Abstract

The paper presents a new programming environment for Logo, developed in Java, where procedures are compiled in-line directly into Java™ bytecode: this brings forward all advantages of portability, hot-spot optimisation, availability of huge libraries, integration with the Web and with other Java-based or native code systems, without losing the traditional powerful interpreter orientation. New Logo primitives, written in Java, are added in form of class bytecode, without recompiling the entire system. The environment includes an experimental 2D GUI, a graphical-to-Logo generator, and a multithreading debugger. A distributed application to control the LEGO® Mindstorms™ RCX is also shortly presented.

Keywords: JxLogo, Logo, Java, programming environment, LEGO Mindstorms

1. Introduction

For years Logo has inspired the development of programming environments which, even referring to the core idea of that language as a learning engine, followed different directions and objectives. Mainly considering the language's aspects, we could mention three main lines:

- Logo as an extensible language: this led to the enrichment and sometimes to the specialisation of its set of primitives in order to satisfy specific application requirements, e.g. in the robotics context (LEGO-Logo, Resnick & Ocko, 1991, Pogo, Qaiyumi & others, 1998) and to allow network communications (Zaveršnik & Batagelj, 1999);
- Logo as an effective graphical environment: taking advantage from the improvements of modern operating systems, new powerful graphical and multimedia user interfaces have been introduced, and consequently the language has been enhanced with new features, sometimes masking the language level with the user interface itself (e.g. Microworlds™ [1]);
- Logo as a basic programming paradigm: some new programming paradigms have been more or less directly derived from that language, as if it was necessary 'to update' its programming philosophy (e.g. many proposals to make Logo object-oriented, like Elica (Boychev 1999) and Object Logo [2]).

The introduction of Java and of its development environment have given a further opportunity to new researches, due to the possibility to face and solve two important problems at the same time: the availability of a portable Logo graphical environment, and its 'putting on the net', in the sense of being able to develop applications that can be displayed/downloaded from the Web using a browser. Actually this second aspect has been already faced in the past but it was solved in many cases requiring the installation of a specific plug-in, in contrast with the Java approach that only requests a browser integrating a JVM (Java Virtual Machine) of a suitable version.

The paper presents the motivations and some preliminary results obtained in the development of an experimental environment named JxLogo which is different from several previous experiences in Java essentially because any Logo procedure is in-line compiled and translated into Java class bytecode. This renders the executive environment uniform (actually the Java interpreter acts as the interpreter of the Logo application) and allows some other interesting possibilities that have been explored in this work. The
principles that guided the experimentation will be presented in the paper referring first to the two more investigated aspects, portability and open language; then the main components of the current prototype will be briefly described and some conclusions with possible future developments and open questions will follow.

2. Logo portability

In Logo environments the portability requirements are usually limited to some essential aspects: at the source level, substantially preserving the syntax and semantics at least of the main commands of the language in the different implementations; at the operating level, realising a (totally or partially) portable interpreter. These conditions allow user defined procedures to be executed from their sources in different environment with substantially equivalent effects. On the other hand the unavailability of an environment specifically designed to operate on various platforms, mainly due to the incompatibility of their host graphical systems, produced a proliferation of different development environments even when descending from a common core (as in the case of UCBLogo for Unix and MSWLogo under Windows).

The more recent Java system based on the WORA (Write Once Run Anywhere) assertion allowed new portability improvements in the field of Logo applications. A first level is intrinsic in Java graphical applications: even compiled they can be executed with the same effects on any platform implementing a JVM (Java Virtual Machine) of a suitable level. When the Logo development system is realised in Java the portability of the environment, interpreter included, is automatically guaranteed together with the portability of any Logo source program. This is true also in case of a dialect as far as a compatible interpreter is concerned. Moreover in implementing its basic features great advantage can be taken from the availability of a huge function library related to arithmetic processing, 2D-3D graphical resources, multimedia animation, multiprogramming.

A second intrinsic level, with inter-platform characteristics, is the possibility to implement the environment in a distributed fashion, with its user graphical interface inside the Web browser window and the executive Logo engine running on the server. Even with some limitations, caused by the security mechanism imposed by Java, a sufficient degree of interactivity is allowed, including the definition and testing of new user procedures.

