

# Basic Overview Of Research Logo System

Pavel Boychev  
*pavel@elica.net*

## Abstract

Research Logo System (RLS) is programming environment based on the Logo programming language. It is developed to provide research environment suitable for pupils, students, teachers and scientists. RLS is a base upon which the user can construct different modelling environments and can investigate relationships between objects, thus simulating the world [CP].

RLS itself is only a kernel supporting RLS Logo programming language. It provides reduced set of reserved words. RLS works under Windows and has up-to-date design and performance. In the current material will be discussed the main language features.

## 1. RLS History

RLS has a long history. It starts with TopLogo++ - a logo-like programming language. It was alternative to the already existing Plane Geometry System (now called Geomland [ES]). Later was partly developed TopTeam Logo PGS System (TGS) that run under TopLogo++ environment. Then TGS upgraded to Logo Geometry System (LGS). LGS was a MS DOS compatible application. When Windows became wide spread LGS was rewritten in Turbo Pascal for Windows and Logo Geometry System for Windows (LGSW) appeared.

TopLogo++, PGS, TGS, LGS and LGSW were all kinds of Logo dialects. They did not appear quite different ideologically from other Logo dialects, developed by other vendors.

Because it was not necessary to produce more LOGO dialects that provided nearly the same functionality a new concept was designed. Its name was Relational Logo System and revealed theoretically all features of a system that can produce nearly any wide spread Logo environment.

When first attempts were made to turn the abstract Relational Logo System into a real program, several things changed because of some hardware and performance considerations. Any way the programmed version of Relational Logo System appeared and was called RLS (Research Logo System).

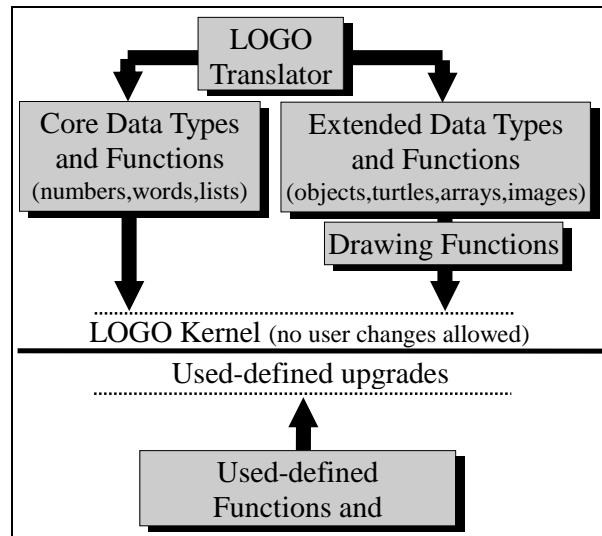


Fig. 1. User definable elements in conventional LOGOs.

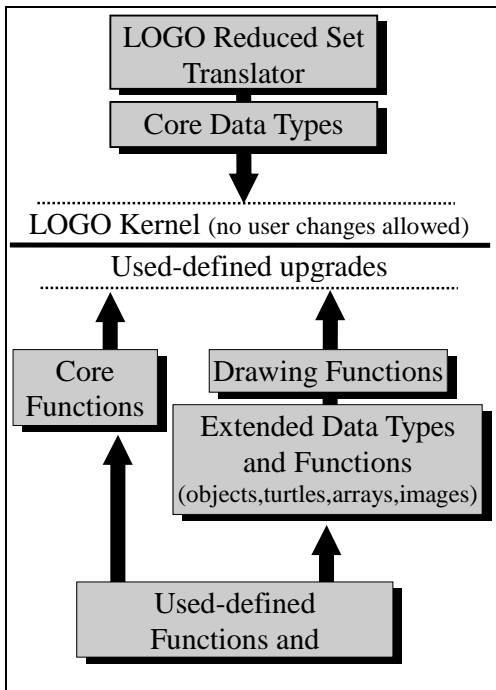


Fig. 2. User definable elements in RLS.

## 2. RLS is a kernel to many Logo implementations

In fact, RLS is only a Windows environment and kernel for interpreting RLS Logo environments. One of the main feature of RLS is that it provides a reduced set of several commands. Some of them as TO, END, IF, MAKE, OUTPUT, PRINT and RUN are command to all known Logo dialects [LW], others as COND, LOCAL, REPEAT, WHILE and OBJECT are used by advanced dialects [OL]. Of course there are several words that are unique to RLS kernel: EXTERNAL, ITERATION and RULE. So the reduced set of reserved words consists of only 15 reserved words that are proven to be enough to build different Logo applications.

