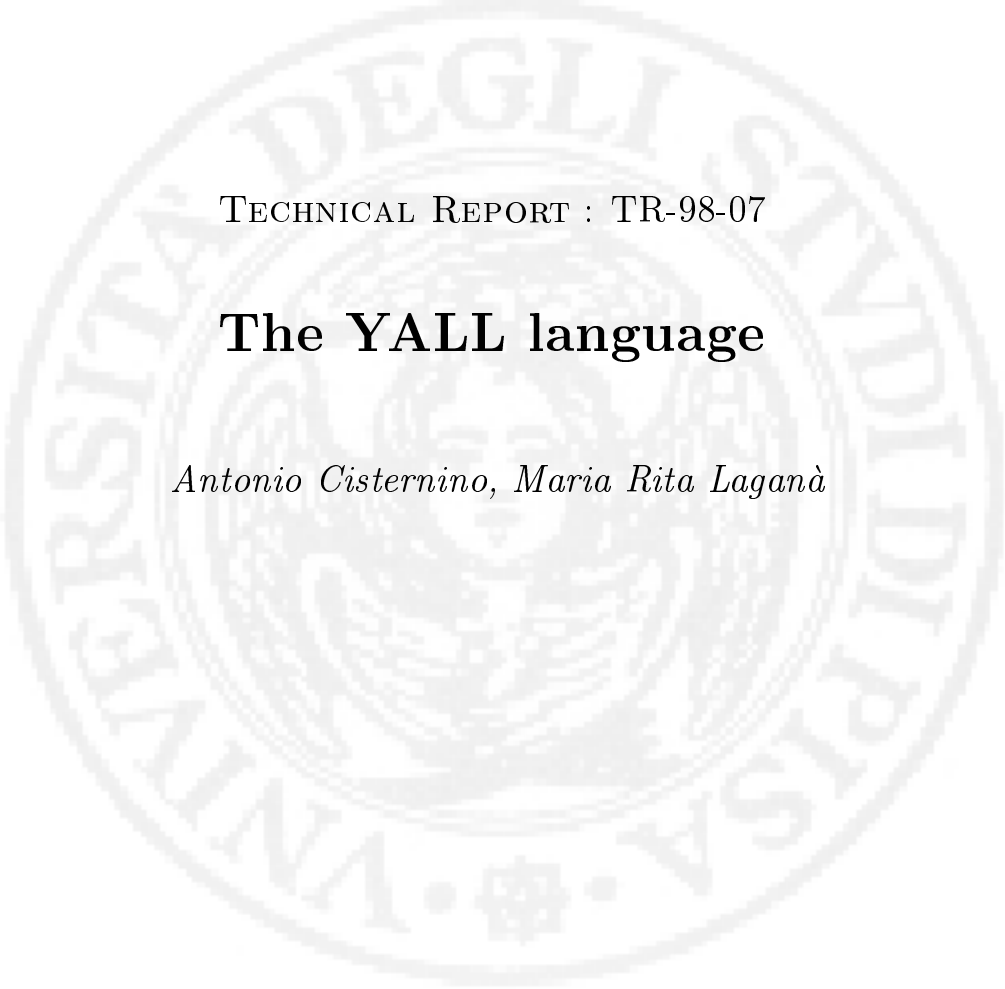
Università di Pisa

Dipartimento di Informatica



Technical Report : TR-98-07

The YALL language

Antonio Cisternino, Maria Rita Laganà

July 02, 1998

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

The YALL language

Antonio Cisternino
cisterni@di.unipi.it

Maria Rita Laganà
lagana@di.unipi.it

Dipartimento di Informatica
Università di Pisa
Corso Italia, 40
I-56125, Pisa
Italy

Phone: +39 050 887261
Fax: +39 050 887226

Abstract: The authors of Logo wanted to create a tool to help children build and explore the world of programming as means of building and exploring their own logical thought. The strength of this idea is particular and the turtle has carried his work out well. Today, however, new programming paradigms call for new metaphors and the technological gap between the turtle and modern graphic and multimedia programs seems to make less powerful the original Logo metaphor. In this paper we propose to enhance standard Logo by introducing modern program paradigms. In particular we extend Logo with object-oriented programming and multi-agent programming features. This new language is called YALL and try to change the way of programming Logo without abandon the original philosophy of the language. We introduce a new approach to express the inheritance relation that is more suitable for children. We discuss about these new extensions to Logo describing the new commands introduced. We also discuss a possible implementation of YALL using the Java language by Sun Microsystems.

