www.logofoundation.org

Creating a Logo Tool Box

by

Brian Silverman and Michael Tempel

It is often said that if Logo doesn't have a particular primitive procedure you need, you can write it in Logo. Here are some tips about how to do it.

Problem Solving vs Implementing Solutions

Most programming languages are designed for implementing the solutions to problems. The ideology, if not always the practice, of many data processing professionals is that problems should first be solved in broad outline, then refined and only at the last step translated into a program. Interaction with the computer does not aid in the solution of the problem. The problem has presumably been solved before coding begins.

Logo is ideally suited to a different sort of problem solving. A problem may not be well defined, or exploration may begin without a specific problem in mind. Logo provides an interactive environment in which thinking may be expressed and reflected upon. Problems are explored and solved while interacting with Logo.

Most programming languages are well suited to a particular class of problems. This is quite reasonable when the problems are generally familiar and predictable such as managing the company payroll or processing tax returns. For example, Cobol makes it relatively easy to format and display numerical information. It is good for business applications. Fortran is designed for solving equations and is useful to engineers. Each language contains the tools needed for its special class of problems.

Logo is more general. If problems are loosely defined, you may not know ahead of time what tools you will need. Therefore it's best to have what you need to make tools as you go.

If you were messing around with probability and statistics, you might want to have procedures that compute **factorials**, **means**, **medians** and various kinds of random distributions. Logo gives you the basic arithmetic operations and a simple random number generator. You can use these to build what else you need.

If you were working with natural language, you might want procedures that reverse lists, delete words from lists or insert words into lists. The simple list manipulation procedures provided in Logo may be used to create such tools.

Why not include all these various tools in Logo instead of asking people to make them? For one thing, such an approach would produce an enormous Logo. More important, since Logo is designed for exploration and the creation of new knowledge, the designers of Logo can't presume to know what tools will be needed by users. A well designed Logo provides flexible general purpose building blocks.

Literacy vs Fluency

You might be feeling a bit uneasy about the preceding discussion. You have been handed a Logo. Now get out there and explore, making your tools as you go. Oh, by the way, you do, of course, know how to write procedures like **factorial**, **median**, **reverse**, **delete** and **insert**, don't you?

Well, if you don't, stick around for the rest of this workshop.

We will start by going through a series of procedures that illustrate various points about Logo that are important for tool building. Things generally get more difficult as we go.

The first step is to reach a level of literacy with these procedures. That is, you should be able to follow how they work, not be surprised at their results and even predict outcomes.

Beyond literacy you will achieve fluency, the ability to create your own procedures of the kinds illustrated.

Procedures

Logo programs are built out of procedures. Some of these procedures are *primitives.* You may also define your own procedures. The only important difference between these two categories of procedures is that the scripts of the primitives are written in machine language while your procedures have Logo scripts. There are other, more important distinctions between types of procedures.

Here are some procedures you are probably familiar with. They are primitives. What categories can you divide them into?

forward
cleartext
print
word
heading
name
setpos
pos
home

Here's one categorization:

forward	word
cleartext	heading
setpos	pos
name	
print	
home	

The procedures in the first column are *commands.* The others are *reporters* (also called operations). Commands do something. reporters report, or output a Logo Thing. Logo Things may be words or lists. When a procedure outputs a Logo Thing, there must be another procedure there to accept it as an input. This leads us to a second way to divide our list of procedures:

cleartext	heading
home	pos

forward	word
name	
print	
setpos	

Cleartext and **home** are commands that do not take any inputs. **Forward, name**, **print** and **setpos** are commands that require inputs. If you type **forward** with no number following it, Logo will complain. **Forward 50** is fine.

Heading and **pos** are **reporters** that don't require inputs. **Word** needs two inputs.

Let's look at some interactions with Logo that show how all this works:

?Cleartext	This command clears text from the screen.
?forward Not enough inputs to forward	This command needs an input.
?forward 100	This works.
?word "Logo "Writer I don't know what to do with LogoWriter	Word is a reporter but there's no procedure to accept its output as an input.
?word 7 5 I don't know what to do with 75	The same problem.
?forward word 7 5	The turtle moves forward 75 steps.

