Jamie Samdal
Ricardo Carlos
5/6/2008
CSc 520 Final Report

Logo

History

The first version of Logo was created in 1967 by Seymour Papert, an MIT Professor, and Wallace Feurzeig, a researcher at Bolt, Baranek, and Newman Technologies. Logo was designed to be an educational programming language, and is particularly rooted in the idea of the constructivist educational philosophy. This philosophy essentially boils down to the idea that learners construct knowledge, and learners learn in a self-initiated manner. As such, creating the Logo programming language was not the goal, but rather facilitating learning activities, including mathematics, language, music, robotics, and science.

Logo is best known for its computers graphics support known as “turtle graphics.” The term “turtle” was first used to describe the mechanical robot first developed at MIT in 1969, which had a shell that was used to cover and protect its mechanical and electrical components. The turtle had a pen on its underside (or “belly”) that was used to draw lines, and was directed to move by computer instructions. Since then, the turtle has moved to the computer screen as a cursor, where Logo (and other programming languages) allows users to draw a variety of lines and shapes through a series of commands.

Logo was designed to be easy for a novice programmer, and provides an interactive programming environment so that they do not become discouraged easily. Users are able to provide simple instructions to the interpreter without having to know about saving files, compiling and running programs.

A prime example of Logo’s ease of use is the creation of the Lego/Logo kit. Through a partnership with Lego, Lego/Logo kits were sold around the world to allow children to learn about constructing devices and “robots” using Lego bricks, and programming them to perform specific tasks using Logo programs. Eventually, Lego decided to move in a different direction through a partnership with LabVIEW, which is used in the Lego Mindstorm products available today.

Language Details

Logo was first designed to manipulate language constructs such as words and sentences. In fact, Logo's name came from the Greek word for *word.* But Logo is best known as the educational programming language that introduced turtle graphics. Its main features are interactivity, modularity, extensibility, and flexibility.

Interactivity

Logo is a dialect of Lisp, and is primarily an interpreter since it directly executes instructions from user programs without translating it to a different language. Some Logo versions are developed as compilers to allow users to have an integrated development environment to create, compile, and debug programs.

Logo provides an interactive environment where users create and debug programs "on the fly" as

opposed to compilers which require a program, and the procedures it invokes, to be complete before attempting to compile and execute it. Logo users also have the option of creating procedures and saving them in source code files to be loaded within the Logo interpreter.

Developers of Logo wanted the user to utilize their experience with getting the computer to help humans learn to think more efficiently. A user can create programs, run them, and immediately use the results to think of new ideas in solving a problem. The UCB Logo interpreter initially looks like a text command prompt, but after executing a graphical command, such as `cs` which clears the screen, the top portion of the interpreter turns into a graphical screen on which turtle graphics can be displayed, as shown in Figure 1.

