Error Reporting in Elica

Pavel Boytchev

Elica Team
Bulgaria, Sofia
pavel@elica.net

Abstract

Some of the latest improvements in Elica are in the area of proper error reporting. Many Logo implementations do not provide enough context information about syntax and run-time errors. This is a topic routinely ignored by many software vendors. The presented system, however, tries to feed the user with error-related information that is complete and well structured into different levels of abstractions. It locates errors at per-character precision, provides error-specific hints, reveals error history and converts problematic Logo commands in various representations.

Keywords

Logo, Elica, error reporting, source representation, Error Inspector, AI

"If I were trying to sell a Logo interpreter for money, I'd probably spend a year or two working on nothing but error messages"
Brian Harvey

1. Why error reporting?

Long time ago there was a short discussion at LogoForum, an online Logo community, about Logo reaction to program errors [Boytchev P, Gorman B, Harvey B and Tomcsanyi P (2002)]. At that time Elica was almost a kid – it was a pretty new Logo implementation, and was enjoying all the things it can do due to its flexibility of expressiveness [Elica Team (2003)]. However, as a kid, it paid minimal attention to the dark side of programming – dealing with bugs. In most cases, bugs were ignored and bypassed, and this gave hard time to many Elica users.

Now, being mature, Elica understands not only its rights but also its responsibilities. An essential part of them is the error reporting. This includes the data that the system gives to the user when there is a problematic situation.

In March 2003 the first draft version of the Error Inspector was completed. It is the module that reports program errors, structures and displays hints and translates Logo.

2. Error details hierarchy

When there is an error, Elica's main concern is not who is wrong, the user or the computer, but what can be done to fix the problem. To achieve this, all the information about errors is grouped into levels. The first level contains only general information that all Logo implementations provide. The second level contains detailed explanation of the problem (as seen by Elica), precise error location and hints that can help the user see the problem from his own viewpoint. The third level tracks the execution hierarchy that led to the problem, because the place of the actual error is often different from the place of error detection. Finally, the forth level translates Logo to Lisp, assembler or English.

3. General error information (Level I)

To better show Elica error reporting let's start with the common error of missing parenthesis inside brackets – see Figure 1. When this source is executed, Elica shows a short error message, the name of the source file and an arrow that points the line where the error is detected. The error message is parametric[1] where the parameter is the word "(". Many Logo implementations provide this or similar information. Unfortunately, for many of them this is the only thing that the user is informed about.

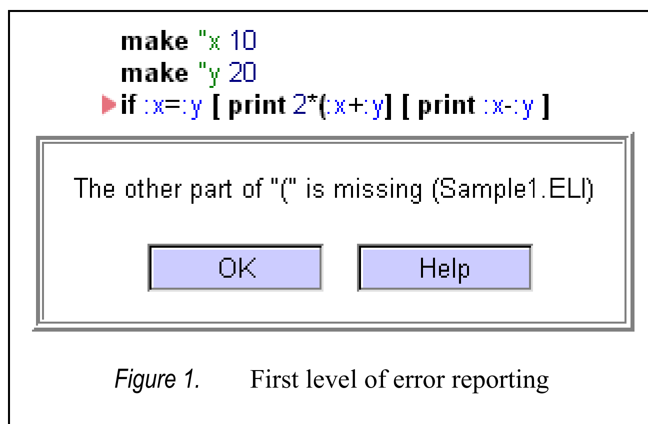

Figure 1. First level of error reporting

There are two buttons below the error message text. If the [OK] button is pressed Elica will show the file containing the error and the will put the cursor right before "("[2]. On the other hand, the [Help] button will activate the Error Inspector that will prepare error data for Level II.

4. Detailed explanation, localisation and hints (Level II)

The Inspector collects lots of data about the error, structures it in a user-friendly way and displays it in the Information window. The Level II data contains detailed error explanation, complete error reference and on-topic hints.

4.1. Error explanation

The Inspector gives three explanations of the error. The short one in our case is:

---

[1] Many other messages are parametric too. For example, when a variable is not found, its name is included in the error message.