You probably know how to write your own commands in Logo:

```
to square
repeat 4 [fd 50 rt 90]
end
```

Square is a command that takes no inputs. Here's a version of **square** that takes one input:

```
to square :side
repeat 4 [fd :side rt 90]
end
```

In order to write a reporter in Logo you have to use the primitive **Output**. Here's an

example:

```
to five
output 5
end

?print five + five
10
```

Output takes a Logo Thing as an input. The number 5 is the input to **output**. **Output** stops the procedure **Five**. **Output** also causes **Five** to be a reporter whose output is 5. In general, the input to **output** becomes the output of the procedure. (Is that not less than unclear?)

Here's an example of a reporter that requires an input:

```
to double :number
output :number + :number
end

?print double 4
8
?print double five
10
```

Here's a procedure that pads one and two digit numbers to three digits and leaves numbers with three or more digits alone.

```
to padnumber :num
if :num < 10 [output word "00 :num]
if :num < 100 [output word "0 :num]
output :num
end
```

Notice that if **:num** is less than 10, the second line is never reached because **output** in the first line stops **padnumber**. If **:num** is 10 or more but less than 100, **padnumber** stops at the second line and the last line is not reached. If **:num** is 100 or more the last line is executed.

Recursion

It turns out that many useful Logo reporters are recursive. Before looking at some recursive reporters, here are some recursive commands:

```
to countdown :num
if :num < 1 [stop]
print :num
countdown :num - 1
end

to countdown :num
if :num < 1 [stop]
print :num
countdown :num - 1
print :num
end
```

Does the behavior of the second version surprise you? If so, here are some things to think about:

- When a Logo procedure calls a *subprocedure* it waits until the subprocedure finishes and then continues. A procedure doesn't go away when it calls a subprocedure.
- Primitive procedures and user-defined procedures work the same way.
- Procedures don't care what the names of their subprocedures are.
- Subprocedures don't care about the names of the procedures that call them.

Here's another pair of procedures that are equivalent to **countdown:**

```
to squiral :side
if :side > 100 [stop]
forward :side right 90
squiral :side + 5
end

to squiral :side
if :side > 100 [stop]
forward :side right 90
squiral :side + 5
left 90 back :side
end
```

Recursive Reporters

Here's an example of a recursive reporter:

```
to factorial :number
if :number = 0 [output 1]
output :number * factorial :number - 1
end

?print factorial 0
1
?print factorial 5
120
```

This translates into English as: "The factorial of 0 is 1. The factorial of a number greater than 0 is the number times the factorial of one less than the number."

```
to sum.of :list
if empty? :list [output 0]
output (first :list) +
(sum.of butfirst :list)
end

?print sum.of [1 2 3 2 5]
13
```

In English this is: "If a list is empty, then the sum of the numbers in it is 0. Otherwise, the sum of a list of numbers is the first number plus the sum of the rest of the numbers in the list".

Here's one more example:

```
to reverse :list
if equal? 1 count :list [output :list]
output lput first :list reverse bf :list
end
```

"The reverse of a list of one element is just that list. The reverse of a longer list is the first element of the list stuck on the end of the reverse of the rest of the list."

Here are some rules to observe when writing recursive reporters:

1. There must be at least two different ways of handling the inputs, a simple one that immediately produces an answer and another one that, in part, uses the same procedure to produce the answer.

   In **factorial**, an input of 0 produces 1 as an answer. Otherwise **factorial**, in part, uses **factorial** to compute an answer.

Both parts are needed. Try running this:

```
to factorial :number
output :number * factorial :number - 1
end
```

or this

```
to factorial :number
if :number = 0 [output 1]
end
```

2. The input to the recursive subprocedure must be different from the input to the procedure that called it. The input to the subprocedure **factorial** is :**number- 1**. The input to the subprocedure **sum.of** is **butfirst :list**. If the input to the subprocedure were not different from the input to the calling procedure, there would be no way for the simple case to ever be reached. Try running each of these altered versions of **factorial**, **sum.of** and **reverse:**

```
to sum.of :list
if empty? :list [output 0]
output (fisrt :list) + (sum.of :list)
end

to factorial :number
if :number = 0 [output 1]
output :number * factorial :number
end

to reverse :list
if equal? 1 count :list [output :list]
output lput first :list reverse :list
end
```

3. All cases must result in something being **output**.

   Here's a procedure that doesn't work:

```
to reverse :list
if equal? 1 count :list [output :list]
lput first :list reverse butfirst :list
end

Here's another flawed procedure:

to sum.of :list
```

```
if empty? :list [stop]
output (first :list) +
(sum.of butfirst :list)
end
```

Look at the appendix for some more interesting examples of recursion.