There is a big difference between RLS environment and other LOGO environments. Usually LOGO environments include a LOGO translator - mainly interpreter, but in some cases

compiler. That translator is aware of all built-in data types - from the most common like numbers, words and list, to more complex ones - turtles, images, arrays, objects, etc. [L].

All core and extended functions that deal with all that data type should embed into the translator. This is the conventional schema - fig 1. Users can only type their own LOGO functions or commands. If they need to add extra data type users should either simulate it with LOGO instructions, or make the LOGO vendor to embed the new type into a new version of their product [IL], [OL].

What about RLS? Let's start from the most internal elements. There are only two of them that the user cannot modify. The first is the RLS LOGO reduced set translator, that deals with only 15 reserved words. The second is the definition of three data types: numbers, words and lists.

As seen from figure 2, core functions that process numbers, words and lists, are not a part of the kernel. They are supplied as external Dynamic Link Libraries (DLL), and the user is free to modify and accommodate the DLL according to his or her needs.

Core functions include math processing (ABS, SIN, etc.), word processing (WORD?, FIRST, etc.) and list processing (LIST, SE, etc.).

The user can also modify the operators +, -, \* and /. So if someone looks at the command MAKE "A :B+:C it may not be quite clear what exactly happens. The variables B and C may be numbers, or vectors, or even result recordsets of querying a DB (Data Base) table.

Other redefinable RLS elements are the extended data types. In RLS they are all represented by objects and memos. Objects are structured types with local fields and local methods. Memos are unstructured type and the user must supply a program code that interprets their structure.

If an object or a memo has a graphical representation on the screen the user may write a procedure that draws it.

### 3. Libraries

To achieve maximum functionality RLS can create and use libraries. Other non LOGO languages also use libraries. These libraries are files that hold a set of functions and declarations. That kind of libraries cannot be executed as separate programs. Examples of such libraries are DLLs, TPUs, TPLs, OBJs, etc.

Any program in RLS is a library with no need of additional source adapting. And any library is a standalone program. With the command RUN any RLS Logo file may be executed as if it is a library. When a file is run as a library all global variables, objects, procedures and functions are ported back to the calling program and become available in it. The commands that do not belong to any object, procedure or function definition are used to initialise the library.

It is not forbidden to load a library from within another library. There is no limit how deep libraries can be nested. Several libraries can form an iterational circular execution, each library calling the next, until a given condition satisfies.

RLS give the opportunity to call libraries in two ways. The first way of calling libraries is to call them as an external program. This ensures that if the library is called several times, the initialisation commands of the libraries are executed after each call. The second way of calling a library is as an ordinary library. In this mode if a library is called several times, only the first one launches the library and executes the initialisation. All subsequent calls are ignored.

Here is a simple example of a HELLO program, where LOGO is a library that defines all core functions:

```
RUN "LOGO
PRINT WORD "HELLO 5
```

### 4. Defining simple objects

To illustrate how the user can define simple objects let us think of the following task: *"Task 1: Define a library PGS (Plane Geometry System) that includes the objects POINT and VECTOR."*

To solve the problem we should specify the meaning of POINT and VECTOR. Let's at the beginning assume that a point is a structure composed by two numbers, called ABSC and ORD, that contains point's coordinates. Let's also assume that a vector is a structure composed by two points, called INITIAL and FINAL. Now it is easy to make the library. Its source will look something like the following lines:

```
TO OBJECT Point :Absc :Ord
END
TO OBJECT Vector :Initial :Final
END
```

It is simple enough. Let's now try to use these objects [OOP]. We first have to save the lines above as a text file, called PGS.LGO. A short example follows, showing how to use these simple objects:

```

RUN "PGS                                {-Define point and vector}
MAKE "A Point 0 0                       {-Make a point}
MAKE "V Vector :A Point 10 20           {-Make a vector}
PRINT :A.Absc                            {-Test the point...}
PRINT :V.Initial.Absc                   {...and the vector}

```

Fig. 3.

## 5. External functions

Because RLS is an interpreter in some cases it appears to work slower. For example it is not convenient and fast to do three dimensional computer graphics based only on standard logo. It is also hard to think of any qualitative animation, speech synthesises, image recognition, ray tracing and so on. When a portion of the program is time or memory critical, or it performs low-level hardware-specific operations the traditional Logo will fail to do this in time. Such program fragments may be written in another language (Borland Pascal, C++) and later compiled to dynamic link library (DLL).