[2] Logo implementations usually provide per-line or per-command precision of error location. However, Elica tries to point the beginning and the end of the error at a per-character precision.

"Pair mismatch (...]"

Then the Inspector adds a longer explanation:

"Elica found (but did not find any matching). Instead it found] for which there is no ["

followed by a source reference and a full explanation:

"The (symbol is used to identify the beginning of an expression or a command. The error has been reported because Elica was not able to find the end of the expression or the command which should be marked with). Instead of) Elica found the symbol] which is used to identify the end of a list. An expression and a list could be either nested one in another or independent. They cannot cross-intersect."

4.2. Source reference

The source reference shown on Figure 2 contains various helpful pieces of information. Except for the file name and line number, it cites the source. The part that confuses Elica is marked in red. There are two links next to the file name. [Locate] activates the source window and moves the cursor to the beginning of the error. [History] points to the next level of error information – the error history.

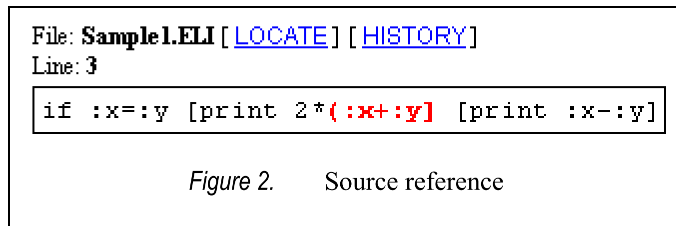

Figure 2. Source reference

If the explanations or the source reference are not enough to resolve the problem, users could consult the section with hints or view Level III data.

4.3. Hints

This is the most advanced advice that the Inspector could provide in respect to the error[3]. Among the five hints specific to our case, there is one that is quite relevant:

"Check whether there is a missing) that corresponds to the opening (, or a missing [that already have closing]".

The selection of hints depends not only on the type of the error. Consider these two obviously incomplete commands:

```
repeat 5
repeat [ make "a :a+1]
```

They are accompanied by different sets of hints. The first one contains the suggestion to add a list of commands, while the second suggests inserting a number of repetitions.

---

[3] The author has studied more than a hundred distinct cases of errors and has included unique set of hints to each of them.

Hints are something dynamic and get more precise and richer as times goes on. However this does not happen by its own. Users are key factors in this process. They can vote on the best hints[4] or suggest a new hint to Elica developers. Voting is used to examine the frequency of reasons for each type of errors. This provides further ideas for Inspector's improvement.

5. Error history (Level III)

Source reference from Level II has a link to the error history. When a program is executed some actions call other actions. In case of a problem, the error could be on any level of execution. Let's have a procedure that generates two random numbers and passes them to another procedure that calculates an expression and prints it on the screen:

```
to expr :x :y
    print (1+:y)*(sin :x)-count :y
end

to try
    local "a "b
    make "a random
    make "b random
    expr "a :b
end

try
```

The bug here is that instead of the value of variable `a` we pass its name – the word 'a'. When we start the program we get the message:

"'a' cannot be used as a number"

and the cursor lands just before `sin`. If we then go to the error history we could see that when the error occurred there were four commands that have not been completed – Figure 3.

---

[4] Some errors have huge lists of hints. For example, if Elica finds `:foo` but cannot find `foo`, the Inspector provides 10 hints. This means there are at least 10 different reasons for getting the error message *"Unknown variable foo"*. The hints are:

- *The name **foo** does not exist among all known and accessible variables' names;*
- *The name might be misspelled, or the variable which value is supposed to be accessed might have been created with misspelled name;*
- *If **foo** is a name of input then the corresponding action might have been called without a value for this input;*
- *Check whether a variable called **foo** is created before using its value;*
- *The name **foo** might not be found if it is used as a name of a local variable in another object which is not directly accessible from the current command;*
- *If the name is used as a name of a local variable in an object then use a compound name to access it;*
- *If **foo** is supposed to be defined in an Elica library then load it with run;*
- *If **foo** should be used as a word then remove : or replace it with ";*
- *If the name is generated at run time then check the way it is generated and the values of all used values;*
- *When :**foo** should be used as one word then use single quotes to protect : from being used as a system symbol.*

History helps users to track the execution stack even down to the definition of sine. The official error position is reported in record #2, the actual exception occurred in #1, the wrong statement is in #3 and everything happened when Elica executed the command in #4.

