if processing a log file the following line is found:

```
<< quit
```

then `RLisp` is also closed.

13 To Edit

In the `Edit` section of the toolbar you can find the usual commands to `Cut`, `Copy`, and `Paste`. They are used to transfer information between aplications.

14 The Path

The `Action-Path` command opens an explorer to select a directory. Once selected, the Java `ClassLoader` can access the classes that are accessible from the directory. All of the warnings written in the `Tree` section concerning class accesibility apply again to this case.

To execute this action from Lisp, where `URL` is the directory, type:
`(path URL)`

15 Lisp Code

When you execute `Action-Lisp code`, `RLisp` opens an explorer to select a file. By default this explorer only shows the files with extension `lisp`, though you can instruct it to show all of the files. Once a file is selected, `RLisp` executes each and every Lisp expression in the file.

You can execute this action in Lisp, where `URL` is the file, typing:
`(load URL)`

16 To Name

If you want to give the current object a name, you can use the `Action-Name` command. The current object is the object shown in the last line of the main window. After naming an object, we can refer to it by its name, being this way accessible.

The Lisp equivalent to this action, where `@` is the way used in `RLisp` to refer to the last result and `name` is the name we want to give it, is:
`(def name @)`

17 To Unname

If you want that a named and accessible object stops being so, then you should use the `Action-Unname` command. You will be then presented a list with all the named objects to choose one of them.

The same action can be performed in Lisp, where `name` is the unnamed name, doing:
`(set! name)`

18 To List

The `Action-List` command shows in the main window each and every named object, with its Java class and its textual (`toString()`) representation.

To get the class of an individual `object`, you can use the Lisp command:
`(method object 'getClass)`.

19 User Manual

To know the name of this document, you can execute the `Help-Manual` command. And to know the name of the Spanish version of this document, you can execute the `Help-Spanish Manual` command. If `RLisp` is run in a Windows environment and both files, `RLispManE.pdf` (or `RLispManS.pdf`) and `RLisp.jar`, are in the same directory, then the predetermined action is executed on the pdf file. The usual predetermined action on a pdf file is to open it with the Adobe's Acrobat Reader.

20 The Source Code

To know the name of a document where you can read the source code of `RLisp`, you can execute the `Help-Code` command. If `RLisp` is run in a Windows environment and both files, `RLispCode.pdf` and `RLisp.jar`, are in the same directory, then the predetermined action is executed on the pdf file. The usual predetermined action on a pdf file is to open it with the Adobe's Acrobat Reader.

21 About RLisp

The command `Help-About RLisp` shows you the version of `RLisp` that you are running, the copyright, and the name of its author, that is, my name. If you don't like `RLisp`, you can write me; if you don't dislike it, you can also write me!

22 Lisp

In this section we will explain the main characteristics of our Lisp. Note that you can use `RLisp` without writting a single line in Lisp; just inside the tree window you can create Java objects, you can give the crated objects names, you can execute the created objects methods, and you can see the results. So, in this working mode, Lisp is just a syntax to note the session events. But using Lisp the possibilities of `RLisp` are enhanced enormously.

22.1 Scheme

Scheme is the Lisp variant we have chosen to develope the `RLisp` Lisp. So we will explain the differences between our Lisp and Scheme. Scheme, being a well known Lisp dialect, will not be explained here.

22.2 Tail recursivity

The main conceptual difference is that, while Scheme is tail recursive, our Lisp is not. And this is so because Java (or more precisely the Java compiler) is not tail recursive. If some tail recursive Java compiler were developed, then we should revise the `RLisp` code in order to conform it to the compiler requirements.

22.3 Improper Lists

Improper lists, or pairs, that are noted with a dot ("`.`") in Scheme, are noted with a comma ("`,`") in our Lisp. The reason behind this difference is that in Java we use the

dot to refer to objects, and this is a very frequent operation in RLisp. For example, to refer to a class method we have to write the class name, a dot, and the method name. In addition, the comma is used in Mathematics to express pairs, (car,cdr), so it is the most natural replacement for the dot.

22.4 Pair Assignment

Our Lisp assigner is very powerful. The primitive operation to give an object a name in our Lisp is def:

```
(def name object)
```

where name is not evaluated, object is evaluated, and then:

- If name is not a pair, that is, if (atom? name) is t (for true), then the name name is added to the current dictionary with object (evaluated, as we have said) as its meaning.
- If name is a pair and object (evaluated) is also a pair, then def is transformed in two defs, and so on recursively:
  ```
  (def (car name) (car object))
  (def (cdr name) (cdr object))
  ```
- Lastly, if name is a pair, but object (evaluated) is not a pair, then no assignment takes place.

The primitive to change a name meaning is set!, and works with respect to pairs the same way as def works. The name should be already defined, otherwise set! do nothing.

```
(set! name object)
```

If you omit object, (set! name), then name gets undefined.

22.5 Atom Evaluation

If a name was already defined, and it was not undefined, then it evaluates to its definition. If a name is a primitive name, then it evaluates conforming to the primitive meaning. Any other name is the name of itself, that is, it evaluates to the String literal that is itself (three evaluates to the five characters String "three", and 12 to the String "12"), so in our Lisp all atoms are evaluable.

