



The Very Logo Way

By Pavel Boytchev

© 2002 Logo Foundation

You may copy and distribute this document for educational purposes provided that you do not charge for such copies and that this copyright notice is reproduced in full

1. Introduction

A couple of weeks ago I watched a documentary about several villages in Rhodopa mountain. What impressed me most was the way peasants spoke. They used very basic words in simple sentence structures. If I were them I would express the same with much more words. That documentary made me think about my experience as a software developer. In the past I used to work with a dozen different programming languages. Nowadays I'm generally stuck to C, Logo, and Pascal (not in this order). That documentary plugged a question in my mind.

Today, while I was spring cleaning my mind, I decided to face the question. There is a saying that a good programmer can use virtually any language. I tend to agree on this, but the question needs an answer. The question is: "What is the very Logo way? " I could rephrase it as "Does Logo (as a language!) provide functionality almost impossible with most other languages? "Note that I talk about Logo as a language, not as an educational tool. Most of the Logo features like list processing and turtle graphics can be implemented in other languages relatively easy. There must be something else, something which an ordinary Logo user will never use just because (s)he never thought about it.

My intention while writing this article is to introduce you to three interesting problems and their unusual Logo solutions.

2. The first problem: "A bureaucratic chain"

I'm sure that most of you have experienced the power of bureaucracy. Call it bubbledom or red tape, it is an invisible but quite tangible force. Imagine you need to be serviced by the red-tapist named Anthony. He is the perfect bureaucrat. The only thing he knows is the name of one of his colleagues – Barbara - and he always tells people to go to her office. When you visit his office, he redirects you to Barbara. But she is as good a bureaucrat as he is. So, you go to her office and she says you must go to Clark's office.

You go there just to find out that the right office is that of Donna. You can go on this way all your day (or life). The problem I'm talking about is if you know in advance the clerks and how they service visitors, then you can figure out whether the saga will end or you will go from office to office forever.



Figure 1 A bureaucracy chain

This is just one of the many possible dressings of the same problem. You might have encountered it as the whispers-are-going-round problem. Whenever a person hears a rumour (s)he whispers it to his(her) best friend. Everybody has at most one best friend, and best friends remain best friends forever. The question in this interpretation is whether the rumour will vanish at some point or will circulate forever.

Both the bureaucracy and the rumour problems can be formalized to whether a chain of object is cyclic or not. Let's now try to solve this problem. The input data contains pairs of names. Each of the pairs defines a single connection in the chain. For example, the chain of names above can be formalized to the following three pairs:

(Anthony, Barbara)

(Barbara, Clark)

(Clark, Donna)

If I start to solve this problem in C I need to dive in an ocean of pointers. The Logo way to solve it is to use lists. Logo has a pretty good set of list-management subroutines, some of which are extremely fast. A typical Logo solution is to start from the first node and go along the chain. At each step we store the name of the current node in a list. At some point we will either find a dead end (surprisingly this is the happy end in the case of bureaucracy problem) or visit a node which we have already visited. The latter is checked easily as long as we keep track of all visited nodes. This solution is the additive one.

<p><u>List of visited offices</u></p> <ol style="list-style-type: none"> 1. Anthony 2. Barbara 3. 4. 	<p><u>List of offices</u></p> <ol style="list-style-type: none"> 1. Anthony 2. Barbara 3. Clark 4. Donna
<p>Figure 2 The additive solution is to keep track of all visited offices. When we visit an already visited office, then there is a loop in the chain.</p>	<p>Figure 3 The subtractive solution starts with a full list of offices and every visited one is removed from the list.</p>

There is a subtractive solution too. It starts with a list of all nodes and at each visit the node is excluded from the list.

Although both these solutions represent the Logo way, they do not show the glitter of Logo. They are the Logo way, but they are not the very Logo way. I found a dozen solutions of the problem, but one of them is so different and so crazy that I'm eager to share it with all of you.