RLS provides unified interface to DLLs. It is not influenced from the compiler that is used to produce the DLLs. The only compulsory thing is to use the RLS DLL Interface Convention. That interface allows any DLL function to access the variables and the procedures of the currently running RLS program.

A good example of using external functions is the library LOGO. The library defines all core functions, used in RLS Logo. Here is a short extract from the library. The three functions declare WORD? and BF.

```

TO Word? :X
  OUTPUT EXTERNAL DLLLogo WordP
END
TO BF :X [:Y]
  OUTPUT EXTERNAL DLLLogo BF
END

```

## 6. Extending objects: adding graphics

It is easy to define and use simple objects. We already showed how to create a point and a vector. A step further is the following task: *"Add a graphical representation of vectors."*

RLS can draw any user-defined object if it has a method called DrawObject. Let's think we have already written PGS.DLL that has functions to draw a vector. The only thing that remains is to change the definition of vector, so that RLS knows how to draw it.

```

TO OBJECT Vector :Initial :Final
  TO LOCAL DrawObject
    OUTPUT EXTERNAL PGS DrawVector
  END
END

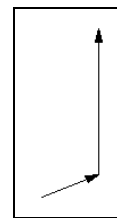
```

If we run the following tree commands, the screen will look like this:

```

MAKE "V Vector :A Point 50 20
MAKE "W Vector :V.Final Point 50 150

```



## 7. Polishing objects: adding operators

Concerning objects, RLS provides a unique feature that few other

programming languages do [PL]. The user is able to define the any object's behaviour in an expression. In other words, the mathematical operations +, -, \* and / may be redeclared to use them with objects.

The user has no full access to operators. For examples it is not possible to change operators' precedence.

To refine the vector's behaviour it is a good idea to solve the following task: *"Add to vectors two methods DX and DY that return the corresponding offset between the final and initial points. Also make possible to add and subtract vectors, and to multiply or divide them by a scalar factor."*

Although this task seems too difficult, it is quite easy to solve it. For example the DX method may look like this:

```
TO LOCAL DX
  OUTPUT :Final.Absc - :Initial.Absc
END
```

The other function DY is nearly the same - only Absc should be replaced by Ord.

To define the parameters, we should know how RLS resolves them. When the sum of two vectors is going to be calculated RLS calls the method + of the left vector and passes the second argument as a variable Arg. That means when a vector is multiplied by a scalar factor, the vector should be the left argument, and the scalar - the right one. That is overcome by defining an object Scalar. Here is the full description of the object vector:

```
TO OBJECT Vector :Initial :Final
  TO LOCAL DX
    OUTPUT :Final.Absc - :Initial.Absc
  END
  TO LOCAL DY
    OUTPUT :Final.Ord - :Initial.Ord
  END
  TO LOCAL +
    OUTPUT Vector :Initial Point :Final.Absc+Arg.DX :Final.Ord+Arg.DY
  END
  TO LOCAL -
    OUTPUT Vector :Initial Point :Final.Absc-Arg.DX :Final.Ord-Arg.DY
  END
  TO LOCAL *
    (OUTPUT Vector :Initial Point :Initial.Absc+DX*:Arg
      :Initial.Ord+ DY*:Arg)
  END
  TO LOCAL /
    (OUTPUT Vector :Initial Point :Initial.Absc+DX/:Arg
      :Initial.Ord+DY/:Arg)
  END
  TO LOCAL DrawObject
    OUTPUT EXTERNAL PGS DrawVector
  END
END {-End of vector definition}
```

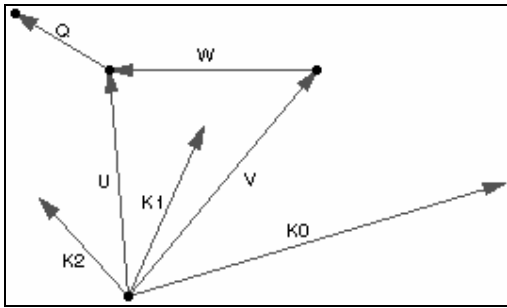


Fig. 4. Mathematical operators with vectors.