Keywords: *YALL*, *Logo*, *Object-Oriented Programming*, *Multi-Agent System*

1. Introduction

The authors of Logo [Papert 1971] wanted to create a tool to help children build and explore world of programming as means of building and exploring their own logical thought. The strength of this idea is particular and the turtle has carried his work out well. Today, however, new programming paradigms call for new metaphors and the technological gap between the turtle and modern graphic and multimedia programs seems to make less powerful the original Logo metaphor. To address this problem various researchers have suggested different, good solutions. We can take, for example, Kidsim [Smith 1994], Multilogo [MultiLogo 1990], StarLogo [StarLogo 1996] and MicroWorlds [MicroWorlds] to quote a few.

We propose to enhance standard Logo by introducing modern program paradigms. In particular we extend Logo with object-oriented programming and multi-agent programming features. We also introduce multimedia features that make it much more interesting for children. Our solution differs from the others because we try to change the language structure rather than patch the language by adding new constructs that adds new functionalities to the language.

The main idea behind our work is to make it possible to create different turtles and to put them in a world. A world defines a turtle and thus the child can define a hierarchy of turtles where one or more turtles build and draw another turtle. This structure is inherently recursive and defines a hierarchy of turtles and worlds; each turtle is a world for the turtles that have drawn it.

This is a very powerful idea because gets the Logo into line with current programming trends allowing the student to learn complex paradigms in a very simple way.

We describe more in detail the features of YALL (Yet Another Logo Language) in section 2. Section 3 is devoted to the language extensions needed to the standard Logo syntax to support new paradigms. Section 4 describes a possible implementation in Java [Gosling 1996] of YALL and finally section 5 is devoted to conclusions.

2. YALL

YALL is an enhanced version of the Logo language. We assume that the basic features of the Logo are familiar to the reader and we focus on the description of the enhancements that we propose.

Two are the main differences between YALL and standard Logo: There are many turtles that can communicate and there are multiple worlds where turtles can interact. Each turtle offers his back as a possible world for other turtles. For example the user can draw the shape of a new turtle using the turtles that already exist. In this way we define a hierarchy of worlds and in each one there are one or more turtles. Each world is indeed a multi agent environment where the agents are the turtles.

Thus YALL simply extends Logo with some key aspects of modern programming: Concurrent programming in a multi-agent environment, and object-oriented design. There are many ways to implements those aspects in a programming language, but the real difficult task is to find the right constructs of the language that are suitable for a child.

Logo's turtle metaphor was thought as a way of teaching mathematical [Papert2 1971] and programming concepts by playing the game of draw shapes with the computer. Today a picture

can be made in a very simple way using the modern graphics interfaces and bitmap editors that come with all operating systems. This situation compromises the game of draw something with Logo because the use of a programming language is more complex than the use of a point and drag interface. Various extensions proposed to the language try to address this problem by introducing new language constructs or more turtles but none of these introduces the possibility of using the shape of a turtle to create new turtle shapes.

To describe more in detail our extensions we propose an example that shows the main features of YALL. Suppose that we want to build a world where there are many cars moving. We have two main problems: Drawing a complex world and animating it.

Let us now to consider the first problem, first of all we can draw the first shape: The driver of the car. To do this we start using the classic triangular turtle to draw it. When we have created the first shape we close the editing session and the driver becomes a possible shape for a turtle. To avoid confusion and to favourite simplicity we don't require any command to say which code is really necessary to draw the new turtle. Thus every variable created with **make** becomes an instance variable of the turtle and each procedure defined becomes a method of the turtle. This way of proceed implies that is possible that we store in the turtle definition garbage instructions, also if we try to individuate which instructions can be eliminated. When we have drawn the driver of the car we can terminate our editing session and the driver is added to the turtle's palette.

Now we want to draw the car using as a turtle the driver already designed. We open a new canvas and we select the driver turtle from the turtle's palette. This operation is important because, as we will seen later, we substantially create an instance of the class of the turtles that are driver of some car. To define the car we proceed in the same way as we have done for the driver. We can store the various colour used to draw in instance variables of the turtle car. When the editing session is terminated the car is added to the turtle's palette.