Imagine a man and his father start going from office to office. The man is young and can pass two offices while his father passes only one. Because the son progresses faster, sooner or later he will either finish his journey or will catch up with his father. In the first case he has proved that there is no cycle so he can patiently wait in the lobby for his father to finish. In the other case the best he can do is to advice his father to give up, because there is a cycle and they will never finish the saga.

How to implement this in Logo in the very Logo way? How to find a solution which does not use temporary list(s) that can grow as much as the number of nodes? Is it possible to construct a solution which represents the reality as closely as possible?

Let's assume that all people involved in the scenario of the problem are named differently. Each person is represented as a variable. Everyone knows only one thing - the name of the next person in the chain. We'll store that knowledge as variable's value. If there is no next person, then the value is "Lobby".

Here is one sample set of data:

```
make "Anthony "Barbara
make "Barbara "Clark
make "Clark "George
make "Donna "Emily
make "Emily "Frank
make "Frank "Lobby
make "George "Barbara
make "Lobby "Lobby
```

The commands above describe that Anthony always redirects visitors to Barbara, while after visiting Frank you will finish your job and will go to the Lobby. Pay attention to the

last line where we say that if the someone goes to the lobby (s)he will stay there and wait.

The next picture visualizes the same set of data. In addition, there are two paths - blue and green. The first one starts from Anthony and the second from Donna. The first path contains a loop, as long as George will redirect visitors to Barbara. The second path ends in the lobby.

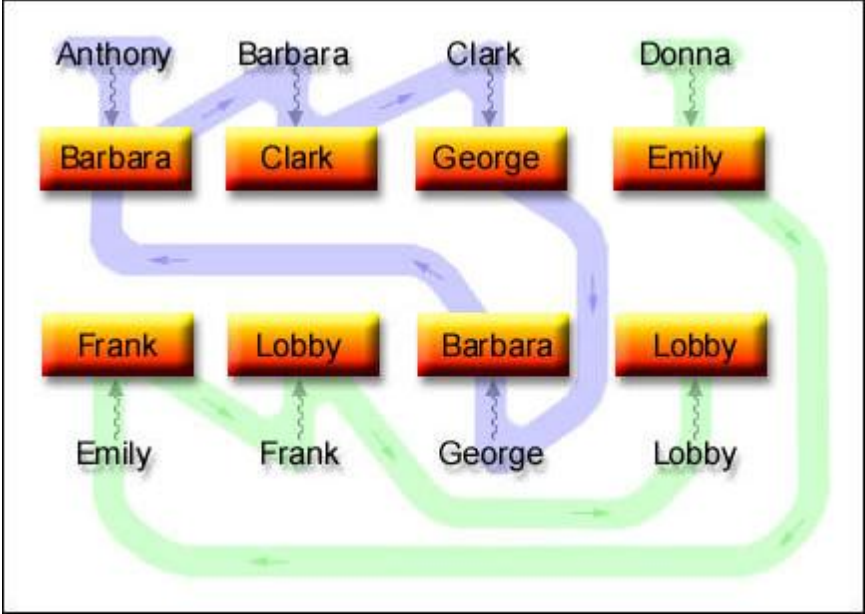


Figure 4 The blue path starting from Anthony contains a loop. The green path starting from Donna ends in the lobby.

This example shows that the existence of a loop depends not only on the internal links, but also on the starting point.

To complete the data representation, we need to define the father and the son. The simplest way is to define two variables, called Father and Son. Their values (i.e. the facts that they know) are the names of the officer where they are at the moment. If the starting office is Anthony's, then at the beginning both variables will have the value "Anthony. In a step, Father will be set to "Barbara, and Son - to "Clark.

The task is to build a function called *saga_will_end?* It takes one argument which is the name of the first visited officer. The output is either the word "Yes or "No. If the saga is endless (somewhere in the chain there is a loop) then the output must be "No. Otherwise it must be "Yes.

We will build the function in three easy steps.