Transportable Procedures

Good tools are useful in many contexts. You should be able to use the same Logo procedure as a part of several programs. There may, for example, be different situations in which you will need to find the average of some numbers.

Here's one way of doing this:

```
to average :number.list
output (sum.of :number.list) /
(count :number.list)
end

?print average [1 2 3 4 5 6 7]
4
```

Here's another way:

```
to average
name [1 2 3 4 5 6 7] "numbers
name sum.of :numbers "total
name count :numbers "count
name :total / :count "average
end

?average
?print :average
4
```

The first version of **average** may be added into any Logo program that doesn't already have a procedure called **average**. You hand the procedure a list of numbers and you get back the average. What goes on inside **average** does not affect names or other procedures that may be around. It works with any list of numbers as input.

The second version works with only a specific list of numbers. It also uses the global names **numbers, total, count** and **average**. If these names already exist, their values will be changed by the procedure **average**.

In general, tools should be easily movable from one program to another. In most

cases, global names are to be avoided. The procedure should not produce side effects upon the workspace or be inadvertently affected by what is already around.

Projects

Here are some suggested projects on which to exercise your tool building skills:

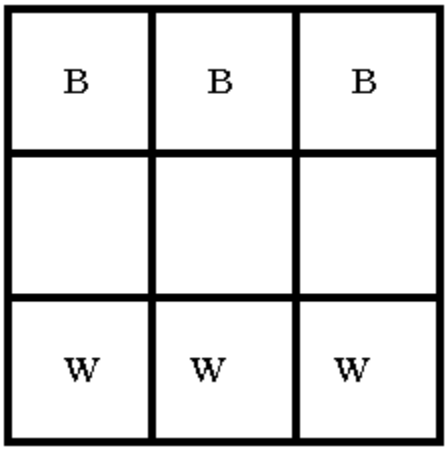
Games

Computers may be used to play games, such as chess or tic tac toe, in several, increasingly complex ways:

1. The computer displays the game board. It accepts moves that are input by human players and simply shows the resulting positions.
2. In addition to displaying moves, the computer knows what the legal moves are and prevents illegal moves from being made.
3. The program plays against a human. It makes legal moves, but has no strategy for winning.
4. The program has a playing strategy based on learning from previous games. It has no initial game plan other than to make legal moves, but it learns not to repeat fatal errors made in prior games.
5. The program has a strategy built in from the start. This strategy may be modified by experience.

The collection of procedures that you need at each step may be incorporated into the program you create at the next level. Different people may work on different aspects of the solution and mesh their results together.

Here's a very simple game to try this on. It's called Hexapawn. It is played on an abbreviated chess board that is three squares on a side. Each player has three pawns. The opening position looks like:



The legal moves in Hexapawn are the same as for the pawns in chess. A pawn may

move forward to the space in front of it if that space is vacant. A pawn may capture an opposing piece by moving diagonally forward. The game ends in one of three ways:

- If a player can make no legal moves his opponent wins.
- If a player captures all of his opponent's pieces he wins.
- If a player moves a piece to the opposite side of the board he wins.

The game is limited enough to be of little interest to people, but still presents a reasonable challenge for a computer.

After working with Hexapawn, you might make things more complicated by playing Octopawn on a four by four board. Then try tic tac toe and possibly chess.

When you get up to working on programming playing strategies you should be aware of two general approaches to take. First, you can program all possible game outcomes. If your program knows all of the possibilities that can occur from a given point on, it also knows which moves will win. It can always select the correct move based on this knowledge. This works well for hexapawn and even tic tac toe, but it would take the fastest computers available about 30 billion years to run through all the possible outcomes of a chess game. Since you only have an Apple, it would take even longer. It is this impracticality of an "exhaustive search" strategy that makes chess playing programs interesting. Some rules and approaches need to be programmed.