Lastly we want to draw a world where many cars may move in it. This can be achieved by using a new canvas and the cars previously defined as a turtle. In this case we add several turtles to the canvas to add various cars to the world. Thus we open a new canvas and we select from the turtle's palette the car previously defined several times. Remark that when we make an instance of a car we implicitly make an instance of the driver of the car. When a turtle is added to a canvas the system prompt to the user asking the name of the turtle that is begin added to the canvas. This name is used to select the active turtle and for communication purposes. The first problem that arises is how subdivide the task of drawing the world among the various turtles created. We can use only one turtle or use all turtles by operating with the synchronisation mechanisms provided by the communication primitives. Other problems are related to the animation of the various cars.

Let us now consider the problem of animating this world and analyse which problems arise in this task. Suppose that we want to animate the driver of the car and the cars in the world. The animation is very complex because it has to compose two different animations. Using YALL the problem can simply solved by animating the driver in the car and the cars in the world. To animate the driver we define the procedure called *animate* during the definition of the driver. The *animate* procedure is a particular procedure because can refer the turtle defined by the world in which is defined. In this procedure we can also change the world as another procedure. The *animate* procedure is needed to individuate the part of code that must be executed during the animation phase rather than the drawing phase. In this procedure we can define an infinite loop that animates the driver by repeatedly moves of the turtle.

To animate the cars in the world we must define the *animate* procedure during the definition of the car. The correct animation of the cars in the final world requires some sort of synchronisation mechanism to resolve conflicts between moving cars. Thus YALL provides some language constructs to allow communication among turtles that are used in the same canvas. Via communication turtles can synchronise and exchange information among them.

We plan to introduce some sort of perception of the turtle to allow a child to program a turtle as an agent that interacts with the world where he lives.

This simple example shows the main features of YALL: The use of turtles and canvas supplies recursion, object-oriented programming and concurrent programming. We also think that the use of our language can be a good motivation to play with this new Logo.

The recursive use of canvases, which can be also used as new turtles, allows a child to experiment and understand problem decomposition techniques. This is because children may decompose complex world into objects focusing their attention to subsets of the world that are much simpler. When the various parts of the world are ready children may merge it building the desired world. For example if we want to draw an house we can defines the windows and the door as separate objects using separate canvas and then we create a new canvas where we draw the house and we place the windows and the door simply by moving turtles.

The possibility of making multiple instances starting from a previously defined canvas introduces the concept of class and the natural canvas composition shows the use of the *part-of* relation. In the previous example of the house a child may deduce that the windows and the door are parts of the entire house.

We also plan to introduce the inheritance relations among classes to allow children use a complete object-oriented paradigm. We have in mind of build the inheritance graph starting from the leaves toward to the roots because the process of abstraction is more intuitive rather than the opposite. Usually program languages that support object-oriented paradigm build the hierarchy from the more abstract concepts to the more specific. In the real world humans proceed in the opposite direction: starting from the experience they build successive layer of abstraction. The main reason of the radical difference from the top down approach used by programming languages and the bottom up approach used by humans is due to the difficulty that systems have to individuate similarities. If two classes share a method and we proceed top down it is easy to define the method in a base class and derive the two classes from this one. If we proceed bottom up we must individuate the two methods that are equivalent and we must generate the base class using one of the two methods. The problem of demonstrate that two programs (in this case two methods) are equivalent is in general not decidable, thus programming languages delegate to the programmer this responsibility. We want introduce in YALL a system that unifies many classes, if it is possible, building base classes as a result of the unification. We are working in this direction but it is not trivial to implement such algorithm.

Communication mechanisms are provided to allow the various turtles to exchange information and for synchronisation purposes. The communication model is described in the next section.

When we add a previously defined turtle to a canvas we introduce the concept of local co-ordinate system. Thus the extensions introduced by YALL introduces the fundamental mathematical concept of multiple co-ordinate systems in the world of Logo. When a child draw the car uses the co-ordinate system defined by the canvas of the car. The co-ordinate systems of the world where cars are placed have its own co-ordinate system but the commands used to draw

the car are referred to the original co-ordinate system. The possibility of define nested co-ordinate systems is typically used in 3D graphics. For example the VRML language [VRML97] uses the *Transform* node to define a local system of co-ordinates in the 3D space.

3. Language extensions

Proposing extensions to Logo is difficult because one of the key aspects of this language is simplicity. The main constraints are that the extensions can be learned incrementally and that new language constructs must be simple to understand and to use. The constructs introduced by YALL are suitable for children because it is much simpler to simulate the world with them. By using the definition of various objects a child can defines the various aspects of the world that he want to define. Through communication can define the interactions among the various active entities in the world.