2.1. Step 1 - Building the frame

We want to define a function with one input - the name of the first visited officer. So we can name this input *Father* as long as this is the first officer that he will meet. Apparently the situation is the same with the son. Here is what we have so far. The ellipses show where we will include more commands at later steps.

```

to saga_will_end? :Father
make "Son :Father
...
end

```

To those interested in programming methods, this is a pure form of top-to-bottom approach. We start from the topmost level and smoothly (or not so smoothly) go deeper into details.

2.2. Step 2 - Visiting offices

The second step is to define how the father and his son visit offices. Whenever the father moves an office ahead, the son moves twice more. To program this, we add an endless *while* cycle. The first two commands in it are to move the father one step and the son two steps further. Here is the source of the function.

```

to saga_will_end? :Father
make "Son :Father
while "true
[make "Father (:Father)
 make "Son :(:Son)
...
]
end

```

Do not hurry to blame me. There is no typo in the text above. The defect is in fact an effect.

Definitely you've seen the strange use of *colon* `:`. In Elica Logo the colon character is not only a part of the syntax structure or a shortcut of *thing* ". It is a function. It gives the value of a variable with a given name. The name can be a word (as it is in all other Logo implementations that I'm aware of) or an expression (as it is shown above). I could give a piece of advice for people whose favourite Logo does not have this functionality: *Do not give up*. You can use `THING` instead of colon.

Multiple colons in conventional Logo

If you do not want to or cannot use so advanced *colon* `:` you might consider using the good old traditional `THING`:

<code>THING "Anthony</code>	<code>= :Anthony</code>	<code>= "Barbara</code>
<code>THING THING "Anthony</code>	<code>= (:Anthony)</code>	<code>= "Clark</code>
<code>THING THING THING "Anthony</code>	<code>= (:(:Anthony))</code>	<code>= "George</code>

We still have to make the third step. It will contain checks for two conditions – whether the chain is cyclic, not cyclic, or is still undetermined.

It is easy to figure out that if the son goes in the lobby, then there is no cycle in the chain. On the other hand if the son enters an office and he finds his father there then this is a proof for a loop. The two checks are the final step and the function below is the final one.

```
to saga_will_end? :Father
make "Son :Father
while "true
[make "Father :(:Father)
 make "Son :(:(:Son))
 if :Son=:Lobby [output "yes]
 if :Son=:Father [output "no]
]
end
```

We can try it with different initial offices. Depending on the first visited office we will get different results.

```
print saga_will_end? "Anthony
print saga_will_end? "Barbara
print saga_will_end? "Donna
```

For the first two queries we will get *no*, for the last one - *yes*. This is the whole solution. Of course, it is not the shortest, nor it is the fastest. It is just one of the many possible solutions.

As I wrote earlier, in this article I will discuss two problems and their Logo solutions. Although the solution of the first problem can be implemented in all Logo implementations, the second problem will be a problem to most of them.

3. The second problem: Artificial "Artificial Intelligence"


Before getting into the problem, have a look at the dialogue in the right-hand part of the page. You may get the feeling that the answering part understands what it is told and replies appropriately.

This is a typical problem of the Artificial Intelligence. I'm pretty sure that if you have an hour or two free time you could make a Logo program that accepts statements like *john is smart* and answers to *what* and *who* questions. Of course, if you have Prolog in your hand, you can solve the problem in a blink of an eye.

Well, most likely your Logo program will follow the Logo way. Your program accepts user's input, analyzes it and prints back the result. This is the so called *inputand- calculate method*.

But the very Logo way is somewhat different - it is based on the *understandand- reply method*. From the dialog exclude all answers. What is left is a set of sentences and questions that are the Logo program itself.