Graphs

Make a collection of tools that produce bar or line graphs from some data. Some of the tools you would need would:

- draw a bar of specified size in a specified position
- plot points
- connect plotted points
- compute averages and totals
- draw axes and scale them properly
- put titles and labels on the graph

Pluralization

Write a program to pluralize any word you give it as input. It should work like this:

```
?print plural "dog
dogs
?print plural "mouse
mice
?print plural "fox
foxes
```

As you work on this, you'll find that the Logo primitives **first**, **last**, **butfirst**, **word** etc. may not always be convenient to use directly. You might want to construct tools like **last.two** or **next.to.last**. Other tools such as **vowel?** might be useful.

Logo for Little Kids

Attempts have been made to make Logo accessible to very young children. One category of solutions rests on the assumption that kids can't find the **Return** key. These "Instant" type programs move the turtle with single keystrokes. While easy to start using, these programs are discontinuous with Logo as a whole since any Logo procedures not assigned to keystrokes are unavailable, and the method of invoking procedures - without pressing **Return** - will not apply when the Instant program is discarded.

Can you create an Instant program that also allows the use of regular Logo commands?

Instead of an Instant program, can you write a collection of tools that simplify Logo in similar ways, but work at top level? For example, write a procedure **f** that causes the turtle to go **forward 10**.

Try making a "slow turtle" whose moves and turns are slower than normal so that the drawing process may be more easily followed.

Create a collection of procedures that scale the turtle movements so that small numbers produce big effects.

You might also want to create tools like **square**, **circle, rectangle** and **triangle**. Using these procedures children can achieve satisfying results before they are able to program such shapes themselves.

Appendix

Here are some interesting recursive Logo procedures:

```
to printvals :list
if empty? :list [stop]
printval first :list
printvals bf :list
end

to printval :var
type :var type char 32
if name? :var [pr thing :var][pr "\-\-\-]
end
```

```
?printvals [num size foo]
```

will print the values of num, size and foo if they are names. If they're not names, --- is printed.

```
to map :func :list
if empty? :list [stop]
run se :func [first :list]
map :func bf :list
end

to square :side
repeat 4 [fd :side rt 90]
end

?map "square [10 20 30 40 50]
```

displays five squares of the specified sizes.

```
to assign :names :vals
if empty? :names [stop]
make first :names first :vals
assign bf :names bf :vals
end

?assign [n1 n2 n3 n4] [10 20 30 40]
?shownames
:n1 is 10
:n2 is 20
:n3 is 30
:n4 is 40
```

This procedure plays the Tower of Hanoi game:

```
to hanoi :from :to :spare :n
if :n = 1 [(pr :from "to :to) stop]
hanoi :from :spare :to :n - 1
(pr :from "to :to)
hanoi :spare :to :from :n - 1
end
```

Try this:

```
?hanoi "A "B "C 1
A to B
?hanoi "A "B "C 2
```

```
A to C
A to B
C to B
?hanoi "A "B "C 3
A to B
A to C
B to C
A to B
C to A
C to B
A to B
?hanoi "A "B "C 10

to same.as? :a :b
if word? :a [output :a = :b]
if word? :b [output "false]
if subset? :a :b [output subset :b :a]
output "false
end

to subset? :a :b
if empty? :a [output "true]
if part.of? first :a :b [output subset? butfirst :a :b]
output "false
end

to part.of? :a :b
if empty? :b [output "false]
if same.as? :a first :b [output "true]
output part.of? :a butfirst :b
end

?print same.as? [a b c] [b c a]
true
?print same.as? [a b c] [b c [a]]
false
?print same.as? [[a b c] d e] [e [c a b] d]
true
?print subset? "a [a b c]
true
?print subset? [a] [a b c]
false
?print subset? [a] [[a][b][c]]
true
?print subset? [a b] [b a c]
true
```

```
?print part.of? "a [a b c]
true
?print part.of? [a][a b c]
false
```