

Programming As Poetry (The Power of the Simplicity in Programming)

Pavel Boytchev
pavel@elica.net

Abstract

The paper will present the result of an attempt to design a programming language with a vary limited set of reserved words, but the ability to define functions, procedures, operators and objects. Although the language syntax rules are very simple, it is possible to construct very complex programs and command-data structures, which are not possible in other languages.

Introduction

It is known, that some of the greatest poets of the world can write superb poems using a few distinct words. The trick is how these words are combined and how the phrases are interpreted. Human language and the poetry as a way of expressing can be both very condensed and sparse in matters of information density. It can be at the same time well determined and quite ambiguous.

Following these thoughts, this paper will present the result of an attempt to design a programming language and its concepts which implement a poetry-like programming style - this is **how to make complex programs with a very limited set of initial reserved words**. Before going further into details, it will be good to say that this paper is neither about artificially generated poetry, nor about linguistic software.

This paper will be entirely about the Elica system which base language is a member of the Logo family. Traditionally, Logo language has no world- and time-wide specification, although some statements and constructions are the same for all Logo implementations. The Elica's Logo originates ideologically from Geomland - a Bulgarian software package for the teaching and learning of school Geometry through explorations. What differs Geomland's Logo from Elica's Logo is that the second has been improved in many directions, some of which will be explained later.

The following table will list some typical for Logo language way of writing Logo statements, which will be used in this paper:

Numbers:	45, 1, 98.5
Words:	"ABC, "Engine
Lists:	[X Y], [a [j 8] p o]
Variables:	:A, :dist, :X_Y
Assignment:	make "A 6 make "B :A + 1 + (count [X Y Z])
Functions	to sqr :X make "X :X*:X output :X end
Output to screen:	print "A "is :A
Conditional execution:	if :a<0 [print "Negative] [print "Poitive_or_Zero]

Elica's Logo

It is interesting to explain what is the relationship between Elica's Logo and other Logo implementations. First of all, most of the reserved words are removed. **This reduction is extended to the level of leaving only 8 reserved words in Elica ... and among them there are no identifiers for definition of procedures and functions.** The second step is to remove all complex data types and to leave only the most basic one: numbers, words and lists. The third step is to remove all procedures and functions both user-defined and system-predefined. This means that there is no definition, for example, of how to find the sum of two numbers. The last step is to make the internal representation as less complex as it is possible. Currently all data is held in blocks with unified structure.

Having in mind all these simplifications, one could ask what is the intention to do with such a limited system. First of all, to be able **to redefine all removed reserved words, functions and procedures.** To be able **to define operators** with their priority and associativeness. To complete the list, it should be possible **to make objects** (with fields and methods, multiple inheritance, etc.) and all other eliminated complex data structures. This limited system is expected **to provide events handlers** for different events, non-virtual and virtual properties, libraries, domains (or namespaces) and more.

Because of the unification of the internal structures, variables, procedures, functions, objects, operators, domains and libraries are represented in one and the same way, which naturally leads to the idea that for the user all these items can be presented in an unified manner. This is one of the basic achievements of Elica: **for the user there is no difference between a procedure and a variable, between operator and object.** All this separation is done on semantic basis and when the user starts to think in the terms of Elica, it will be possible to make programming simpler and more expressive.

Elica data

As said above, all the data in Elica is managed in unified blocks. Each block has two parts: informational and relational part. The informational part is capable to store numbers, words or lists.

The relational part contains data that defines the internal hierarchy and relationship between blocks. All blocks are organized in the form of a single tree. This requires a parent-child (vertical) relationship. Every block may or may not have a parent. Every block may or may not have children. Another relationship (horizontal) is brother-brother. All the children with one and the same parent are brothers each other.

This structure is quite enough to make it possible to achieve the main goals of Elica.

Procedures and Functions Are the Same

The following part of the paper will attempt to explain how the terms variable, procedures, functions, operators and objects are combined in one both physically and logically.

The first and most natural step is to combine procedures and functions. In Logo this unification is relatively easy, as both of them have identical definitions. The only difference is that a procedure terminates either when the word END is met, or when an OUTPUT statement is executed without arguments. Functions should always terminate with an OUTPUT statement with one argument.