Yes, I'm not kidding. It is possible to write in Logo a small library which allows the user to write simple programs in almost natural language. The dialog shown here is not imaginary. It is the output of a Logo program which I wrote some time ago.



mike is tall
john is tall
peter is old
john is old

what is john
[JOHN IS TALL AND OLD]

john is smart

what is john
[JOHN IS TALL, OLD AND SMART]

who is smart
[JOHN IS SMART]

who is old
[PETER AND JOHN ARE OLD]

mike is old

who is old
[PETER, JOHN AND MIKE ARE OLD]

Figure 5 A simple dialogue between a human and a computer. Human's words are in handwrite-style font. Computer's response is framed in square brackets.

The idea about such use of Logo originated when I made the Elica North Pole Project, where the user can write programs using commands like *start from A* , *wait for 6 hours*, *land in B* , etc. Having this idea in mind it was just a matter of time to write a few natural language commands. Let me try and explain you how I did it. There are several important things. The first one is that most of the words are in fact names of functions. For example, *mike* is a function which always outputs the word *"mike*. Here is the definition:

```
to mike
output "mike
end
```

In a similar way most of the other words are functions too. However there is no need to define all functions explicitly. We can use a single function which defines all other functions.

```
to words :x
while :x<>[]
[ make first :x se [[:x]] se [output "] first :x
make "x bf :x
]
end
```

```
words [who what mike john peter tall old smart]
```

One of the possible realizations of such a function is shown above. It accepts as an input a list of names. Then it creates functions corresponding to these names. All created functions return as results their names. Although this technique is not quite efficient, it allows us to write:

```
mike is tall
```

instead of:

```
"mike is "tall
```

Anyway, both expressions are equivalent. This is the first important idea.

The next one is to find what to do with the word *is*. Elica is quite flexible and you can define operators directly and entirely in Logo. Thus, defining *is* as operator is one of the many possible solutions. To define an operator you do not need to learn anything new. Remember, you are now on the very Logo way and there are no obstacles ahead of you. All the things you do must and will remain Logo-like all the time. So, to define an operator called *is* you just write:

```
to :x is :y
...
end
```

Pay attention on the heading of the definition. It is a *to...end* definition except that one of the parameters is before the name *is*. This informs Elica translator that this is a definition of an operator which expects one left argument and one right.

The body of the operator processes the three different sentence patterns:

1. what is <someone>
2. who is <somewhat>
3. <someone> is <somewhat>

We will look at the cases one by one. Let's start from the last one. If *is* is used as a statement, then we have to update our 'knowledge base'. We can do this in numerous ways. The one which I prefer is to create variables for each piece of knowledge. If we process a statement like *mike is tall*, then we create the variables *mike.iswhat.tall* and *tall.iswho.mike*.

When we have a question like *who is tall* all we need is to scan for all variables which names start with *tall.iswho*.. If two persons are tall, then there will be two variables that start with *tall.iswho*.. When we have a *what* question we find an answer in a similar way. Consider *what is mike*. To answer it we look for variables starting with *mike.iswhat* .

The realization of the operator *is* is much easier to implement in Elica Logo rather than in any other Logo. Here is the full definition:

```
to :x is :y
if :x="what
[print (se :y "is sent names :(word :y ".iswhat)) ]
[if :x="who
[if (count names :(word :y ".iswho))<2
[make "i "is]
[make "i "are]
print (se sent names :(word :y ".iswho) :i :y)
]
[make (word :x ".iswhat. :y) []
make (word :y ".iswho. :x) []
]
]
end
make "is.onpriority 10
```