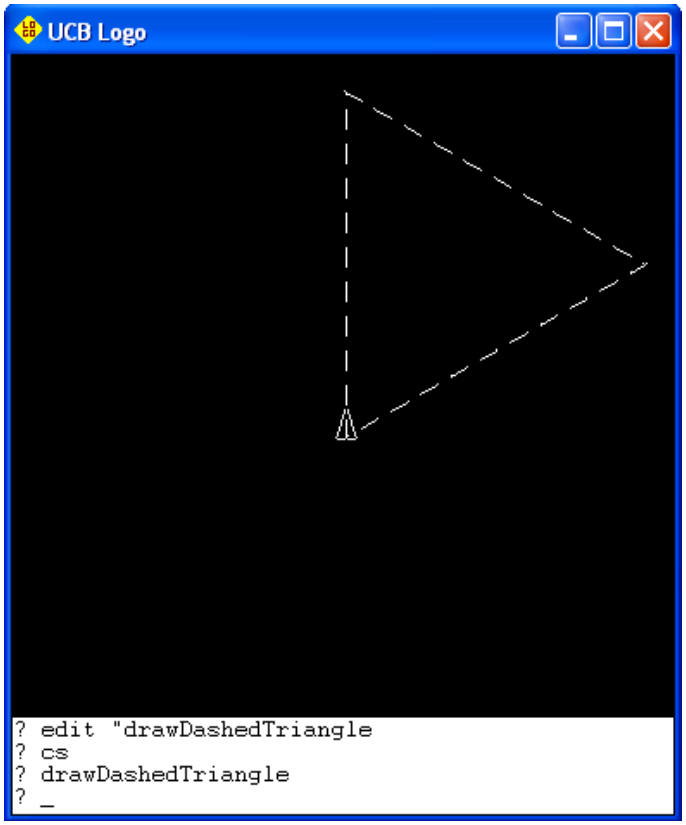

Figure 1 - UCB Logo interpreter, displaying final output of the triangle example

The following example shows what a student might enter into the interpreter while attempting to draw a dashed triangle, as shown in Figure 1. The student would begin by drawing a few dashed lines with the commands:

```
? forward 10     ; moves the pen 10 pixels in its current direction
? penup          ; lifts the pen so that it does not draw
? forward 10     ; moves the pen 10 pixels in its current direction
? pendown        ; puts the pen down so that it draws again
```

Comments are signaled by the ; character, which is in effect for the remainder of the line. Then, realizing how much effort it would take to type that into the interpreter 10 times, the student may experiment with a different tactic to draw one side of the triangle:

```
? repeat 10 [forward 10 penup forward 10 pendown]
```

After verifying that this worked, the student could then execute the following commands in the interpreter to draw a dashed triangle, step by step:

```
? repeat 10 [forward 10 penup forward 10 pendown]
? right 120     ; turns the pen 120 degrees to the right
? repeat 10 [forward 10 penup forward 10 pendown]
? right 120     ; turns the pen 120 degrees to the right
? repeat 10 [forward 10 penup forward 10 pendown]
```

Modularity

Each procedure is defined as a stand-alone entity that is available to be used within other Logo procedures as subprocedures to build even larger modules. For a student that wants to create a dashed triangle procedure, they would simply take the code executed successfully via the interpreter and make it a procedure. Having had so much fun learning about the `repeat` command, they might implement the function this way (the ~ character is for line continuation, meaning that the current line continues on the next line):

```
to drawDashedTriangle
     repeat 3 ~           ; repeat the given instructions 3 times
          [repeat 10 ~    ; repeat the given instructions 10 times
               [forward 10 ~   ; move the pen forward 10 pixels
                penup ~        ; lift the pen
                forward 10 ~   ; move the pen forward 10 pixels
                pendown] ~     ; return the pen
          right 120]      ; turn the pen 120 degrees
end
```

Extensibility

Logo allows users to extend the "language" it understands by creating new procedures or macros, which tell Logo “how to” perform something, and also using them in a variety of ways. They can be used as control structures, and they can act as both operations and commands.

For example, `ifelse` can be used as an operation to return a value as follows:

```
? print ifelse emptyp [] [sum 2 3] [sum 6 7]
5
```

The ifelse command can also be used as a command to execute other Logo procedures:

```
? ifelse 4 = 2 + 2 [print “Y] [print “N]
Y
```

Flexibility

Logo allows variables to hold any type of value without being defined as a specific type (or needing to redefine it to a different type). This allows users to easily implement a procedure that handles a variety of data types, as well as not having to specify specific bounds (e.g. array size).

For example, the built-in `count` procedure can be executed with both a word or list parameter, and will count the number of elements either as characters in a word or elements of a list

appropriately.

```
? print count "word
4

? print count [The quick brown fox jumped over the lazy dog]
9

? print count [[element1] element2 [[element3] [element3 part2]]
element4]
4
```

Memory Management

The latest version of UCB Logo implements a generational garbage collector. As we discussed in class, the GC will only release pointers that were recently created, unless a full garbage collection is requested either manually with the `gc` command or by virtue of extremely low available memory.

Types

There are three basic data types in Logo: a word, a list, and an array.

A word is a group of zero or more continuous alphanumeric and punctuation characters. Each word evaluates to a specific value. If Logo finds a word, it interprets it as a procedure or variable and attempts to evaluate it unless it is preceded by a quotation mark (`"`). If it is preceded by a quotation mark, Logo will treat the word as itself. For example, if Logo finds `word`, it will attempt to find the value of the variable named `word`, or execute the instruction named `word`. If Logo finds `"word` in an instruction, then it will treat it as a string, and not a variable or procedure.