The following two sequences of commands do one and the same job, but the left one uses a procedure, while the right - a function:

```
to sqr :x          to sqr :x
  make "x :x*:x    make "x :x*:x
  print :x         output :x
end               end
sqr 5             print sqr 5
```

This invokes the following question: if we have a routine, where with the help of a conditional command like IF we exist from it either without result, or with a result, then is this routine a procedure or a function? The answer is: **it is both of them, because it can be used it as procedure and as function.**

There are two independent factors, which can help conventional thinking programmer to distinguish procedures from functions. The first is whether the routine returns a value or does not. The second factor is whether the statement, which uses the routine, expects a value. In most of the languages both factors should be synchronized, otherwise an error is reported. Some languages allow semidesynchronization, which is naturally available in Elica. If a routine returns a value that is not expected, then this value is ignored. The full desynchronization is natural for Elica too, though it is almost forbidden in other languages. The behaviour of Elica when a value is expected from a procedure will be discussed later as it is very important.

Procedures and Functions as Variables (And Vice Versa)

Because Logo language originated as a simplification (mainly at a syntax level) of LISP, it is natural to represent commands as data. This feature is present in most of the Logo implementations including Elica. Commands are described as lists of commands and as far as all commands are composed of identifier, numbers and other special symbols, any command can be written as a list.

This does not mean that internally procedures and functions are stored as lists, but at least for Elica this is true - in Elica all commands are stored as lists even internally they are managed as lists are. **This gives the power to use variables as routines or to use routines as variables.**

In the first case (variables as routines), we can assign a variable a list value. If this list contains valid commands we can execute the list or even we can execute the variable as if it is a procedure. When executing the last command from the example below, Elica will treat the identifier A as a name of a procedure. The value of the variable contains a list of commands, which is actually executed.

```
make "a [print "Hello]
run :a          -> Will print "Hello"
a              -> Will print "Hello" too
```

Now let's have a look at the opposite case. Provided we have a procedure that prints out OK followed by a name given by the user, it is possible to run it, but also to use it as if it is a variable. In this case the value of the procedure as a variable is a list of all commands in

the procedure. This feature makes it possible to retrieve, analyze and change the definition of a procedure.

```
to b :x
  print "OK :x
end
b "Tom          -> Will print "OK Tom"
print :b        -> Will print "[[:x], print "OK :x, ]
```

Another important consequence is that there is no need for the system to support TO...END statement, which is used for declaring routines. The system automatically converts every occurrence of TO..END to an equivalent MAKE. This gives the reason to claim that routines in Elica are standard variables.

Operators in Elica

One of the most interesting features of Elica is that the definition of routines has a quite free form. The traditional form is after the name of the routine to list all parameters. In Elica it is possible to change the position of the routine's name among the parameters. The example below shows two ways of defining addition.

```
to + :x :y          to :x + :y
  ...
end                 end
```

The left example shows how to define + as a function. The right one - as an operator. Elica can handle both of the definitions. In fact, Elica can handle any position of the name and all these cases are internally processed in one and the same way. **So, procedures and functions are examples of operators. A more formal conclusion is that prefix, infix and suffix notations can be mixed even in one and the same expression.**

To provide more functionality, each operator can contain as child variables some important characteristics like priority and associativeness. These characteristics are used to resolve conflicts of how to interpret expressions and how values are distributed among parameters.

One unusual result is that it is possible to define an operator with 3 parameters to left of it and two to right. In addition, every routine can be executed with less or more arguments and this rule applies independently to both sides of the operator.

Objects Definitions and Object Instances as Variables

Elica supports two types of object creation. The first one is the traditional for OOP programming - you have an object definition, which is used to construct instances of an object. The other possibility is to create the object directly, field by field. In terms of Elica, the object is a variable, which has children.

```
make "a.x 5
make "a.p [print :x]
a.p
```

In the example above it is shown how to construct manually an object. In this case the object is called A and have two subvariables: X and P. X is a number, while P is a list of

commands. As it is mentioned above, every list can be used as a procedure (local procedures in an object are often called methods).