In this section we illustrate the main ideas behind the language key constructs of YALL. The first command that we want to introduce is **AssumeShape** that allows the user to modify the shape of the basic turtle to use images as possible turtles. This extension is provided to make it more interesting playing with YALL and to allow beginners to obtain good results with a small subset of the language. The opportunity of changing the shape of a turtle in this way can be useful to enrich a world of details. In the example of the world of cars we can define the driver as a turtle with the image of the driver using the **AssumeShape** primitive rather than draw using turtle primitives.

This command is also interesting because allow the user to animate the shape of a turtle using the usual frames based technique. It is possible to write an animated turtle as:

```
make "i 0
while [ TRUE ] [ make "i :i + 1 AssumeShape :i wait 30]
```

After the initialisation of the variable **i** the program begin to execute the **while** cycle. The body of the **while** command change the image of the turtle every 30 milliseconds. We suppose for simplicity that there is an infinite number of frames and that **AssumeShape *n*** selects the *n*th frame.

The YALL language supports concurrence because allow the user to add many turtles in a canvas. Thus communication mechanisms are required for synchronisation and communication purposes. Many models of communication are possible and Logo has been extended to support concurrence and communication in many ways. StarLogo [StarLogo 96] is an example of this kind of systems, another example is the Logo PL described in [CaLaRi 98].

To support communication among turtle instances that reside in the same canvas we provide four primitives: **WaitForMessage**, **HearMessage**, **TellMessage** and **PostMessage**. The first two commands are supplied to receive messages and the others must be used to send messages to other turtles. These primitives account for both synchronous and asynchronous communication. All primitives are both symmetric and asymmetric [Hoare 1978]. The user can wait until a message arrives and then listen to the message, or try to listen to a message without blocking. Similarly we provide two primitives for sending messages in a synchronous or in an asynchronous way. The real problem is how we manage destinations? We provide a very simple

management to keep the language simple: Each turtle has a name that can be referred to by communication primitives. We also provide broadcast mechanisms to communicate with all the turtles that share the same canvas.

Suppose that in a world there are two instances of turtles named Alice and Bob. If Alice want to send a message to Bob without blocking must simply include in his program:

```
…
PostMessage "Bob "Hello
…
```

If Bob want to wait until arrive a message from Alice he can use the **WaitForMessage** primitive joined with the **HearMessage** primitive:

```
…
WaitForMessage "Alice
make "msg HearMessage "Alice
…
```

We think that is much more intuitive for a child to separate the two activities of waiting for and receiving a message. If Alice want to say something to Bob and waiting until Bob has received the message can use the primitive **TellMessage**:

```
…
TellMessage "Alice "Bye
…
```

Finally Bob can receive a message from Alice if available using the primitive **HearMessage** without using the **WaitForMessage**:

```
…
HearMessage "Alice
…
```

The communication system uses the name **everybody** to identify all available turtles in the canvas. Thus if this name is used in the receive primitives the turtle receives the first available message in the message queue. If **everybody** is specified in sending primitives it is possible to sends broadcast messages.

This rich set of communication primitives is simple to use but allow a child to experiments sophisticated interaction in concurrent programming. The communication primitives of YALL mimics the human communication acts and thus can be very intuitive to understand.

YALL also defines commands to control the turtle's instantiation and all other aspects that usually are controlled via graphics interface. This is an important aspect of the language because a child can starts using the graphics interface and gradually learns the various command of the language. In particular the instantiation of a turtle is an important operation because the user can

specify the instance variables of the new created turtles. In the example of the cars moving in a world when the child creates the moving car can specify the colour of each car that is an attribute for that object.

We use the classic dot notation to access all the variables and procedures (methods) defined in a turtle used in a canvas. All the objects defined during the editing phase of a turtle are associated with the turtle. No mechanisms are provided to declare which procedures and variables are included in the class. This choice implies that usually there is garbage associated with turtle, on the other hand the child can define a class without knowing about object-oriented programming.

As we have seen the turtle has two responsibilities in a canvas: she can draw the shape on the canvas and she can move in the world defined. The code that animates the turtle must be defined in a procedure called **Animate**.