Tracking errors that span over several layers of execution could be tricky. That's why when a record refers to an action, except for the full source reference, the Inspector provides action name, a link to its definition and a link to the values of its inputs. Every record in the error history has also an [Explain] link to Level IV data.

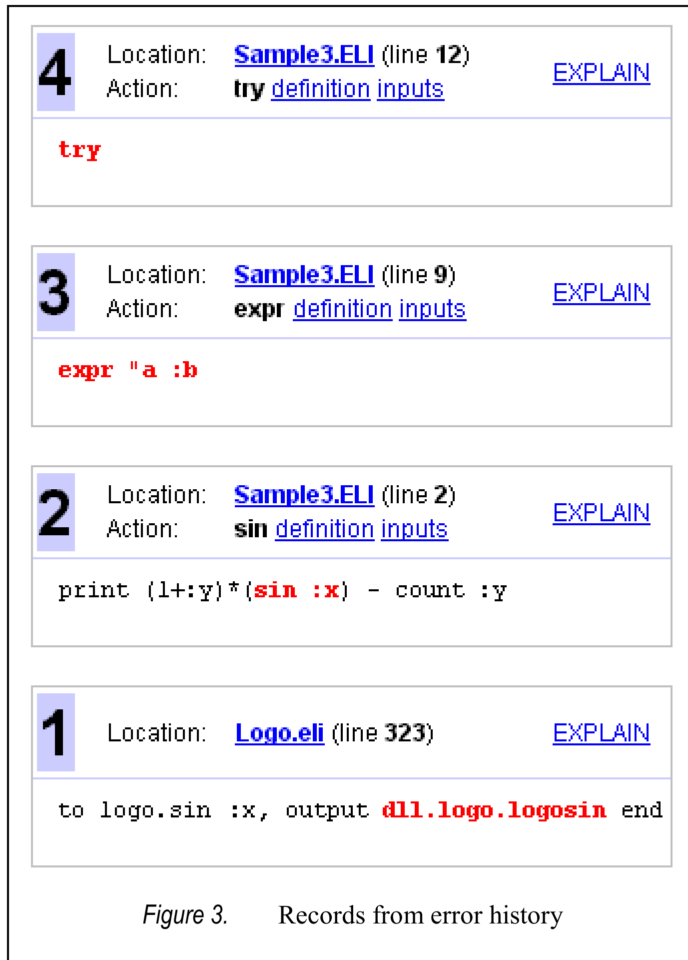

Figure 3. Records from error history

6. Translating Logo (Level IV)

The main purpose of the Level IV is to assist the user in locating bugs that are hard to find even with the help of all the information from previous levels.

Historically, this level originated because of two distinct reasons. The first one is that there are several types of bugs that are very hard to locate and there should be a tool that helps users to find them. The other one is the planned integration of Elica with AI.