A third level has been explored by the present work: from the simple observation that the JVM is a complete and efficient interpreter, the hypothesis to translate the Logo user procedure directly into Java bytecode has been experimented. The positive consequences of this choice are (see Figure 1):

- a compiled procedure becomes a portable executable unit, expressed in an intermediate language (bytecode);
- implementing all the Logo primitives, and translating all the Logo user and library procedures, in Java classes (and methods), primitives and procedure become really homogeneous at the bytecode level;
- an additional level of interpretation is avoided, with benefits in efficiency made more effective by just-in-time compilation of the bytecode done by the JVM, reaching performances comparable with code compiled in machine language;
- the object oriented development allows the implementation of a dynamic incremental environment where new primitives and procedures can be added only adding bytecode visible classes to the basic ones: no overall knowledge or source code is strictly required;
you need only the JRE (Java Runtime Environment) in order to compile and test new Logo procedures; the entire Java SDK (Source Development Kit) is necessary only when new primitives written in Java must be added to the system.

Figure 1: JxLogo architecture

The execution from the command line of a single primitive/procedure or of a list of primitives and procedures is preceded by the compilation of the given command in a temporary class, that is thereafter normally executed. This partly replies to the usual objection against compiled Logo, related to commands like `RUN :list`, whose exact meaning is known only at run-time. An effective form of interpretation of procedures is anyhow available as an alternative to compilation: under the control of configuration parameters, the structured memory representation of the source code of the scanned procedures is used as a form of internal caching to improve interpretation. The speed of modern computers, and the manipulation of procedures and lists as separate compilation units, make the choice in favour of compilation comparable in performance, and better in perspective, with respect to traditional interpretation.

3. Logo as open language

As known, Logo stands on an intermediate level between imperative and functional programming language. Being procedure- and recursion-oriented and including relatively simple structures, Logo is an interesting experimental platform suitable for investigating extensions and specialisations. These were actually developed in the past with a lot of examples in different areas. Even in the usual interpretative form, considering the set of primitives and library procedures as the run-time support for applications, the simple adding of the source files of new library and user procedures enriches the capabilities of the system.

In JxLogo this aspect has been further improved thanks to the uniformity allowed by Java bytecode. In fact a JVM executes any class after its loading in memory from a secondary memory or from the network. Such a loading is usually done the first time some code inside the class is to be executed. The rules that steer the searching and loading of classes are implemented by one or more 'class loaders' which are redefinable components of the virtual machine. In JxLogo the same symbolic name used in Logo is assigned to a procedure source file and to a bytecode file (both for primitives and procedures); the name of the derived Java class is also obtained as a transliteration of the Logo symbol. On the basis of these simple rules,
specifically defined loader provides to search the requested class and decides to load it if it is not yet present in memory or, in case of a procedure, when the corresponding bytecode file is obsolete with respect to its Logo source file. In the latter case the compiler is activated before loading/reloading the class in order to update the bytecode. The search is done first in the current work directory, then in one or more library directories, following a logical separation between run-time support and application.

The operating mode described above allows interesting possibilities. First of all updates of application user files are possible within a work session; their effects can be immediately tested executing the involved program/procedure from the command line. Short delays are introduced only by the *una-tantum* compilation and loading of single updated classes.

A second implemented feature is a complete freedom in assigning names to primitives and procedures: any dependence from static tables or other limits of this type have been carefully avoided. This fact, together with a potentially unlimited number of definable primitives and procedures, allows you to enrich the run-time support simply adding new bytecode files in the foreseen directories, obtaining a remarkable flexibility in the definition of specific dialects.

A third possibility, derived from the previous one, and actually explored, is the introduction of a sort of simplified mechanism of overriding for primitives: they can be partitioned in subsets called *alternatives*, identified by an (incremental) integer, and put in separate directories which represent, in the Java viewpoint, different packages. Basic primitives are put in the root directory/package (alternative 0), whereas different alternatives may contain homonymous primitives whose interface is equivalent but which are alternative in their bytecode implementation. On the basis of configuration parameters and of the current alternative level (the level is dynamically settable by means of a specific basic primitive), the class loading mechanism decides whether limiting its search to the root package or to the current alternative, or starting it from the current or the maximum predefined alternative downstream. In the latter case the first version of the requested primitive found in such a searching path is actually loaded and executed at the current level. At any level it is known which previously loaded alternative primitives are visible and active. It is for example possible to define the basic *FORWARD* primitive acting on the graphical turtle, and a second robotic alternative which controls the motion of an external physical object.

The inheritance mechanism of Java has been easily used to realise another extension, usable in conjunction with the alternative feature: the definition of alias names for primitives and procedures. The *ALIAS* primitive provides to compile an empty body class with the new alias name, which inherits from the class with the corresponding old name. This feature allows the definition of abbreviated aliases (like *FD* for *FORWARD*) and an easier international localisation of the environment.

The implementation of primitives in Java does not prevent from activating native code from within the Logo application thanks to the standard JNI (*Java Native Interface*). In particular this allows the dialogue between the application and special hardware or software with real-time constraints. Another aspect being explored, which can similarly take advantages from the Java architecture, is the definition of object-oriented extensions of the language, as suggested by previous experiences.