The disadvantage of this way of object construction is that it is not suitable for creating numerous instances, especially if an object is very complex and contains many fields and methods. To eliminate this disadvantage, Elica provides a way to define an object description, which can be used to construct object instances. Because the ideology of Elica prohibits the introduction of new reserved words, **it is a real challenge to make object definition with the available reserved words only.**

The solution is to use procedures which to initialize object instance. It is now the time to go back to the last case about procedures and functions. If a procedure is called as a function, it will have nothing to return to the caller, except all currently local variables. Elica assignment mechanism automatically attaches all these local variables to the result of the function. So, **if a procedure is called as a function, the result is a variable, which has children - the former local variables of the procedure.**

We can use this way of object definition and creation and rewrite the example above:

```
to myobject :x
  to p
    print :x
  end
end
make "a myobject 5
```

A side effect of the method of object definition is that when looking at it it is impossible to recognize whether it is a definition of a procedure or of an object. The functionality depends only upon how it is used. **The same rule applies to all other structures in Elica - whether a definition is a procedure, a function, an object, an operator etc. can be found or, better to say, determined by the style of usage.** It is like the evolution of Homo Sapience – it is not important what tool is in your hand, it is important how you use it.

To close the topic of objects let's remark that an object definition is in fact a procedure definition, which is just assignment, a value to a variable. So, an object definition can (and is internally) done as a standard MAKE statement.

One interesting example, which shows the confusion of implementing OOP terminology without understanding Elica is that in Elica some structures cannot be determined by this terminology. For example, there is no word for an object, which is operator at the same time, or for a variable with numerical value, which is object at the same time too.

Final thoughts

The size of the paper does not allow to explain all the possibilities that Elica provides to the user, but a simplified 3D-diagram can explain them in a structured form. The diagram below displays different interpretations of a variable. Each variable in Elica has three independent properties, **which in combination give a total of 24 different interpretations. While only 6 of them are available in other languages.** The simplest property is whether a variable has children or does not. The first (the closer) layer on the diagram is for

interpretations where the variable has no children. The second layer is when it has 1 or more children. The columns in the diagram define how the variable is used: either as a standard variable, or as a procedure, or as a function, or it is a procedure used as a function (functional procedure). Rows reflect the contents of the variable (children are not a part of this contents). The three different cases are when the content is not a list of commands, or it is a list of commands but either in prefix format or in non-prefix. Prefix format is when the definition starts with an identifier followed by arguments and then by the commands. In the non-prefix format at least one argument comes before the identifier.

The 6 grayed blocks are what other languages support: variables (var), procedures (proc), functions (func), operators (oper), object definitions (obj), and object instances (inst). All the other blocks are unused. But this does not apply to Elica!

