МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2000
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2000
Proceedings of Twenty Ninth Spring Conference of
the Union of Bulgarian Mathematicians
Lovetch, April 3–6, 2000

PROGRAMMING AS POETRY (THE POWER OF THE SIMPLICITY IN PROGRAMMING)

Pavel Boytchev

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

1. Introduction. It is known that some of the world's greatest poets can write superb poems using only a few distinct words. The trick is how these words are combined and how the phrases are interpreted. Human language and poetry as a way of expressing can be both very condensed and sparse in matters of information density. They 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 – that is **how to make complex programs with a very limited set of init ial 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, whose 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. Elica's Logo originates ideologically from Geomland – a Bulgarian software package for the teaching and learning of school Geometry through explorations. What distinguishes Geomland's Logo from Elica's Logo is that the latter has been improved in many directions, some of which will be explained later.

The following table will list some ways of writing Logo statements, typical for the Logo language 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+(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 "Positive_or_Zero ]
```

2. Elica's Logo. It is interesting to explain what the relationship is 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 ones: 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 the least complex possible. Currently all data is held in blocks with unified structure.

Having in mind all these simplifications, one could ask what one intends 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. Finally, 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 event 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 the same way, which naturally leads to the idea that for the user all these items can be presented in a 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 a 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.

3. 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 of each other.

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

4. 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 unified, 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 the same job, but the left-hand one uses a procedure, while the right-hand one uses 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 evokes the following question: if we have a routine from which we exit by means of a conditional command such as IF either with or without a result, then is this routine a procedure or a function? The answer is: **it is both, because it can be used as a procedure and as a function**.

There are two independent factors which can help the programmer who thinks conventionally distinguish procedures from functions. The first is whether the routine returns a value or not. The second factor is whether the statement which uses the routine expects a value. In most languages both factors should be coordinated, otherwise an error is reported. Some languages allow semidiscoorination, which is naturally available in Elica. If a routine returns a value that is not expected, then this value is ignored. Full discoordination 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.

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

This does not mean that procedures and functions are stored internally 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 we can even execute the variable as if it were a procedure. When executing the last command from the example below, Elica will treat the identifier A as the 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 were 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 the TO...END statement, which is used for declaring routines. The system automatically converts every occurrence of TO..END to an equivalent MAKE. This gives one a reason to claim that routines in Elica are standard variables.

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

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

The left-hand example shows how to define + as a function, the right-hand one as an operator. Elica can handle both definitions. In fact, Elica can handle any position of the name and all these cases are internally processed in 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 the same expression**.

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

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

7. Objects definitions and object instances as variables. Elica supports two types of object creation. The first one 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 an object manually. In this case the object is called A and has two subvariables: X and P. X is a number while P is a list of commands. As 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 an object definition with the available reserved words only**.

The solution is to use procedures which would initialize object instance. It is now 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 current local variables. Elicas 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 form 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 on 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, determined by the style of usage**. It is like the evolution of Homo Sapiens – it is not important what tool is in your hand, it is important how you use it.

To close the topic of objects let us remark that an object definition is in fact a procedure definition, which is just assignment of a value to a variable. So an object definition can (and internally is) 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 an operator at the same time, or for a variable with numerical value, which is object at the same time too.

8. Final thoughts. The size of the paper does not permit 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 not.

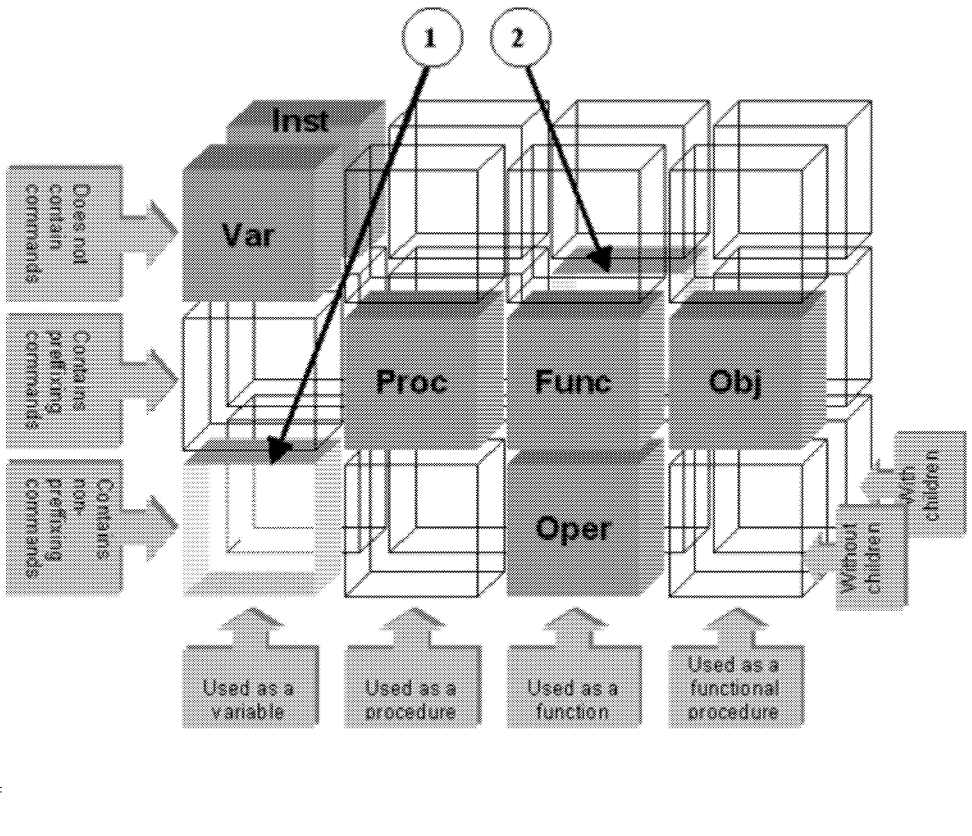
The first (closer) layer on the diagram is for interpretations where the variable has no children. The second layer is when it has 1 or more children. The columns in the diagram define how the variable is used: either as a standard variable, or as a procedure, or as a function, or as a procedure used as a function (functional procedure). The rows reflect the content of the variable (children are not a part of this content). The three different cases are when the content is not a list of commands, and when it is a list of commands but in prefix or non-prefix format. The 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 an infix (or suffix) declaration and is used as a variable. One may think that such a structure does not make any sense, but here is how this interpretation makes sense in Elica: let us have the operator "+". If we execute this command:

