

Elica Logo and Objects

Pavel Boychev
pavel@elica.net

Abstract

The Educational Logo Interface for Creative Activities (Elica) is one of the newly developed programming environments that enables the interactive visual programming with the well-established Logo language. It is based on a limited set of Logo commands that provide a complete and rapid take off towards very complex programs. The concepts of Elica make learning objects easy and intuitive task. They provide a perfect balance between simple syntax, clear semantics, acceptable speed and amazing results.

Keywords: Elica, Logo, objects, EOD, Easy Object Declaration

Records, Procedures, Functions and Objects

The history of programming follows closely the history of many sciences. The programming as a process deals with a set of elements. The similarity between programming and other sciences comes from similarities in the evolution of these elements. At the beginning some basic elements are declared, later more are created. But from a given level, the efforts are not directed towards introducing new elements, but towards consolidating the existing ones.

The first manipulated elements in programming are *data* and *instructions*. If we follow strictly the concepts of J. von Neumann the instructions are just another interpretation of data, but because of their importance, they are separated as a different element. These two elements are the base of programming.

The next step in the evolution is to create more complex elements. In this way, the simple data gave birth to *records*, which are a kind of data containers. On the other hand, instructions led to *procedures* and *functions*. These elements are not fundamental, because for every program with records, procedures and functions it is possible to produce another functionally equivalent program based only on data and instructions elements.

Some years ago it was necessary to make the next step - consolidation. The new element, which is sometimes overused, is the *object*. Although it introduces some new concepts, in fact it is a container of data *and* instructions. Data in objects are referred as *fields*, the procedures and functions are referred as *methods*.

It is not accepted to treat objects only as containers, because they possess some features, which expand the set of programming techniques. Inheritance, polymorphism, abstract and virtual methods are just few to mention. The last element in the hierarchy is the *component*.

The evolution of concepts in Logo

Fortunately and unfortunately Logo stands at the border between imperative and functional programming. Fortunately, because Logo takes the advantages from both worlds (Burke, 1987). Unfortunately, because takes the disadvantages, too. If we have to make the grand total now, we will see that Logo almost rejected the concepts of objects and components (Billstein, Libeskind and Lott, 1987). There are some implementations, which give us the opportunity to work with objects (Hain, 1990) or just the feeling as if we work with objects.

This paper will not discuss other Logo implementations than Elica, nor it will compare Elica with them.

Looking more positively, in spite of the fact that objects are not traditionally implemented in Logo, this programming language has some features that make a part of the consolidation done (Nikolova and Georgiev, 1985). One of the features is that procedures and functions are lists. The other feature is that procedures and functions are one and the same element - they are defined the same way and whether we are working with a function or with a procedure becomes obvious just when we use them (LogoWriter Reference Guide, 1986). It is possible the same source to be used either as function or as procedure.

```
make "a point 10 20
print absc :a
absc "a 12
print absc :a
```

The example above shows how a routine is used as a function (to retrieve the value of point's abscissa) and as a procedure (to set it). The source in the example is taken from Elica, but it will work without any modifications in Geomland (Sendov and Boycheva, 1997).

Functional and procedural use of routines applies to all user-defined routines, although some have to be used only as procedures, others - only as functions.

Elica introduces a new level of consolidation

What we need to make a better consolidation is just a good way of defining objects. As Logo is close to both imperative and functional programming, it can easily take from them the way of defining objects. This step was made in the previous version of Elica (called Research Logo System) (Boychev, 1997).

To define an object of type *point* with three fields - *abscissa*, *ordinate* and *type*, we use a `to-end` definition, which is nearly, the same as the one used for routine definition. The only difference is that the reserved word `object` is inserted right after `to`. This is indicator for the interpreter, that the definition is neither a function, nor a procedure, but an object:

```
to object point :absc :ord
  local "type
  make "type "point
  to distance
    ...
  end
end
```

RLS was the first to introduce the *EOD (Easy Object Declaration)*. This term is used to denote simple or well-known syntax structures, used to define objects. In the example above the word `object` changes the meaning of all the elements of the declaration (without changing their visual appearance). If we skip `object` then it will become a definition of a procedure with two arguments `absc` and `ord`, a local variable `type` and a local routine `distance`. If we keep the word `object` it will become a definition of an object with two initializing fields `absc` and `ord`, another field `type` and a method `radius`.

To see the importance of EOD, you may have a look at how objects are declared in other languages: C++, Delphi, ObjectLogo (Hain, 1990). Definitely, you will see that the definition is supported by a number of new syntax structures, which may put a novice user in a puzzle.