To demonstrate the usage of vector's operators on figure 4 is displayed the result of a small test program. The program uses vectors V, W and Q, and defines four other vectors. Here is an extract of the test source:

```
MAKE "U :V+:W
MAKE "K1 :V+:W-:Q
MAKE "K0 :V-:Q*2
MAKE "K2 (:V-:Q/2)/2+:W
```

## 8. Unstructured data type: a key to

### images, databases, etc.

Objects are structured data types. The user defines all parts of the object and RLS is aware of that. Memos, on the other hand, are unstructured data type. RLS allocates a buffer in the computer's virtual memory, and leaves the user-supplied functions to process and to interpret the data in the buffer.

Memos are not intended to be used by beginners, because they require more computer knowledge and some system programming skills.

To illustrate the usage of memos we will have a look at two examples. When we start RLS it displays an initial picture. That picture is a standard BMP file. Here is the new task: *"Define a way of displaying images."*

On figure 5 is displayed a snapshot taken from a computer screen. The lower half of the screen contains the source that defines and displays the image. The upper half of the screen is the image itself [EL].

In a little bit more complicated way memos are used to implementing multimedia support. Audio and video streams are stored into memory buffers and are ready to be played [OL].

The second example of using memos is something that very few specialists imagine a logo can do. Using ODBC (Open Database Connectivity) drivers that are supplied by many vendors make RLS able to connect to database servers. So a RLS Logo program may retrieve, store and query data from Oracle, Informix, Sybase and SQL Server databases. It is also possible to manipulate data from PC-sized databases like Access, FoxPro, dBase and MS DOS flat files. The symbiosis between Logo and RDBMSs (Relational Database Management Systems) allows RLS Logo programs to process huge amounts of data. Let's just think of a 3D object composed by millions of planes, or a long list of knowledge based rules, used by ES (Expert Systems) or MIS (Manager Information System).

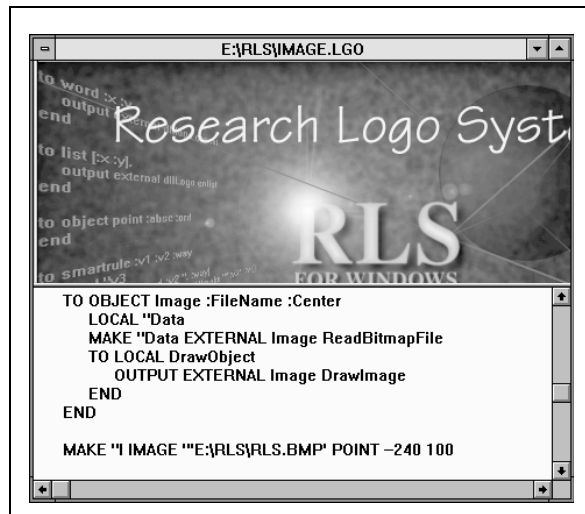


Fig. 5. Displaying images.

## 9. Make objects alive

To make the user-defined objects alive it is not enough to supply them with methods, as it is done in conventional OOP (Object Oriented Programming). RLS gives the opportunity to define links between objects, so when one of them changes, another may react by executing a piece of code. The definition of object's behaviour is done by *rules*.

Rules in RLS form a special category, as variables and programs do. Rules are the only way to build objects that form one construction. For example it is possible to define acute triangle what will make its best to remain acute. If someone tries to change one of its angles to 90 degrees or over, then the triangle will respond by changing other elements so keeping itself an acute triangle.

Of course rules are defined by the user. The user is responsible to associate necessary rules to each variable. Once rules are created, they remain in effect until the variable is destroyed. One variable may contain as many rules as it is necessary. Variables with no rules are treated as standard variables.

Formally, the rules are list of commands assigned to a variable. That list is executed each time the variable changes its value. A variable is called to be *active* if at the current moment RLS interprets one of its rules. While a variable is active the change of value will not reexecute its rules, because they are already in execution state. This ensures that any combination of variables and rules between them is *stable*. A stable combination is any combination that needs a limited number of rule activations (in respond to an external interaction) in order to satisfy all rules.

Construction with many variables and many rules between them seem to work as a neuro net. When a part of the construction is changed the activation of the rules is passed from one variable to another spreading partly or fully over all variables. This process will disappear gradually as the construction becomes more stable.

Let's take as an example the triangle. It is composed by three points and three segments. If any of the points is changed, then some or all the segments will be affected. On the other hand if a segment changes, the points will change too.

RLS deals with two types of rules - internal (also called local) and external (called global). Sometimes it is impossible to detect whether a given rule is internal or external. The distinction is formal.