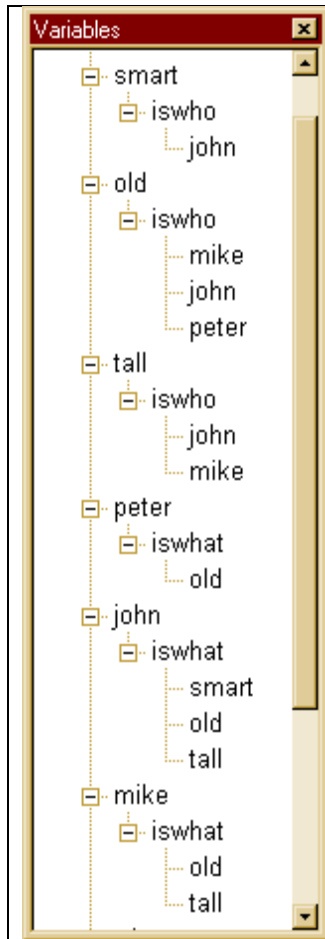


Figure 6 Variable hierarchy

There is an interesting question. Why we use names like *mike.iswhat.tall*, rather than *mike_iswhat_tall*, or *mike\iswhat\tall*? The answer is that if we use dots to separate subwords in a word then Elica will structure these names in a way which is very efficient in terms of memory storage and access time.

Consider the statements in **Figure 5**. As a result of executing operator *is*, the following variables will be defined (in order of creation):

```

make.iswhat.tall
tall.iswho.mike
john.iswhat.tall
tall.iswho.john
peter.iswhat.old
old.iswho.peter
john.iswhat.old
old.iswho.john
john.iswhat.smart
smart.iswho.john
mike.iswhat.old
old.iswho.mike

```

The image on the left is a snapshot of Elica screen. The window "Variables" displays hierarchically all the variables in the memory. When a name of a variable contains a dot, that dot splits the name.

Thus the name *mike.iswhat.tall* is split into three subwords: *mike*, *iswhat* and *tall*. When the heading subwords of several names are the same, then these subwords are merged. You can see in the bottom of **Figure 6** than *mike.iswhat.tall* is merged with *mike.iswhat.old*. The first two subwords of both names are shared between them.

What is the easiest way to find all variables with the same header? Of course you can get a list of all names and check each of the names. In Elica there is a function which does this for you. Its name is *names*. Note that the argument of this function is not a word, but a variable.

Elica allows you to use the value of variable giving its partial name. Thus the value of *:mike.iswhat* is a set of two variables. Their names are *old* and *tall*. The function *names* expects as an input a variable whose value is a set of variables. The result of executing *names :mike.iswhat* is the list *[old tall]*.

In the source of operator *is*, partial names are built on-the-fly. Accessing their values can be done with the functional *:*.

Using *names* is not sufficient. We need to format its output and to make it more natural. We can do this with the help of another function, called *sent*. The function must convert a list of words into a natural language phrase.

```
print sent [mike] -> mike
print sent [mike john] -> mike and john
print sent [mike john ken] -> mike, john and peter
```

Needless to say, there is no any problem to realize this function. Without being very picky, I would suggest the following quick'n'dirty implementation:

```
to sent :x
  if 1>=(count :x) [output :x]
  make local "y ""
  while 2<(count :x)
    [make "y (word :y first :x "' , ' )
     make "x bf :x
    ]
  make "y (word :y first :x "' and ' first bf :x)
  output :y
end
```

These definitions (*word*, *is*, and *sent*) are enough to build our example of artificial AI. Why I call it artificial? That's because the example is very simple and does not reveal the complexity of AI problems.

As I wrote earlier, the solutions of the two problems in the article are not intended to be the fastest, the simplest or the shortest. And definitely, they are not the only possible solutions.

Some programmers argue that using unique features of a programming language is a bad habit. They say this is against portability. As long as the goal of computer science is multidimensional, these programmers are right in respect to one particular projection.

My personal and thus subjective opinion is that using the power of a language is very beneficial. If we don't use the Logoish features of Logo, then it might be better to switch to another programming language. Of course, the other case is as bad as this one. Using the extreme tricks of a language could make a program hard to read and understand.

4. The third problem: "Selfregenerator"

Some 20 years ago I took part in a contest. The problem was to write a program that after its execution prints out its source. Of course, it was not allowed for the program to access its source. Instead, it had to be regenerated by the program itself. Needless to say, all spaces, new lines and other special symbols had to be printed exactly.