As for Elica, it makes a step further to EOD. You might want to see how to declare an object in Elica in the next example. You may compare it with the previous definition. The only difference is that the reserved word `object` is missing. As it was said above, this word is used as the only identification that a definition is an object definition, rather than another one.

```
to point :absc :ord
  local "type
  make "type "point
  to distance
  ...
end
end
```

When we have that identification removed, is it possible to recognize the type of the definition? The first answer to come in mind is “No, it’s not possible!” Wrong answer. The root of object-oriented programming in Elica is based on the idea, that routines and objects are one and the same thing - a piece of code can be used as a function, as a procedure or as an object. And how it is used is not determined by its definition (it is the same, really the same) but by the way it is used.

To understand the Elica concept of objects we have to start from the very beginning.

What is a procedure? A piece of code that performs some actions, usually initialized by arguments, and eventually uses locally defined variables and routines. After a procedure is used, its instance, but not definition, is forgotten - all memory resources used for storing its attributes are released. What will happen if we do not release this memory, but try to keep the procedure ‘alive’? First of all, we will have access to all local variables defined during its execution. Also, we will be able to start any locally defined routine. If we run again the procedure a new memory will be allocated, but at the end it will not be released. So we will have two instances of this procedure - each of them unique, because could be produced with different set of initializing parameters. If we run the procedure a hundred times more, all we get is a hundred new ‘alive’ instances. In Elica an object is a procedure instance that is not destroyed, but left ‘alive’.

It is obvious, that every part of the procedure has its own meaning when the procedure is used as an object. The only part, not explained till now, is the body of the definition. As far

as it is executed only once, at the beginning, when the object is constructed, that part is de facto the object initialization.

Although, it is already clear that objects are active procedures, we have still not answered to the question how to distinguish functions and procedures from objects. The interpreter of Elica solves this task at the end of execution of the routine's code or object's initialization code.

First of all, there are two ways to execute a routine - functional and procedural. Within the functional way, the interpreter expects a result after the execution, while the procedural does not. Secondly there are two ways of exiting a piece of code - functional and procedural. The functional is when a piece of code is exited by the command `output` followed by expression. The procedural way is when there is no argument of `output` or the execution terminates at the final `end` command. The combinations between these two ways of calling and two ways of exiting produce four different behaviours.

```
to routine :x
  output :x
end
make "a routine 5
```

When the call and the exit are of the same type, then the result is clear. The functional call and exit tell that this is a function. In the example above `routine` is uses as a function.

```
to routine :x
  output
end
routine 5
```

The procedural call and exit tells the interpreter to treat the code as a procedure. In this example above `routine` is used as a procedure.

```
to routine :x
  output :x
end
routine 5
```

When the call is procedural, the interpreter does not expect any result, but if the exit is functional, as the example above, then the result is ignored. This is sometimes called procedural function - a function, which is used as a procedure and its result is not stored anywhere.

```
to routine :x
  output
end
make "a routine 5
```

The last combination is when the call is functional and the interpreter expects a result, but because the exit is procedural there is no result at all. In such case the procedure gives itself as a result. That's why the interpreter does not 'kill' the instance of the routine, but preserves it 'alive' as an instance of an object.

These four examples show how Elica interpreter determines what it deals with - a procedure, a function or an object. To make a step further follows an example how to use one and the same routine as these four different ways.

```

to anything :param
  if :param [output 1]
end

anything "false
anything "true
make "variable anything "true
make "object anything "false

```

There is a definition of a routine called `anything`. The parameter `param` is used to generate the different types of exits. When it is `"true`, then the exit from `anything` is functional, otherwise it is procedural. The four commands that follow the definition represent the four ways of using `anything`. The first two use procedural calls, the last two - functional.

As it is expected, the first use of `anything` is as if it is a procedure. In the second use it acts like a procedure again, but the returned value is ignored. The third use of `anything` is as a function - the interpreter expects a value and `anything` satisfies its expectations. The last use is as an object.

Logo and Objects

As it was noted earlier in this paper, Logo take advantages from both worlds – imperative and functional programming. Introducing objects to Logo environment does not only give us all the strength and usefulness of objects, but discovers new horizons. Many features, specific to Logo can be successfully applied to object-oriented Logo (Boychev, 1997; Hain, 1990; Sendov and Boycheva, 1997). This makes Logo objects much more useful and flexible.