An external is any rule that is assigned to a variable from outside the variable itself. Internal rule is just the opposite. Internal rule is a rule defined from inside a variable. Rules between fields in one object are internal rules. Rules between different objects or between a field in an object and another variable, external to that object, are external.

Here is a very simple example of two variables X and Y that automatically keep the difference between them equal to 1.

```
MAKE "X 1, MAKE "Y 0
RULE "X [MAKE "Y :X-1]
RULE "Y [MAKE "X :Y+1]
MAKE "X 5           {-Makes X=5 and Y=4}
MAKE "Y 5           {-Makes X=6 and Y=5}
```

This example shows some Prolog-like features of RLS [CO]. Imagine  $Y=F(X)$  and we are searching Y's roots. We can implement Newton's algorithm for finding roots and assign a value 0 to Y - rules will automatically do all the necessary actions to find the roots.

Let's now try to use rules and define a triangle [MIT]. Here is the task: "*Define the object triangle, constructed by three points, and including three segments.*" We assume a triangle is a construction of three points and three segments between them. When one of these elements changes (or when a part of one of them changes) some of the others elements should change too.

```
TO OBJECT TRIANGLE :A :B :C
  LOCAL "SA "SB "SC
  MAKE "SA SEGMENT :B :C
  MAKE "SB SEGMENT :C :A
  MAKE "SC SEGMENT :A :B
  RULE "A [MAKE "SC.INITIAL :A MAKE "SB.FINAL :A]
  RULE "B [MAKE "SA.INITIAL :B MAKE "SC.FINAL :B]
  RULE "C [MAKE "SB.INITIAL :C MAKE "SA.FINAL :C]
  RULE "SA [MAKE "B :SA.INITIAL MAKE "C :SA.FINAL]
  RULE "SB [MAKE "C :SB.INITIAL MAKE "A :SB.FINAL]
  RULE "SC [MAKE "A :SC.INITIAL MAKE "B :SC.FINAL]
END
```

If you have a look at the rules in TRIANGLE, you will notice that all points and segments have rules. This ensures that if any of the elements changes, the construction will remain triangle. Let's try and move point C. What will happen? The first result is that C is torn out from the triangle - it is not the connecting point between SA and SB. But during the execution of the C's rules SB moves its initial point to match C. Later SA moves its final point again to match C.

## 10. Summary

RLS is a fully integrated logo based environment that helps the user to construct different models of real-world phenomena. The RLS Logo language is as simple as to be easily understood by children. At the same time it allows the user to build complex programs and data structures that are oriented towards software specialists. The language is simple but provides variety of user-defined upgrades and in the same time remains a logo language. The RLS environment is based on Windows 3.1 and Windows 95 and follows the Windows interface conventions.

The current RLS (at the time of preparing this material) is not a commercial-ready version. It is still under development. A beta version is available and any comments, offers, proposals are welcome.

## References

- [CO] Coelho, H. *Prolog by Example*, Springer-Verlag Berlin Heidelberg, 1988
- [OOP] Wal, R. V. D., *Trees and Objects*, Eindhoven, 1995, 18-39
- [CP] Weizenbaum, J., *Computer Power and Human Reason*, Penguin Books, London, England, 1993, 132-153
- [ES] Sendova, E., Azalov, P., Muirhead, J., *Informatics in the Secondary School - Today and Tomorrow.*, UNESCO International Workshop, Sofia, Bulgaria, 1995, 99-109
- [LW] *LogoWriter Reference Guide*, Logo Computer Systems Inc., 1986
- [PL] Pratt, T. W., *Programming Languages - Design and Implementation*, Department of CS, University of Texas at Austin, 1975, 342-553
- [IL] Nikolova, I., Georgiev, I., *Programming with Logo*, Publishing House "Technica", Sofia, 1992

- [MIT] Billstein, R., Libeskind, Sh., Lott, J. W., *Logo - MIT Logo for the Apple*, University of Montana, Missoula, Montana, 1985, 196-208
- [L] Burke, M. E., *Logo and Models of Computations*, San Jose state University, 1987
- [EL] Blaho, A., Kalas, I., Playing, Developing and Computing With Images in Comenius Logo for Windows, *EuroLogo Proceedings 95*, 1995, 15-19
- [OL] Hain, St., *ObjectLogo for the Apple Macintosh*, Paradigm Software, Cambridge, Massachusetts, 1990