Words do not require a trailing quotation mark. A number is a special case of a word that evaluates to itself when appearing with or without requiring a quotation mark. A single alpha character is considered a word. If a word has zero characters, then it is considered to be an empty word.

A list is a group of zero or more words or lists combined by a matching set of square brackets. An example of a list is `[one [2 3] four [five!]]`. As with words, if a list has zero members, then it is considered to be an empty list. Lists cannot be altered after their creation, in order to change a value in a list you would need to create a new list with the first portion of the list, the new value, and the end of the list.

An array is a group of zero or more words or lists combined by a matching set of braces. An example of an array is `{1 2 3}`. To create an empty array, a user must use the `array` operation. To retrieve an item in the array, a user must use the `item` operation. Arrays are very similar to lists, but the key difference is that you can modify an array after creation, one element at a time. This is done with the `setitem` command.

Users may also create abstract data types that are based on the basic data types. For example, a user can create a stack data type based on a list, with the top of the stack being the 1st element (e.g. word, list) in the list. A user can also create procedures to manipulate abstract data types.

For example, the push and pop operations of a stack can be created by a user.

There is no static typing, since the interpreter uses dynamic typing by context.

Type Compatibility and Inference

For many of the built-in Logo procedures, any input type will be accepted. The example of `count` mentioned earlier in the report is an example of this fact. Usually as long as it seems that the operation or command makes sense with a given type or format of input, then the operation or command is executed. This is determined via type inference.

In the event that a provided input type isn't applicable to a given procedure, an error is displayed when it is determined at runtime, as follows:

```
? print sum "word "word
sum doesn't like word as input
```

This is because it doesn't make sense to add two strings. A similar error occurs if you attempt to add a string to an integer as well. If the input provided is not in the correct format, even if it is the correct type, the same error is output:

```
? print first []
first doesn't like [] as input
```

This is because first, which returns the first element of a list or word, can't find a first element if the list is empty.

Name and Binding

Each variable has 2 parts: name and thing. Name is the variable's name, and thing represents the value of the variable. A variable can hold any type of value (e.g. word, list, array). To retrieve a variable's value, a user can enter `thing "name`. A user can also enter `:name` (the colon is referred to as "dots") as a shorthand way of stating `thing "name`.

This illustrates the fact that a variable acts more like a pointer reference to some place in memory that holds the value. This is evident when executing instructions such as `make "x :y`, that changes the thing of `x` to point to the value of `y`.

Scope

Logo has dynamic scope since it will first refer to a variable that is local to a procedure, and if no variable of the same name is defined in that procedure, it will look for a variable of the same name that belong to the procedure's superprocedures (i.e. procedures that invoke the current procedure at run-time).

Variables do not need to be declared before use – if they are declared in an expression then they are automatically global. Variables can be declared local, which makes its scope limited to the procedure in which it is defined and any procedures that the defining procedure calls. Input arguments to a procedure are also local values.

In the following example, a and b are local variables to the procedure caller, and b is a local variable to the procedure callee, because parameters to a procedure are local.

```
to caller :a :b
     print :a
     print :b
     callee "c
end

to callee :b
     print :a
     print :b
end

? caller "a "b
a
b
a
c
```

Executing the `caller` procedure with the parameters “a and “b gives the result a, b, a, c, because `callee` has access to the local variables in `caller`, but its own b variable has tighter scope.

Procedures

Logo has approximately 200 "primitive" procedures that handle many of the basic functions to create and execute code. Each procedure has a defined number and type of inputs (parameters). Procedure names appear before its parameters, except for infix arithmetic operations (e.g. +, -, *, /). There are also primitive procedures for these operations that can be used as prefix operators (e.g. product 2 3, sum 4 5).

There are 2 types of procedures: commands and operations. A command does not have an output, but performs some effect such as printing something to the screen. An operation computes a value and returns it as the procedure’s output (e.g. sum takes 2 numbers and returns the numerical value equal to adding the 2 numbers together).