Like procedures and functions, Elica Logo keeps the definition of an object as an ordinary list. This list can be retrieved, modified, stored or manipulated in any way. In the example below the output of the commands is formatted in italics.

```

to obj :a :b
  print "sum "is sum :a :b
end

make "a obj 5 7
SUM IS 12

print :obj
[ : A : B , PRINT " SUM " IS SUM : A : B , ]

make "obj se :obj [print "ok]
make "b obj 5 7
SUM IS 12
OK

print :obj
[ :A :B, PRINT "SUM "IS SUM :A :B, PRINT "OK ]

```

This example shows clearly how to retrieve the definition of a routine and how to modify it (we append `print "ok` command at the end).

Using definition as a list is not the basic advantage of Elica Logo. Because Logo is a language suitable more for interpretation rather than for compilation, run-time access to objects is empowered with the ability to change not only their definition (manipulating them as lists), but also to change individually each instance of the object.

Elica Logo allows (even after an object is created) to change any of its fields, any methods, or to add new fields and methods. A small part of this functionality is realized in other non-Logo programming languages. They introduce things like dynamic and static access to fields, virtual methods, inheritance and many others just to make objects more convenient and powerful. Fortunately, all these features are naturally available by Elica Logo's EOD and the user does not 'loose' attention on how to express one thing or another.

Just as an example to show how simply and painlessly Elica Logo performs object inheritance, we can define an object `fruit`, which has one field `name` containing the name of the fruit and a method `what` to print this name.

```
to fruit
  local "name
  make "name "fruit
  to what
    print :name
  end
end
```

Now, we want to define a new object `apple`, which inherits from `fruit` both the field and the method. There are two ways of doing this. One of them is to declare the object `apple` and from within its initialization code to run the initialization code of `fruit`, then to make the appropriate changes. The next example shows us some very important thing. First of all, to inherit an object we use ordinary commands - no new syntax, no new reserved words.

```
to apple
  run :fruit
  make "name "apple
end

make "a fruit
a.what
FRUIT

make "a apple
a.what
APPLE
```

Secondly, Elica Logo allows the user to construct 'immoral' objects. If we run many objects' initialization codes from within the initialization code of a single object, then this object will have many parents. Although this is not normal in most of the countries around the world, object-oriented programming classifies this feature as very important. Unfortunately, for many languages it is impossible to make an object inherit fields and methods from multiple parents. Fortunately, Elica Logo is one of the few to make it possible.

Of course, the method of inheriting an object, described above, will create a new definition of an object and will copy all fields and methods from the parent to the child. If we have a

simpler task, where the child differs from the parent in the values of few fields, and we do not want to make a new declaration, we can create an instance of the parent and modify it.

```
make "b fruit
make "b.name "apple
b.what
APPLE
```

In this case, we firstly create an instance of the parental object. Then we modify its field. The result, compared with the previous example, is the same – one and the same instance is created.

Of course, there are more ways to do the same task. For example, we may define one object `fruit`, which has one initializing argument. According to the value of this argument the initialization may have different execution paths. Is such a simple way Elica allows to incorporate two or more definitions into one routine.

Step by step, object by object

To show how a beginner can start using objects without knowing much about them, we will solve a simple task. To complete it we will have to construct three objects, each of them using the previous ones.

As for the task itself, it is really very simple – to define the object *point* initialized by two numbers (*abscissa* and *ordinate*), the object *segment* (initialized by two points – *initial* and *final*) and the object *triangle* (initialized by three points and containing three segments). The definitions of these objects will be as simple as it is *theoretically* possible.

Let's start with the definition of a point. What we have to do is to define a procedure (or something that looks like a procedure, but is used as object):

```
to point :absc :ord
end
```

The initialization parameters are defined as arguments to the procedure. The definition consists of 7 words. It is the shortest definition! Four of the items are used because of Logo syntax considerations (`to`, `end` and two`":`), one word contains the object's name (`point`) and the last two words – the names of the arguments.

The definition of segment is nearly the same:

```
to segment :initial :final
end
```

The segment consists of two points, named *initial* and *final*. Again, this definition is the shortest possible. It is hard to think of another language, or another Logo implementation, which will give a shorter definition.

Maybe there is no need to give more explanations about the definitions of point and segment. They are really very simple and should not cause any problems even to newcomers.