### 4. Developed tools

#### 4.1. The compiler and the run-time environment

The Logo run-time environment (Figure 1) requires the permanent availability of the compiler and of all the supporting library functions. When the compiler is activated, the source code is analysed by a parser which is responsible for checking the correctness of the code and for building a parse tree in memory. A visit of a tree either generates the corresponding bytecode, if compilation is active, or produces the interpreted actions, otherwise. The parser has been implemented using well known automated tools (JavaCC and JJTree [3]). The generation of the class bytecode is aided by the use of the free JAS tool which provides, in addition to a bytecode symbolic assembler, a library of classes which makes it possible to generate bytecode through simplified method interfaces.
The procedure-class implements a predefined Java interface and therefore it exhibits a known, relatively simple structure: every time the procedure is activated, an object of that class is created and represents part of the state of such an activation. After the input parameters have been checked, the specific method to include the procedure body's code is activated. The dynamic management of variables exploits the object oriented approach: every Logo variable is represented by an object maintaining all information necessary to manage the variable during its lifetime (remember that variables are non strong typed in Logo). Variables are grouped in contexts (frames) subject to a LIFO policy which corresponds to the nesting of activations of procedures. The 0-level context is the global one and it is visible at any time, apart from the normal masking of preceding variable instances when a new instance of the same variable is created.

In the current prototype the most important primitives that you can find in MSWLogo (our 'model') have been implemented. A set of further primitives, called 'extrinsic' (Loethe, 1992) (POST, DISTANCE, DIRECTION, etc.) in contrast with the 'intrinsic' primitives that refer to an absolute cartesian coordinate system, has been implemented as well. The JxLogo environment integrates the usual 2D graphical turtle action plane (Figure 2) and the input command line with other tools by which interesting pilot activities have been performed: a window-based debugger, a graphical editor able to generate Logo code, a remote control system evaluated on a specialised robotic system based on LEGO® MINDSTORMSTM. A brief description of these components follows.

![Figure 2: JxLogo user interface](image)

### 4.2. The Debugger

When the development of this component started, we thought to fully adopt JPDA (Java Platform Debugger Architecture) [4]. This recent proposal from Sun Microsystems®, still in progress, states that the debugger should be remotely executed (either on a different physical machine or on a different JVM on the same machine) with respect to the JVM executing the debugged application; this latter virtual machine must implement a special native interface (JVMDI - Java VM Debug Interface) and communicates with the
debugger using a native protocol (JDWP - *Java Debug Wire Protocol*), through a couple of interface components (BackEnd for the first JVM, FrontEnd for the debugger). The implementation of debug tools is simplified by the definition of the JDI (*Java Debug Interface*) which is implemented by the FrontEnd. Our aim of debugging Logo procedures compiled in bytecode, possibly enriched by some accessory debugging information, was compatible with the hypothesis to adopt the JPDA approach. Actually this choice has been temporarily suspended in favour of an *ad hoc* but portable solution. Such a solution required limited modifications on the basic code of JxLogo and preserved the JPDA requirements, like those regarding the structure of debugging information inside the class bytecode (e.g. line numbers), and the separation between debugger and application: these design principles make it possible to use any compatible debugger tool for Java (like JDB included in the SDK).

![Diagram](image)

**Figure 3:** JxLogo-Debugger communications

The architecture of the developed debugger foresees a BackEnd, a FrontEnd and a debugger thread; the communication between the two interface threads takes place on shared memory (a common *CommunicationHandler* object) (Figure 3), and is synchronized by semaphores. The communication is bi-directional: both executor and debugger may solicit its counterpart due to asynchronous events, e.g. turtle state changes and reached breakpoints on the one hand, user interactions on the other. Most of the implementation basic structures and functions were already available from the Java language or its library.

![Diagram](image)

**Figure 4:** Debugger tools

The architecture of the developed debugger foresees a BackEnd, a FrontEnd and a debugger thread; the communication between the two interface threads takes place on shared memory (a common *CommunicationHandler* object) (Figure 3), and is synchronized by semaphores. The communication is bi-directional: both executor and debugger may solicit its counterpart due to asynchronous events, e.g. turtle state changes and reached breakpoints on the one hand, user interactions on the other. Most of the implementation basic structures and functions were already available from the Java language or its library.
The debugging functions realised are as usual: breakpoint/resume, step-in (execution of a single primitive or, in case of a procedure, stop just after entering that called procedure), step-over (differs from step-in in stopping after the execution of an entire procedure call or list of commands), step-out (execution until the end of the current activation), view of the turtle state, view/modify of the value of a variable.