A set of one of more procedures make up an instruction (e.g. print sum 2 product 3 4).

To define new procedures, a user begins by entering the words `to <name>` (e.g. `to Hello`). Logo sees this instruction as the user telling it "how to" perform this procedure. A procedure name must be one word only and may include certain punctuation marks (e.g. `push, pop, pop.all`). The user then enters a set of instructions. To complete the procedure, a user will enter `end` to notify Logo that the procedure has been completely defined.

Procedures may accept user input by entering the word `readchar` or `readlist` which is an operation that reads input on a single line (until a RETURN is read) and produces a character or list as its output.

Nested Procedures

Logo does not support nested procedures. Every procedure is a separate entity that must be defined at the top level (e.g. Logo interpreter user prompt, source code file).

Parameters

Logo will begin to evaluate a procedure by verifying that the correct number of parameters has been specified, as well as the correct type of each parameter. Logo will evaluate a word, list, or array to determine its value. If a word matches the name of a defined procedure, it will evaluate that procedure first and consider its "output" (return) value as the input for the current procedure. Once the required number and type of inputs are confirmed, then Logo will evaluate the procedure and return its output. Otherwise, it will report an error.

Logo will report an error if there are more inputs than required by the current procedure, unless the procedure name and inputs are surrounded by matching parentheses. This tells Logo to override the procedure to accept a different number of inputs.

Higher Order Functions

Logo provides procedures that can accept other procedures as inputs. Logo does not treat procedures as data that can be manipulated, as other languages such as Scheme do. Rather, manipulate other procedures in certain ways.

The command `text` takes the name of a procedure as its only input, and returns the contents of the procedure. Running show text will print out the contents of the procedure.

```
? show text "opinion
[[yes no] [print sentence [I like] :yes] [print sentence [I hate] :no]]
```

The command `define` allows a user to override the definition of an existing procedure, or create a new one. This can be used to write programs that write other programs on their own!

```
to ordinals
ord1 [second third fourth fifth sixth seventh] [output first butfirst]
end

to ord1 :names :instr
if emptyp :names [stop]
define first :names list [thing] (lput ":thing :instr)
ord1 (butfirst :names) (lput "butfirst :instr)
end

? ordinals
? po "fifth
to fifth :thing
output first butfirst butfirst butfirst butfirst :thing
end
```

As you can see, each of these commands takes the names of procedures (e.g. `"opinion`) as input, rather than the procedure itself (e.g. `opinion`). There is also another procedure called `cascade` that allows users to repeatedly invoke a procedure on itself. This is similar to

composing on itself many times (e.g. f(f(f(x)))).

Control Structures

Common control structures in Logo include "`ifelse test [do if true list] [do if false list]`", "`while condition [instruction list], until condition [instruction list], repeat number [instruction list]`".

Other control structures, such as `for` are present in some versions of Logo (but not all) because their developers created them using procedures or macros, which we will discuss now.

Inlining

Certain versions of Logo support macro expansion, where a macro serves as a procedure that returns a set of instructions. These instructions are evaluated in the macro caller's context since the instructions are just copied into the caller's body, where regular Logo procedures would be evaluated in their own context. This can be very useful when dealing with local variables within the context of the caller, as with control structures, but also detrimental to the performance of the caller due to the creation of the same set of instructions each time the macro is called.

Every macro is an operation because it must return a set of instructions, so it cannot be used as a command, as some regular Logo procedures can be used.

UCB Logo version 5.5 includes the `.macro` and `.defmacro` procedures to create macros. The `.macro` procedure is invoked in the same way as `to` is invoked. Similarly, `.defmacro` is invoked in the same way that `define` is invoked.