My solution was based on the modern at that time BASIC. Some years later, I wrote a similar solution in Pascal.

Both solutions explore the same idea - the program contains two parts. The first one is a list of assignments to an array. The assigned values are the source lines of the second part. The second part should print itself (this is easy, as long as it can read itself from

the array). But before this, it must regenerate the first part. That's easy too because the source of the first part be produced by the values of the same array. Of course, there are some minor problems and you will definitely meet them if you try to solve the problem by yourself.

Well, how to solve the same problem in Logo? What follows a solution which is based on the same idea as my ancient BASIC and Pascal solutions. The first part is formed by the first 7 source lines; the second part - by the next 7 lines. To make the solution shorter, I've used a trick - I've used numbers as names of variables (that's legal in Elica). This is done just to make the source shorter.

```
make 1 "'make "t 1'
make 2 "'repeat 14'
make 3 "'[ if :t<8'
make 4 "' [print "make :t word "" char 39 word :(:t) char 39]'
make 5 "' [print :(:t-7)]'
make 6 "' make "t :t+1'
make 7 "']'
make "t 1
repeat 14
[ if :t<8
[print "make :t word "" char 39 word :(:t) char 39]
[print :(:t-7)]
  Make "t :t+1
]
```

That's a nice more-than-300-(including-spaces)-characters-long Logo solution. I think there is no need to show the result of the execution. It is the same as the text above except for the bold face and color formatting.

But it raises a question. Is there a shorter solution? Definitely, we can easily get a shorter one by removing all unnecessary spaces from the source and using the short names of the commands. But that's not the goal. There is one interesting command (and function) in Logo. It is called *run*. Maybe with the help of *run* we might be able to find an entirely new solution?

Fortunately, the answer of this question is positive. There is a solution of only 70 characters:

```
make "a [print " make " " " a : a word char 10 " run " : " a]
run : a
```

Again the program contains two parts - the first one contains an assignment, and the second one executes the assigned value. The key factor here is that the assigned value prints the entire program by self-referring. It uses itself to print itself

Note: Do not let the spaces to confuse you. The MAKE statement is the same as this one:

```
make "a [print "make "" "a :a word char 10 "run ": "a]
```

but because PRINT inserts spaces between printed values, we need to provide the source with spaces too.

This resemblance to recursion makes me feel that we might like a new discount. What about 51 characters? Here it is the next version:

```
make "a [print " make " " " a : a char 44 " a] , a
```

Here we have almost the same idea, but instead of running the contents of variable *a* it is executed as if it is a procedure. Thus we get rid of *run*. I'm a little bit sad because we've used *run* to reach a better solution and then we abandoned it.

Want a shorter Logo solution? What about 40 characters? 30? Well, let this be your homework if you are really interested and want to give it a try. Try to find a solution which is less than 30 characters long. As a hint I can say that there must be 4 or less occurrences of *double quotes* " and at most one occurrence of *colon* .:

5. Noitcudortni

No, this word is not wrong, although my spell-checkers spell on me that it is. In fact, it is the reversed writing of "introduction". I'm not able to decide how to title this final section. Definitely it is not a summary. It is not a conclusion either. Maybe it is the moral of the whole story?

Trying to show the very Logo way is not easy. There are many ways of programming for those of us who use Logo. If we need we can use it to write programs the C/Pascal way. If we need we can use it to write programs the Lisp way. And of course the best is if we use Logo to write programs the Logo way.

If we imagine the different ways of programming as paths, then the very Logo way does not map to any of them, because it is the power and the freedom to choose any way. Thus we get the flexibility to manoeuvre the way we want, not being obstructed by the landscape.

Although I talk a lot about the very Log way, I still think I'm on my first steps on this way. It is not so important what is the position. What matters is the heading.

We, the Logo community of users and developers have the responsibility and the pleasure to invent the very Logo way and to build it ... in the very Logo way.

Volunteers are welcome.