```
print :+
```

we will use "+" as a variable, it has no children and has an infix content (provided the definition starts as 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.



1
2
Inst
Var
Proc
Func
Obj
Oper
Does not contain commands
Contains preffixing commands
Contains non-preffixing commands
Without children
With children
Used as a variable
Used as a procedure
Used as a function
Used as a functional procedure

The second example, marked as (2) on the diagram, is a little more complex. Reading the dimensions 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 to be used as function. This flexibility shows why it is stated that variables, procedures, functions, operators and objects are the same thing, but most importantly, it shows that **we should not apply standard programming terms to 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 were 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 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 adultness. 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 discovering new laws, direct a significant part of their efforts to understanding Nature and to unifying these laws in an attempt to find the sole unique Law. This attempt to find the essence made possible the union of the basic physical forces (still except gravity), the discovery of atoms, the 'invention' of the dualism of elementary particles, and so on. On the other hand, programming languages get more and more powerful, introducing many new features but at the cost 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 by saying less. As it is in all good poems.

REFERENCES

[1] R. Billstein, Sh. Libeskind, J. M. Lott. Logo – MIT Logo for the Apple. University of Montana, Missoula, Montana, 1985, 196-208

[2] P. Boychev. Elica and Objects. Eurologo'99 Proceedings, 1990.

[3] P. Boychev. Overview of Research Logo System, PEG97, Proceedings, 1997, 30-37.

[4] M. E. Burke. Logo and Models of Computations. San Jose state University, 1987.

[5] St. Hain. ObjectLogo for the Apple Macintosh. Paradigm Software, Cambridge, Massachusetts LogoWriter Reference Guide (1986) Logo Computer Systems Inc., 1990.

[6] I. Nikolova, I. Georgiev. Programming with Logo. Publishing House "Technica", Sofia, 1992.

[7] T. W. Pratt. Programming Languages – Design and Implementation. Department of CS, University of Texas as Austin, 1975, 342-553

[8] B. Sendov, S. Boycheva. Geomland. University of Sofia, 1997.

Department of Information Technologies
Faculty of Mathematics and Informatics
University of Sofia
5, James Bourchier Str.
1164 Sofia, Bulgaria
e-mail: mgozi@yahoo.com

ПРОГРАМИРАНЕТО КАТО ПОЕЗИЯ (СИЛАТА НА ПРОСТОТАТА В ПРОГРАМИРАНЕТО)

Павел Бойчев

Докладът представя резултата от опит да се проектира език за програмиране със силно ограничено множество от запазени думи, но в същото време позволяващ да се дефинират функции, процедури, оператори и обекти. Въпреки че синтактичните правила на езика са прости, възможно е да се съставят сложни програми и структури, които са неосъществими при другите езици за програмиране.