Here is an example of a `.macro` call:

```
.macro returnthing :name
output (list "thing word "" :name)
end
```

If the following program is defined and executed:

```
to testretthing
make "myvar "xyz
print returnthing "myvar
print :myvar
end
```

The macro `returnthing` will return `[thing "myvar]` to `testretthing` and this will be printed to the screen:

```
xyz
xyz
```

There are 2 supporting procedures related to macros:

- `macrop` (or `macro?`) returns true if the input is the name of a defined macro
- `macroexpand` returns the output produced by the macro invoked by input expression

Exceptions

UCB Logo version 5.5 supports exception handling. When an exception occurs, Logo checks whether or not a variable named ERRACT is available. If so, the program will evaluate it to retrieve a list of instructions to execute to handle the error condition. This allows the code to handle exceptions on its own, but also provides the option of allowing programmer interaction.

One useful characteristic of the execution of ERRACT instructions is that a loop will be preempted if another exception is encountered. If the same error condition is produced twice in a row while executing the ERRACT instructions, then control is returned to the top-level (interpreter user-prompt), and the message "Erract loop" is reported to the programmer. If ERRACT invokes PAUSE (allowing the programmer to interactively check the value of active variables), and the user provides an invalid input, then the program will report "Without user interaction" and will wait for the programmer (user) to provide another input.

If the ERRACT variable is not available to handle an error, then Logo will call THROW "ERROR. The program is expected to provide a CATCH "ERROR instruction to handle and report this error condition, otherwise the top-level will report the error. A programmer can also use the throw and catch instructions to perform their own exception handling.

The set of valid error codes are integers 0 through 34. If error codes 0 (Fatal internal error) and 32 (APPLY doesn't like BADTHING as input) are raised, Logo will exit immediately since those errors cannot be handled.

Iterators

UCB Logo has a foreach command that will iterate over a list and perform the specified action for each element in the list. The following example shows some of the values that can be used in the action portion of the foreach call, using the ?, #, and ?REST words.

```
? foreach [a b c] ~   ; foreach element in the list [a b c]
      [print ?   ~    ; print the value of the current element
      print #    ~    ; print the position of the current element
      print ?REST ~   ; print the rest of the data to be processed
      print "-----]   ; mark the end of this iteration
a
1
b c
-----
b
2
c
-----
c
3

-----
```

Functional Programming

Logo is a functional programming language where complicated computations are done by function composition and recursion: solve a larger problem by solving sub-problems first and combining

the results in an appropriate manner.

Recursion is encouraged over sequential programming that primarily use iterative control structures (for, while loops) which produce/modify a partial solution. Mainly because Logo uses lists which work best when they are built up one member at a time.

Other programming languages such as Pascal have a block structure that are better suited for sequential programming since their block structure makes it easier to read iterative code. Logo does not have a block structure which makes it difficult to read iterative code.

Logo is not a pure functional programming language since it allows assignment and mutation operators. Turtle graphic programs also use sequential programming techniques (e.g. iteration) which fall outside of the functional programming paradigm.

Object-oriented programming is not supported by all versions of Logo, but Object Logo was specifically designed and developed to introduce object-oriented programming capability. One key aspect is the ability to manage multiple turtles using a defined set of procedures that are common across all turtle instantiations.

Summary

In summary, Logo is best known as an educational programming language that was designed in 1967 as an interactive interpreter that provides an easy-to-use interface for novice programmers. Logo is a LISP derivative, and features turtle graphics capabilities and language processing. Its main features are interactivity, modularity, extensibility, and flexibility, allowing users to assign different types of values (e.g. word, list, sentence) to a variable and create their own predicates and control structures.

Logo uses dynamic scoping meaning scope is in the context of the procedures that are called during execution. Logo is a compromise between the functional and sequential programming language paradigms, due to its focus on recursion, and use of iteration in turtle graphics.

References

Harvey, Brian. *Symbolic Computing*. Computer Science Logo Style. 1997 Vol 1. [Online]. Available: http://www.cs.berkeley.edu/~bh/v1-toc2.html

Logo Foundation. “What is Logo?” 2000. [Online]. Available: http://el.media.mit.edu/Logo-foundation/logo/index.html

“Logo (programming language).” Wikipedia, The Free Encyclopedia. Apr 2008. Wikimedia Foundation, Inc. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Logo_%28programming_language%29&oldid=205443520.

Harvey, Brian. *Berkeley Logo 5.5 User Manual.* Aug 2005 Available with UCB Logo distribution: /ucblogo/DOCS/usermanual.html