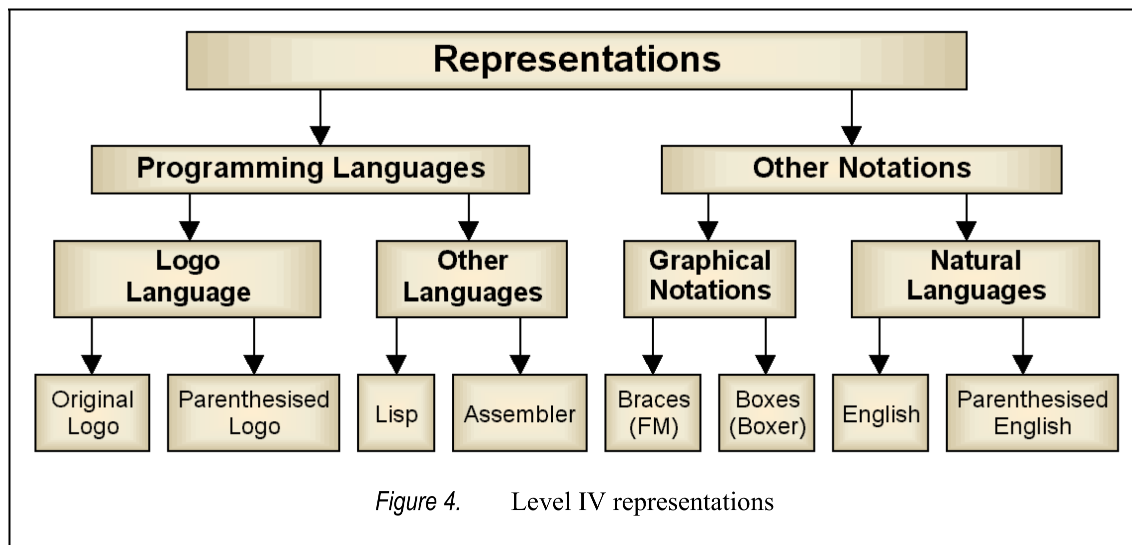

Figure 4. Level IV representations

Formally, Level IV shows different representations [Doyle M (1995)] of the Logo commands in the error history. As of March 2003, there are 8 different representations, which classification is shown on Figure 4.

6.1. Original Logo source

This is the most natural representation, because is shows the Logo command as it is in the source of the program. This is the only representation that relies directly on the Logo source. The rest representations are based on structures and relations generated by the Inspector.

6.2. Completely parenthesised Logo source

Right after the original Logo source, the Inspector puts its parenthesised version[5] – Figure 5. It is a reconstruction of the source, where all missing parentheses are included and all extra ones are removed. The purpose of this representation is to solve problems with ambiguity of values when some of them could be used as inputs to different actions at the same time.

```
(print (((1 + :y) * (sin :x)) - (count :y)))
```

Figure 5. Parenthesised Logo source

The other role of the representation is to show the pairs of parentheses. Each pair is coloured depending on the depth level. The number of colours is limited, so several pairs could share the same colour.

6.3. Graphical representation with braces

Braces notation is a graphical representation of Logo commands – Figure 6. Left inputs are placed on the left hand of a horizontal brace; right inputs are on the other side. The name of the action that uses these inputs is positioned below. Subexpressions are visualised as nested braces. Direct values like constants and references to variables are framed. Missing inputs are marked as red boxes with question marks inside.

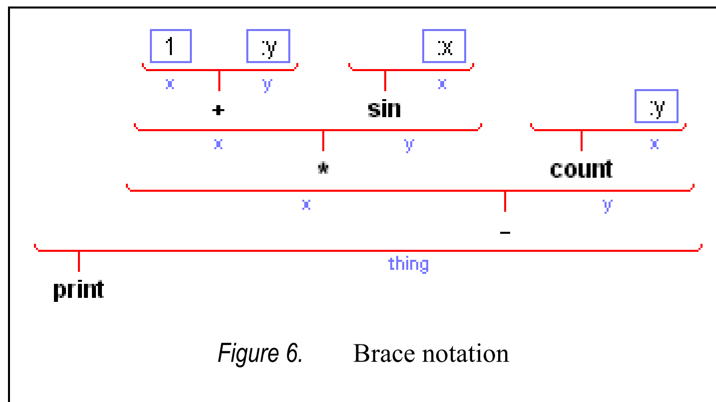

Figure 6. Brace notation

Brace notation is generally used to visualise the internal dependency between values within an expression. In many cases beginners would prefer the brace notation to the pa-

[5] We focus only on record #2 because it is the most interesting from the error history

rentheses notation. Showing expressions as a set of braces is similar to Function Machines [Feurzeig (1999)]. Users can assume that inputs are “falling down” into brace-machines, which do what they are supposed to do and drop their results downwards as inputs to lower-level brace-machines.