Other commands may be provided with implementations of YALL but those defined above are required for any implementation. The extensions to Logo proposed in this paper are few in number, to keep the language much simple as possible, but are very powerful and change radically the use of Logo without abandoning the original philosophy of the language.

4. A possible implementation using Java

Java is an ideal language to implement a Logo that uses our extensions. The standard support to the graphics interfaces makes it simple to implement the basic turtle. The `Tokenizer` class makes it simple to write the language parser. The support to concurrence mechanisms allows to simply defining multiple turtles and communication by creating an execution thread for each turtle.

Even if Java seems to be an ideal language to implement YALL, there are two main problems to be solved: Making the system efficient, and implementing the hierarchical structure of the animation.

The use of standard 2D techniques like double buffering can solve the problem of efficiency and help to eliminate flickering during animation. The new release of Java [Javasoft] also supports new functionalities like image rotation that is useful to define the turtle as an image.

To animate the world we must compose the animations present on the various canvases. We split the job of drawing the world in two steps: first we draw on the canvases using the drawing shape section in the code associated to the turtles, second we animate the world using the animation code of the various turtles. To correctly handle with the various co-ordinate systems defined by the nested canvas we must draw the various canvases from the one that contains all turtles to those that not contains turtles following the hierarchical structure of the turtles from the leaves to the roots.

The job of translating the turtles can be simplified in Java using the lightweight components defined in the version 1.1 of the language. These components allow defining "logical" components without allocate a window like the normal components. The use of lightweight components can simplify the translation of co-ordinate systems defined by the various nested canvases.

5. Conclusions and future works

In this paper we have described the YALL language which is a superset of the classic Logo language. The main features that introduce YALL in the world of Logo are elements of object-oriented programming and multi-agent programming.

The support of object-oriented programming paradigm is due to the possibility of instantiating many turtles from a single defined shape. It is also possible to build *part-of* hierarchies by using previous defined turtles in defining new ones. For the moment we don't provide a mechanism to specify the inheritance relation among turtles.

The existence of multiples turtles allow the user to experiment concurrent programming by using agents (the turtles) that interact by using communication primitives that also supplies synchronisation.

The recursive mechanism provided by YALL to specify the various turtles and the final world provide a simple way to specify composition of animations. It is simple to describe a world as a set of objects rather than a complex set of lines and movements. Thus complex world are naturally decomposed using YALL.

Java by Sun Microsystems is an ideal language to implement YALL because is a concurrent, object-oriented language with standard support for graphics. Thus the implementation of the message system can be simply reflected in Java using the standard synchronisation mechanisms rather than starting from scratch.

We are working to an implementation of YALL in Java. The system will be available from the Web as soon as possible. We also plan to start many experiments with children to verify if the language is so intuitive as may appear. Finally we are working to the unification algorithm needed to express inheritance relation in the language.

6. References

[Papert 1971] Papert, S. (1971). *A Computer Laboratory for Elementary Schools*. Publication no. 246 of MIT AI Laboratory.

[Papert2 1971] Papert, S. (1971). *Teaching Children to be Mathematicians vs. Teaching About Mathematics*. Publication no. 249 of MIT AI Laboratory.

[Smith 1994] Smith, D.C., Cypher, A. & Spohrer, J. (1994). *KidSim: Programming Agents Without a Programming Language*. Communications of the ACM, 37(7), July 1994, pp. 54 - 67.

[MultiLogo 1990] Resnick, M. (1990). *MultiLogo: A Study of Children and Concurrent Programming*. Interactive Learning Environments, vol. 1, no. 3.

[StarLogo 1996] Epistemology and Learning Group (1996). *StarLogo reference Manual*. MIT Media Laboratory.

[MicroWorlds] http://www.kidscanprogram.com/main/tofc.cfm

[Gosling 1996] Gosling, A.J. (1996). *The Java Programming Language*. Addison Wesley Publishing Company, Reading, Ma.

[CaLaRi 98] Capretti, G., Laganà, M.R., Ricci, L. (1998). *Microworld to learn concurrency*. To be appear in the proceedings of the International Conference on Systems, Signals, Control, Computers (SSCC'98).

[Hoare 1978] Hoare, C.A.R. (1978). *Communicating Sequential Process*. Comm. Of the ACM, Vol. 21, No. 8, pp. 666-677.
[VRML97] VRML 97, ISO/IEC 14772-1 standard. http://www.vrml.org
[Javasoft] http://www.javasoft.com