The debugger includes a simple text editor, implemented following the usual Model-View-Control paradigm which separates respectively data representation, user interaction and viewing modes. The editor performs the create/modify/save functions for source files, and its user interface is integrated with the debugger functions, including the compile and execute commands (Figure 4).

4.3. The graphical editor

This is a separate Java application, equipped with some common drawing tools: its aim was to verify the integration in JxLogo of a mouse-based graphical user interface able to build geometric scenarios, as an alternative to the traditional Logo constructive approach. Instead of expressing actions that build the scene step by step, using the mouse and the editor interface it is possible to draw and modify complex objects directly on the drawing plane although on the basis of limited simple items like segments, arcs and elementary figures. Drawn items are graphical objects (not bitmaps) that can be grouped into subsets the user can singularly name: for example, it is possible to assign the name house to the selected cluster of objects of Figure 5. Any object or cluster may be successively modified, resized, moved or rotated.

![Figure 5: The graphical editor](image)

When the scene is complete, you can activate a post-processor generating Logo code: every cluster generates a procedure, with the same name assigned to the cluster, containing some useful commands and one call to the basic geometric primitive/procedure for each elementary figure in the cluster. In fact any figure is associated with a predefined primitive/procedure whose parameters are general enough (position, size, angle) to simplify geometric transformations like roto-translations and scaling. The generated procedure, compiled
and executed in JxLogo context, is able to reconstruct the scene. In this sense, the Logo language is used as a saving graphical representative language. Unluckily the actual prototype of the editor does not provide the upload of a previously drawn and saved scene.

### 4.4. The robotic control system

The described mechanism for defining alternatives and aliases is used to implement an extension that makes the environment a remote control system. The chosen controlled system is LEGO MINDSTORMS (Baum, 1999) [6], the so called 'robotic invention system'. Its heart is an actuator component named RCX, a little microcontrolled system with some communication ports on board used to drive little motors, and to connect simple sensors and an infrared interface through which the RCX can communicate with a host PC. The kit development system includes a graphical programming environment base on a block diagram programming language. After having developed the robotic application on the PC and having preliminarly downloaded a (volatile) firmware via IR, the program is translated into a proprietary bytecode and downloaded to the RCX: the loaded firmware includes its interpreter which drives actuators and gets system state information through the services provided by the non-volatile firmware. The system state can be sent back to the host if requested by the host itself.

The architecture of our control system includes the following components:
- a set of primitives representing elementary services provided by the robotic controlled system (single motor moving, power control, moving duration) and a set of alternative to normal turtle primitives which drive the robot instead of the graphical turtle;
- an RMI (Remote Method Invocation) server which exports services to the JxLogo client and communicates with it through such a standard interface of Java;
- on the RCX an alternative firmware is loaded (LeJOS [5]) which is the implementation of a tiny JVM able to execute some classes acting as interfaces towards the basic services of the resident firmware; moreover a small interpreter for elementary robotic commands, written as a Java application, is also loaded.

From JxLogo, calling a specific primitive, it is now possible to connect to the remote server which causes the export of a set of interface objects from the server to the client. During the entire connection these objects will be the driving interfaces for the robotic commands. The server is in charge of calculating trajectories and controlling durations and synchronisations. Any elementary command is sent by the server to the RCX through the IR connection. The possible solicited replies from the RCX, when supplied with data, are sent back to the application through the same path. It is worth noting that the dependancy from the RCX architecture is confined within the server, thus the implementation of robotic primitives in JxLogo is in principle independent of the characteristics of the used robot. Moreover the distributed architecture makes it possible to drive the same RCX from many different clients, possibly on different platforms, and distributes the processing load, dedicating one machine to real-time aspects.

### 5. Conclusions and future works

The present experimental phase in using JxLogo has confirmed the many advantages in the core idea of implementing the run-time environment for Logo on the JVM. With respect to other experiences that use a compilation to a native language, the relative inefficiency of our solution is within acceptable operating limits and can be further reduced introducing some form of optimisation at the compile level. The experimentation with the robotic control system is going on and will give the opportunity to improve the flexibility of the environment (e.g. alternatives not depending from indexes) and to introduce multiprogramming in the language. The integration with the Web and the multimedia support are also currently under development. As soon as these developments will produce a stable and enough complete version, the environment will be freely distributed to the Logo community for a large testing and evaluation. Whereas the current experimentation has been limited to our undergraduate students, we hope that teachers at different levels will appreciate its flexibility using it in activities with their children or students. It is an open question whether to extend the language to object-oriented and component-oriented (Koutlis &
Oikonomou, 1997) approaches: it is the opinion of the author that many ideas discussed in literature may lead to proposals of semantics that can be rather easily mapped onto the Java model.

References


Web links