22.6 The Syntactic Primitives: Lisp

RLisp is minimalist, and it only employs the following syntactic primitives:

- quote is as in Scheme (and the single quote "'" is also its abbreviation).
- eval is as in Scheme.
- def and set! are as explained above.
- cond and eq?, where cond is as in Scheme, except that there is not else (but you can use t instead), and where eq? is as equal? in Scheme.
- cons, car, and cdr are as in Scheme, but (cons) evaluates to nil and (cons 1) to (1).
- atom? is as in Scheme.
- lambda is as in Scheme, but it employs, when asigning formals to actuals, all of the assigner power.

- `rho` is the way to create new special forms to extend the syntax. We will explain it in the next section.

22.7 The extensions

To create new special forms and syntaxes in our Lisp, we will employ `rho`. To evaluate the `rho`-expression:

```
((rho name expander) expression)
```

`RLisp` evaluates first:

```
(expander '(name expression))
```

And then it evaluates the result.

22.8 Initial Definitions

When `RLisp` starts, it loads the file `RLisp.lisp`, which is located inside the file `RLisp.jar`. In this file other known Lisp names are defined: `nil`, `t`, `null?`, `not`, `list`, `cadr`, `macro`, `syntax`, `define`, `if`, `sequence`, `or`, `and`, `mapcar`, `let`, `GENV`, `Gdefine`. You can browse this file to see some examples of `rho` usage, though hidden behind `syntax`. An easy way to extend `RLisp` is to add definitions to this file, or to the files called by this one, which are also inside `RLisp.jar`. We will see these files below.

22.9 The Semantic Primitives: Java

Our Lisp can access Java objects. In order to do this, it provides the following seven semantic primitives:

- `(string that is all)` evaluates to the Java `String` `"that is all"`.
- `(new cl arg0 arg1 ...)`, where `arg` is `ob` or `(cons 'Class ob)`, evaluates to a new Java object of class `cl`, using as arguments `arg0`, `arg1`, etc.; please see further details below.
- `(method [cl | ob] mt arg0 arg1 ...)`, where `arg` is `ob` or `(cons 'Class ob)`, evaluates to the result of executing the method `mt` of object `ob`, or of class `cl` (which then should be `static`), using as arguments `arg0`, `arg1`, etc.; please see further details below.
- `(field [cl | ob] f [val | ])` evaluates to the current value of field `f` of object `ob`, or of class `cl`. If the optional `val` is present, then the new value of the field `f` is `val`.
- `(array cl ob0 ob1 ...)` evaluates to a new array of class `cl` objects, initialized with objects `ob0`, `ob1`, etc.
- `(path URL)`, already seen.
- `(load URL)`, already seen.

In primitives that use a list of arguments, which are `new` and `method`, you will need to indicate the class of an argument, using variant `(cons 'Class ob)`, when you want to cast the object `ob` to a superclass of its proper class.

This happens, for example, when using a constructor or a method with an argument of class `java.lang.Object`, as `equals(java.lang.Object)`, because, in spite that every

Java object belongs to some subclass of that class, none of them belongs to it. So, assuming that both `object1` and `object2` belongs to a class that only defines `equals` that way, then to call the method we should write:

```
(method object1 'equals (cons 'java.lang.Object object2))
```

Something similar happens when we need to use the value `null`. While `null` can be the value of any class, it does not properly belong to any class, and therefore its class, `void`, does not appear in the signature of any constructor nor any method. For this reason we should cast the value of `null` using the `(cons 'Class (car nil))` variant. Note that Java `null` cannot be included in the dictionary, so you should use a Lisp expression, as `(car nil)`, to get the `null`.

22.10 Other Initial Definitions

In file `RLispJava.lisp`, which is called by `RLisp.lisp`, eight functions to get values of the eight basic types of Java are defined. You can use them like this: `(boolean false)`, `(char c)`, `(byte 5)`, `(short 233)` `(int -12)`, `(long 1234567)`, `(float 1.4)`, `(double 4.234567e4)`.

In file `RLispMaths.lisp`, also called by `RLisp.lisp`, the basic arithmetics functions are defined: addition (`+`), substraction (`-`), multiplication (`*`), quotient (`/`), and remainder (`%`); and two conditions: equal to (`=`), and greater than (`>`). It employs `java.math.BigInteger` numbers, and expression `(# 4)` evaluates to the `BigInteger` number `4`. If you need another type of arithmetics, you should redefine this file.

In file `RLispArray.lisp`, also called by `RLisp.lisp`, the following functions to work with Java arrays are defined: `(isArray? o)`, `(array-length a)`, `(array-get a i)`, `(array-set! a i v)`; where `o` is any `Object`, `a` an `Array`, `i` an `int` used as index, and `v` (for the new value) an `Object`. To create an array you can use the semantic primitives `(array cl ob0 ob1)`, which creates a unidimensional array of length two initialized, or `(new cl[] dim1 dim2)`, which creates a bidimensional array, sized `dim1` × `dim2`, not initialized (that is, initialized to `null` values).