6.4. Box notation

Box notation represents the Logo command as a system of nested boxes. Actions are put in boxes and their inputs are boxes within these boxes – Figure 7. The placement of boxes is horizontal and follows the order defined by the Logo source. Boxes of left inputs will be to the left of action’s name. Boxes for right inputs will be on the other side. The box notation could be treated as looking at the brace notation from above so that different words are ordered the same way as in the original Logo source.

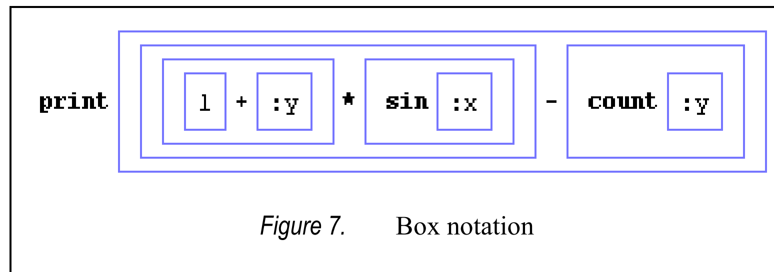

Figure 7. Box notation

Although functionally equivalent to the brace notation, the box notation is useful, because in some cases it provides clearer[6] representation. The development of box notation in Elica has been inspired by Boxer [DiSessa A (1997)])

6.5. Translating Logo to Lisp-like style

The fifth representation of the Logo source is in Lisp style. All parentheses are explicitly shown and actions are formatted with proper indentation – Figure 8.

Translating Logo into Lisp converts all operators and functions into prefix ones. All their names are extended to include the name of the library where they are defined. Showing full names helps a lot when there is confusion caused by duplicate names in two or more libraries.

Lisp version of Logo commands is not intended to be executed with any particular version of Lisp. This representation is included in order to ease those users who feel more comfortable with Lisp.

```
(print
  (logo.-
    (logo.*
        (logo.+
          1
          :y
        )
        (logo.sin
          :x
        )
    )
    (logo.count
      :y
    )
  )
)
```

Figure 8. Lisp notation

6.6. Compiling Logo into pseudo-assembler instructions

One of the most questionable representations is the one

[6] Different people would prefer different representations for the same Logo command. This is entirely subjective.

with assembler instructions. Figure 9 shows the instructions that are generated when the original Logo source is compiled.

```
PUSH 1
PUSH y
CALL :1
CALL logo.+2
PUSH x
CALL :1
CALL logo.sin1
CALL logo.*2
PUSH y
CALL :1
CALL logo.count1
CALL logo.-2
CALL print1
POP
RET
```

Figure 9. Assembler

The assembler instructions are not bound to any specific microprocessor, but are some of the most basic, so in the future Elica could be turned into a true compiler.

The small numbers at the end of CALL instructions indicate how many values from the stacked must be popped by the called action. These numbers are the same as the number of actual inputs to each action.

The benefit of viewing Logo commands in assembler is that it reveals the internal mechanism of the run-time processing. Elica Logo is compiled into internal machine code, which is later interpreted. The assembler notation dumps this internal code.

There is one interesting feature perfectly demonstrated by the assembler instructions in Figure 9. Retrieving the value of a variable using colons pops the name of the variable from the stack (see 3rd, 6th or 10th instruction). This name could be actually prepared by an expression. In other words, a colon can be used as a function, so it is legal to write `[print :(bf :x)]` in Elica.

6.7. Explaining Logo using human language

One of the most advanced features of the Inspector is to translate Logo code into English[7]. This is the first attempt to incorporate AI into Elica and there are serious plans to continue in this direction.

The translation of the command `[print (1+:y)*(sin :x)-count :y]` is:

> *"Print the difference of* ***X1*** *and the number of elements in the value of* ***y****, where* ***X1*** *is the product of* ***X2*** *and the sine of the value of* ***x****.* ***X2*** *is the sum of 1 and the value of* ***y****."*

The AI module of the Inspector is responsible for this translation. This module is written entirely in Logo. Otherwise it will be difficult to provide translations into other languages.