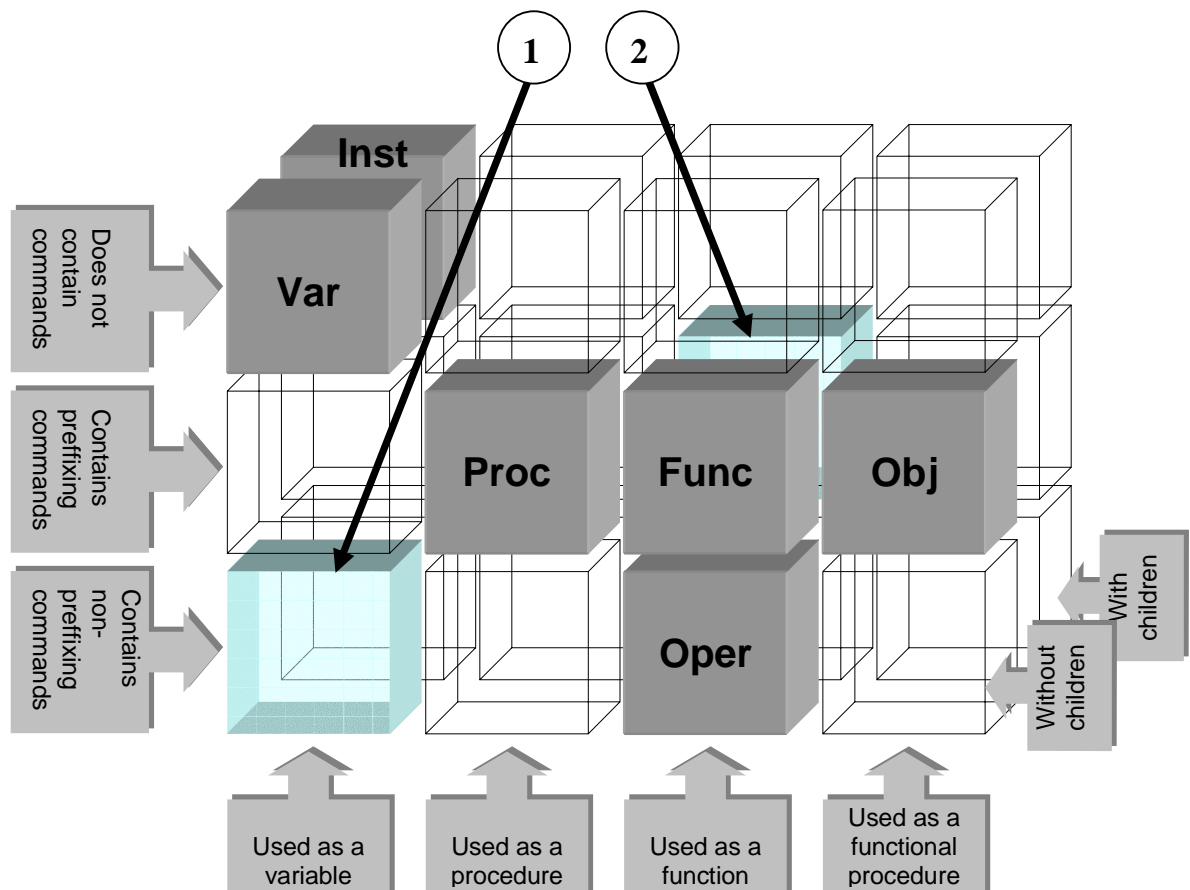
For example, the block marked with (1) is for a structure, which has no children, contains infix (or suffix) declaration and is used as a variable. One may think that such a structure does not have any sense, but here is how this interpretation has sense in Elica: let we have the operator "+". If we execute this command:

```
print :+
```

we will use "+" as a variable, it has no children and has an infix contents (provided the definition starts like TO :X + :Y ... END). The final result will be that the definition of "+", interpreted as a list of words, will be printed out on the screen.

The second example, marked as (2) on the diagram, is a little bit more complex. Reading the dimensions values of the properties, we can find that it corresponds to a variable, which has children, is used as a function and has a prefix content. If we want to identify such a variable in the context of other languages, we have to find an object instance, which is used as a function, and returns a value after evaluation.

As written before, every structure in Elica can be used in different ways and can be



modified at run time. This allows the user to modify a variable in such a way, that it becomes an object instance, being initially, for example, an operator definition, and later, the instance to be used as function. This flexibility shows why it is declared that variables, procedures, functions, operators and objects are one and the same thing, but most importantly, it shows that **we should not apply standard programming terms on Elica variables. When it is said that A is an object, this means that in this particular case the variable with name A is used as if it is an object.**

This multifunctionality will cause some trouble to standard-OOP programmers, as moving from complex environment to a simpler but more powerful one is not a trivial process. But once the user understands the concepts of Elica variables, he or she **obtains the magic force to freely express thoughts in Logo, concentrating on the sense rather than on the form.**

From a more global point of view, programming as a part of computer science stays much behind other human activities in terms of adulthood. Computer science stuff gets more and more complex and almost nobody tries to simplify it. Just an example: in Chemistry and in Physics, the scientists along with finding new laws, direct significant part of their efforts to understand the Nature and to unify these laws in attempt to find the sole unique Law. This attempt to find the essence made possible the union of the basic physical forces (still except the gravity), the discovery of atoms, the 'invention' of elementary particle's dualism and so on. On the other hand, programming languages get more and more powerful, introducing many new feature but at the price of heavier programs and more complex syntax.

Maybe it is too late to move back to simplicity without sacrificing functionality and flexibility, but one of the general aims of Elica is to show how to say more with less. As it is in all good poems.