To complete the task we have to describe the triangle. As it might be expected, the definition will be a standard `to-end` Logo definition:

```

to triangle :a :b :c
  local "ab "bc "ca
  make "ab segment :a :b
  make "bc segment :b :c
  make "ca segment :c :a
end

```

As long as any triangle is defined by three point, there are three arguments, called a, b and c. Also, the triangle consist of the segments that connect the points in pairs. These segments are stored in the local fields of the triangle and are called ab, bc and ca. To construct the segments we use three make commands and ... that's all! Now we have a complete definition of a triangle.

Because arguments to Logo procedures are not type-determined, the current definition of a point, segment and triangle cover not only the geometrical idea of geometrical points, segments and triangles, but something more common and more abstract. If we work in a space where coordinates are not numbers, but let's say words or lists, then a point is just a pair of two coordinates, while a segment will be a pair of two pairs of coordinates. If we exclude these exotic domains, the definitions will work with standard numbers, with complex numbers, with homogeneous numbers and so on without any change in the source.

If it is compulsory to put some type checking, the user may do so with placing some if's in the initialization part of the definitions.

The definition of a triangle has one very disappointing feature. If we construct a triangle and move one of the initializing points, the segments will not be recalculated. For example, if we construct the triangle t based on the points p, q and r, then all these four different points should be coincident: p, t.a, t.ab.initial and t.ca.final. All the definitions above keep this rule only when objects are constructed, but if any part of them is changed, none of the others will be change. This is very crucial, because if we break apart the elements of the triangle, they will not try to glue back again.

To eliminate this situation, Elica introduces rules, which are de facto, user-defined events executed when a value is changed. Every variable that 'want' to cause some recalculation when its value is changed, has to have assigned a rule.

Rules were implemented in the previous version of Elica that was called RLS (Research Logo System). Those rules seemed to be a little bit difficult to manage. To show how they looked like in the predecessor of Elica, you may have a look at the next example that shows how to define a triangle with rules (Boychev , 1997).

```

to object triangle :a :b :c
  local "ab "bc "ca
  make "ab segment :a :b
  make "bc segment :b :c
  make "ca segment :c :a
  rule "a [make "ab.initial :a make "ca.final :a]
  rule "b [make "bc.initial :b make "ab.final :b]
  rule "c [make "ca.initial :c make "bc.final :c]
  rule "bc [make "b :bc.initial make "c :bc.final]
  rule "ca [make "c :ca.initial make "a :ca.final]
  rule "ab [make "a :ab.initial make "b :ab.final]
end

```

As you may notice, every rule is a set of commands applied to each field. In the case of a triangle there are six rules – three rules applied to points and three to segments.

The problem with RLS was that if we want to define an object with more fields, then we have to write many new rules. Another problem is that it is not quite intuitive to add new fields. If we want to transform the example to a definition of a square, it will not be enough just to do some make up.

The Easy Object Declaration technique empowers Elica to solve the same task without all this inconvenience. Elica supports automatic rule generation, which will work in most of the cases. In this particular case, to write a definition of a triangle with rules, there are only three pairs of parentheses to add:

```
to triangle :a :b :c
  local "ab "bc "ca
  (make "ab segment :a :b)
  (make "bc segment :b :c)
  (make "ca segment :c :a)
end
```

This is the Elica EOD definition of a triangle. All triangle instances built with it will be real triangles – if one moves one of the points, the segments will turn and extend or contract to reach that point. The same happens if one of the segments is relocated – the instance will remain a triangle.

Summary

Although Elica does not support components, it makes people accept Logo as a modern programming language. The flexible syntax structure of Logo makes it possible to implement new concepts as objects and object-oriented programming, without adding new commands, new reserved words, or new syntax enhancements. This is a feature that no other language and no other implementation of Logo is able to deliver to the user. The idea of declaring functions, procedures and objects in one and the same way will speed up the process of learning and using object-oriented programming.

References

- Billstein R, Libeskind Sh and Lott J W (1985) *Logo - MIT Logo for the Apple*, University of Montana, Missoula, Montana, 196-208
- Boychev P (1997) Overview of Research Logo System, *PEG97 Proceedings*, 30-37
- Burke M E (1987) *Logo and Models of Computations*, San Jose state University
- Hain St (1990) *ObjectLogo for the Apple Macintosh*, Paradigm Software, Cambridge, Massachusetts
- LogoWriter Reference Guide* (1986) Logo Computer Systems Inc.
- Nikolova I and Georgiev I (1986) *Programming with Logo*, Publishing House "Technica", Sofia, 1992
- Pratt T W (1975) *Programming Languages - Design and Implementation*, Department of CS, University of Texas at Austin, 342-553
- Sendov B and Boycheva S (1997) *Geomland*, University of Sofia