Logo encourages users to define their own commands. So there should be a way to teach the AI module how to explain these new commands. Let's define a function that is expected to reverse a list and teach the AI module to explain it:

---

[7] There are attempts to convert natural language of specific domains into Logo [Wong W K (1999)]. However translating Logo to natural language is equally important for beginners.

```
to reverse :lst
   if ...
end

oninspector.add
    [ to reverse :lst
         output list se [the reversed list of] :lst
      end
     ]

make "a count reverse bf [1 2 three 4 5]
```

The body of `reverse` is left incomplete on purpose, because we want to force an error in order to activate the Inspector. Note how the Inspector gets familiar with our new function. We just add its definition to the Inspector knowledge base, and instead of its body we provide a command that build an explanation phrase.

While analysing a command, the role of the Inspector is to gather all explanation phrases, group them in one or more sentences and create appropriate co-references between them (also known as discourse).

When the example with `reverse` is executed, we expect to set the value of variable `a` to the number of elements of the reversed list without its first element. The AI module explains it in this way:

> *"Make a variable with name the text **a** and set its value to the number of elements in the reversed list of the list [1 2 three 4 5] without its first element."*

An interesting thing here is that the whole command is translated into a single sentence. Why is the previous example split into two sentences? The reason is that the Inspector uses some algorithms to determine the ambiguity and readability of generated text. The previous example contains too many *and*'s and without a split it would be hard to understand. Why? The answer is in the next representation.

6.8. Using parentheses to remove ambiguity in natural language

The last representation provides the translation of Logo into one sentence where fragments of the sentence are put in parentheses to make it easier to determine internal relationships. In this notation, our example would be translated like this:

> *"Print (the difference of (the product of (the sum of 1 and (the value of **y**)) and (the sine of (the value of **x**))) and (the number of elements in (the value of **y**)))."*

Parentheses are important. Without them we get:

> *"Print the difference of the product of the sum of 1 and the value of **y** and the sine of the value of **x** and the number of elements in the value of **y**."*

Although shorter than the example with `reverse`, this explanation is really fuzzy. That's why the Inspector either cuts it or uses parentheses.

7. Future work

It is obvious that the Error Inspector is too young to properly handle all possible cases. However, the author has studied several areas in which error reporting can be improved. For example, there is a formal method to handle syntax mismatches of brackets, parentheses, `to...end`, and others, and to provide automatic (and correct!) error correction in more that 50% of the cases. When this method is improved to cover at least 80% it will be included as a part of the Inspector.

The second level of error reporting can be extended to cover user-defined errors and include user-defined hints, because current sets are predefined. On a more abstract layer, the ambition of the author is to implement a more sophisticated AI into Elica, especially into areas like translating from natural language into Logo, and providing active and supported by AI search of bugs. Also, the Error Inspector could be used to analyse Logo source not only when there is an error, but also when the user needs to understand any portion of the program.

Currently Elica environment can be translated on the fly into 8 languages. The future AI module could provide translations of Logo source and help files into other human languages. And finally, it can use Logo as an intermediate language for description of a knowledgebase used to translate one natural language into another.

8. References

Boytchev P, Gorman B, Harvey B and Tomcsanyi P (2002, Feb 7-9), *Re: [LogoForum] frustrating error msg.*, `<http://groups.yahoo.com/group/LogoForum>` (2003, Mar. 22)

Elica Team (2003), *Elica*, `<http://www.elica.net>` (2003, Mar. 22)

Feurzeig W (1999), *Visualizing Algorithms*, Proceedings of the Seventh European Logo Conference EuroLogo'99, 40-49

DiSessa A (1997), *Twenty reasons why you should use Boxer (instead of Logo)*, Proceedings of the Sixth European Logo Conference EuroLogo'97, 7-27

Doyle M (1995), *Logo and our new Instrument of Representation*, Proceedings of the Fifth European Logo Conference EuroLogo'95, 125-143

Wong W K (1999), *Chinese Logo and Its Drawing Tools for WWW*, Proceedings of the Seventh European Logo Conference EuroLogo'99, 389-394